

GumNaaM_Helpers



Mega File for MiD TerM

All Files in ONE FILE

(Handouts_waqar_moaz_junaaid_vu toper_orange_monkey)

Alhamdulillah GumNaaM_Helpers Bs..IT/CS ky Tamam Semesters ki tamam Books ki help krty hn.

Group Join krty hi description Read krna na bholyn

Semester Name	Group Link
Bs..IT/CS...smstr:1	https://chat.whatsapp.com/Ciya2X0DI0l8iQ14m0VNNj
Bs..IT/CS...smstr:2	https://chat.whatsapp.com/Kwh85sRK02z1Qk5B7aMR19
Bs..IT/CS...smstr:3	https://chat.whatsapp.com/ljlpdzi8pIA4lQVNUMmNYE
Bs..IT/CS...smstr:4	https://chat.whatsapp.com/CiGpHsSftir0oyhM47WfKh
Bs..IT/CS...smstr:5	https://chat.whatsapp.com/Hf7NWWhdW06uCW7Mbt1n4Db
Bs..IT/CS...smstr:6	https://chat.whatsapp.com/DyAoT7VFkwy8DuvrYDgm2L
Bs..IT/CS...smstr:7	https://chat.whatsapp.com/ENiyfIDRWatJFPbXCUGYNN
Bs..IT/CS...smstr:8	https://chat.whatsapp.com/Bnkk4n7utpdEuXazHLxRcA

...Paid Service Available for LMS Activities...

Only for those students who are so busy in office or work or job that they can't watch their LMS due to their Busyness

FOR MORE INFO. CONTACT US ONLY WHAT'S APP 0302-1500025

Data Structures

Table of Contents

Data Structures.....	1
Lecture No. 01.....	3
Lecture No. 02.....	12
Lecture No. 03.....	20
Lecture No. 04.....	32
Lecture No. 05.....	46
Lecture No. 06.....	56
Lecture No. 07.....	63
Lecture No. 08.....	71
Lecture No. 09.....	82
Lecture No. 10.....	93
Lecture No. 11.....	105
Lecture No. 12.....	123
Lecture No. 13.....	135
Lecture No. 14.....	146
Lecture No. 15.....	158
Lecture No. 16.....	170
Lecture No. 17.....	189
Lecture No. 18.....	203
Lecture No. 19.....	211
Lecture No. 20.....	220
Lecture No. 21.....	231
Lecture No. 22.....	240
Lecture No. 23.....	257
Lecture No. 24.....	273
Lecture No. 25.....	284
Lecture No. 26.....	296
Lecture No. 27.....	307
Lecture No. 28.....	321
Lecture No. 29.....	333
Lecture No. 30.....	348
Lecture No. 31.....	360
Lecture No. 32.....	370
Lecture No. 33.....	379
Lecture No. 34.....	386
Lecture No. 35.....	393
Lecture No. 36.....	404
Lecture No. 37.....	416
Lecture No. 38.....	424
Lecture No. 39.....	430
Lecture No. 40.....	441
Lecture No. 41.....	449
Lecture No. 42.....	459
Lecture No. 43.....	468
Lecture No. 44.....	474
Lecture No. 45.....	485

Data Structures

Lecture No. 01

Reading Material

Data Structures and algorithm analysis in C++

Chapter. 3

3.1, 3.2, 3.2.1

Summary

- Introduction to Data Structures
- Selecting a Data Structure
- Data Structure Philosophy
- Goals of this Course
- Array
- List data structure

Welcome to the course of data structure. This is very important subject as the topics covered in it will be encountered by you again and again in the future courses. Due to its great applicability, this is usually called as the foundation course. You have already studied Introduction to programming using C and C++ and used some data structures. The focus of that course was on how to carry out programming with the use of C and C++ languages besides the resolution of different problems. In this course, we will continue problem solving and see that the organization of data in some cases is of immense importance. Therefore, the data will be stored in a special way so that the required result should be calculated as fast as possible.

Following are the goals of this course:

- Prepare the students for (and is a pre-requisite for) the more advanced material students will encounter in later courses.
- Cover well-known data structures such as dynamic arrays, linked lists, stacks, queues, trees and graphs.
- Implement data structures in C++

You have already studied the dynamic arrays in the previous course. We will now discuss linked lists, stacks, queues, trees and graphs and try to resolve the problems with the help of these data structures. These structures will be implemented in C++ language. We will also do programming assignments to see the usage and importance of these structures.

Information about Data Structure subject is available at: “<http://www.vu.edu.pk/ds>”.

Introduction to Data Structures

Let's discuss why we need data structures and what sort of problems can be solved with their use. Data structures help us to organize the data in the computer, resulting in more efficient programs. An efficient program executes faster and helps minimize the usage of resources like memory, disk. Computers are getting more powerful with the passage of time with the increase in CPU speed in GHz, availability of faster network and the maximization of disk space. Therefore people have started solving more and more complex problems. As computer applications are becoming complex, so there is need for more resources. This does not mean that we should buy a new computer to make the application execute faster. Our effort should be to ensure that the solution is achieved with the help of programming, data structures and algorithm.

What does organizing the data mean? It means that the data should be arranged in a way that it is easily accessible. The data is inside the computer and we want to see it. We may also perform some calculations on it. Suppose the data contains some numbers and the programmer wants to calculate the average, standard deviation etc. May be we have a list of names and want to search a particular name in it. To solve such problems, data structures and algorithm are used. Sometimes you may realize that the application is too slow and taking more time. There are chances that it may be due to the data structure used, not due to the CPU speed and memory. We will see such examples. In the assignments, you will also check whether the data structure in the program is beneficial or not. You may have two data structures and try to decide which one is more suitable for the resolution of the problem.

As discussed earlier, a solution is said to be efficient if it solves the problem within its resource constraints. What does it mean? In the computer, we have hard disk, memory and other hardware. Secondly we have time. Suppose you have some program that solves the problem but takes two months. It will be of no use. Usually, you don't have this much time and cannot wait for two months. Suppose the data is too huge to be stored in disk. Here we have also the problem of resources. This means that we have to write programs considering the resources to achieve some solution as soon as possible. There is always cost associated with these resources. We may need a faster and better CPU which can be purchased. Sometimes, we may need to buy memory. As long as data structures and programs are concerned, you have to invest your own time for this. While working in a company, you will be paid for this. All these requirements including computer, your time and computer time will decide that the solution you have provided is suitable or not. If its advantages are not obtained, then either program or computer is not good.

So the purchase of a faster computer, while studying this course, does not necessarily help us in the resolution of the problem. In the course of “Computer Architecture” you will see how the more efficient solutions can be prepared with the hardware. In this course, we will use the software i.e. data structures, algorithms and the recipes through which the computer problems may be resolved with a faster solution.

Selecting a Data Structure

How can we select the data structure needed to solve a problem? You have already studied where to use array and the size of array and when and where to use the pointers etc. First of all, we have to analyze the problem to determine the resource constraints that a solution must meet. Suppose, the data is so huge i.e. in Giga bytes (GBs) while the disc space available with us is just 200 Mega bytes. This problem can not be solved with programming. Rather, we will have to buy a new disk.

Secondly, it is necessary to determine the basic operations that must be supported. Quantify the resource constraints for each operation. What does it mean? Suppose you have to insert the data in the computer or database and have to search some data item. Let's take the example of telephone directory. Suppose there are eight million names in the directory. Now someone asks you about the name of some particular person. You want that this query should be answered as soon as possible. You may add or delete some data. It will be advisable to consider all these operations when you select some data structure.

Finally select the data structure that meets these requirements the maximum. Without, sufficient experience, it will be difficult to determine which one is the best data structure. We can get the help from internet, books or from someone whom you know for already getting the problems solved. We may find a similar example and try to use it. After this course, you will be familiar with the data structures and algorithms that are used to solve the computer problems.

Now you have selected the data structure. Suppose a programmer has inserted some data and wants to insert more data. This data will be inserted in the beginning of the existing data, or in the middle or in the end of the data. Let's talk about the arrays and suppose you have an array of size hundred. Data may be lying in the first fifty locations of this array. Now you have to insert data in the start of this array. What will you do? You have to move the existing data (fifty locations) to the right so that we get space to insert new data. Other way round, there is no space in the start. Suppose you have to insert the data at 25th location. For this purpose, it is better to move the data from 26th to 50th locations; otherwise we will not have space to insert this new data at 25th location.

Now we have to see whether the data can be deleted or not. Suppose you are asked to delete the data at 27th position. How can we do that? What will we do with the space created at 27th position?

Thirdly, is all the data processed in some well-defined order or random access allowed? Again take the example of arrays. We can get the data from 0th position and traverse the array till its 50th position. Suppose we want to get the data, at first from 50th location and then from 13th. It means that there is no order or sequence. We want to access the data randomly. Random access means that we can't say what will be the next position to get the data or insert the data.

Data Structure Philosophy

Let's talk about the philosophy of data structure. Each data structure has costs and benefits. Any data structure used in your program will have some benefits. For this,

you have to pay price. That can be computer resources or the time. Also keep in mind that you are solving this problem for some client. If the program is not efficient, the client will not buy it.

In rare cases, a data structure may be better than another one in all situations. It means that you may think that the array is good enough for all the problems. Yet this is not necessary. In different situations, different data structures will be suitable. Sometimes you will realize that two different data structures are suitable for the problem. In such a case, you have to choose the one that is more appropriate. An important skill this course is going to lend to the students is use the data structure according to the situation. You will learn the programming in a way that it will be possible to replace the one data structure with the other one if it does not prove suitable. We will replace the data structure so that the rest of the program is not affected. You will also have to attain this skill as a good programmer.

There are three basic things associated with data structures. A data structure requires:

- space for each data item it stores
- time to perform each basic operation
- programming effort

Goals of this Course

Reinforce the concept that costs and benefits exist for every data structure. We will learn this with practice.

Learn the commonly used data structures. These form a programmer's basic data structure “toolkit”. In the previous course, you have learned how to form a loop, functions, use of arrays, classes and how to write programs for different problems. In this course, you will make use of data structures and have a feeling that there is bag full of different data structures. In case of some problem, you will get a data structure from the toolkit and use some suitable data structure.

Understand how to measure the cost of a data structure or program. These techniques also allow you to judge the merits of new data structures that you or others might develop. At times, you may have two suitable data structures for some problem. These can be tried one by one to adjudge which one is better one. How can you decide which data structure is better than other. Firstly, a programmer can do it by writing two programs using different data structure while solving the same problem. Now execute both data structures. One gives the result before the other. The data structure that gives results first is better than the other one. But sometimes, the data grows too large in the problem. Suppose we want to solve some problem having names and the data of names grows to 10 lakhs (one million). Now when you run both programs, the second program runs faster. What does it mean? Is the data structure used in program one not correct? This is not true. The size of the data, being manipulated in the program can grow or shrink. You will also see that some data structures are good for small data while the others may suit to huge data. But the problem is how can we determine that the data in future will increase or decrease. We should have some way to take decision in this regard. In this course we will do some mathematical analysis and see which data structure is better one.

Arrays

You have already studied about arrays and are well-versed with the techniques to utilize these data structures. Here we will discuss how arrays can be used to solve computer problems. Consider the following program:

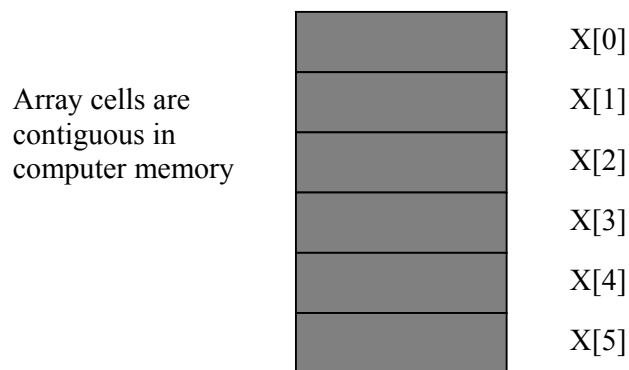
```
main( int argc, char** argv )
{
    int x[6];
    int j;
    for(j = 0; j < 6; j++)
        x[j] = 2 * j;
}
```

We have declared an *int* array of six elements and initialized it in the loop.

Let's revise some of the array concepts. The declaration of array is as *int x[6]*; or *float x[6]*; or *double x[6]*; You have already done these in your programming assignments. An array is collection of cells of the same type. In the above program, we have array *x* of type *int* of six elements. We can only store integers in this array. We cannot put *int* in first location, *float* in second location and *double* in third location. What is *x*? *x* is a name of collection of items. Its individual items are numbered from zero to one less than array size. To access a cell, use the array name and an index as under:

x[0], x[1], x[2], x[3], x[4], x[5]

To manipulate the first element, we will use the index zero as *x[0]* and so on. The arrays look like in the memory as follows:



Array occupies contiguous memory area in the computer. In case of the above example, if some location is assigned to *x[0]*, the next location can not contain data other than *x[1]*. The computer memory can be thought of as an array. It is a very big array. Suppose a computer has memory of 2MB, you can think it as an array of size 2 million and the size of each item is 32 bits. You will study in detail about it in the computer organization, and Assembly language courses. In this array, we will put our programs, data and other things.

In the above program, we have declared an array named x . ' x ' is an array's name but there is no variable x . ' x ' is not an *lvalue*. If some variable can be written on the left-hand side of an assignment statement, this is *lvalue* variable. It means it has some memory associated with it and some value can be assigned to it. For example, if we have the code `int a, b;` it can be written as `b = 2;` it means that put 2 in the memory location named b . We can also write as `a = b;` it means whatever b has assign it to a , that is a copy operation. If we write as `a = 5;` it means put the number 5 in the memory location which is named as a . But we cannot write `2 = a;` that is to put at number 2 what ever the value of a is. Why can't we do that? Number 2 is a constant. If we allow assignment to constants what will happen? Suppose ' a ' has the value number 3. Now we assigned number 2 the number 3 i.e. all the number 2 will become number 3 and the result of $2 + 2$ will become 6. Therefore it is not allowed.

' x ' is a name of array and not an *lvalue*. So it cannot be used on the left hand side in an assignment statement. Consider the following statements

```
int x[6];
int n;
x[0] = 5; x[1] = 2;
x = 3;           //not allowed
x = a + b;       // not allowed
x = &n;          // not allowed
```

In the above code snippet, we have declared an array x of *int*. Now we can assign values to the elements of x as `x[0] = 5` or `x[1] = 2` and so on. The last three statements are not allowed. What does the statement `x = 3;` mean? As x is a name of array and this statement is not clear, what we are trying to do here? Are we trying to assign 3 to each element of the array? This statement is not clear. Resultantly, it can not be allowed. The statement `x = a + b` is also not allowed. There is nothing wrong with $a + b$. But we cannot assign the sum of values of a and b to x . In the statement `x = &n`, we are trying to assign the memory address of n to x which is not allowed. The reason is the name x is not *lvalue* and we cannot assign any value to it. For understanding purposes, consider x as a constant. Its name or memory location can not be changed. This is a collective name for six locations. We can access these locations as `x[0]`, `x[1]` up to `x[5]`. This is the way arrays are manipulated.

Sometimes, you would like to use an array data structure but may lack the information about the size of the array at compile time. Take the example of telephone directory. You have to store one lakh (100,000) names in an array. But you never know that the number of entries may get double or decline in future. Similarly, you can not say that the total population of the country is one crore (10 million) and declare an array of one crore names. You can use one lakh locations now and remaining will be used as the need arrives. But this is not a good way of using the computer resources. You have declared a very big array while using a very small chunk of it. Thus the remaining space goes waste which can, otherwise, be used by some other programs. We will see what can be the possible solution of this problem?

Suppose you need an integer array of size n after the execution of the program. We have studied that if it is known at the execution of the program that an array of size 20 or 30 is needed, it is allocated dynamically. The programming statement is as follows:

```
int* y = new int[20];
```

It means we are requesting computer to find twenty memory locations. On finding it, the computer will give the address of first location to the programmer which will be stored in `y`. Arrays locations are contiguous i.e. these are adjacent. These twenty locations will be contiguous, meaning that they will be neighbors to each other. Now `y` has become an array and we can say `y[0] = 1` or `y[5] = 15`. Here `y` is an *lvalue*. Being a pointer, it is a variable where we can store the address of some variable. When we said `int* y = new int[20];` the `new` returns the memory address of first of the twenty locations and we store that address into `y`. As `y` is a pointer variable, so it can be used on the left-hand side. We can write it as:

```
y = &x[0];
```

In the above statement, we get the address of the first location of the array `x` and store it in `y`. As `y` is *lvalue*, so it can be used on left hand side. This means that the above statement is correct.

```
y = x;
```

Similarly, the statement `y = x` is also correct. `x` is an array of six elements that holds the address of the first element. But we cannot change this address. However we can get that address and store it in some other variable. As `y` is a pointer variable and *lvalue* so the above operation is legal. We have dynamically allocated the memory for the array. This memory, after the use, can be released so that other programs can use it. We can use the *delete* keyword to release the memory. The syntax is:

```
delete[] y;
```

We are releasing the memory, making it available for use by other programs. We will not do it in case of `x` array, as ‘`new`’ was not used for its creation. So it is not our responsibility to delete `x`.

List data structure

This is a new data structure for you. The *List* data structure is among the most generic of data structures. In daily life, we use shopping list, groceries list, list of people to invite to a dinner, list of presents to give etc. In this course, we will see how we use lists in programming.

A list is the collection of items of the same type (grocery items, integers, names). The data in arrays are also of same type. When we say `int x[6];` it means that only the integers can be stored in it. The same is true for list. The data which we store in list should be of same nature. The items, or elements of the list, are stored in some particular order. What does this mean? Suppose in the list, you have the fruit first which are also in some order. You may have names in some alphabetical order i.e. the names which starts with *A* should come first followed by the name starting with *B* and so on. The order will be reserved when you enter data in the list.

It is possible to insert new elements at various positions in the list and remove any

element of the list. You have done the same thing while dealing with arrays. You enter the data in the array, delete data from the array. Sometimes the array size grows and at times, it is reduced. We will do this with the lists too.

List is a set of elements in a linear order. Suppose we have four names $a1, a2, a3, a4$ and their order is as $(a3, a1, a2, a4)$ i.e. $a3$, is the first element, $a1$ is the second element, and so on. We want to maintain that order in the list when data is stored in the list. We don't want to disturb this order. The order is important here; this is not just a random collection of elements but an *ordered* one. Sometimes, this order is due to sorting i.e. the things that start with A come first. At occasions, the order may be due to the importance of the data items. We will discuss this in detail while dealing with the examples.

Now we will see what kind of operations a programmer performs with a list data structure. Following long list of operations may help you understand the things in a comprehensive manner.

Operation Name	Description
<code>createList()</code>	Create a new list (presumably empty)
<code>copy()</code>	Set one list to be a copy of another
<code>clear();</code>	Clear a list (remove all elements)
<code>insert(X, ?)</code>	Insert element X at a particular position in the list
<code>remove(?)</code>	Remove element at some position in the list
<code>get(?)</code>	Get element at a given position
<code>update(X, ?)</code>	Replace the element at a given position with X
<code>find(X)</code>	Determine if the element X is in the list
<code>length()</code>	Returns the length of the list.

`createList()` is a function which creates a new list. For example to create an array, we use `int x[6]` or `int* y = new int[20];` we need similar functionality in lists too. The `copy()` function will create a copy of a list. The function `clear()` will remove all the elements from a list. We want to insert a new element in the list, we also have to tell where to put it in the list. For this purpose `insert(X, position)` function is used. Similarly the function `remove(position)` will remove the element at position. To get an element from the list `get(position)` function is used which will return the element at position. To replace an element in the list at some position the function `update(X, position)` is used. The function `find(X)` will search X in the list. The function `length()` tells us about the number of elements in the list.

We need to know what is meant by “particular position” we have used “?” for this in the above table. There are two possibilities:

- Use the actual index of element: i.e. insert it after element 3, get element number 6. This approach is used with arrays
- Use a “current” marker or pointer to refer to a particular position in the list.

The first option is used in the data structures like arrays. When we have to manipulate the arrays, we use index like $x[3]$, $x[6]$. In the second option we do not use first, second etc for position but say wherever is the current pointer. Just think of a pointer in the list that we can move forward or backward. When we say get, insert or update

while using the current pointer, it means that wherever is the current pointer, get data from that position, insert data after that position or update the data at that position. In this case, we need not to use numbers. But it is our responsibility that current pointer is used in a proper way.

If we use the “current” marker, the following four methods would be useful:

Functions	Description
start()	Moves the “current” pointer to the very first element
tail()	Moves the “current” pointer to the very last element
next()	Move the current position forward one element
back()	Move the current position backward one element

In the next lecture, we will discuss the implementation of the list data structure and write the functions discussed today, in C++ language.

Data Structures

Lecture No. 02

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 3

3.1, 3.2, 3.2.1, 3.2.2

Summary

- 1) List Implementation
 - *add* Method
 - *next* Method
 - *remove* Method
 - *find* Method
 - Other Methods
- 2) Analysis Of Array List
- 3) List Using Linked Memory
- 4) Linked List

Today, we will discuss the concept of list operations. You may have a fair idea of ‘*start* operation’ that sets the current pointer to the first element of the list while the *tail* operation moves the current pointer to the last element of the list. In the previous lecture, we discussed the operation *next* that moves the current pointer one element forward. Similarly, there is the ‘*back* operation’ which moves the current pointer one element backward.

List Implementation

Now we will see what the implementation of the list is and how one can create a list in C++. After designing the interface for the list, it is advisable to know how to implement that interface. Suppose we want to create a list of integers. For this purpose, the methods of the list can be implemented with the use of an array inside. For example, the list of integers (2, 6, 8, 7, 1) can be represented in the following manner where the current position is 3.

A	2	6	8	7	1				<u>current</u>	<u>size</u>
	1	2	3	4	5				3	5

In this case, we start the index of the array from 1 just for simplification against the usual practice in which the index of an array starts from zero in C++. It is not necessary to always start the indexing from zero. Sometimes, it is required to start the indexing from 1. For this, we leave the zeroth position and start using the array from index 1 that is actually the second position. Suppose we have to store the numbers from 1 to 6 in the array. We take an array of 7 elements and put the numbers from the

index 1. Thus there is a correspondence between index and the numbers stored in it. This is not very useful. So, it does not justify the non-use of zeroth position of the array out-rightly. However for simplification purposes, it is good to use the index from 1.

add Method

Now we will talk about adding an element to the list. Suppose there is a call to add an element in the list i.e. *add(9)*. As we said earlier that the current position is 3, so by adding the element 9 to the list, the new list will be (2, 6, 8, 9, 7, 1).

To add the new element (9) to the list at the current position, at first, we have to make space for this element. For this purpose, we shift every element on the right of 8 (the current position) to one place on the right. Thus after creating the space for new element at position 4, the array can be represented as

A	2	6	8		7	1			<u>current</u>	<u>size</u>
	1	2	3	4	5				3	5

Now in the second step, we put the element 9 at the empty space i.e. position 4. Thus the array will attain the following shape. The figure shows the elements in the array in the same order as stored in the list.

A	2	6	8	9	7	1			<u>current</u>	<u>size</u>
	1	2	3	4	5	6			4	6

We have moved the current position to 4 while increasing the size to 6. The size shows that the elements in the list. Whereas the size of the array is different that we have defined already to a fixed length, which may be 100, 200 or even greater.

next Method

Now let's see another method, called '*next*'. We have talked that the next method moves the current position one position forward. In this method, we do not add a new element to the list but simply move the pointer one element ahead. This method is required while employing the list in our program and manipulating it according to the requirement. There is also an array to store the list in it. We also have two variables- *current* and *size* to store the position of current pointer and the number of elements in the list. By looking on the values of these variables, we can find the state of the list i.e. how many elements are in the list and at what position the current pointer is.

The method *next* is used to know about the boundary conditions of the list i.e. the array being used by us to implement the list. To understand the boundary conditions, we can take the example of an array of size 100 to implement the list. Here, 100 elements are added to the array. Let's see what happens when we want to add 101st element to the array? We used to move the current position by *next* method and reached the 100th position. Now, in case of moving the pointer to the next position (i.e. 101st), there will be an error as the size of the array is 100, having no position after this point. Similarly if we move the pointer backward and reach at the first position regardless that the index is 0 or 1. But what will happen if we want to move backward from the first position? These situations are known as boundary conditions and need attention during the process of writing programs when we write the code to use the list. We will take care of these things while implementing the list in C++

programs.

remove Method

We have seen that the *add* method adds an element in the list. Now we are going to discuss the *remove* method. The *remove* method removes the element residing at the current position. The removal of the element will be carried out as follows. Suppose there are 6 elements (2, 6, 8, 9, 7, 1) in the list. The current pointer is pointing to the position 5 that has the value 7. We remove the element, making the current position empty. The size of the list will become 5. This is represented in the following figure.



A	2	6	8	9		1			<u>current</u>	<u>size</u>
	1	2	3	4	5	6			5	~~6~~ 5

We fill in the blank position left by the removal of 7 by shifting the values on the right of position 5 to the left by one space. This means that we shift the remaining elements on the right hand side of the current position one place to the left so that the element next to the removed element (i.e. 1) takes its place (the fifth position) and becomes the current position element. We do not change the current pointer that is still pointing to the position 5. Thus the current pointer remains pointing to the position 5 despite the fact that there is now element 1 at this place instead of 7. Thus in the *remove* method, when we remove an element, the element next to it on the right hand side comes at its place and the remaining are also shifted one place to the right. This step is represented by the following figure.



A	2	6	8	9	1				<u>current</u>	<u>size</u>
	1	2	3	4	5				5	5

find Method

Now let's talk about a function, used to find a specific element in the array. The *find(x)* function is used to find a specific element in the array. We pass the element, which is to be found, as an argument to the *find* function. This function then traverses the array until the specific element is found. If the element is found, this function sets the current position to it and returns 1 i.e. true. On the other hand, if the element is not found, the function returns 0 i.e. false. This indicates that the element was not found. Following is the code of this *find(x)* function in C++.

```
int find (int x)
{
    int j ;
    for (j = 1; j < size + 1; j++ )
        if (A[j] == x )
            break ;
    if (j < size + 1)           // x is found
    {
        current = j ;          //current points to the position where x found
        return 1 ;             // return true
    }
}
```

```

    }
    return 0 ;                //return false, x is not found
}

```

In the above code, we execute a *for* loop to traverse the array. The number of execution of this loop is equal to the size of the list. This *for* loop gets terminated when the value of loop variable (*j*) increases from the size of the list. However we terminate the loop with the *break* statement if the element is found at a position. When the control comes out from the loop, we check the value of *j*. If the value of *j* is less than the size of the array, it means that the loop was terminated by the *break* statement. We use the *break* statement when we find the required element (*x*) in the list. The execution of *break* statement shows that the required element was found at the position equal to the value of *j*. So the program sets the *current* position to *j* and comes out the function by returning 1 (i.e. true). If the value of *j* is greater than the size of the array, it means that the whole array has traversed and the required element is not found. So we simply return 0 (i.e. false) and come out of the function.

Other Methods

There are some other methods to implement the list using an array. These methods are very simple, which perform their task just in one step (i.e. in one statement). There is a *get()* method, used to get the element from the current position in the array. The syntax of this function is of one line and is as under

```
return A[current] ;
```

This statement returns the element to which the *current* is pointing to (i.e. the current position) in the list A.

Another function is *update(x)*. This method is used to change (set) the value at the current position. A value is passed to this method as an argument. It puts that value at the current position. The following statement in this method carries out this process.

```
A [current] = x ;
```

Then there is a method *length()*. This method returns the size of the list. The syntax of this method is

```
return size ;
```

You may notice here that we are returning the size of the list and not the size of the array being used internally to implement the list. This *size* is the number of the elements of the list, stored in the array.

The *back()* method decreases the value of variable *current* by 1. In other words, it moves the current position one element backward. This is done by writing the statement.

```
current -- ;
```

The *--* is a decrement operator in C++ that decreases the value of the operand by one. The above statement can also be written as

```
current = current -1 ;
```

The *start()* method sets the current position to the first element of the list. We know that the index of the array starts from 0 but we use the index 1 for the starting position. We do not use the index zero. So we set the current position to the first element by writing

```
current = 1 ;
```

Similarly, the *end()* method sets the current position to the last element of the list i.e. *size*. So we write

```
current = size ;
```

Analysis of Array List

Now we analyze the implementation of the list while using an array internally. We analyze different methods used for the implementation of the list. We try to see the level upto which these are efficient in terms of CPU's time consumption. Time is the major factor to see the efficiency of a program.

Add

First of all, we have talked about the *add* method. When we add an element to the list, every element is moved to the right of the current position to make space for the new element. So, if the current position is the start of the list and we want to add an element in the beginning, we have to shift all the elements of the list to the right one place. This is the worst case of adding an element to the list. Suppose if the size of the list is 10000 or 20000, we have to do the shift operation for all of these 10000 or 20000 elements. Normally, it is done by shifting of elements with the use of a *for* loop. This operation takes much time of the CPU and thus it is not a good practice to add an element at the beginning of a list. On the other hand, if we add an element at the end of the list, it can be done by carrying out 'no shift operation'. It is the best case of adding an element to the list. However, normally we may have to move half of the elements. The usage of add method is the matter warranting special care at the time of implementation of the list in our program. To provide the interface of the list, we just define these methods.

Remove

When we remove an element at the current position in the list, its space gets empty. The current pointer remains at the same position. To fill this space, we shift the elements on the right of this empty space one place to the left. If we remove an element from the beginning of the list, then we have to shift the entire remaining elements one place to the left. Suppose there is a large number of elements, say 10000 or 20000, in the list. We remove the first element from the list. Now to fill this space, the remaining elements are shifted (that is a large number). Shifting such a large number of elements is time consuming process. The CPU takes time to execute the *for* loop that performs this shift operation. Thus to remove an element at the beginning of the list is the worst case of *remove* method. However it is very easy to remove an

element at the end of the list. In average cases of the *remove* method we expect to shift half of the elements. This average does not mean that in most of the cases, you will have to shift half the elements. It is just the average. We may have to shift all the elements in one operation (if we remove at the beginning) and in the second operation, we have to shift no element (if we remove at the end). Similarly, in certain operations, we have to shift just 10, 15 elements.

Find

We have discussed that the *find* method takes an element and traverses the list to find that element. The worst case of the find method is that it has to search the entire list from beginning to end. So, it finds the element at the end of the array or the element is not found. On average the find method searches at most half the list.

The other methods *get()*, *length()* etc are one-step methods. They carry out their operation in one instruction. There is no need of any loop or other programming structures to perform the task. The *get()* method gets the value from the specified position just in one step. Similarly the *update()* method sets a value at the specific position just in one-step. The *length()* method returns the value of the size of the list. The other methods *back()*, *start()* and *end()* also perform their tasks only in one step.

List using Linked Memory

We have seen the implementation of the list with the use of an array. Now we will discuss the implementation of the list while using linked memory. In an array, the memory cells of the array are linked with each other. It means that the memory of the array is contiguous. In an array, it is impossible that one element of the array is located at a memory location while the other element is located somewhere far from it in the memory. It is located in very next location in the memory. It is a property of the array that its elements are placed together with one another in the memory. Moreover, when we have declared the size of the array, it is not possible to increase or decrease it during the execution of the program. If we need more elements to store in the array, there is need of changing its size in the declaration. We have to compile the program again before executing it. Now array will be of the new size. But what happens if we again need to store more elements? We will change the code of our program to change the declaration of the array while recompiling it.

Suppose we have used the dynamic memory allocation and created an array of 100 elements with the use of *new* operator. In case of need of 200 elements, we will release this array and allocate a new array of 200 elements. Before releasing the previous array, it will wise to copy its elements to the new array so that it does not lose any information. Now this new array is in 'ready for use' position. Thus the procedure of creating a new array is not an easy task.

To avoid such problems, usually faced by the programmers while using an array, there is need of using linked memory in which the various cells of memory, are not located continuously. In this process, each cell of the memory not only contains the value of the element but also the information where the next element of the list is residing in the memory. It is not necessary that the next element is at the next location in the memory. It may be anywhere in the memory. We have to keep a track of it. Thus, in this way, the first element must explicitly have the information about the location of the second element. Similarly, the second element must know where the

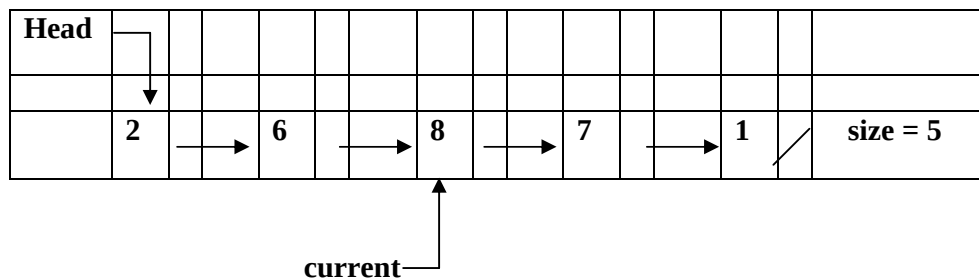
third element is located and the third should know the position of the fourth element and so on. Thus, each cell (space) of the list will provide the value of the element along with the information about where the next element is in the memory. This information of the next element is accomplished by holding the memory address of the next element. The memory address can be understood as the index of the array. As in case of an array, we can access an element in the array by its index. Similarly, we can access a memory location by using its address, normally called memory address.

Linked List

For the utilization of the concept of linked memory, we usually define a structure, called linked list. To form a linked list, at first, we define a node. A node comprises two fields. i.e. the *object* field that holds the actual list element and the *next* that holds the starting location of the next node.



A chain of these nodes forms a linked list. Now let's consider our previous list, used with an array i.e. 2, 6, 8, 7, 1. Following is the figure which represents the list stored as a linked list.



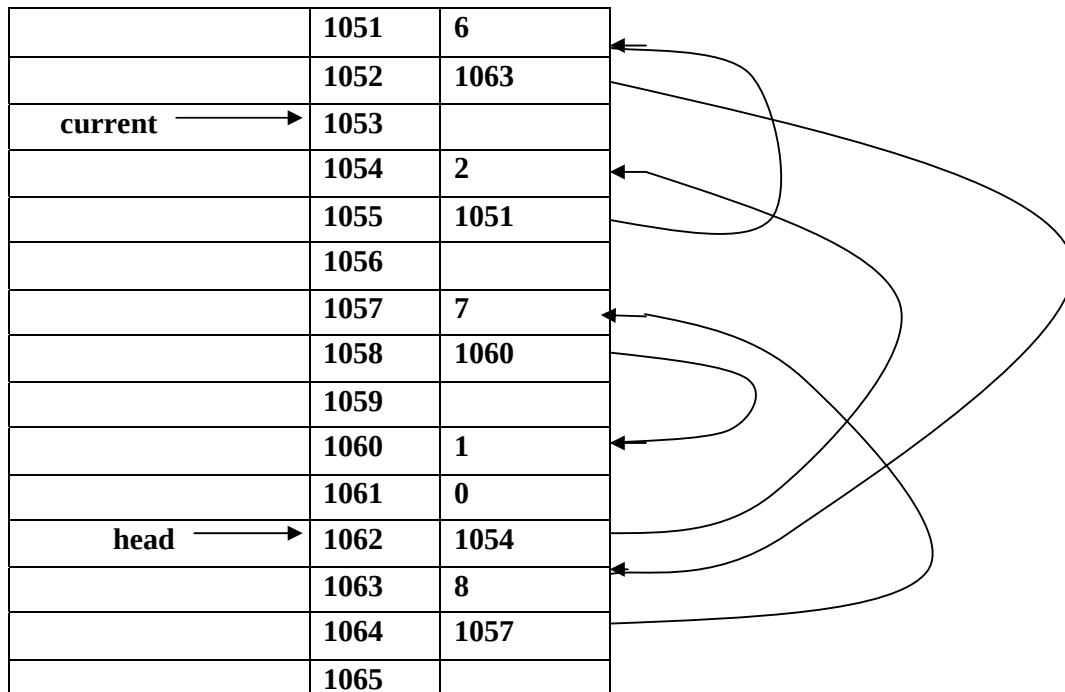
This diagram just represents the linked list. In the memory, different nodes may occur at different locations but the *next* part of each node contains the address of the next node. Thus it forms a chain of nodes which we call a linked list.

While using an array we knew that the array started from index 1 that means the first element of the list is at index 1. Similarly in the linked list we need to know the starting point of the list. For this purpose, we have a pointer *head* that points to the first node of the list. If we don't use *head*, it will not be possible to know the starting position of the list. We also have a pointer *current* to point to the current node of the list. We need this pointer to add or remove current node from the list. Here in the linked list, the *current* is a pointer and not an index as we used while using an array. The *next* field of the last node points to nothing. It is the end of the list. We place the memory address NULL in the last node. NULL is an invalid address and is inaccessible.

Now again consider the list 2, 6, 8, 7, 1. The previous figure represents this list as a linked list. In this linked list, the *head* points to 2, 2 points to 6, 6 points to 8, 8 points to 7 and 7 points to 1. Moreover we have the current position at element 8.

This linked list is stored in the memory. The following diagram depicts the process

through which this linked list is stored in the memory.



We can see in the figure that each memory location has an address. Normally in programming, we access the memory locations by some variable names. These variable names are alias for these locations and are like labels that are put to these memory locations. We use *head* and *current* variable names instead of using the memory address in numbers for starting and the current nodes. In the figure, we see that *head* is the name of the memory location 1062 and the name *current* is used for the memory address 1053. The *head* holds the address 1054 and the element 2, the first one in the list, is stored in the location 1054. Similarly *current* holds the address 1063 where the element 8 is stored which is our current position in the list. In the diagram, two memory locations comprise a node. So we see that the location 1054 holds the element 2 while the next location 1055 holds the address of the memory location (1051) where the next element of the list (i.e. 6) is stored. Similarly the *next* part of the node that has value 6 holds the memory address of the location occupied by the next element (i.e. 8) of the list. The other nodes are structured in a similar fashion. Thus, by knowing the address of the next element we can traverse the whole list.

Data Structures

Lecture No. 03

Reading Material

Data Structures and algorithm analysis in C++

Chapter. 3
3.2.2, 3.2.3, 3.2.5

Summary

- Linked List inside Computer Memory
- Linked List Operations
- Linked List Using C++
- Example Program

In the previous lectures, we used an array to construct a list data structure and observed the limitation that array being of fixed size can only store a fixed number of elements. Therefore, no more elements can be stored after the size of the array is reached.

In order to resolve this, we adopted a new data structure called *linked list*. We started discussing, how linked lists are stored in computer memory and how memory chains are formed.

Linked List inside Computer Memory

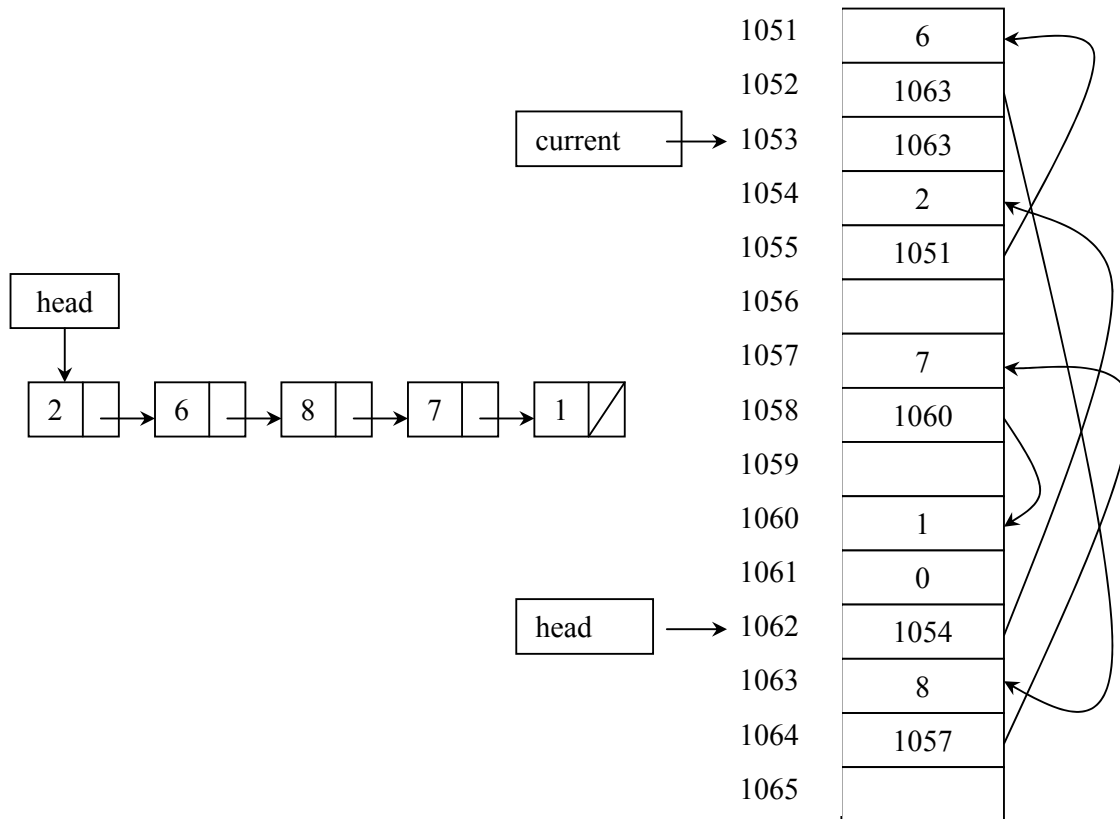


Fig 1. Linked list in memory

There are two parts of this figure. On the left is the linked list chain that is actually the conceptual view of the linked list and on the right is the linked list inside the computer memory. Right part is a snapshot of the computer memory with memory addresses from 1051 to 1065. The *head* pointer is pointing to the first element in the linked list. Note that *head* itself is present in the memory at address 1062. *It is actually a pointer containing the memory address 1054.* Each node in the above linked list has two parts i.e. the data part of the node and the pointer to the next node. The first node of the linked list pointed by the *head* pointer is stored at memory address 1054. *We can see the data element 2 present at that address.* The second part of the first node contains the memory address 1051. So the second linked list's node starts at memory address 1051. We can use its pointer part to reach the third node of the list and in this way, we can traverse the whole list. The last node contains 1 in its data part and 0 in its pointer part. 0 indicates that it is not pointing to any node and it is the last node of the linked list.

Linked List Operations

The linked list data structure provides operations to work on the nodes inside the list. The first operation we are going to discuss here is to create a new node in the memory. The *Add(9)* is used to create a new node in the memory at the current

position to hold '9'. You must remember while working with arrays, to add an element at the current position that all the elements after the current position were shifted to the right and then the element was added to the empty slot.

Here, we are talking about the internal representation of the list using linked list. Its *interface* will remain the same as in case of arrays.

We can create a new node in the following manner in the *add()* operation of the linked list with code in C++:

```
Node * newNode = new Node(9);
```

The first part of the statement that is on the left of the assignment is declaring a variable pointer of type *Node*. It may also be written as *Node * newNode*. On the right of this statement, the *new* operator is used to create a new *Node* object as *new Node(9)*. This is one way in C++ to create objects of classes. The name of the class is provided with the *new* operator that causes the constructor of the class to be called. The constructor of a class has the same name as the class and as this a function, parameters can also be passed to it. In this case, the constructor of the *Node* class is called and '9' is passed to it as an *int* parameter.

Hence, the whole statement means:

"Call the constructor of the *Node* class and pass it '9' as a parameter. After constructing the object in memory, give me the starting memory address of the object. That address will be stored in the pointer variable *newNode*."

To create an object of *int* type in the same manner, we can write as:

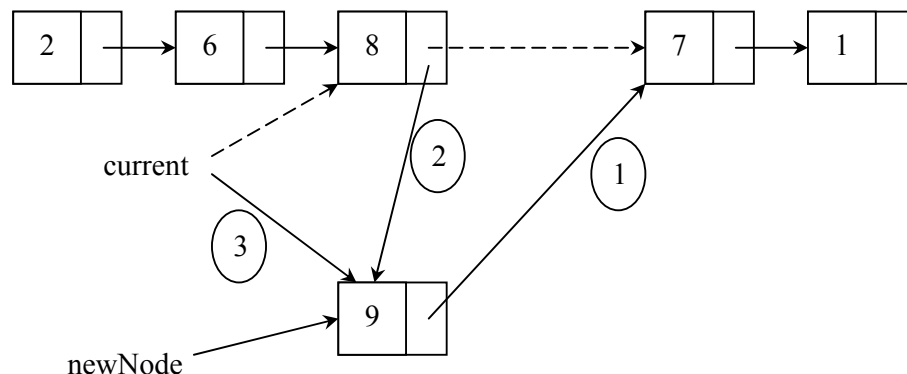
```
int * i = new int ;
```

Previously, we used the same technique to allocate memory for an array of *ints* as:

```
int * i = new int [10] ;
```

Now after the node has been created, how the node is fit into the chain of the linked list.

Fig 2. Insertion of new Node into the linked list



In the above figure, there is a linked list that contains five nodes with data elements as 2, 6, 8, 7 and 1. The *current* pointer is pointing to the node with element as 8. We want to insert a new node with data element 9. This new node will be inserted at the current position (the position where the *current* pointer is pointing to). This insertion

operation is performed in a step by step fashion.

- The first step is to point next pointer of the new node (with data element as 9) to the node with data element as 7.
- The second step is to point the next pointer of the node with data element 8 to the node the new node with data element 9.
- The third step is to change the *current* pointer to point to the new node.

Now, the updated linked list has nodes with data elements as 2, 6, 8, 9, 7 and 1. The list size has become 6.

Linked List Using C++

```
/* The Node class */

class Node
{
public:
    int  get() { return  object; };
    void set(int  object) { this->object =  object; };

    Node * getNext() { return  nextNode; };
    void setNext(Node * nextNode) { this->nextNode =  nextNode; };
private:
    int  object;
    Node * nextNode;
};
```

Whenever, we write a class, it begins with the word *class* followed by the *class-name* and the body of the class enclosed within curly braces. In the body, we write its *public* variables, methods and then *private* variables and methods, this is normally the sequence.

If there is no code to write inside the constructor function of a class, we need not to declare it ourselves as the compiler automatically creates a default constructor for us. Similarly, if there is nothing to be done by the destructor of the class, it will be better not to write it explicitly. Rather, the compiler writes it automatically. Remember, the default constructor and destructor do nothing as these are the function without any code statements inside.

Let's start with the data members first. These are given at the bottom of the class body with the scope mentioned as *private*. These data members are actually two parts of a linked list's node. First variable is *object* of type *int*, present there to store the data part of the node. The second variable is *nextNode*, which is a pointer to an object of type *Node*. It has the address of the next element of the linked list.

The very first *public* function given at the top is *get()*. We have written its code within the class *Node*. It returns back the value of the variable *object* i.e. of the type of *int*.

When we write class in C++, normally, we make two files (*.h* and *.cpp*) for a class. The *.h* file contains the declarations of *public* and *private* members of that class. The *public* methods are essentially the interface of the class to be employed by the users of this class. The *.cpp* file contains the implementation for the class methods that has the

actual code. This is usually the way that we write two files for one class. But this is not mandatory. In the code given above, we have only one file *.cpp*, instead of separating into two files. As the class methods are very small, so their code is written within the body of the class. This facilitates us in carrying on discussion. Thus instead of talking about two files, we will only refer to one file. On the other hand, compiler takes these functions differently that are called *inline* functions. The compiler replaces the code of these *inline* functions wherever the call to them is made.

The second method in the above-mentioned class is *set()* that accepts a parameter of type *int* while returning back nothing. The accepted parameter is assigned to the internal data member *object*. Notice the use of *this* pointer while assigning the value to the internal data member. It is used whenever an object wants to talk to its own members.

The next method is *getNext()* which returns a pointer to an object of type *Node* lying somewhere in the memory. It returns *nextNode* i.e. a pointer to an object of type *Node*. As discussed above, *nextNode* contains the address of next node in the linked list.

The last method of the class is *setNext()* that accepts a pointer of type *Node*, further assigned to *nextNode* data member of the object. This method is used to connect the next node of the linked list with the current object. It is passed an address of the next node in the linked list.

Let's discuss a little bit about classes. A very good analogy of a class is a factory. Think about a car factory. On the placement of order, it provides us with the number of vehicles we ordered for. Similarly, you can see number of other factories in your daily-life that manufacture the specific products.

Let's take this analogy in C++ language. Suppose, we want to make a factory in C++. By the way, what is our *Node* class? It is actually a factory that creates nodes. When we want to make a new node, a new operator is used. By using *new* operator with the *Node* class, actually, we send an order to *Node* factory, to make as many as nodes for us.

So we have a good analogy, to think about a class as a factory. The products that are made by the factory have their own characteristics. For example, a car made by an automobile factory has an engine, wheels, steering and seats etc. These variables inside a class are called *state variables*. Now the kinds of operations this car can do are the *methods* of its class. A car can be driven, engine can be started, gears can be shifted and an accelerator can be pressed to run it faster.

Similarly, the *Node* class creates nodes, where every node has two-state variables i.e. *object* and *nextNode*. We have already seen its operations in the above code. We use *new* to create new object or an array of new objects, stored in memory.

Let's see the code below.

```
/* List class */
#include <stdlib.h>
#include "Node.cpp"

class List
{
public:
```

```
// Constructor
List() {
    headNode = new Node();
    headNode->setNext(NULL);
    currentNode = NULL;
    size = 0;
}
```

We are creating a list factory here employed to create list objects. Remember the list operations; *add*, *remove*, *next*, *back* and *start* etc. Let's see the above class declaration code in detail.

There are two *include statements* at the start. The first line is to include a standard library *stdlib.h* while the second line includes the *Node* class file *Node.cpp*. This *Node* class is used to create nodes that form a *List* object. So this *List* factory will order *Node* class to create new nodes. The *List* class itself carries out the chain management of these *Node* objects.

We have written our own constructor of *List* class as the default constructor is not sufficient enough to serve the purpose. The *List* constructor is parameterless. The very first step it is doing internally is that it is asking *Node* class to create a new node and assigning the starting address of the new *Node*'s object to the *headNode* data member. In the second statement, we are calling *setNext()* method of the *Node* class for the object pointed to by the *headNode* pointer. This call is to set the *nextNode* data member to *NULL*, i.e. *Node*'s object pointed to by the *headNode* pointer is not pointing to any further *Node*. The next statement is to set the *currentNode* pointer to *NULL*. So at the moment, we have initialized the *currentNode* pointer to *NULL* that is not pointing to any *Node* object. The next statement is to initialize the *size* data member to 0 indicating that there is no node present in the list. All this processing is done inside the constructor of *List* class, as we want all this done when a list object is created. Considering the analogy of car factory, the constructor function can perform certain tasks: The oil is poured into the engine, the tyres are filled-in with air etc.

Let's see the *add* method of the *List* class:

```
/* add() class method */
void add (int addObject)
{
1.   Node * newNode = new Node();
2.   newNode->set(addObject);
3.   if( currentNode != NULL )
4.   {
5.       newNode->setNext(currentNode->getNext());
6.       currentNode->setNext( newNode );
7.       lastCurrentNode = currentNode;
8.       currentNode = newNode;
9.   }
10.  else
11.  {
12.      newNode->setNext(NULL);
13.      headNode->setNext(newNode);
}
```

```
14.         lastCurrentNode = headNode;
15.         currentNode = newNode;
16.     }
17.     size ++;
}
```

The interface or signatures of *add()* method is similar to the one discussed in case of an array. This method takes the object to be added as a parameter. The implementation of this *add()* method is a bit longer as the method is being implemented for linked list. In the first statement, a new *Node* object is created with its address stored in the *newNode* pointer variable. The second statement is to call *set()* method of the *Node* object pointed to by the *newNode* pointer. You can note the way the method is called. A pointer variable is at the left most side then an arrow sign (->), then the name of the method with appropriate arguments within parenthesis. It is followed by the if-statement that checks the *currentNode* is not *NULL* to perform certain operations inside the if-code block. Inside the if-statement, at line 5, the *nextNode* pointer of the new node is being set to the *nextNode* of the object pointed to by the *currentNode* pointer. In order to understand the statements given in this code properly, consider the fig 2 above, where we added a node in the linked list. We have done step 1 at line5. At line 6, we are performing the second step by setting the *newNode* in the *nextNode* pointer of the object pointed to by the *currentNode*. At line 7, we are saving the current position (address) of the *currentNode* pointer in the pointer variable *lastCurrentNode*, which might be useful for backward traversing. Although, the fig 1 (left part) indicates movement in one direction from left to right but the *lastCurrentNode* pointer node can be used by the *back()* member function to traverse one position back from right to left. At line 8, the *currentNode* pointer is assigned the address of the object pointed to by *newNode*. This way, a new node is added in already existent linked list.

Line 10 is start of the *else* part of if-statement. This is executed if the *currentNode* is *NULL*. It means that there is no node present in the list previously and first node is going to be added. At line 12, we are setting the *nextNode* pointer of the object pointed to by *newNode* pointer. The *nextNode* is being set to *NULL* by calling the *setNext()* method. Then at line 13, we point the *head* pointer (*headNode*) to this new node pointed to by *newNode* pointer. Note that *headNode* is pointing to a node that is there despite the fact that the *size* of the linked list is 0. Actually, we have allocated a *Node* object for *headNode* pointer. Although, we don't need a *Node* object here, yet it will be helpful when we perform other operations like *remove()* and *find()*.

At line 14, the *headNode* address is being assigned to *lastCurrentNode*. At line 15, *currentNode* pointer is assigned the address of *newNode*. At the end i.e. at line 17, the *size* of the list is incremented by 1.

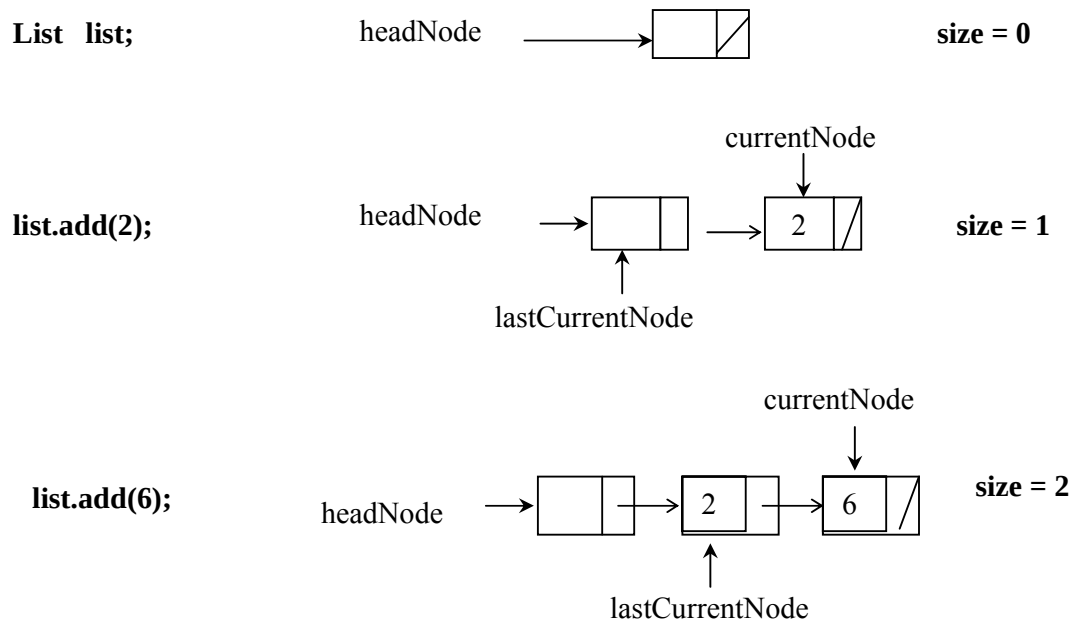


Fig 3. Add operation of linked list

Following is the crux of this `add()` operation :

Firstly, it will make a new node by calling Node class constructor. Insert the value e.g. 2. of the node into the node by calling the set method. Now if the list already exists (has some elements inside or its size is non-zero), it will insert the node after the current position. If the list does not already exist, this node is added as the first element inside the list.

Let's try to add few more elements into the above linked list in the figure. The following are the lines of code to be executed to add nodes with values 8, 7 and 1 into the linked list.

`list.add(8); list.add(7); list.add(1);`

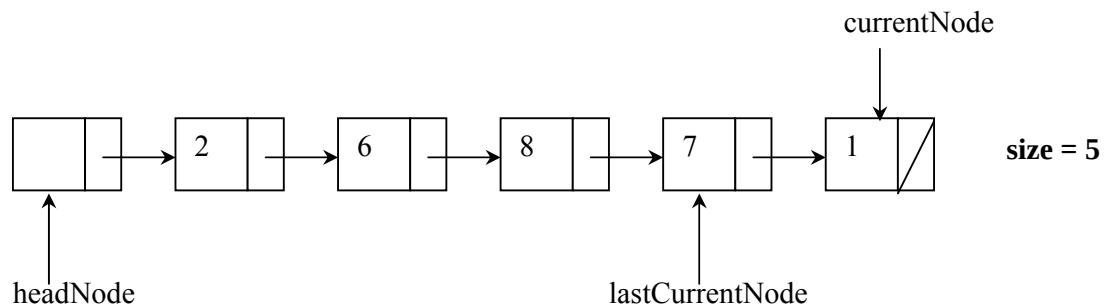


Fig 4. More Nodes added into linked list

Now we will see the remaining methods of the linked list. The `get()` method of the `List` class is given below


```
/* get() class method */
int get()
{
    if (currentNode != NULL)
        return currentNode->get();
}
```

This method firstly confirms that the *currentNode* pointer is not *NULL*. If it is not *NULL*, then it must be pointing to some *Node* object as inside the constructor of the *List* class, we have initialized this pointer variable to *NULL*. That indicates that the *currentNode* is *NULL* when there is no element inside the list. However, when a *Node* object is added into it, it starts pointing to it. So, this *get()* returns the address of the node pointed to by the *currentNode* pointer.

Further, we have another method given below:

```
/* next() class method */
bool next()
{
    1.    if (currentNode == NULL) return false;
    2.
    3.    lastCurrentNode = currentNode;
    4.    currentNode = currentNode->getNext();
    5.    return true;
};
```

This is *next()* method, used to advance the *currentNode* pointer to the next node inside the linked list. At line 1, the *currentNode* is being checked to confirm that there are some elements present in the list to advance further. At line 1, the method is returning *false* if there is no element present in the list. At line 3, it is storing the value of the *currentNode* pointer into the *lastCurrentNode*. At line 4, *currentNode* is calling the *getNext()* method to get the address of next node to be stored in the *currentNode* pointer to advance the *currentNode* pointer to the next element. At line 5, it returns *true* indicating the method is successful in moving to the next node.

Example Program

Given below is the full source code of the example program. You can copy, paste and compile it right away. In order to understand the linked list concept fully, it is highly desirable that you understand and practice with the below code.

```
#include <iostream.h>
#include <stdlib.h>

/* The Node class */
class Node
{
    public:
        int get() { return object; };
        void set(int object) { this->object = object; };
};
```

```
        Node * getNext() { return  nextNode; };
        void  setNext(Node * nextNode) { this->nextNode =  nextNode; };

    private:
        int  object;
        Node * nextNode;
};

/* The List class */
class List
{
public:
    List();
    void add (int  addObject);
    int  get();
    bool next();
    friend void  traverse(List list);
    friend List  addNodes();

private:
    int  size;
    Node *  headNode;
    Node *  currentNode;
    Node *  lastCurrentNode;

};

/* Constructor */
List::List()
{
    headNode = new Node();
    headNode->setNext(NULL);
    currentNode = NULL;
    lastCurrentNode = NULL;
    size = 0;
}

/* add() class method */
void List::add (int  addObject)
{
    Node * newNode = new Node();
    newNode->set(addObject);
    if( currentNode != NULL )
    {
        newNode->setNext(currentNode->getNext());
        currentNode->setNext( newNode );
        lastCurrentNode = currentNode;
        currentNode = newNode;
    }
    else
    {

```

```
        newNode->setNext(NULL);
        headNode->setNext(newNode);
        lastCurrentNode = headNode;
        currentNode = newNode;
    }
    size++;
}

/* get() class method */
int List::get()
{
    if (currentNode != NULL)
        return currentNode->get();
}

/* next() class method */
bool List::next()
{
    if (currentNode == NULL) return false;

    lastCurrentNode = currentNode;
    currentNode = currentNode->getNext();
    if (currentNode == NULL || size == 0)
        return false;
    else
        return true;
}

/* Friend function to traverse linked list */
void traverse(List list)
{
    Node* savedCurrentNode = list.currentNode;
    list.currentNode = list.headNode;

    for(int i = 1; list.next(); i++)
    {
        cout << "\n Element " << i << " " << list.get();
    }
    list.currentNode = savedCurrentNode;
}

/* Friend function to add Nodes into the list */
List addNodes()
{
    List list;
    list.add(2);
    list.add(6);
    list.add(8);
    list.add(7);
    list.add(1);
}
```

```
    cout << "\n List size = " << list.size << "\n";  
    return list;  
}  
  
main()  
{  
    List list = addNodes();  
    traverse(list);  
}
```

The output of the example program is as follows:

List size = 5

Element 1 2

Element 2 6

Element 3 8

Element 4 7

Element 5 1

Data Structures

Lecture No. 04

Reading Material

Data Structures and algorithm analysis in C++

Chapter. 3
3.2.3, 3.2.4, 3.2.5

Summary

- Methods of Linked List
- Example of list usage
- Analysis of Link List
- Doubly-linked List
- Circularly-linked lists
- Josephus Problem

Methods of Linked List

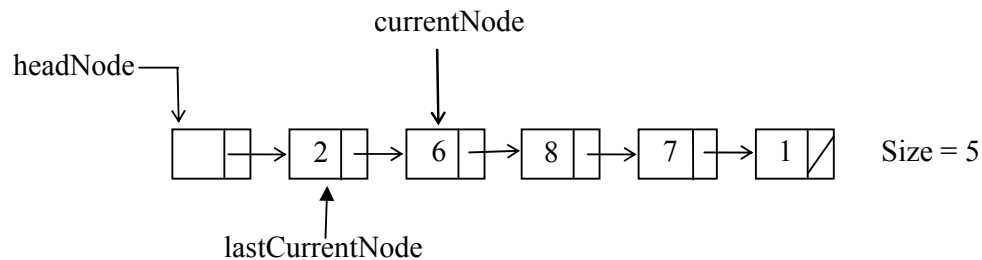
In the previous lecture, we discussed the methods of linked list. These methods form the interface of the link list. For further elucidation of these techniques, we will talk about the *start* method that has the following code.

```
// position currentNode and lastCurrentNode at first element
void start() {
    lastCurrentNode = headNode;
    currentNode = headNode;
};
```

There are two statements in this method. We assign the value of *headNode* to both *lastCurrentNode* and *currentNode*. These two pointers point at different nodes of the list. Here we have pointed both of these pointers at the start of the list. On calling some other method like *next*, these pointers will move forward. As we can move in the singly-linked list in one direction, these pointers cannot go behind *headNode*.

We will now see how a node can be removed from the link list. We use the method *remove* for this purpose.

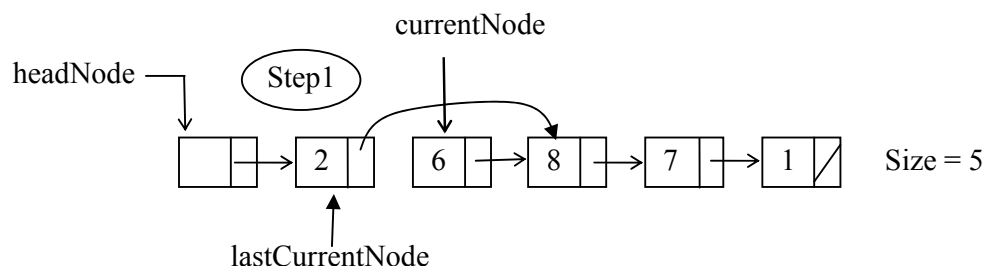
```
void remove() {
    if( currentNode != NULL && currentNode != headNode) {
(step 1)    lastCurrentNode->setNext(currentNode->getNext());
(step 2)    delete currentNode;
(step 3)    currentNode = lastCurrentNode;
(step 4)    size--;
    }
};
```



Suppose that the *currentNode* is pointing at the location that contains the value 6. A request for the removal of the node is made. Resultantly, the node pointed by *currentNode* should be removed. For this purpose, at first, the *next* pointer of the node with value 2 (the node pointed by the *lastCurrentNode* pointer), that is before the node with value 6, bypasses the node with value 6. It is, now pointing to the node with value 8. The code of the first step is as:

```
lastCurrentNode->setNext(currentNode->getNext());
```

What does the statement *currentNode->getNext()* do? The *currentNode* is pointing to the node with value 6 while the *next* of this node is pointing to the node with value 8. That is the *next* pointer of node with value 6 contains the address of the node with value 8. The statement *lastCurrentNode->setNext(currentNode->getNext())* will set the *next* pointer of the node pointed by the *lastCurrentNode* to the node with value 8. So the *next* pointer of the node with value 2 is pointing to the node with value 8.

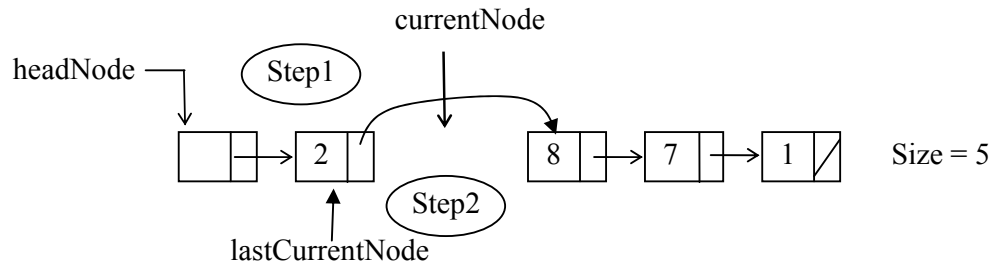


You see that the next pointer of the node having data element 2 contains the address of the node having data element 8. The node with value 6 has been disconnected from the chain while the node with value 2 is connected to the node with the value 8.

The code of the next step is:

```
delete currentNode;
```

You already know, in case of allocation of the memory with the help of the *new* keyword, the *delete* statement releases this memory which returns the memory to the system. Pictorially it can be represented as:

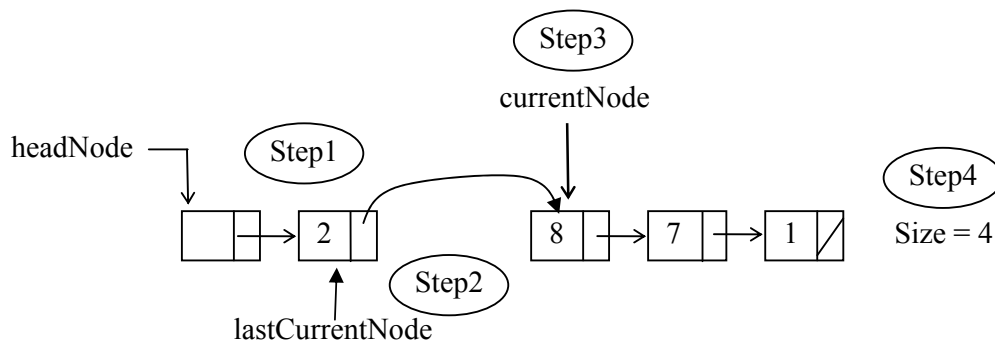


In the next step, we have moved the *currentNode* to point the next node. The code is:

```
currentNode = lastCurrentNode;
```

In the fourth step, the size of the list has been reduced by 1 after the deletion of one node i.e.

```
size--;
```



The next method is *length()* that simply returns the size of the list. The code is as follows:

```
// returns the size of the list
int length()
{
    return size;
};
```

The private data members of the list are:

```
private:
    int size;           // contains the size of the list
    Node *headNode;     // points to the first node of the list
    Node *currentNode, // current node
    Node *lastCurrentNode; // last current node
```

The list class completed just now, can be termed as list factory. We have included all

the required methods in it. We may employ more methods if required. A programmer can get the size of the list, add or remove nodes in it besides moving the pointers.

Example of list usage

Now let's see how we use the link list. Here is an example showing the use of list:

```
/* A simple example showing the use of link list */

#include <iostream>
#include <stdlib.h>
#include "List.cpp" // This contains the definition of List class

// main method
int main(int argc, char *argv[])
{
    List list; // creating a list object

    // adding values to the list
    list.add(5);
    list.add(13);
    list.add(4);
    list.add(8);
    list.add(24);
    list.add(48);
    list.add(12);

    // calling the start method of the list
    list.start();

    // printing all the elements of the list
    while (list.next())
        cout << "List Element: " << list.get() << endl;
}
```

The output of the program is:

```
List Element: 5
List Element: 13
List Element: 4
List Element: 8
List Element: 24
List Element: 48
List Element: 12
```

Let's discuss the code of the above program. We have included the standard libraries besides having the "List.cpp" file. Usually we do not include .cpp files. Rather, the .h files are included. Whenever you write a class, two files will be created i.e. .h (header file containing the interface of the class) and .cpp (implementation file). Here for the sake of explanation, we have combined the two files into "List.cpp" file. At the start

of the main method, we have created a list object as:

```
List list;
```

Here the default constructor will be called. If you understand the concept of factory, then it is not difficult to know that we have asked the *List* factory to create a *List* object and named it as *list*. After creating the object, nodes have been added to it. We have added the elements with data values 5, 13, 4, 8, 24, 48 and 12. Later, the *start()* method of list is called that will position the *currentNode* and *lastCurrentNode* at the start of the list. Now there is no need to worry about the implementation of the List. Rather, we will use the interface of the List. So the *start* method will take us to the start of the *list* and internally, it may be array or link list or some other implementation. Then there is a while loop that calls the *next()* method of the *List*. It moves the pointer ahead and returns a boolean value i.e. true or false. When we reach at the end of the list, the *next()* method will return false. In the while loop we have a *cout* statement that prints the value of the list elements, employing the *get()* method. The loop will continue till the *next()* method returns true. When the pointers reach at the end of the list the *next()* will return false. Here the loop will come to an end.

Please keep in mind that list is a very important data structure that will be used in the entire programming courses.

Analysis of Link List

As stated earlier, we will be going to analyze each data structure. We will see whether it is useful or not. We will see its cost and benefit with respect to time and memory. Let us analyze the link list which we have created with the dynamic memory allocation in a chain form.

- **add**

For the addition purposes, we simply insert the new node after the current node. So 'add' is a one-step operation. We insert a new node after the current node in the chain. For this, we have to change two or three pointers while changing the values of some pointer variables. However, there is no need of traversing too much in the list. In case of an array, if we have to add an element in the centre of the array, the space for it is created at first. For this, all the elements that are after the current pointer in the array, should be shifted one place to the right. Suppose if we have to insert the element in the start of the array, all the elements to the right one spot are shifted. However, for the link list, it is not something relevant. In link lists, we can create a new node very easily where the current pointer is pointing. We have to adjust two or three pointers. Its cost, in terms of CPU time or computing time, is not much as compared to the one with the use of arrays.

- **remove**

Remove is also a one-step operation. The node before and after the node to be removed is connected to each other. Update the current pointer. Then the node to be removed is deleted. As a result, the node to be removed is deleted. Very little work is needed in this case. If you compare it with arrays, for the deletion of an element from the array, space is created. To fill this space, all the right elements are shifted one spot left. If the array size is two thousand or three thousand, we need to run a loop for all these elements to shift them to left.

- **find**

The worst-case in find is that we may have to search the entire list. In find, we have to search some particular element say x . If found, the *currentNode* pointer is moved at that node. As there is no order in the list, we have to start search from the beginning of the list. We have to check the value of each node and compare it with x (value to be searched). If found, it returns true and points the *currentNode* pointer at that node otherwise returns false. Suppose that x is not in the list, in this case, we have to search the list from start to end and return false. This is the worst case scenario. Though time gets wasted, yet we find the answer that x is not in the list. If we compare this with array, it will be the same. We don't know whether x is in the array or not. So we have to search the complete array. In case of finding it, we will remember that position and will return true. What is the average case? x can be found at the first position, in the middle or at the end of the list. So on average, we have to search half of the list.

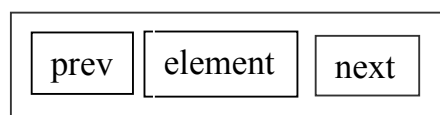
- **back**

In the back method, we move the *current* pointer one position back. Moving the *current* pointer back, one requires traversing the list from the start until the node whose *next* pointer points to current node. Our link list is singly linked list i.e. we can move in one direction from start towards end. Suppose our *currentNode* pointer and *lastCurrentNode* are somewhere in the middle of the list. Now we want to move one node back. If we have the pointer of *lastCurrentNode*, it will be easy. We will assign the value of *lastCurrentNode* to *currentNode*. But how can we move the *lastCurrentNode* one step back. We don't have the pointer of previous node. So the solution for this is to go at the start of the list and traverse the list till the time you reach the node before the *lastCurrentNode* is pointing. That will be the node whose next pointer contains the value *lastCurrentNode*. If the *currentNode* and the *lastCurrentNode* are at the end of the list, we have to traverse the whole list. Therefore back operation is not a one step operation. We not only need a loop here but also require time.

Doubly-linked List

If you look at single link list, the chain is seen formed in a way that every node has a field *next* that point to the next node. This continues till the last node where we set the *next* to NULL i.e. the end of the list. There is a *headNode* pointer that points to the start of the list. We have seen that moving forward is easy in single link list but going back is difficult. For moving backward, we have to go at the start of the list and begin from there. Do you need a list in which one has to move back or forward or at the start or in the end very often? If so, we have to use double link list.

In doubly-link list, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node. Now our node factory will create a node with three parts.



First part is *prev* i.e. the pointer pointing to the previous node, second part is *element*,

containing the data to be inserted in the list. The third part is *next* pointer that points to the next node of the list. The objective of *prev* is to store the address of the previous node.

Let's discuss the code of the node of the doubly-link list. This node factory will create nodes, each having two pointers. The interface methods are same as used in singly link list. The additional methods are *getPrev* and *setPrev*. The method *getPrev* returns the address of the previous node. Thus its return type is *Node**. The *setPrev* method sets the *prev* pointer. If we have to assign some address to *prev* pointer, we will call this method. Following is the code of the doubly-linked list node.

```
/* this is the doubly-linked list class, it uses the next and prev pointers */

class Node {
public:
    int get() { return object; }; // returns the value of the element
    void set(int object) { this->object = object; }; // set the value of the element

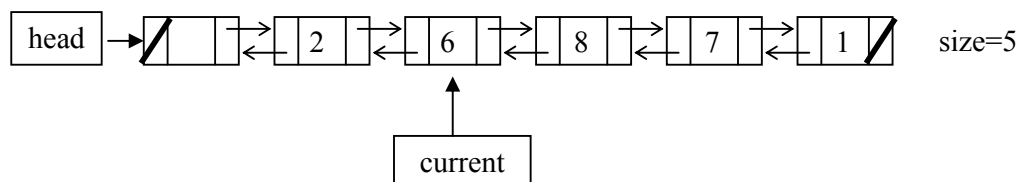
    Node* getNext() { return nextNode; }; // get the address of the next node
    void setNext(Node* nextNode) // set the address of the next node
        { this->nextNode = nextNode; };

    Node* getPrev() { return prevNode; }; // get the address of the prev node
    void setPrev(Node* prevNode) // set the address of the prev node
        { this->prevNode = prevNode; };

private:
    int object; // it stores the actual value of the element
    Node* nextNode; // this points to the next node
    Node* prevNode; // this points to the previous node
};
```

Most of the methods are same as those in singly linked list. A new pointer *prevNode* is added and the methods to get and set its value i.e. *getPrev* and *setPrev*. Now we will use this node factory to create nodes.

You have to be very cautious while adding or removing a node in a doubly linked list. The order in which pointers are reorganized is important. Let's have a pictorial view of doubly-link list. The diagram can help us understand where the *prevNode* and *nextNode* are pointing.



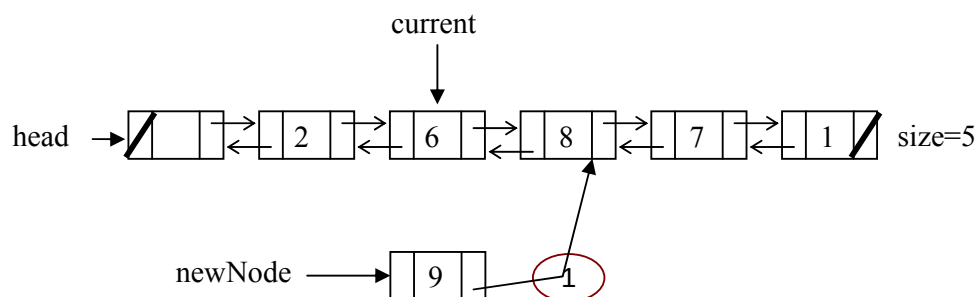
This is a doubly link list. The arrows pointing towards right side are representing *nextNode* while those pointing towards left side are representing *prevNode*. Suppose

we are at the last node i.e. the node with value 1. In case of going back, we usually take the help of *prevNode* pointer. So we can go to the previous node i.e. the node with value 7 and then to the node with value 8 and so on. In this way, we can traverse the list from the end to start. We can move forward or backward in doubly-link list very easily. We have developed this facility for the users to move in the list easily.

Let's discuss other methods of the doubly-linked list. Suppose we have created a new node from the factory with value 9. We will request the node factory to create a new object using new keyword. The newly created node contains three fields i.e. *object*, *prevNode* and *nextNode*. We will store 9 into object and connect this new node in the chain. Let's see how the pointers are manipulated to do that. Consider the above diagram, the current is pointing at the node with value 6. The new node will be inserted between the node with value 6 and the one with value 8.

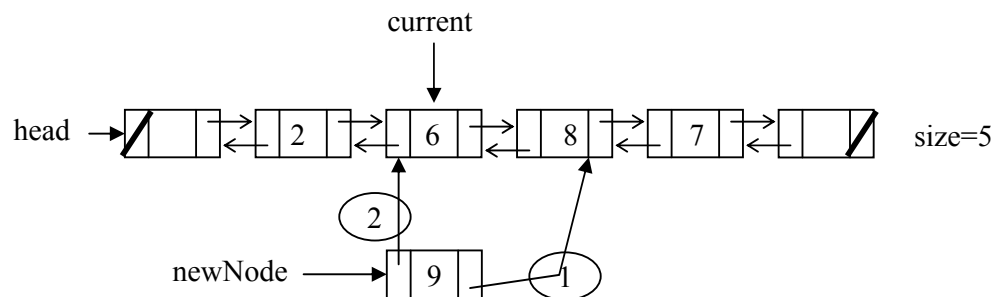
In the first step, we assign the address of the node with value 8 to the *nextNode* of the new node.

```
newNode->setNext( current->getNext() );
```



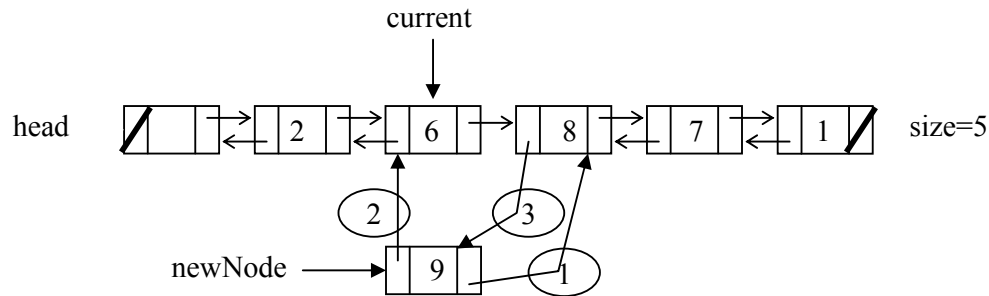
In the next step, a programmer points the *prevNode* of the *newNode* to the node with value 6.

```
newNode->setprev( current );
```



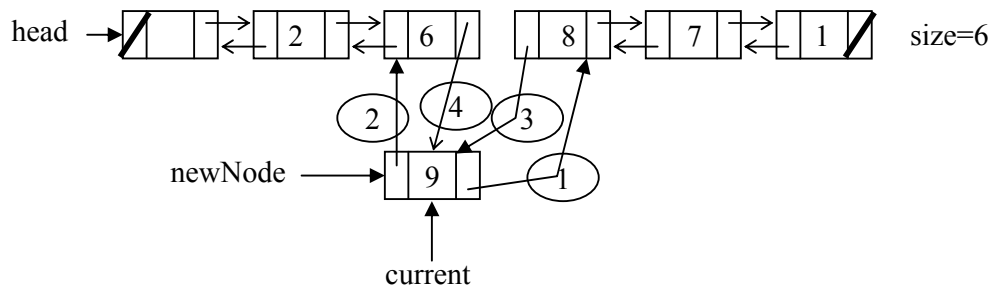
In the third step, we will set the previous node with value 8 to point to the *newNode*.

```
(current->getNext())->setPrev(newNode);
```



Now the *prevNode* of the node with value 8 is pointing to the node with value 9. In the fourth step, the *nextNode* of the node with value 6 is pointing to the *newNode* i.e. the node with value 9. Point the *current* to the *newNode* and add one to the *size* of the list.

```
current->setNext( newNode );
current = newNode;
size++;
```



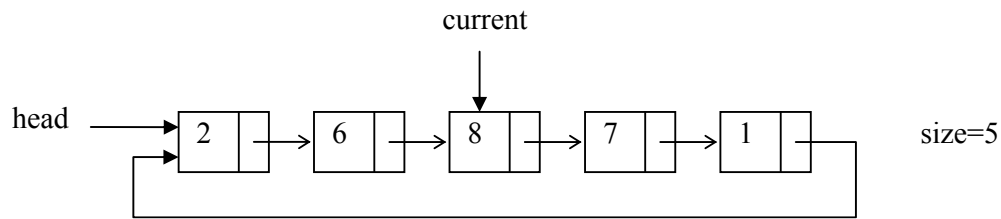
Now the *newNode* has been inserted between node with value 6 and node with value 8.

Circularly-linked lists

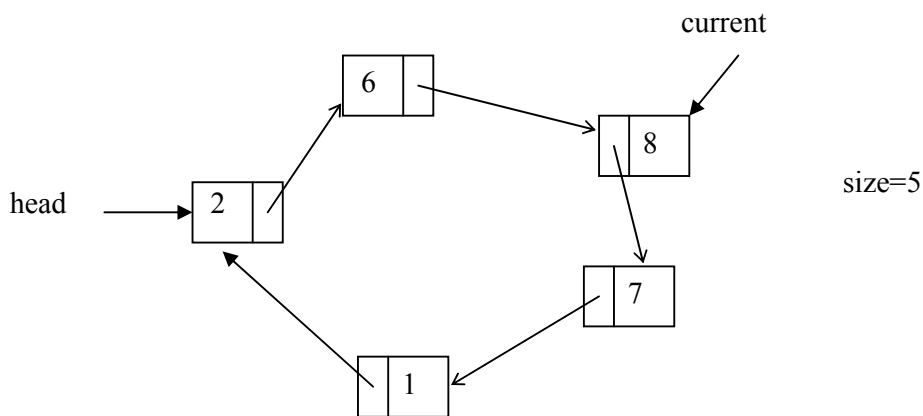
Let's talk about circularly linked list. The next field in the last node in a singly-linked list is set to NULL. The same is the case in the doubly-linked list. Moving along a singly-linked list has to be done in a watchful manner. Doubly-linked lists have two NULL pointers i.e. *prev* in the first node and *next* in the last node. A way around this potential hazard is to link the last node with the first node in the list to create a *circularly-linked list*.

The *next* method in the singly-linked list or doubly-linked list moves the *current* pointer to the next node and every time it checks whether the *next* pointer is NULL or not. Similarly the *back* method in the double-linked list has to be employed carefully if the current is pointing the first node. In this case, the *prev* pointer is pointing to NULL. If we do not take care of this, the current will be pointing to NULL. So if we try to access the NULL pointer, it will result in an error. To avoid this, we can make a circularly linked list.

We have a list with five elements. We have connected the last node with the first node. It means that the *next* of the last node is pointing towards the first node.



The same list has been shown in a circular shape.



You have noticed that there is no such node whose *next* field is NULL. What is the benefit of this? If you use the *next* or *back* methods that move the *current* pointer, it will never point to NULL. It may be the case that you keep on circulating in the list. To avoid this, we get help from the *head* node. If we move the *head* node in the circularly linked list, it will not be certain to say where it was pointing in the start. Its advantages depend on its use. If we do not have to move too much in the list and have no problem checking the NULL, there is little need a circularly-linked list. But this facility is available to us.

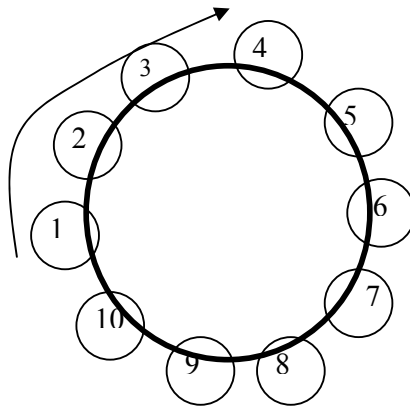
In this example, we made a circular linked list from a singly link list. In a singly link list we move in one direction. We point the *next* pointer of the last node to the first node. We can do the same with the doubly-linked list. The *prev* pointer of the first node will point to the last node and the *next* pointer of the last node will point to the first node. If you arrange all the nodes in a circle, one of the pointers (i.e. *next* pointer) will move in clockwise direction while the *prev* pointers in anti-clockwise direction. With the help of these pointers, you can move in the clockwise direction or anti-clockwise direction. Head node pointer will remain at its position. You don't need to change it. If there is a need to remove the node pointed by head node than you have to move the head pointer to other node. Now we don't have any NULL pointer in the doubly-linked list. We will not get any exception due to NULL pointers.

Josephus Problem

Now we will see an example where circular link list is very useful. This is Josephus Problem. Consider there are 10 persons. They would like to choose a leader. The way they decide is that all 10 sit in a circle. They start a count with person 1 and go in clockwise direction and skip 3. Person 4 reached is eliminated. The count starts with the fifth and the next person to go is the fourth in count. Eventually, a single person remains.

You might ask why someone has to choose a leader in this way. There are some historical stories attached to it. This problem is also studied in mathematics. Let's see its pictorial view.

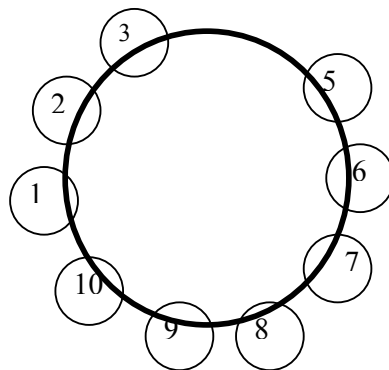
$N=10, M=3$



We have ten numbers representing the ten persons who are in a circle. The value of M shows the count. As the value of M is three, the count will be three. N represents the number of persons. Now we start counting clockwise. After counting up to three, we have the number four. The number four is eliminated and put in the eliminated column.

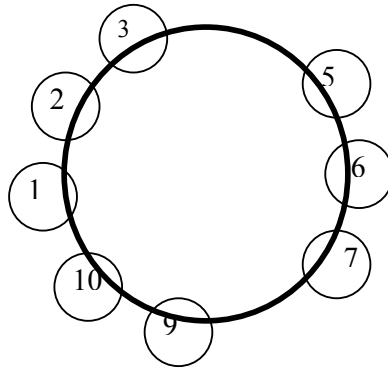
$N=10, M=3$

Eliminated



After eliminating the number four, we will start our counting from number five. Counting up to three, we have number eight which is eliminated and so on.

$N=10, M=3$

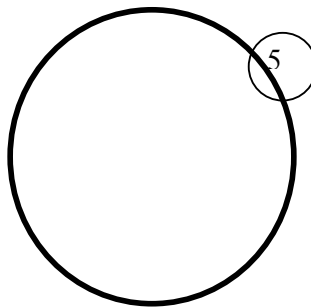


Eliminated

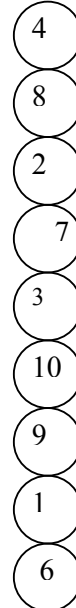


In the end, only number five will remain intact.

$N=10, M=3$



Eliminated



If we have ten persons ($N = 10$) in a circle and eliminate after counting up to three ($M = 3$). If we start our count from one, who will be the leader? We have studied this earlier and know that the person who is sitting at the fifth position will become the leader.

Suppose if the value of N is 300 or 400 and the value of M is 5 or 10. Now who will be the leader? This is a mathematical problem where we can change the values of N and M . There is a formula where the values of N, M are allotted. You can calculate who should become the leader. Here we will not solve it mathematically. Rather, it will be tackled as a computer problem. If you analyze the pictures shown above, it gets clear that this can be solved with the circular link list. We arrange these numbers in a circularly-linked list, point the *head* pointer at the starting number and after calling the *next* method for three times, we will reach the node which is to be removed. We will use the *remove* method to remove the node. Then the *next* method is called thrice from there and the node is removed. We will continue this till we have

only one node.

We are not concerned with the NULL pointers, internal to link list. However, if you want to solve this problem and choose the best data structure, then circular link list is the best option. We can also use the list to solve this.

Let's see the code of the program by which we can solve this problem. The code is as under:

```
/*This program solves the Josephus Problem */

#include <iostream.h>
#include "CList.cpp" //contains the circularly-linked list definition

// The main method
void main(int argc, char *argv[])
{
    CList list;          // creating an object of list
    int i, N=10, M=3;
    for(i=1; i <= N; i++ ) list.add(i); // initializing the list with values

    list.start();        // pointing the pointers at the start of the list

    // counting upto M times and removing the element
    while( list.length() > 1 ) {
        for(i=1; i <= M; i++ ) list.next();
        cout << "remove: " << list.get() << endl;
        list.remove();
    }

    cout << "leader is: " << list.get() << endl;    //displaying the remaining node
}
```

We have included the “CList.cpp”. It means that we are using the circularly-linked list. In the main method, *CList* factory is called to create a circular link list as *CList list*; After this, we assign the values to *N* and *M*. We have used for loop to add the nodes in the list. When this loop finishes, we have ten nodes in the list having values from 1 to 10. But here a programmer may not pay attention to the internal details of the list. We have created a list and stored ten numbers in it. Then we moved the pointers of the list at the start of the list using the *start* method. It means that the pointers are pointing at the position from where we want to start the counting of the list.

There is a while loop that will continue executing until only one node is left in the list. Inside this loop, we have a for loop. It will execute from 1 to *M*. It has only one statement i.e. *list.next()*. This will move the pointer forward three times (as the value of *M* is 3). Now the current pointer is at the 4th node. We called the *remove* method. Before removing the node, we display its value on the screen using *cout*. Again we come into the *while* loop, now the length of the list is 9. The ‘for loop’ will be executed. Now the *list.next()* is not starting from the start. It will start from the

position where the *current* pointer is pointing. The *current* pointer is pointing at the next node to the node deleted. The count will start again. The *list.next()* will be called for three times. The current pointer will point at the 8th node. Again the remove method will be called and the *current* pointer moved to the next node and so on. The nodes will be deleted one by one until the length of the list is greater than one. When the length of the list is one, the while loop will be terminated. Now only one node is left in the list i.e. the leader. We will display its value using the *get* method.

We can change the values of *M* and *N*. Similarly, these values can be read from the file or can use the command line arguments to get values. There are many variations of this problem. One variation is that the value of *M* keeps on changing. Sometimes, it is 3, sometimes 4 or 5 and so on. Due to this, it will become difficult to think that who will become leader. Make a picture in your mind that ten persons are sitting in a circle. Every time the value of *M* is incremented by one. Now try to ascertain which position you should sit to get chosen as a leader. You may like to write a program to solve this or use the mathematical formula.

Data Structures

Lecture No. 05

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 3

3.1, 3.2.5, 3.3.1, 3.3.2
(array implementation)

Summary

- 5) Benefits of using circular list
- 6) Abstract Data Type
- 7) Stacks
- 8) Stack Implementation using arrays

In the previous lecture, we demonstrated the use of the circular list for the resolution of the Josephus problem. After writing a program with the help of this data structure, a leader among ten persons was selected. You must have noted many things while trying to solve the problem. These things will help us to understand the usage of data structures in C++, thus making the programming easy. The code of the program is given below.

```
#include "CList.cpp"
void main(int argc, char *argv[])
{
    CList list;
    int i, N=10, M=3;
    for(i=1; i <= N; i++ ) list.add(i);

    list.start();

    while( list.length() > 1 ) {
        for(i=1; i <= M; i++ ) list.next();

        cout << "remove: " << list.get() << endl;
        list.remove();
    }
    cout << "leader is: " << list.get() << endl;
}
```

In the program, we include the file of the class *CList* and create its object i.e. *list*. Then we solve the problem by using the *add*, *start*, *length*, *next*, *remove* and *get* methods of the class *CList*.

In the program, we have included already-defined data structure *CList*. After defining its different methods, we have an interface of *Clist*. There is no need to be worry about the nature of the list i.e. whether it is linked list, doubly linked list or an array. For us, it is only a list to be manipulated according to our requirement. You will see that a programmer may use different methods of the list object to solve the problem. We add elements to the list by a simple call of *add* method and go to the first element of the list by *start* method. Here, the length method is used in the condition of the *while* loop. Then we remove elements from the list and use the *next*, *get* and *remove* methods during this process. We get the current element by using the *get* method, then remove it by calling the *remove* method and then go to the next element by the method *next*. This way, all the elements are removed from the list except one element, called the leader. This one element remains there as we execute the while loop one less than the length of the list.

In singly linked list, the '*next*' returns false when it reaches to the last node due to the fact that the *next* field of the last node is set to NULL. But in a circularly linked list there is no NULL. It will be there only when there is no node in the list.

The whole process, which we carried out to solve the Josephus problem, can also be carried out with functions in C++. While adopting this way (of writing functions), we have to write these functions whenever we write another program that manipulates a list. In this method, we define a class of the data structure list and its different methods for the purpose of manipulation. This way, this class, obviously its methods too, can be used in any program where the manipulation of a list is needed. Thus there is re-usability of the code. In a class, we encapsulate the data and its methods. This shows that we are no longer interested in the internal process of the class. Rather, we simply use it wherever needed. The circular linked list, earlier used for the resolution of the Josephus problem, can also be employed in other problems. We have a class *CList* of this circular linked list through which any number of objects of data type of circular linked list can be created. Thus we can assume the class *CList* as a factory, creating as many objects of list as needed. This class and its objects in any program can be used to solve the problems with the help of its interface. The interface of this class consists of some methods like *add*, *remove*, *next*, *back*, *get* and some other simple ones. While carrying out programming, we will see that these classes (objects) help us very much to solve different problems.

Benefits of using circular list

While solving the Josephus problem, it was witnessed that the usage of circular linked list helped us make the solution trivial. We had to just write a code of some lines that solved the whole problem. In the program, we included the class *CList* (which is of our data structure i.e. circular linked list) and used all of its methods according to the requirements. There was no problem regarding the working of the methods. We just called these methods and their definition in the class *CList* worked well.

Now we will see what happens if we solve the Josephus problem by using an array instead of the class in our program. In this case, we have to define an array and write code to move back and forth in the array and to remove different elements properly in a particular order. A programmer needs to be very careful while doing this, to reach the solution of the problem. Thus our code becomes very complex and difficult for

someone to understand and modify it. Moreover we cannot use this code in some other problem. Note that here we are talking about the use of an array in the main program, not in the class that defines the *CList* data structure. There is no need to be worried whether an array, singly linked list, doubly linked list is used or circular linked list being employed internally in implementing the list in defining the class of list data type. We only want that it should create objects of list. The usage of the class of a data structure simplifies the code of the program. We can also use this class wherever needed in other programs. This shows that the choice of appropriate data structures can simplify an algorithm. It can make the algorithm much faster and efficient. In this course, we will see that there are different data structures, which makes the algorithms very easy to solve our problems. Later, we will see how some elegant data structures lie at the heart of major algorithms. There is also a course dedicated to study different algorithms and recipes that can be used to solve host of complex problems. Moreover, we will study different data structures in detail and see that with the use of a proper data structure, we can solve a problem efficiently. A properly constructed data structure will always help in the solution of problems.

Abstract Data Type

A data type is a collection of values and a set of operations on those values. That collection and these operations form a mathematical construct that may be implemented with the use of a particular hardware or software data structure. The term abstract data type (ADT) refers to the basic mathematical concept that defines the data type. We have discussed four different implementations of the list data structure. In case of implementation of the list with the use of an array, the size of the array gives difficulty if increased. To avoid this, we allocate memory dynamically for nodes before connecting these nodes with the help of pointers. For this purpose, we made a singly linked list and connected it with the next pointer to make a chain. Moving forward is easy but going back is a difficult task. To overcome this problem, we made a doubly linked list using *prev* and *next* pointers. With the help of these pointers, we can move forward and backward very easily. Now we face another problem that the *prev* pointer of first node and the *next* pointer of the last node are NULL. Therefore, we have to be careful in case of NULL pointers. To remove the NULL pointers, we made the circular link list by connecting the first and last node.

The program employing the list data structure is not concerned with its implementation. We do not care how the list is being implemented whether through an array, singly linked list, doubly linked list or circular linked list. It has been witnessed that in these four implementations of the list, the interface remained the same i.e. it implements the same methods like *add*, *get*, *next*, *start* and *remove* etc. This proves that with this encapsulation attained by making a class, we are not concerned with its internal implementation. The implementation of these abstract data types can be changed anytime. These abstract data types are implemented using classes in C++. If the list is implemented using arrays while not fulfilling the requirements, we can change the list implementation. It can be implemented with the use of singly-link list or doubly link list. As long as the interface is same, a programmer can change the internal implementation of the list and the program using this list will not be affected at all. This is the abstract data type (ADT). What we care about is the methods that are available for use, with the List ADT i.e. *add*, *get*, and *remove* etc methods. We have not studied enough examples to understand all the

benefits of abstract data types. We will follow this theme while developing other ADT. We will publish the interface and keep the freedom to change the implementation of ADT without effecting users of the ADT. The C++ classes provide a programmer an ability to create such ADTs. What benefits can we get with the help of these ADTs and classes? When we develop an ADT or a class or factory then the users of this factory are independent of how this factory works internally. Suppose that we have ordered the car factory (car class) to produce a new car and it replies after a long time. If we ordered the remove method to remove one node and we are waiting and it keeps on working and working. Then we might think that its implementation is not correct. Although, we are not concerned with the internal implementation of this ADT yet it is necessary to see whether this ADT is useful for solving our problem or not. It should not become a bottleneck for us. If the method we are using is too much time consuming or it has some problem in terms of algorithm used. On one side, we only use the interfaces provided by these ADTs, classes, or factories as long as they do what they promise. We are not concerned with the internal details. On the other hand, we have to be careful that these factories or methods should not take too much time so that these will not be useful for the problem.

This distinction will always be there. Sometimes, the source code of classes is not provided. We will be provided libraries, as standard libraries are available with the compiler. These classes are in compiled form i.e. are in object form or in binary form. On opening these files, you will not see the C++ code, rather binary code. When you read the assembly language code, it will give some idea what this binary code is about. You can view the interface methods in the `.h` file. As an application programmer, you have to see that the ADTs being used are written in a better way. The point to be remembered here is that you should not worry about the internal implementation of these ADTs. If we want to change the internal implementation of the ADTs, it can be done without affecting the users of these ADTs. While writing a program, you should check its performance. If at some point, you feel that it is slow, check the ADTs used at that point. If some ADT is not working properly, you can ask the writer of the ADT to change the internal implementation of that ADT to ensure that it works properly.

Stacks

Let's talk about another important data structure. You must have a fair idea of stacks. Some examples of stacks in real life are stack of books, stack of plates etc. We can add new items at the top of the stack or remove them from the top. We can only access the elements of the stack at the top. Following is the definition of stacks.

“Stack is a collection of elements arranged in a linear order”

Let's see an example to understand this. Suppose we have some video cassettes. We took one cassette and put it on the table. We get another cassette and put it on the top of first cassette. Now there are two cassettes on the table- one at the top of other. Now we take the third cassette and stack it on the two. Take the fourth cassette and stack it on the three cassettes.

Now if we want to take the cassette, we can get the fourth cassette which is at the top and remove it from the stack. Now we can remove the third cassette from the stack

and so on. Suppose that we have fifty cassettes stacked on each other and want to access the first cassette that is at the bottom of the stack. What will happen? All the cassettes will fall down. It will not happen exactly the same in the computer. There may be some problem. It does not mean that our data structure is incorrect. As we see in the above example that the top most cassette will be removed first and the new cassette will be stacked at the top. The same example can be repeated with the books. In the daily life, we deal with the stacked goods very carefully.

Now we will discuss how to create a stack data structure or a factory, going to create stack object for us. What will be the attributes of this object? During the discussion on the list, we came to know that a programmer adds values in the list, removes values from the list and moves forward and backward. In case of a stack too, we want to add things and remove things. We will not move forward or backward in the stack. New items can be added or removed at the top only. We can not suggest the removal of the middle element of the stack.

Let's talk about the interface methods of the stacks. Some important methods are:

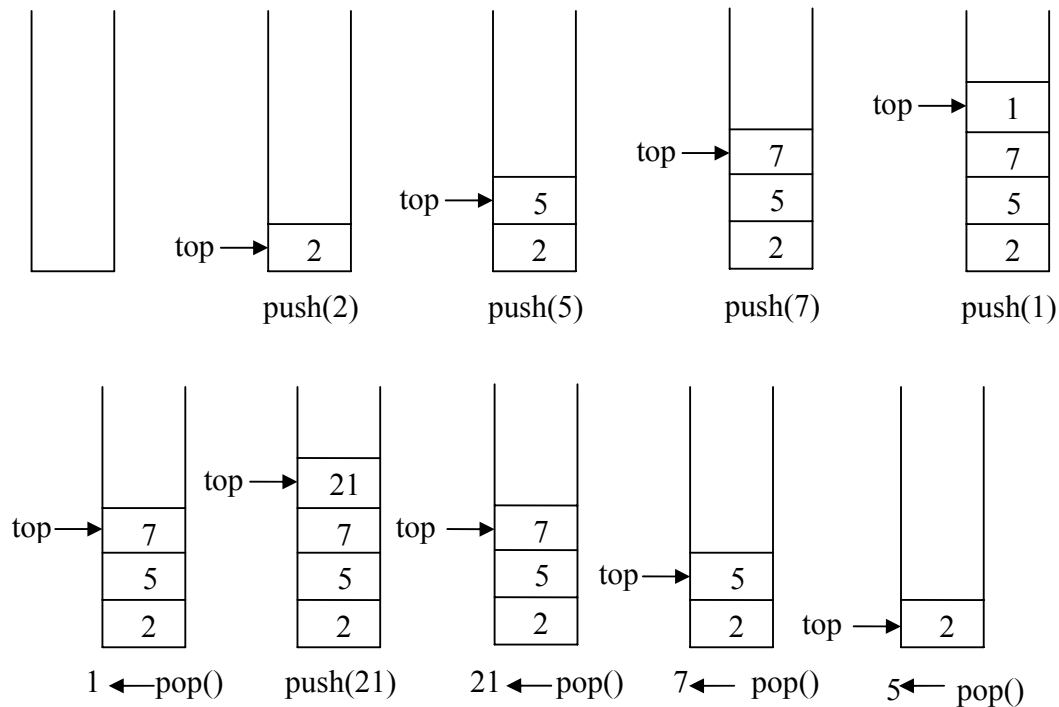
Method Name	Description
<code>push(x)</code>	Insert x as the top element of the stack
<code>pop()</code>	Remove the top element of the stack and return it.
<code>top()</code>	Return the top element without removing it from the stack.

The *push(x)* method will take an element and insert it at the top of the stack. This element will become top element. The *pop()* method will remove the top element of the stack and return it to the calling program. The *top()* method returns the top-most stack element but does not remove it from the stack. The interface method names that we choose has special objective. In case of list, we have used *add*, *remove*, *get*, *set* as the suitable names. However, for stack, we are using *push*, *pop* and *top*. We can depict the activity from the method name like push means that we are placing an element on the top of the stack and pushing the other elements down.

The example of a hotel's kitchen may help understand the concept of stacks in a comprehensive manner. In the kitchen, the plates are stacked in a cylinder having a spring on the bottom. When a waiter picks a plate, the spring moves up the other plates. This is a stack of plates. You will feel that you are pushing the plates in the cylinder and when you take a plate from the cylinder it pops the other plates. The top method is used to get the top- most element without removing it.

When you create classes, interfaces and methods, choose such names which depicts what these method are doing. These names should be suitable for that class or factory.

Let's discuss the working of stack with the help of a diagram.



At the start, the stack is empty. First of all, we push the value 2 in the stack. As a result, the number 2 is placed in the stack. We have a *top* pointer that points at the top element. Then we said *push(5)*. Now see how 2 and 5 are stacked. The number 5 is placed at the top of number 2 and the pointer *top* moves one step upward. Then we pushed the number 7 which is placed on the top and the number 2 and 5 are below. Similarly, we push number 1. The last figure in the first row shows the stacked values of the numbers- 1, 7, 5 and 2.

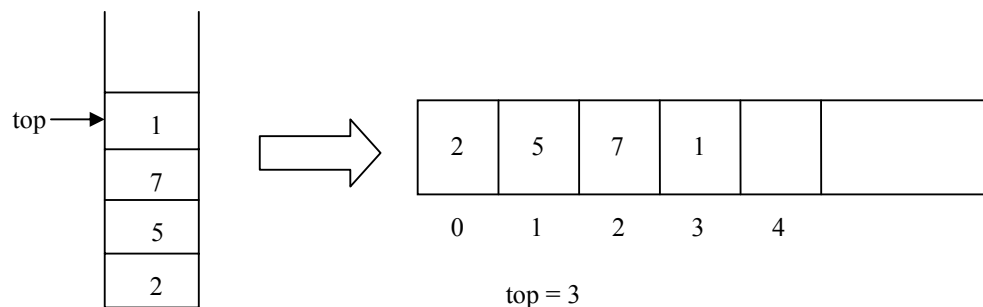
Let's pop the elements from the stack. The first figure of second row shows the pop operation. As a result, the number 1 is popped. Then again we push the number 21 on the stack. The number 7, 5, and 2 are already in the stack and number 21 is pushed at the top. If we pop now, the number 21 is popped. Now number 7 is at the top. If we pop again, the number 7 is popped. Pop again and the number 5 is popped and number 2 remains in the stack. Here with the help of this diagram, we are proving that the values are added at the top and removed at the top in a stack.

The last element to go into the stack is the first to come out. That is why, a stack is known as *LIFO* (Last In First Out) structure. We know that the last element pushed in the stack is at the top which is removed when we call *pop*. Let's see some other scenarios. What happens if we call *pop()* while there is no element? One possible way-out is that we have *isEmpty()* function that returns true if stack is empty and false otherwise. This is a boolean function that returns false if there is no element in the stack. Otherwise, it will return true. The second option is this that when we call *pop* on an empty stack, it throws an exception. This is a concept of advanced C++. Exception is also a way to convey that some unusual condition has arisen or something has gone wrong. Suppose, if we have a division method and try to divide some number with zero. This method will throw 'division by zero' exception.

Currently we will not throw an exception but use the *isEmpty()* method. The user who is employing the stack is responsible to call the *isEmpty()* method before calling the *pop*. Call the *pop* method if *isEmpty()* returns false. Otherwise, there will be a problem.

Stack Implementation using array

Let's discuss the implementation of the stack. Suppose we implement the stack using the arrays. The stack shown in the above diagram may be considered as an array. Here the array is shown vertically. We can implement the stack using array. The interface will remain as *push* and *pop* methods. The user of the stack does not need to know that the stack is internally implemented with the help of array. The worst case for insertion and deletion from an array may happen when we insert and delete from the beginning of the array. We have to shift elements to the right for insertion and left for removal of an element. We face the same problem while implementing the list with the use of the array. If we *push* and *pop* the elements from the start of the array for stack implementation, this problem will arise. In case of *push*, we have to shift the stack elements to the right. However, in case of *pop*, after removing the element, we have to shift the elements of stack that are in the array to the left. If we push the element at the end of the array, there is no need to shift any element. Similarly as the *pop* method removes the last element of the stack which is at the end of the array, no element is shifted. To insert and remove elements at the end of the array we need not to shift its elements. Best case for insert and delete is at the end of the array where there is no need to shift any element. We should implement *push()* and *pop()* by inserting and deleting at the end of an array.



In the above diagram, on the left side we have a stack. There are four elements in the stack i.e. 1, 7, 5 and 2. The element 1 is the extreme-most that means that it is inserted in the end whereas 7, 5, and 2 have been added before. As this is a LIFO structure so the element 1 should be popped first. On the right side we have an array with positions 0, 1, 2, 3 and so on. We have inserted the numbers 2, 5, 7 and 1. We have decided that the elements should be inserted at the end of the array. Therefore the most recent element i.e. 1 is at position 3. The *top* is the index representing the position of the most recent element. Now we will discuss the stack implementation in detail using array.

We have to choose a maximum size for the array. It is possible that the array may 'fill-up' if we push enough elements. Now more elements cannot be pushed. Now what should the user of the stack do? Internally, we have implemented the stack using array which can be full. To avoid this, we write *isFull()* method that will return

a boolean value. If this method returns true, it means that the stack (array) is full and no more elements can be inserted. Therefore before calling the *push(x)*, the user should call *isFull()* method. If *isFull()* returns false, it will depict that stack is not full and an element can be inserted. This method has become the part of the stack interface. So we have two more methods in our interface i.e. *isEmpty()* and *isFull()*.

Now we will discuss the actual C++ code of these operations. These methods are part of stack class or stack factory. We have an array named *A* while *current* is its index. The code of *pop()* method is as:

```
int pop()
{
    return A[current--];
}
```

In this method, the recent element is returned to the caller, reducing the size of the array by 1.

The code of *push* method is:

```
void push(int x)
{
    A[++current] = x;
}
```

We know that *++current* means that add one to the *current* and then use it. That also shows that element *x* should be inserted at *current* plus one position. Here we are not testing that this *current* index has increased from the array size or not. As discussed earlier that before using the *push* method, the user must call *isFull()* method. Similarly it is the responsibility of the user to call the *isEmpty()* method before calling the *pop* method. Therefore there is no *if* statement in the *push* and *pop* method.

The code of the *top()* method is:

```
int top()
{
    return A[current];
}
```

This method returns the element at the current position. We are not changing the value of *current* here. We simply want to return the top element.

```
int isEmpty()
{
    return ( current == -1 );
}
```

This method also tests the value of the *current* whether it is equal to -1 or not. Initially when the stack is created, the value of *current* will be -1. If the user calls the *isEmpty()* method before pushing any element, it will return true.

```
int isFull()
{
    return ( current == size-1);
}
```

This method checks that the stack is full or not. The variable *size* shows the size of the array. If the *current* is equal to the *size* minus one, it means that the stack is full and we cannot insert any element in it.

We have determined the cost and benefit of all the data structures. Now we will see how much time these methods take. A quick examination shows that all the five operations take constant time. In case of list, the *find* method takes too much time as it has to traverse the list. Whereas the *add* and *remove* methods are relatively quick. The methods of stack are very simple. There is no complexity involved. We insert element at one side and also remove from that side not in the middle or some other place. Therefore we need not to carry out a lot of work. During the usage of the array, the stack methods *push*, *pop*, *top*, *isFull* and *isEmpty* all are constant time operations. There is not much difference of time between them.

The complete code of the program is:

```
/* Stack implementation using array */

#include <iostream.h>

/* The Stack class */

class Stack
{
public:
    Stack() { size = 10; current = -1;}           //constructor
    int pop(){ return A[current--];}              // The pop function
    void push(int x){A[++current] = x;}           // The push function
    int top(){ return A[current];}                // The top function
    int isEmpty(){return ( current == -1 );}      // Will return true when stack is empty
    int isFull(){ return ( current == size-1);}   // Will return true when stack is full

private:
    int  object;                                // The data element
    int  current;                               // Index of the array
    int  size;                                  // max size of the array
    int  A[10];                                 // Array of 10 elements
};

// The main method
int main()
{
    Stack stack;                                // creating a stack object
    // pushing the 10 elements to the stack
    for(int i = 0; i < 12; i++)
```

```
{
    if(!stack.isFull())           // checking stack is full or not
        stack.push(i);           // push the element at the top
    else
        cout << "\n Stack is full, can't insert new element";
}

// pop the elements at the stack
for (int i = 0; i < 12; i++)
{
    if(!stack.isEmpty())         // checking stack is empty or not
        cout << "\n The popped element = " << stack.pop();
    else
        cout << "\n Stack is empty, can't pop";
}
}
```

The output of the program is:

```
Stack is full, can't insert new element
Stack is full, can't insert new element
The popped element = 9
The popped element = 8
The popped element = 7
The popped element = 6
The popped element = 5
The popped element = 4
The popped element = 3
The popped element = 2
The popped element = 1
The popped element = 0
Stack is empty, can't pop
Stack is empty, can't pop
```

However, a programmer finds the size-related problems in case of an array. What should we do when the array is full? We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements. Further discussion on this issue will be made in the next lecture.

Data Structures

Lecture No. 06

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 3
3.3.2, 3.3.3
(Postfix expressions)

Summary

- Stack From the Previous Lecture
- Stack Using Linked List
- Stack Implementation: Array or Linked List
- Use of Stack
- Precedence of Operators
- Examples of Infix to Postfix

Stack From the Previous Lecture

We started discussing *Stack* data structure and its implementation in the previous lecture. We also implemented stack structure using an array and wrote code for its *push()*, *pop()* and *top()* operations. We realized that we have to specify the size of the array before using it whether we declare it statically or dynamically. Arrays are of fixed size and when they become full, no more elements can be added to them. In order to get to know that the array has gone full, we wrote the *isFull()* method. It became the responsibility of the user of the stack structure to call *isFull()* method before trying to insert an element using the *push()* method otherwise the whole program could crash.

isEmpty() method is implemented as a *stack* can be empty like a *list* or *set* structures.

It is important to understand that *isFull()* method is there in stack implementation because of limitation of array but *isEmpty()* method is part of the stack characteristics or functionality.

As previously in the implementation of list structure, we used linked list while allocating nodes dynamically in order to avoid the fixed sized limitation of array. Now in this case also, again to overcome the limitation of array, we are going to make use of linked list in place of array to implement the stack data structure. Let's see, how we can implement a stack structure using linked list and how the implementation code will look like internally.

Stack Using Linked List

We can avoid the size limitation of a stack implemented with an array, with the help of a linked list to hold the stack elements.

As needed in case of array, we have to decide where to insert elements in the list and where to delete them so that *push* and *pop* will run at the fastest.

Primarily, there are two operations of a stack; *push()* and *pop()*. A stack carries *lifo* behavior i.e. last in, first out.

You know that while implementing stack with an array and to achieve *lifo* behavior, we used *push* and *pop* elements at the end of the array. Instead of pushing and popping elements at the beginning of the array that contains overhead of shifting elements towards right to *push* an element at the start and shifting elements towards left to *pop* an element from the start. To avoid this overhead of shifting left and right, we decided to *push* and *pop* elements at the end of the array.

Now, if we use linked list to implement the stack, where will we *push* the element inside the list and from where will we *pop* the element? There are few facts to consider, before we make any decision:

- For a *singly-linked list*, insert at start or end takes constant time using the *head* and *current* pointers respectively. As far as insertion is concerned, it is workable and equally efficient at the start and end.
- Removing an element at the start is constant time but removal at the end requires traversing the list to the node one before the last. So removing from the start is better approach rather than from the end.

Therefore, it makes sense to place stack elements at the start of the list because insertion and removal take constant time. As we don't need to move back and forth within the list, therefore, there is no requirement of *doubly* or *circular linked list*. *Singly linked list* can serve the purpose. Hence, the decision is to insert the element at the start in the implementation of *push* operation and remove the element from the start in the *pop* implementation.

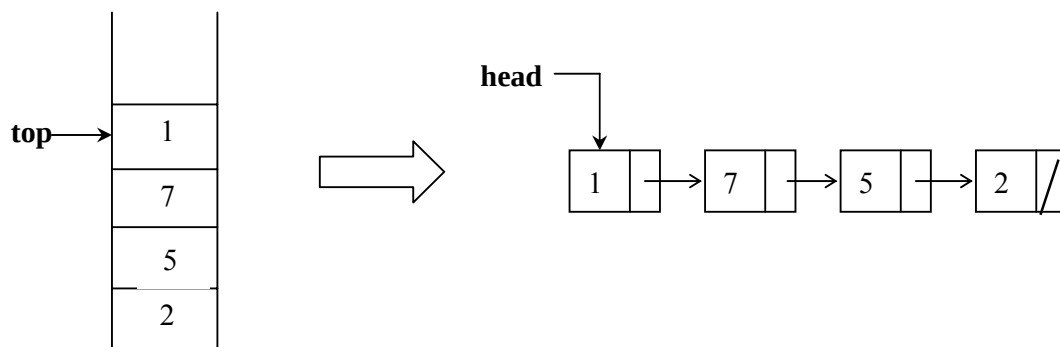


Fig 1. Stack using array (on left side) and linked list (on right side)

There are two parts of above figure. On the left hand, there is the stack implemented using an array. The elements present inside this stack are 1, 7, 5 and 2. The most recent element of the stack is 1. It may be removed if the *pop()* is called at this point of time. On the right side, there is the stack implemented using a linked list. This

stack has four nodes inside it which are linked in such a fashion that the very first node pointed by the *head* pointer contains the value 1. This first node with value 1 is pointing to the node with value 7. The node with value 7 is pointing to the node with value 5 while the node with value 5 is pointing to the last node with value 2. To make a stack data structure using a linked list, we have inserted new nodes at the start of the linked list.

Let's see the code below to implement *pop()* method of the stack.

```
int pop()
{
1.  int x = head->get();
2.  Node * p = head;
3.  head = head->getNext();
4.  delete p;
5.  return x;
}
```

At line 1, we have declared *x* as an *int* and retrieved one element from the node of the stack that is pointed by the *head* pointer. Remember, the *Node* class and its *get()* method that returns the value inside the node.

At line 2, *p* is declared as a pointer of type *Node* and address inside the *head* pointer is being saved inside this *p* pointer.

At line 3, the address of the next node is being retrieved with the help of the *getNext()* method of the *Node* class and being assigned to *head* pointer. After this assignment, the *head* pointer has moved forward and started pointing to the next element in the stack.

At line 4, the node object pointed by the pointer *p* is being deallocated (deleted).

At line 5, the function is returning the value of the node retrieved in step 1.

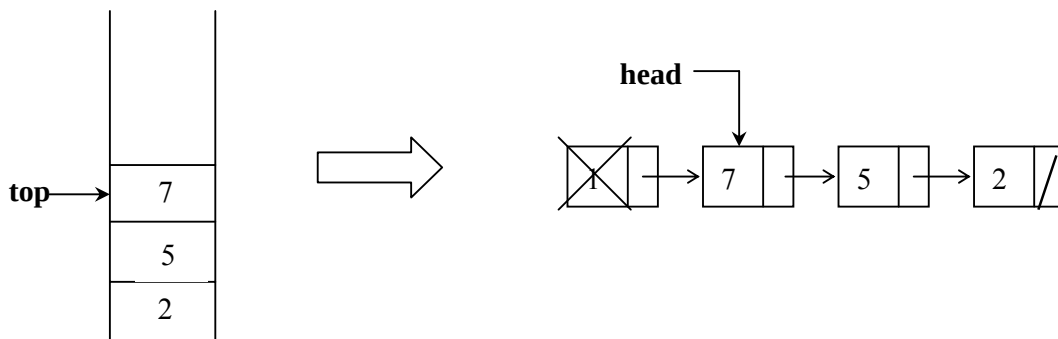


Fig 2. A node removed from the stack after the *pop()* call

Let's see the code of the *push()* method of the stack:

```
void push(int x)
{
1.  Node * newNode = new Node();
2.  newNode->set(x);
}
```

```

3. newNode->setNext(head);
4. head = newNode;
}

```

In line 1, a new node is created, using the *new Node()* statement and returned pointer is assigned to a pointer *newNode*. So *newNode* starts pointing to the newly created *Node* object.

In line 2, the value 2 is set into the newly created *Node* object.

In line 3, the next node of the newly created node is set to the node pointed to by the *head* pointer using *setNext(head)*.

In line 4, the *head* pointer is made to point to the newly created node.

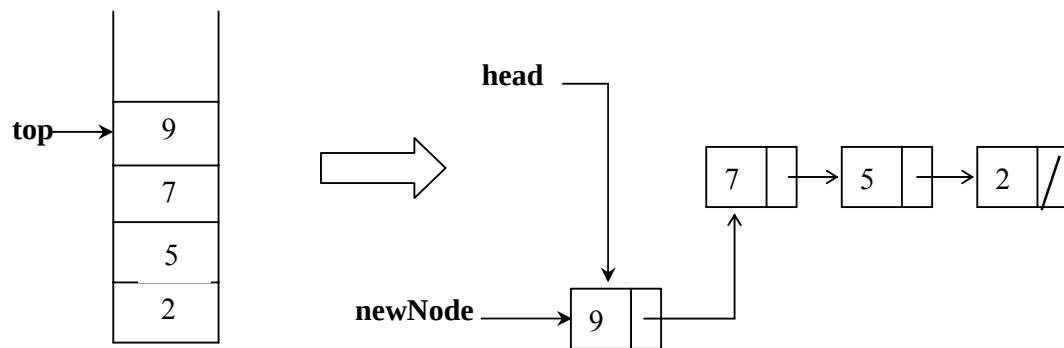


Fig 3. A node added to the stack after the push(9) call

These are two primary methods of a stack. By using the *push()* method, we can keep on pushing elements and using the *pop()* methods. Elements can be removed from the stack till the time, it gets empty. As discussed earlier, *isEmpty()* is the stack characteristic but *isFull()* was implemented because of the size limitation of the array. We are no more using array to implement a stack. Rather, we have used linked list here for stack implementation. Therefore, *isFull()* might not be required here. An interesting question arises here. Can we add infinite elements to the stack now. We should remember that this program of stack will run on computer that definitely has a limited memory. Memory or *Address space* of a computer is the space (physical memory and disk space) that can be addressed by the computer which is limited including the limited physical memory. Disk space is used as the *virtual memory* (we will not discuss *virtual memory* in detail here). A computer with 32-bit addressing can address upto $2^{32}-1$ memory locations and similarly a computer with 64-bit addressing can address upto $2^{64}-1$ addresses. If this address space becomes full, the stack will definitely be full. However, the stack implementation is not liable for this fullness of address space and it is the limitation of a computer address space. Therefore, we don't need to call *isFull()* before *pushing* the element. Rather, *isEmpty()* is called before *popping* an element from the stack.

Let's see the remaining methods of the stack while using linked list to implement it.

```

int top()
{

```



```
    return head->get();
}
int isEmpty()
{
    return ( head == NULL );
}
```

The above-mentioned methods i.e. *top()* and *isEmpty()* are very simple functions. One statement inside the *top()* is retrieving the top element (pointed to by the *head* pointer) from the stack and returning it back by value. It is important to note that *top()* is not removing the element from the stack, but only retrieving it. The one statement inside *isEmpty()* is a check to see if the *head* pointer is not pointing to any node and it is *NULL*. If the *head* pointer is *NULL* that means the stack is empty, the method returns *true* otherwise it returns *false*.

All four operations *push()*, *pop()*, *top()* and *isEmpty()* take constant time. These are very simple methods and don't contain loops. They are also not CPU hungry operation. Also note that we have not written *isFull()* while implementing stack with the linked list.

Stack Implementation: Array or Linked List

Since both implementations support stack operations in constant time, we will see what are the possible reasons to prefer one implementation to the other.

- Allocating and de-allocating memory for list nodes does take more time than pre-allocated array. Memory allocation and de-allocation has cost in terms of time, especially, when your system is huge and handling a volume of requests. While comparing the stack implementation, using an array versus a linked list, it becomes important to consider this point carefully.
- List uses as much memory as required by the nodes. In contrast, array requires allocation ahead of time. In the previous bullet, the point was the time required for allocation and de-allocation of nodes at runtime as compared to one time allocation of an array. In this bullet, we are of the view that with this runtime allocation and de-allocation of nodes, we are also getting an advantage that list consumes only as much memory as required by the nodes of list. Instead of allocating a whole chunk of memory at one time as in case of array, we only allocate memory that is actually required so that the memory is available for other programs. For example, in case of implementing stack using array, you allocated array for 1000 elements but the stack, on average, are using 50 locations. So, on the average, 950 locations remain vacant. Therefore, in order to resolve this problem, linked list is handy.
- List pointers (*head*, *next*) require extra memory. Consider the manipulation of array elements. We can set and get the individual elements with the use of the array index; we don't need to have additional elements or pointers to access them. But in case of linked list, within each node of the list, we have one pointer element called *next*, pointing to the next node of the list. Therefore, for 1000 nodes stack implemented using list, there will be 1000 extra pointer variables. Remember that stack is implemented using 'singly-linked' list. Otherwise, for doubly linked list, this overhead is also doubled as two pointer variables are stored within each node in that case.
- Array has an upper limit whereas list is limited by dynamic memory allocation. In

other words, the linked list is only limited by the address space of the machine. We have already discussed this point at reasonable length in this lecture.

Use of Stack

Examples of uses of stack include- traversing and evaluating prefix, *infix* and postfix expressions.

Consider the expression $A+B$: we think of applying the *operator* “+” to the *operands* A and B . We have been writing this kind of expressions right from our primary classes. There are few important things to consider here:

Firstly, + *operator* requires two operands or in other words “+” is a *binary operator*. Secondly, in the expression $A+B$, the one operand A is on left of the operator while the other operand B is on the right side. This kind of expressions where the operator is present between two operands called *infix* expressions. We take the meanings of this expression as to add both operands A and B .

There are two other ways of writing expressions:

- We could write $+AB$, the operator is written before the operands A and B . These kinds of expressions are called *Prefix* Expressions.
- We can also write it as $AB+$, the operator is written after the operands A and B . This expression is called *Postfix* expression.

The prefixes *pre* and *post* refer to the position of the operator with respect to the two operands.

Consider another expression in *infix* form: $A + B * C$. It consists of three operands A , B , C and two operator $+$, $*$. We know that multiplication ($*$) is done before addition ($+$), therefore, this expression is actually interpreted as: $A + (B * C)$. The interpretation is because of the precedence of multiplication ($*$) over addition ($+$). The precedence can be changed in an expression by using the parenthesis. We will discuss it a bit later.

Let’s see, how can we convert the infix expression $A + (B * C)$ into the postfix form. Firstly, we will convert the multiplication to postfix form as: $A + (B C *)$. Secondly, we will convert addition to postfix as: $A (B C *) +$ and finally it will lead to the resultant postfix expression i.e. : $A B C * +$. Let’s convert the expression $(A + B) * C$ to postfix. You might have noticed that to overcome the precedence of multiplication operator ($*$) we have used parenthesis around $A + B$ because we want to perform addition operation first before multiplication.

$(A + B) * C$	<i>infix form</i>
$(A B +) * C$	<i>convert addition</i>
$(A B +) C *$	<i>convert multiplication</i>
$A B + C *$	<i>postfix form</i>

These expressions may seem to be difficult to understand and evaluate at first. But this is one way of writing and evaluating expressions. As we are normally used to infix form, this postfix form might be little confusing. If a programmer knows the algorithm, there is nothing complicated and even one can evaluate the expression manually.

Precedence of Operators

There are five binary operators, called *addition*, *subtraction*, *multiplication*, *division* and *exponentiation*. We are aware of some other binary operators. For example, all relational operators are binary ones. There are some unary operators as well. These require only one operand e.g. $-$ and $+$. There are rules or order of execution of operators in Mathematics called precedence. Firstly, the *exponentiation* operation is executed, followed by multiplication/division and at the end addition/subtraction is done. The order of precedence is (highest to lowest):

Exponentiation	$\uparrow$
Multiplication/division	$*, /$
Addition/subtraction	$+, -$

For operators of same precedence, the left-to-right rule applies:

$A+B+C$ means $(A+B)+C$.

For exponentiation, the right-to-left rule applies:

$A \uparrow B \uparrow C$ means $A \uparrow (B \uparrow C)$

We want to understand these precedence of operators and infix and postfix forms of expressions. A programmer can solve a problem where the program will be aware of the precedence rules and convert the expression from infix to postfix based on the precedence rules.

Examples of Infix to Postfix

Let's consider few examples to elaborate the *infix* and *postfix* forms of expressions based on their precedence order:

Infix	Postfix
$A + B$	$A B +$
$12 + 60 - 23$	$12 60 + 23 -$
$(A + B) * (C - D)$	$A B + C D - *$
$A \uparrow B * C - D + E / F$	$A B \uparrow C * D - E F / +$

In the next lecture we will see, how to convert *infix* to *postfix* and how to evaluate *postfix* form besides the ways to use stack for these operations.

Data Structures

Lecture No. 07

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 3

3.3.3

Summary

- 9) Evaluating postfix expressions
- 10) An example
- 11) Infix to postfix Conversion

Evaluating postfix expressions

In the previous lecture, we talked about ‘infix and postfix expression’ and tried to understand how to write the postfix notation of mathematical expressions. A programmer can write the operators either after the operands i.e. postfix notation or before the operands i.e. prefix notation. Some of the examples are as under:

Infix	Postfix
$A + B$	$A B +$
$12 + 60 - 23$	$12 60 + 23 -$
$(A + B) * (C - D)$	$A B + C D - *$
$A \uparrow B * C - D + E / F$	$A B \uparrow C * D - E F / +$

The last expression seems a bit confusing but may prove simple by following the rules in letter and spirit. In the postfix form, parentheses are not used. Consider the infix expressions as ‘ $4+3*5$ ’ and ‘ $(4+3)*5$ ’. The parentheses are not needed in the first but are necessary in the second expression. The postfix forms are:

$4+3*5$	$435*+$
$(4+3)*5$	$43+5*$

In case of not using the parenthesis in the infix form, you have to see the precedence rule before evaluating the expression. In the above example, if we want to add first then we have to use the parenthesis. In the postfix form, we do not need to use parenthesis. The position of operators and operands in the expression makes it clear in which order we have to do the multiplication and addition.

Now we will see how the infix expression can be evaluated. Suppose we have a postfix expression. How can we evaluate it? Each operator in a postfix expression refers to the previous two operands. As the operators are binary (we are not talking about unary operators here), so two operands are needed for each operator. The nature

of these operators is not affected in the postfix form i.e. the plus operator (+) will apply on two operands. Each time we read an operand, we will push it on the stack. We are going to evaluate the postfix expression with the help of stack. After reaching an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack. Now we will see an example to comprehend the working of stack for the evaluation of the postfix form. Here is the algorithm in pseudo code form. After reading this code, you will understand the algorithm.

```

Stack s;                                // declare a stack
while( not end of input ) {             // not end of postfix expression
    e = get next element of input
    if( e is an operand )
        s.push( e );
    else {
        op2 = s.pop();
        op1 = s.pop();
        value = result of applying operator 'e' to op1 and op2;
        s.push( value );
    }
}
finalresult = s.pop();

```

We have declared a Stack 's'. There is a 'while loop' along with 'not end of input' condition. Here the input is our postfix expression. You can get the expression from the keyboard and use the enter key to finish the expression. In the next statement, we get the next element and store it in 'e'. This element can be operator or operand. The operand needs not to be single digit. It may be of two digits or even more like 60 or 234 etc. The complete number is stored in the 'e'. Then we have an 'if statement' to check whether 'e' is an operand or not. If 'e' is an operand then we wrote *s.push(e)* i.e. we pushed the 'e' onto the stack. If 'e' is not the operand, it may be an operator. Therefore we will pop the two elements and apply that operator. We pop the stack and store the operand in 'op2'. We pop the stack again and store the element in 'op1'. Then the operator in 'e' is applied to 'op1' and 'op2' before storing the result in *value*. In the end, we push the 'value' on the stack. After exiting the loop, a programmer may have only one element in the stack. We pop this element which is the final result.

Consider the example of $4+3*2$ having a postfix form of $432*+$. Here 4, 3, and 2 are operands whereas + and * are operators. We will push the numbers 4, 3 and 2 on the stack before getting the operator *. Two operands will be popped from the stack and * is being applied on these. As stack is a LIFO structure, so we get 2 first and then 3 as a result of pop. So 2 is store in 'op1' and 3 in 'op2'. Let's have a look on the program again. On applying * on these, we will push the result (i.e. 6) on the stack. The 'while loop' will be executed again. In case of getting the next input as operand, we will push it on the stack otherwise we will pop the two operands and apply the operator on these. Here the next element is the operator +. So two operands will be popped from the stack i.e. 6 and 4. We will apply the operator plus on these and push the result (i.e. 10) on the stack. The input is finished. Now we will pop the stack to get the final

result i.e. 10.

An Example

In the earlier example, we have used the stack to solve the postfix expression. Let's see another comprehensive example. The postfix expression is:

$$6\ 2\ 3\ +\ -\ 3\ 8\ 2\ /\ +\ * \ 2\ \uparrow\ 3\ +$$

We want to evaluate this long expression using stack. Let's try to solve it on paper. We have five columns here i.e. input, op1, op2, value and stack. We will run our pseudo code program. In the start, we read the input as a result we get number 6. As 6 is operand, so it will be pushed on the stack. Then we have number 2 which will also be pushed on the stack. Now 2 is the most recent element. The next element is the number 3 that will also be pushed on the stack. Now, there are three elements on the stack i.e. 3, 2 and 6. The number 3 is the most recent. On popping, we will get the number 3 first of all. The next element is '+', an operator. Now the else part of our pseudo code is executed. We will pop two operands from the stack and apply the operator (+) on these. The number 3 will be stored in variable *op2* and number 2 in *op1*. The operator (+) will be applied on these i.e. 2+3 and the result is stored in *value*. Now we will push the *value* (i.e. 5) on the stack. Now we have two numbers on the stack i.e. 5 and 6. The number 5 is the most recent element. The next element is '-'. As it is also an operator, so we will pop the two elements from the stack i.e. 5 and 6. Now we have 5 in *op2* and 6 in *op1*. On applying the operator (-), we will get the result as 1 (6-5). We can't say *op2* - *op1*. The result (1) will be pushed on stack. Now on the stack, we have only one element i.e. 1. Next three elements are operands so we pushed 3, 8 and 2 on the stack. The most recent element is 2. The next input is an operator in the expression i.e. '/', we will pop two elements from the stack. The number 2 will be stored in *op2* while number 8 in *op1*. We apply the operator (/) on the *op1* and *op2* i.e. (*op1*/*op2*), the result is 4 (i.e. 8/2). We push the result on the stack. We have, now, three elements i.e. 4, 3, and 1 on the stack. The next element is operator plus (+). We will pop the two elements i.e. 4 and 3 and will apply the operator (+). The result (7) will be pushed on the stack. The next input element is operator multiply (*). We will pop the two elements i.e. 7 and 1 and the result (7*1 = 7) is pushed on the stack. You have noted that whenever we have an operator in the input expression, we have two or more elements on the stack. As the operators we are using are binary and we need two operands for them. It will never be the case that you want to pop two elements from the stack and there is only one or no element on the stack. If this happens then it means there is an error in the program and you have popped more values than required. The next input element is 2 that is pushed on the stack. We have, now, the operator ($\uparrow$) in the input. So we will pop the two elements, *op2* will hold 2 and *op1* will have the number 7. The operator ($\uparrow$) will be applied on the operands i.e. (7 $\uparrow$ 2) and the result (49) is pushed on the stack. We have, now, the number 3 in the element being pushed on the stack. The last element is the operator plus (+). So we pop the two elements i.e. 49 and 2 and apply the operator on these. The result (49+3 = 52) is pushed on the stack. The input expression is finished, resulting in the final result i.e. 52.

This the tabular form of the evaluation of the postfix expression.

Input	op1	op2	value	stack
6				6
2				2 6
3				3 2 6
+	2	3	5	5 6
-	6	5	1	1
3	6	5	1	3 1
8	6	5	1	8 3 1
2	6	5	1	2 8 3 1
/	8	2	4	4 3 1
+	3	4	7	7 1
*	1	7	7	7
2	1	7	7	2 7
↑	7	2	49	49
3	7	2	49	3 49
+	49	3	52	52

With the help of stack we can easily solve a very big postfix expression. Suppose you want to make a calculator that is a part of some application e.g. some spreadsheet program. This calculator will be used to evaluate expressions. You may want to calculate the value of a cell after evaluating different cells. Evaluation of the infix form programmatically is difficult but it can be done. We will see another data structure which being used to solve the expressions in infix form. Currently, we have to evaluate the values in different cells and put this value in another cell. How can we do that? We will make the postfix form of the expression associated with that cell. Then we can apply the above algorithm to solve the postfix expression and the final result will be placed at that cell. This is one of the usages of the stack.

Infix to postfix Conversion

We have seen how to evaluate the postfix expressions while using the stack. How can we convert the infix expression into postfix form? Consider the example of a spreadsheet. We have to evaluate expressions. The users of this spreadsheet will employ the infix form of expressions. Consider the infix expressions 'A+B*C' and '(A+B)*C'. The postfix versions are 'ABC*+' and 'AB+C*' respectively. The order

of operands in postfix is the same as that in the infix. In both the infix expressions, we have the order of operands as A, B and then C. In the postfix expressions too, the order is the same i.e. A, B, followed by C. The order of operands is not changed in postfix form. However, the order of operators may be changed. In the first expression 'A+B*C', the postfix expression is 'ABC*+'. In the postfix form multiplication comes before the plus operator. In scanning from left to right, the operand 'A' can be inserted into postfix expression. First rule of algorithm is that if we find the operand in the infix form, put it in the postfix form. The rules for operators are different. The '+' cannot be inserted in the postfix expression until its second operand has been scanned and inserted. Keep the expression A+B*C in your mind. What is the second operand of the plus? The first operand is A and the second operand is the result of B*C. The '+' has to wait until the '*' has not been performed. You do the same thing while using the calculator. First you will multiply the B*C and then add A into the result. The '+' has to be stored away until its proper position is found. When 'B' is seen, it is immediately inserted into the postfix expression. As 'B' is the operand, we will send the operand to the postfix form. Can the '+' be inserted now? In case of 'A+B*C', we cannot insert '+' because '*' has precedence. To perform multiplication, we need the second operand. The first operand of multiplication is 'B' while the second one is 'C'. So at first, we will perform the multiplication before adding result to 'A'.

In case of '(A+B)*C', the closing parenthesis indicates that '+' must be performed first. After sending the A and B to postfix perform, we can perform the addition due to the presence of the parenthesis. Then C will be sent to the postfix expression. It will be followed by the multiplication of the C and the result of A + B. The postfix form of this expression is AB+C*. Sometimes, we have two operators and need to decide which to apply first like in this case '+' and '*'. In this case, we have to see which operator has higher precedence. Assume that we have a function '*prcd(op1,op2)*' where *op1* and *op2* are two operators. The function '*prcd(op1,op2)*' will return TRUE if *op1* has precedence over *op2*, FALSE otherwise. Suppose we call this function with the arguments '*' and '+' i.e. *prcd(*, +)*, it will return true. It will also return true in case both *op1* and *op2* are '+' e.g. if we have A+B+C, then it does not matter which + we perform first. The call *prcd(+, *)* will return false as the precedence of * is higher than the + operator. The '+' has to wait until * is performed.

Now we will try to form an algorithm to convert infix form into postfix form. For this purpose, a pseudo code will be written. We will also write the loops and if conditions. The pseudo code is independent of languages. We will be using a stack in this algorithm. Here, the infix expression is in the form of a string. The algorithm is as follows:

```
Stack s;
while( not end of input ) {
    c = next input character;
    if( c is an operand )
        add c to postfix string;
    else {
        while( !s.empty() && prcd(s.top(),c) ){
            op = s.pop();
            add op to the postfix string;
```



```

    }
    s.push( c );
}
while( !s.empty() ) {
    op = s.pop();
    add op to postfix string;
}

```

First we will declare a stack 's'. The 'while loop' will continue till the end of input. We read the input character and store it in the 'c'. Here the input character does not mean one character, but an operand or an operator. Then we have a conditional if statement. If 'c' is an operand, then we will have to add it to postfix string. Whenever we get an operand in the infix form, it will be added to the postfix form. The order of operands does not change in the conversion. However, in this case, the order of operators may change. If 'c' is the operator, then we will, at first, check that stack is not empty besides identifying the precedence of the operators between the input operator and the operator that is at the top of the stack. In case of the precedence of the operator that is on the stack is higher, we will pop it from the stack and send to the postfix string. For example if we have * on the stack and the new input operator is +. As the precedence of the + operator is less than the * operator, the operands of the multiplication has already been sent to the postfix expression. Now, we should send the * operator to the postfix form. The plus operator (+) will wait. When the while loop sends all such operators to the postfix string, it will push the new operator to the stack that is in 'c'. It has to wait till we get the second operand. Then we will again get the input. On the completion of the input, the while loop will be finished. There may be a case that input may be completed even at the time when there are still some elements on the stack. These are operators. To check this, we have another while loop. This loop checks if the stack is not empty, pops the operator and put it in the postfix string. Let's take a look at a comprehensive example to understand it. In case of the infix expression, $A + B * C$, we have three columns, one each for input symbol, the postfix expression and the stack respectively. Now let's execute the pseudo code. First of all, we get the 'A' as input. It is an operand so we put it on the postfix string. The next input is the plus operator (+) which will be pushed on the stack. As it is an operator and we need two operands for it. On having a look at the expression, you might have figure out that the second operand for the plus operator is $B * C$. The next input is the operand B being sent to the postfix expression form. The next thing we get is the input element as '*'. We know that the precedence of * is higher than that of the +. Let's see how we can do that according to our pseudo code. The *prcd(s.top(), op)* takes two operands. We will get the top element of the stack i.e. + will be used as first argument. The second argument is the input operator i.e. *. So the function call will be as *prcd(+, *)* while the function returns false because the precedence of the plus operator is not higher than the multiplication operator. So far, we have only one operand for multiplication i.e. B. As multiplication is also a binary operator, it will also have to wait for the second operand. It has to wait and the waiting room is stack. So we will push it on the stack. Now the top element of the stack is *. The next symbol is 'C'. Being an operand, C will be added to the postfix expression. At this point, our input expression has been completed. Our first 'while loop' executes till the end of input. After the end of the input, the loop will be terminated. Now the control goes to the second while loop which says if there is something on the stack, pop it and add it the postfix expression. In this case, we have * and + on the stack. The * is at the

top of the stack. So when we pop, we get * which is at the top of the stack and it will be added to the postfix expression. In the result of second pop, we get the plus operator (+) which is also added to the postfix expression. The stack is empty now. The while loop will be terminated and postfix expression is formed i.e. ABC*+.

Symbol	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	* +
C	ABC	* +
	ABC*	+
	ABC*+	

If we have to convert the infix expression into the postfix form, the job is easily done with the help of stack. The above algorithm can easily be written in C++ or C language, specially, if you already have the stack class. Now you can convert very big infix expressions into postfix expressions. Why we have done this? This can be understood with the help of the example of spreadsheet programming where the value of cell is the evaluation of some expression. The user of the spreadsheets will use the infix expressions as they are used to it.

Sometimes we do need the parenthesis in the infix form. We have to evaluate the lower precedence operator before the higher precedence operator. If we have the expression (A+B) *C, this means that we have to evaluate + before the multiplication. The objective of using parenthesis is to establish precedence. It forces to evaluate the expression first of all. We also have to handle parenthesis while converting the infix expression into postfix one. When an open parenthesis '(' is read, it must be pushed on the stack. This can be done by setting *prcd(op, '(')* to be FALSE. What is the reason to put the parenthesis on the stack? It is due to the fact that as long as the closing parenthesis is not found, the open parenthesis has to wait. It is not a unary or binary operator. Actually, it is a way to show or write precedence. We can handle the parenthesis by adding some extra functionality in our *prcd* function. When we call *prcd(op, '(')*, it will return false for all the operators and be pushed on the stack. Also, *prcd('(', op)* is FALSE which ensures that an operator after '(' is pushed on the stack. When a ')' is read. All operators up to the first '(' must be popped and placed in the postfix string. To achieve this our function *prcd(op, ')')* should return true for all the operators. Both the '(' and the ')' will not go to the postfix expression. In postfix expression, we do not need parenthesis. The precedence of the operators is established in such a way that there is no need of the parenthesis. To include the handling of parenthesis, we have to change our algorithm. We have to change the line *s.push(c)* to:

```
if( s.empty() || symb != '(' )
    s.push( c );
else
    s.pop(); // discard the '('
```

If the input symbol is not ‘)’ and the stack is not empty, we will push the operator on the stack. Otherwise, it is advisable to pop the stack and discard the ‘(’. The following functionality has to be added in the *prcd* function.

<code>prcd(‘(’, op)</code>	<code>=</code>	<code>FALSE</code>	for any operator
<code>prcd(op, ‘)’)</code>	<code>=</code>	<code>FALSE</code>	for any operator other than ‘)’
<code>prcd(op, ‘(’)</code>	<code>=</code>	<code>TRUE</code>	for any operator other than ‘(’
<code>prcd(‘)’ , op)</code>	<code>=</code>	<code>error</code>	for any operator.

In the next lecture we will see in detail an example regarding the use of parenthesis.

Data Structures

Lecture No. 08

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 3

3.3.3

Summary

- Conversion from infix to postfix
- C++ Templates
- Implementation of Stack
- Function Call Stack

Conversion from infix to postfix

In the previous lecture, we discussed the way to convert an infix notation into a postfix notation. During the process of conversion, we saw that there may be need of parenthesis in the infix especially at the times when we want to give a higher precedence to an operator of lower precedence. For example, if there is a + operator and * operator in an expression and a programmer wants the execution of addition before the multiplication. To achieve this object, it is necessary to put parentheses around the operands of + operator. Suppose, there is the expression $A + B * C$ in which we want to give the precedence to the + operator over * operator. This expression will be written as $(A + B) * C$. Now we are going to discuss the conversion of infix expression that includes parentheses to the postfix expression. We have defined the return values for opening '(' and closing ')' parentheses in the precedence function. Let's try to understand this process with the help of an example of converting the infix expression $(A + B) * C$ into a postfix expression. We will see how our algorithm, discussed earlier, converts this infix expression into a postfix expression. To carry out the process of conversion we have three columns symbol, postfix and stack. The column symbol has the input symbols from the expression. The postfix column has the postfix string (expression) after each step and the stack is used to put the operators on it. The whole process of converting the infix notation into a postfix is given in the following table. This process of conversion is completed in eight steps. Each of the rows of the table depicts one step.

Step No.	Symbol	Postfix	Stack
1	(		(
2	A	A	(

3	+	A	(+
4	B	AB	(+
5	)	AB+	
6	*	AB+	*
7	C	AB+C	*
8		AB+C*	

First of all, there is the input symbol '(' (i.e. opening parenthesis). As this is not an operand, it may be put on the stack. The next input symbol is 'A'. Being an operand it goes to the postfix string and the stack remains unchanged. Then there is + operator of binary type. Moreover, there is one operand in the postfix string. We push this + operator on the stack and it has to wait for its second operand. Now in the input symbol, there is an operand 'B'. We put this operand in the postfix string. Then after this, there is the closing parenthesis ')' in the input symbol. We know that the presence of a closing parenthesis in the input means that an expression (within the parentheses) has been completed. All of its operands and operators are present within the parentheses. As studied in the algorithm, we discard a closing parenthesis when it comes in the input. Then the operators from the stack are popped up and put in the postfix string. We also pop the opening parenthesis and discard it as we have no need of opening as well as closing parenthesis in the postfix notation of an expression. This process is carried out in the 5th row of the table. The + operator is put in the postfix string. We also discard the opening parenthesis, as it is not needed in the postfix.

Now the next input symbol is *. We put this operator on the stack. There is one operand for the * operator i.e. AB+. The * operator being a binary operator, has to wait for the second operand. 'C' is the Next input symbol that is an operand. We put it in the postfix string. After this, the input string (expression) ends so we come out of the loop. We check if there is anything on the stack now? There is * operator in the stack. We pop the operator and put it into the postfix string. This way, we get the postfix form of the given infix expression that becomes AB+C*. In this postfix expression, the + operator is before the * operator. So addition operation is done before the multiplication. This is mainly due to the fact that in the infix expression, we have put parentheses to give + operator the precedence higher than the * operator. Note that there are no parentheses in the postfix form of the given infix expression.

Now we apply the evaluation algorithm on this postfix expression (i.e. AB+C*). The two operands A and B, will go to the stack. Then operator + will pop these operands from the stack, will add them and push the result back on the stack. This result becomes an operand. Next 'C' will go to the stack and after this * operator will pop these two operands (result of addition and C). Their multiplication will lead to the final result. The postfix notation is simple to evaluate as compared to the infix one. In postfix, we need not to worry about what operation will be carried first. The operators in this notation are in the order of evaluation. However, in the infix notation, we have to force the precedence according to our requirement by putting parentheses in the expression. With the help of a stack data structure, we can do the conversion and evaluation of expressions easily.

C++ Templates

We can use C++ templates for stack and other data structures. We have seen that stack is used to store the operands while evaluating an expression. These operands may be

integers, floating points and even variables. We push and pop the operands to and from the stack. In the conversion of an expression, a programmer uses the stack for storing the operators like +, *, -, and / etc which are single characters. In both cases, the functionality is the same. We push and pop things on and from the stack. At times, we check if the stack is empty or not. Thus identical methods are employed while using stack in evaluating and converting the expressions. However, there may be a difference in the type of the elements (data) used in these cases. We may define a stack, which can store different types of data but for the time being we are restricting ourselves to the stack that can store elements of only one type. In C++ programming, we will have to create two classes *FloatStack* and *CharStack* for operands and operators respectively. These classes of stack have the same implementation. Thus, we write the same code twice only with the difference of data type. Is there any method to write the code for the stack once and then use it for different types of data? This means is there any way that we can make a stack for storing integers, floating points, characters or even objects with the same code written once. The language C++ provides us the facility of writing templates. A template can be understood with the example of a factory that bakes biscuits. The factory may use flour, corn or starch as ingredients of the product. But the process of baking biscuits is the same whatever ingredients it uses. There is no difference in the machinery for producing biscuits with different ingredients. So we call the factory as the template for the biscuits. Similarly in C++ language, a template is a function or class that is written with a generic data type. When a programmer uses this function or class, the generic data type is replaced with the data type, needed to be used in the template function or in the template class. We only give the data type of our choice while calling a template function or creating an object of the template class. The compiler automatically creates a version of that function or class with that specified data type. Thus if we write a template class for stack, then later on we can use it for creating a stack for integers, floating points or characters etc. So instead of writing code for different stacks of different data types, we write one code as a template and reuse it for creating different stacks. We declare the template class in a separate file in addition to the main program file. This file can be used in our program by including it in that file. Following is the code of the template class for stack. This is written in the file **Stack.h**.

```
template <class T>
class Stack
{
    public:
        Stack();
        int empty(void);    // 1=true, 0=false
        int push(T &);      // 1=successful,0=stack overflow
        T pop(void);
        T peek(void);
        ~Stack();
    private:
        int top;
        T* nodes;
};
```

In the above code the line

```
template <class T>
```

shows that we are going to write a template. Here T is a variable name for generic data type. We can use any other name but generally T is used (T evolves from template). A data type will replace this T whenever template is used in a program. Then we declare member functions of the class. To begin with, there is a constructor of the class with the same name as that of the class i.e. `Stack`. It is followed by the `empty()` function and then function `push` that is declared as follows

```
int push(T &) ;
```

We have been using the data (element) type `int` or some other data type in the `push()` function. Now there is T written as the data type in the `push()` function. This means that the function takes an argument of type T , here T is a generic data type and we will use a proper data type while calling this function in our program. This data type will replace T . There are also `pop` and `peek` functions that take no arguments but return the value which is of type T . The `peek` function is similar to the `top` function that returns (shows) the element from the top but does not remove it from the stack.

In the private section of the class, there are two variables. The variable `top` is used to point to the top of the stack. The pointer `nodes` is used to point to nodes of type T . We allocate dynamic memory using this `nodes` pointer.

In the definition of this whole class, T is a substitution parameter. While using the stack class in the program, we will replace this T with a proper data type.

Implementation

Now we will see how to implement this stack class in our program. Following is the code of the program. We save this code in the file named **Stack.cpp**.

```
#include <iostream.h>
#include <stdlib.h>
#include "Stack.h"
#define MAXSTACKSIZE 50

template <class T>
Stack<T>::Stack()
{
    top = -1;
    nodes = new T[MAXSTACKSIZE];
}

template <class T>
Stack<T>::~~Stack()
{
    delete nodes;
}

template <class T>
int Stack<T>::empty(void)
{
    if( top < 0 ) return 1;
```

```

        return 0;
    }
    template <class T>
    int Stack<T>::push(T& x)
    {
        if( top < MAXSTACKSIZE )
        {
            nodes[++top] = x;
            return 1;
        }
        cout << "stack overflow in push.\n";
        return 0;
    }
    template <class T>
    T Stack<T>::pop(void)
    {
        T x;
        if( !empty() )
        {
            x = nodes[top--];
            return x;
        }
        cout << "stack underflow in pop.\n";
        return x;
    }
}

```

In this code, we include different files in which one is Stack.h , written earlier to declare the template class *Stack*. We have defined a constant size of the stack to 50 by writing the line

```
#define MAXSTACKSIZE 50
```

Next is the definition of the constructor. Here before the signature of the constructor function, we have written

```

template <class T>
Stack<T> :: Stack();

```

This means that we are writing a template function that uses *T* wherever a data type is written. As we have declared the Stack class as a template class, <T> will be written with the class name before the access specifier (i.e. ::) while defining a method of the class. Then there is the implementation of the constructor in which we assign the value -1 to *top* and allocate memory by using the *new* operator for stack of size MAXSTACKSIZE of type *T* and put its starting address in the pointer *nodes*. It is pertinent to note that we create an array of type *T* (i.e. a generic type).

Similarly we define the destructor ~Stack, which frees the memory by deleting the nodes. Then there are the different methods of the stack i.e. *empty()*, *push()*, and *pop()*. We define all the methods in the same way as done in case of the constructor. It means that while writing *template <class T>* at the start, we use *T* wherever a data type can be used. The function *empty()* returns a Boolean parameter. It returns 1 that

means TRUE if the stack is empty i.e. if top is less than 0. Otherwise it returns zero i.e. FALSE if the stack is not empty.

We define the *push* function as under

```
int Stack<T>::push(T& x)
{
    if( top < MAXSTACKSIZE )
    {
        nodes[++top] = x;
        return 1;
    }
    cout << "stack overflow in push.\n";
    return 0;
}
```

This function takes an argument *x* by reference. It checks whether there is space in the stack by checking the value of the *top*. If *top* is less than the MAXSTACKSIZE, it means there is space available in the stack. Then it puts the element *x* on the stack and returns 1, indicating that push operation has succeeded. Otherwise, it displays a message of stack overflow and returns zero which indicates that the element was not put on the stack.

Next comes the *pop* method. Here we see that the value returned by this method is of type *T*. In the body of the function, we define a local variable of type *T* and check if the stack is empty. If it is not empty, we pop the value from the top in variable *x* and return it.

Now let's see the use of this template stack. Here we write the *main* program in a separate file including the *stack.cpp* (we have written before shortly) to use the stack. Following is the program written in the file **main.cpp**. This program demonstrates the implementation of the stack.

```
#include "Stack.cpp"
int main(int argc, char *argv[])
{
    Stack<int> intstack;
    Stack<char> charstack;
    int x=10, y=20;
    char c='C', d='D';

    intstack.push(x); intstack.push(y);
    cout << "intstack: " << intstack.pop() << ", " << intstack.pop() << "\n";
    charstack.push(c); charstack.push(d);
    cout << "charstack: " << charstack.pop() << ", " << charstack.pop() << "\n";
}
```

In the above code, consider the line

```
Stack <int> intstack ;
```

This line means that while creating an object *intstack* of *Stack*, the generic data type *T* should be replaced by the type *int*. In other words, it will be a stack for integers. The compiler will replace *T* with *int* wherever it exists in the code, providing a version of code with data type *int*. The compiler does this automatically.

Similarly the next line

```
Stack <char> charstack ;
```

creates an object of Stack that has name *charstack* and replaces the type *T* with *char*. It shows that it will be a stack of characters. Here *T* is replaced with *char* and a version of code is provided by the compiler, used for the *char* data type. Thus we create two objects of two types (i.e. *int* and *char*) of Stack by using the same code of the template class.

To demonstrate the implementation of Stack, we declare two variables of type *int* and two of type *char*. Then we push and pop these variables on and from the proper stack. We push the *int* values on the *intstack* (which we have created for *int* data type). The values of type other than *int* on the stack *intstack* can not be pushed as we have created it to store the *int* data type. And then we pop these values from the stack and show on the screen. Similarly we push the *char* values on the *charstack* (a stack to store *char* values) before displaying these values on the screen with the help of the *pop* method to get these values from the stack.

Now we have the three files of our code i.e. *stack.h*, *stack.cpp* and *main.cpp*. Having these files in the same directory, we compile the main file (*main.cpp*) and execute it. Following is the output of the above program.

```
intstack: 10, 20
charstack: C, D
```

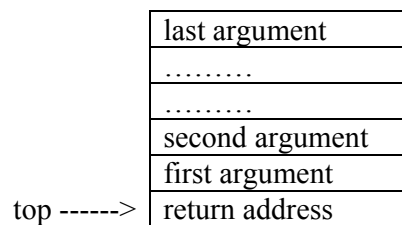
In the above example, we create two objects of Stack to use for two different data types. The compiler automatically provides us two versions of the template code, one for *int* and the other for *char* using the same code, written as template. So it is only due to the use of the template utility, provided by the C++ language only. No other language including C provides this utility of templates. If we write this program in C, the code of the functions of Stack has to be written repeatedly for each data type. Similarly, we will have to write different versions for using different data types. But in templates, a programmer writes the code once and the compiler automatically produces the version of the code with the needed data type. We have implemented the stack by using array. It can also be implemented with the linked list. The use of array or linked list does not matter here. The implementation of the stack remains the same. The templates are so important that C++ provides a library in which a large number of common use functions are provided as templates. This library is a part of the official standard of C++. It is called STL i.e. Standard Template Library. As a library, it is a tested code base. We can use these templates and implement different concepts for our own data types. STL is an important code, pre-developed for us. It is available as a library. Different data structures like stack, queue etc is also there in STL. We can write programs by using them. But here in this course, our goal is to know what the data structures are, what is functioning and how can they be written? So we are writing and discussing the stack templates. You can use data structures from STL in the programming courses or in the professional life. You need not to write the stack or queue from the scratch you can simply use them in your programs from STL.

Function Call Stack

Let's talk about another example of the stack. We know the functionality of the

function calls. Whenever a programmer calls a function, he or she passes some arguments or parameters to the function. The function does work on these arguments and returns a value to the calling function or program. This value is known as the return value of the function. We declare some variables inside the function which are local variables of the function. These variables are demolished when the execution of the function ends. If there are variables in the function that need to be preserved, we have to take care of them. For this purpose, we use global variables or return a pointer to that variable. Now let's see how a stack is used in function calls.

We are using devC++ compiler that actually uses GCC (glue compiler), a public domain compiler. Whenever we call a function, the compiler makes a stack that it uses to fulfill this function call. The compiler puts the entries on the stack in the way that first of all i.e. on the top (i.e. first entry in the stack) is the return address of the function where the control will go back after executing the function. After it, the next entries on the stack are the arguments of the function. The compiler pushes the last argument of the call list on the stack. Thus the last argument of the call list goes to the bottom of the stack after the return address. This is followed by the second last argument of the call list to be pushed on the stack. In this way, the first argument of the call list becomes the first element on the stack. This is shown in the following figure.



In the calling function, after the execution of the function called, the program continues its execution from the next line after the function call. The control comes back here because when the execution of the function ends the compiler pops the address from the stack which it has pushed when the function call was made. Thus the control goes at that point in the program and the execution continues in the calling function or program.

Consider the following code of a function that takes two integer arguments *a*, *b* and returns the average of these numbers.

```
int i_avg (int a, int b)
{
    return (a + b) / 2;
}
```

To understand the use of stack, look at the assembly language code of the above function that is written as under.

```
globl _i_avg
_i_avg:
    movl 4(%esp), %eax
    addl 8(%esp), %eax      # Add the args
```

```
sarl $1, %eax      # Divide by 2
ret                # Return value is in %eax
```

The first statement is *globl _i_avg* which shows that it's a global function that can be called by other functions or programs. After it, there is a label, written as *_i_avg*:

The next statement is *movl 4(%esp), %eax*. Here in this statement, there is the use of stack. Here *esp* is a register in assembly language that is now a stack pointer for us (i.e. top). The *movl* (move long) takes offset 4 from top (4 is number of bytes, we use 4 bytes as in C++ an integer is of 4 bytes.) that means after 4 bytes from the top in the stack it gets the value and put it in the *eax* register. We know that the compiler pushes the arguments of the function in reverse order on the stack. And pushes return address at the end. Thus the order of stack will be that on the top will be the return address and immediately after it will be the first argument. Here in the assembly code generated by the compiler, the compiler pops first argument from offset 4 and puts it in *eax* register. The next statement is

```
addl 8(%esp), %eax
```

The *addl* takes offset 8 from the stack pointer that is second argument and adds it to *eax*. Thus in the previous two statements, the compiler got the first argument i.e. *a* from the stack and the second argument *b* from the stack, added them before putting the result in *eax*. The next statement

```
sarl $1, %eax
```

is the division statement of assembly language. This statement divides the value in *eax* by 2 and thus *eax* has the resultant value. The last statement i.e. *ret*, returns the value on the top of the stack to the caller function.

So we have seen the use of stack in the execution of a function and how the arguments are passed to the function, how the functions return its return value and finally how the control goes back to the caller function. All this process is executed by using a stack. All the things about the functionality of the function calls are necessary to understand as these will be needed while going to write our own compilers. We will read this in the compilers course. The whole process we have discussed about the use of stack in function calling is known as run time environment.

Different data structures are also used in run time environment of the computer. We know that an executable program while in run, is loaded in the memory and becomes a process. This process is given a block of memory which it uses during its execution. Even the operating system, in which we are working, itself takes memory. Suppose we are running many programs simultaneously, which for example include browser, MS Word, Excel and dev-C++. We can also run programs written by us. Every program which we run takes a block of memory and becomes a process. The following figure shows a part of memory in which different programs occupy a block of memory for their execution.

Process 1 (browser)
Process 3 (Word)
Process 4 (Excel)
Process 2 (Dev-C++)
Windows Os

We can also see the details of all the programs running at a specific time. If we press the key combination *Ctrl-Alt-Del*, there appears a window *task manager* on the screen. Following is the figure of the task manager. In the figure, there are many columns i.e. PID (process ID), CPU, CPU time, Memory usage, page faults, I/O Reads, I/O Writes, I/O Read Bytes, I/O Write Bytes. These all things we will read in the course of Operating Systems in detail.

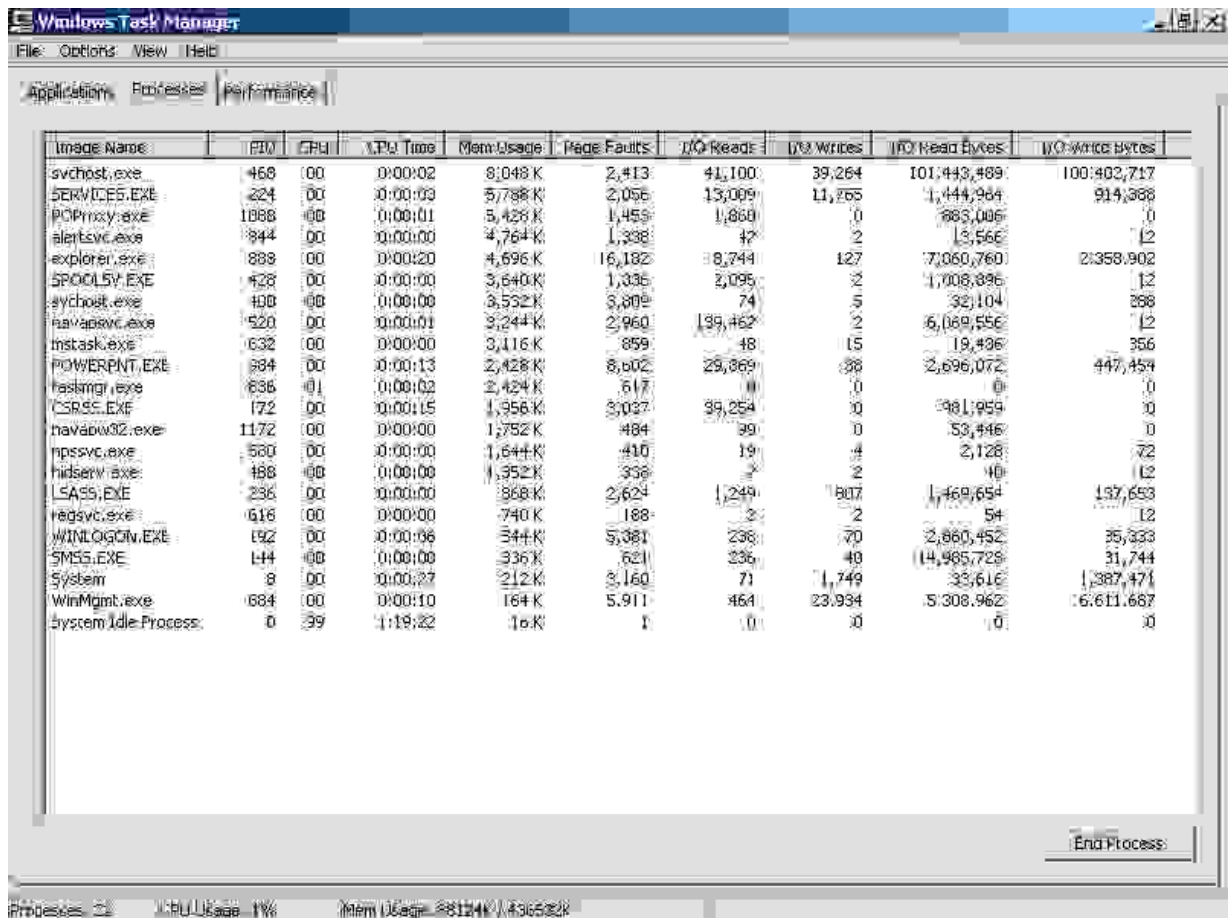
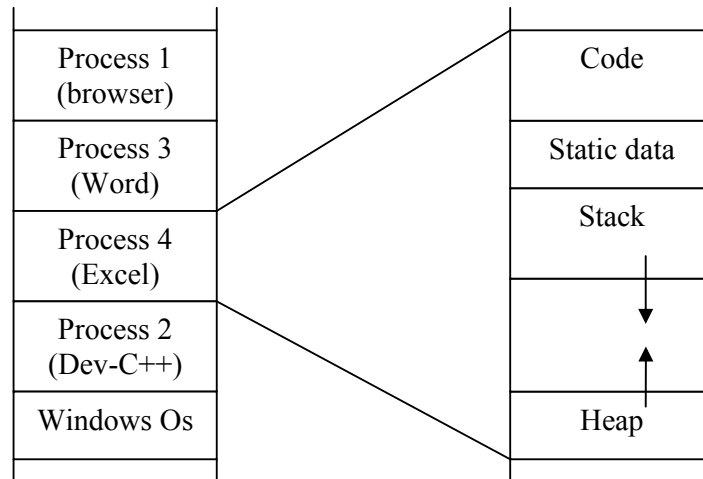


Image Name	PID	CPU	Private Bytes	Working Set	Page Faults	I/O Reads	I/O Writes	I/O Read Bytes	I/O Write Bytes
svchost.exe	468	00	0:00:02	8,048 K	2,413	41,100	39,264	101,443,489	100,403,717
SMSS.EXE	224	00	0:00:00	5,788 K	2,056	13,009	11,265	1,444,964	914,388
POPMGR.exe	1888	00	0:00:01	5,428 K	1,459	1,860	0	883,006	0
alertrvc.exe	844	00	0:00:00	4,764 K	1,338	47	2	13,566	12
explorer.exe	888	00	0:00:20	4,696 K	16,182	8,744	127	7,060,760	2,358,902
SPOOLSV.EXE	428	00	0:00:00	3,640 K	1,335	2,095	2	1,008,896	12
svchost.exe	400	00	0:00:00	3,532 K	3,809	74	5	32,104	888
navapowr.exe	520	00	0:00:01	3,244 K	2,960	139,462	2	6,069,556	12
nttask.exe	632	00	0:00:00	3,116 K	859	48	15	19,436	356
POWERPNT.EXE	884	00	0:00:13	2,428 K	8,602	29,869	88	2,696,072	447,454
taskmgr.exe	636	01	0:00:02	2,424 K	617	0	0	0	0
CSRSS.EXE	172	00	0:00:15	1,956 K	3,037	39,254	0	981,959	0
navapowr32.exe	1172	00	0:00:00	1,752 K	484	99	0	53,446	0
npssvc.exe	580	00	0:00:00	1,644 K	410	19	4	2,128	72
hidserw.exe	488	00	0:00:00	1,352 K	338	2	2	40	12
LSASS.EXE	236	00	0:00:00	868 K	2,624	1,249	807	1,469,654	137,653
regsvr.exe	616	00	0:00:00	740 K	188	2	2	54	12
WINLOGON.EXE	192	00	0:00:06	344 K	5,381	238	70	2,860,452	35,333
SMSS.EXE	144	00	0:00:00	336 K	621	236	40	14,985,728	31,744
System	8	00	0:00:27	212 K	3,160	71	1,749	33,616	1,387,471
WinMgmt.exe	684	00	0:00:10	164 K	5,911	464	23,934	5,308,962	6,611,687
System Idle Process	0	99	1:19:22	16 K	1	0	0	0	0

Here the thing of our interest is the first, second and fifth column. These columns are

Image Name, PID and Mem Usage (i.e. memory usage). Now look at the row where *explorer.exe* is written in the first column. The process ID (PID) of it is 888 and memory usage is 4696K. This means that the process size of *explorer.exe* in the memory is 4696 Kilo Bytes (KB). All the processes in the first column of the task manager are present in the memory of the computer at that time. The column Image name has the names of the processes being executed. These have extension *.exe* but there may be other executable programs that have extension other than *.exe*.

The following figure shows the internal memory organization of a process.



This shows that the first part of the memory of the process is for the code. This is the code generated by the compiler of C++, JAVA or VB etc with respect to the language in which the actual code was written. Then the static data of the program occupies the memory. This holds the global variables and different variables of objects. Then in the memory of the process, there is stack. After it there is heap. The stack and heap are used in function calls. We have discussed the use of stack in function calls. When we allocate memory dynamically in our programs, it is allocated from the heap. The use of heap is a topic related to some programming course or to the operating system course.

Data Structures

Lecture No. 09

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 3
3.3.3, 3.4.1, 3.4.2

Summary

- Memory Organization
- Stack Layout During a Function Call
- Queues
- Queue Operations
- Implementing Queue
- Queue using Array
- Use of Queues

Memory Organization

By the end of last lecture, we discussed the uses of stack to develop a process from an executable file and then in function calls. When you run an executable, the operating system makes a process inside memory and constructs the followings for that purpose.

- A code section that contains the binary version of the actual code of the program written in some language like C/C++
- A section for static data including global variables
- A stack and
- Finally, a heap

Stack is used in function calling while heap area is utilized at the time of memory allocation in dynamic manner.

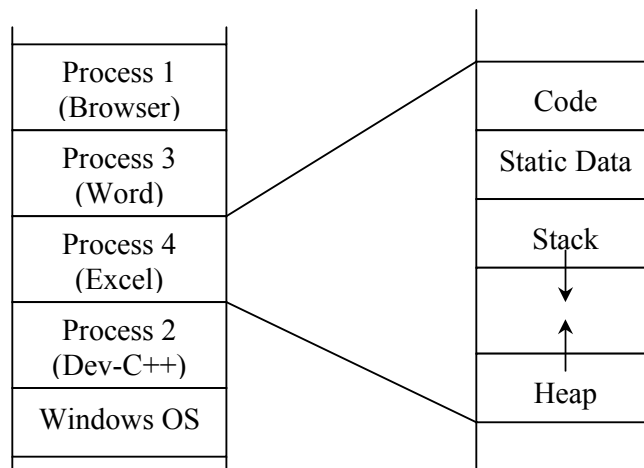


Fig 1. Memory Organization

Stack Layout during a Function Call

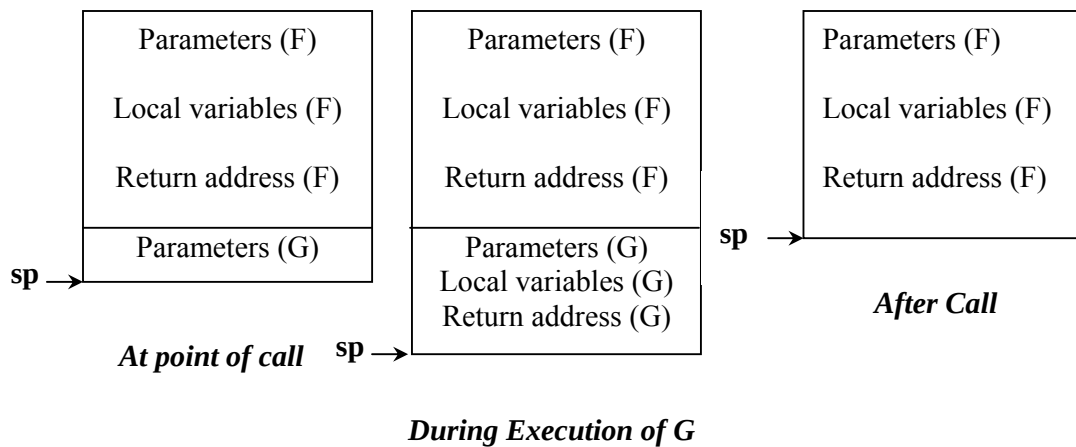


Fig 2: Stack Layout; When function F calls function G

The above diagrams depict the layout of the stack when a function *F* calls a function *G*. Here **sp** stands for stack pointer. At the very left, you will find the layout of the stack just before function *F* calls function *G*. The parameters passed to function *F* are firstly inserted inside the stack. These are followed by the local variables of the function *F* and finally the memory address to return back after the function *F* finishes. Just before function is made to the function *G*, the parameters being passed to the function *G*, are inserted into the stack.

In the next diagram, there is layout of the stack on the right side after the call to the function *G*. Clearly, the local variables of the function *G* are inserted into the stack after its parameters and the return address. If there are no local variables for a function, the return address is inserted (pushed) on to the stack.

The layout of the stack, when the function *G* finishes execution is shown on the right. You can see that the local variables of function *G* are no more in the stack. They have been removed permanently along with the parameters passed to the function *G*. Now, it is clear that when a function call is made, all local variables of the called function and the parameters passed to it, are pushed on to the stack and are destroyed, soon after the the completion of the called function's execution.

In C/C++ language, the variables declared as *static* are not pushed on the stack. Rather, these are stored in another separate section allocated for *static* data of a program. This section for *global* or *static* data can be seen in the fig 1 of this lecture. It is not destroyed till the end of the process's execution. If a variable, say *x* is declared as *static* inside function *G*, *x* will be stored in the *static* data section in the process's memory. Whereas, its value is preserved across *G* function calls. The visibility of *x* is restricted to the function *G* only. But a *static* variable declared as a *class* data is available to *all member functions* of the *class* and a *static* variable declared at *global* scope (outside of any class or function body) is available to all functions of the program.

Now, let's move on to another data structure called *queue*.

Queues

A queue is a linear data structure into which items can only be inserted at one end and removed from the other. In contrast to the stack, which is a LIFO (Last In First Out) structure, a queue is a FIFO (First In First Out) structure.

The usage of queue in daily life is pretty common. For example, we queue up while depositing a utility bill or purchasing a ticket. The objective of that queue is to serve persons in their arrival order; the first coming person is served first. The person, who comes first, stands at the start followed by the person coming after him and so on. At the serving side, the person who has joined the queue first is served first. If the requirement is to serve the people in some sort of priority order, there is a separate data structure that supports priorities. The normal queue data structure, presently under discussion, only supports FIFO behavior.

Now, let's see what are the operations supported by the queue.

Queue Operations

The queue data structure supports the following operations:

Operation	Description
enqueue(X)	Place X at the <i>rear</i> of the queue.
dequeue()	Remove the <i>front</i> element and return it.
front()	Return <i>front</i> element without removing it.
isEmpty()	Return TRUE if queue is empty, FALSE otherwise

Implementing Queue

There are certain points related to the implementation of the queue. Suppose we are implementing queue with the help of the linked -list structure. Following are the key points associated with the linked list implementations:

- Insert works in constant time for either end of a linked list.
- Remove works in constant time only.
- Seems best that head of the linked list be the front of the queue so that all removes will be from the front.
- Inserts will be at the end of the list.

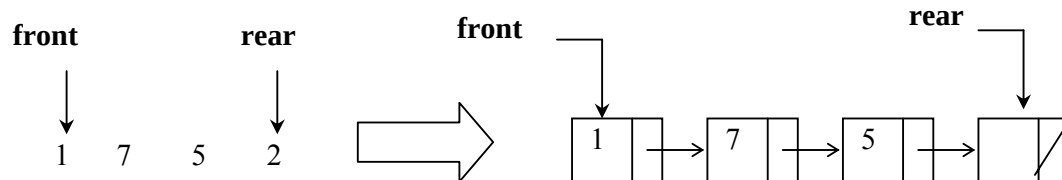


Fig 3. Queue implementation using linked list

The above figure shows *queue* elements on the left with two pointers *front* and *rear*. This is an abstract view of the queue, independent of its implementation method of array or linked list. On the right side is the same *queue* ,using linked list and pointers of *front* and *rear*. When *dequeue()* function is called once, the *front* element 1 is removed. The picture of the *queue* showing one element removal is also depicted below. Note that *front* pointer has been moved to the next element 7 in the list after removing the *front* element 1.

After dequeue() is called once

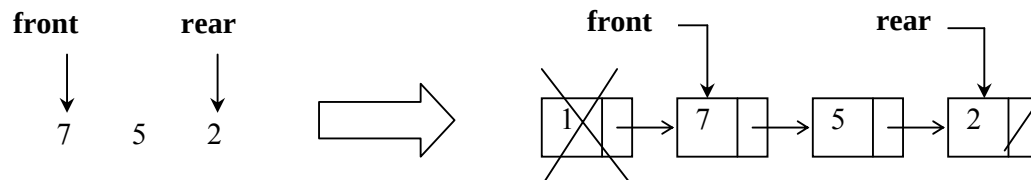


Fig 4. Removal of one element from queue using dequeue()

Now at this stage of the *queue*, we will call *enqueue* (9) to insert an element 9 in it. . The following figure shows that the new element is inserted at the *rear* end and *rear* pointer starts pointing this new node with element 9.

At this point of time, the code of these functions of *dequeue()* and *enqueue()* should not be an issue.

Queue after enqueue(9) call

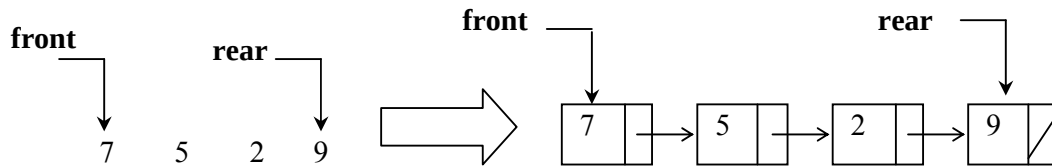


Fig 5. Insertion of one element using enqueue(9)

Note that in this *queue* data structure, the new elements are inserted at *rear* end and removed from the *front*. This is in contrast to *stack* structure where the elements are inserted and removed from the same end.

Let's see the code for queue operations:

```

/* Remove element from the front */
1. int dequeue()
2. {
3.   int x = front->get();
4.   Node* p = front;
5.   front = front->getNext();
6.   delete p;
7.   return x;
8. }
/* Insert an element in the rear */
9. void enqueue(int x)
10. {
11.   Node* newNode = new Node();
12.   newNode->set(x);
13.   newNode->setNext(NULL);
14.   rear->setNext(newNode);
15.   rear = newNode;
16. }

```

In `dequeue()` operation, at line 3, the front element is retrieved from the queue and assigned to the *int* variable *x*.

In line 4, the *front* pointer is saved in *Node* pointer variable *p*.

In line 5, the *front* pointer is moved forward by retrieving the address of the next node by using `front->getNext()` and assigning it to the *front* pointer.

In line 6, the node pointed to by the *front* pointer is deleted by using `delete front` statement.

At the end of `dequeue()` implementation, the value of deleted node that was saved in the *int* variable *x*, is returned back.

The `enqueue(int)` is used to add an element in the *queue*. It inserts the element in the *rear* of the *queue*. At line 11, a new *Node* object is created using the `new Node()` statement and the returned starting address of the created object is assigned to the *newNode* pointer variable.

In line 12, the value of the passed in parameter *x*, is set in the newly created node

object using the *set()* method.

In line 13, the *next* pointer in the newly created node is set to *NULL*.

In line 14, the newly created node is set as the next node of the node currently pointed by the *rear* pointer.

In line 15, the *rear* pointer is set to point to the newly created node.

The code of two smaller functions is as under:

```
/* To retrieve the front element */
int front()
{
    return front->get();
}

/* To check if the queue is empty */
int isEmpty()
{
    return ( front == NULL );
}
```

The *front()* method is used to retrieve the front element. This is the oldest element inserted in the queue. It uses the *get()* method of the *Node* class.

The *isEmpty()* method is used to check whether the queue is empty or not. It checks the address inside the *front* pointer, if it is *NULL*. It will return true indicating that the queue is empty or vice versa.

While studying stack data structure, we implemented it by using both array and linked list. For queue, until now we have been discussing about implementing queue using linked list. Now, let's discuss implementing queue with the help of an array.

Queue using Array

A programmer keeps few important considerations into view account before implementing a queue with the help of an array:

If we use an array to hold the queue elements, both insertions and removal at the front (start) of the array are expensive. This is due to the fact that we may have to shift up to "n" elements.

For the stack, we needed only one end but for a queue, both are required. To get around this, we will not shift upon removal of an element.

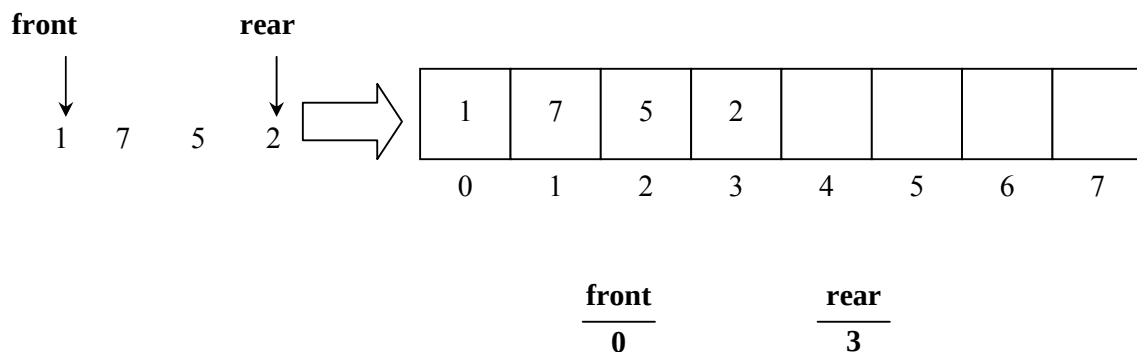


Fig 6. Queue implemented using an array

In the above figure, queue implementation using array is shown. As the array size is 8, therefore, the index of the array will be from 0 to 7. The number of elements inside array are 1, 7, 5 and 2, placed at start of the array. The *front* and *rear* in this implementation are not pointers but just indexes of arrays. *front* contains the starting index i.e. 0 while *rear* comprises 3.

Let's see, how the *enqueue()* works:

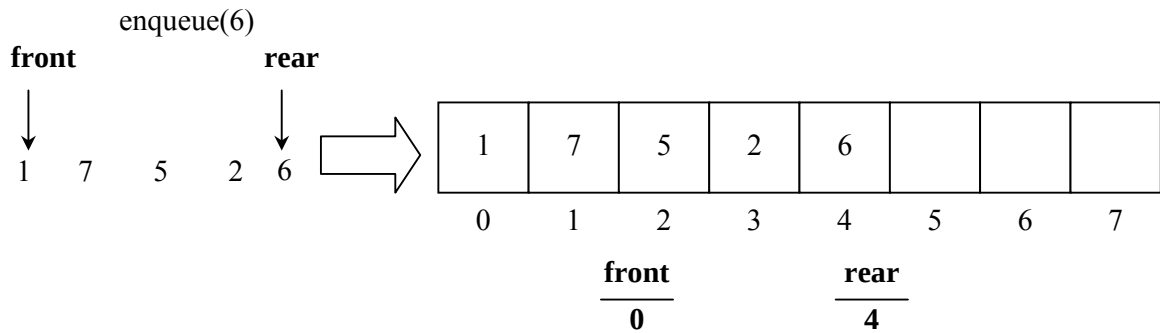


Fig 7. Insertion of one element 6

As shown in the above diagram, an element i.e. 6 has been inserted in the *queue*. Now, the *rear* index is containing 4 while the *front* has the same 0 index. Let's see the figure of the array when another element 8 is inserted in the queue.

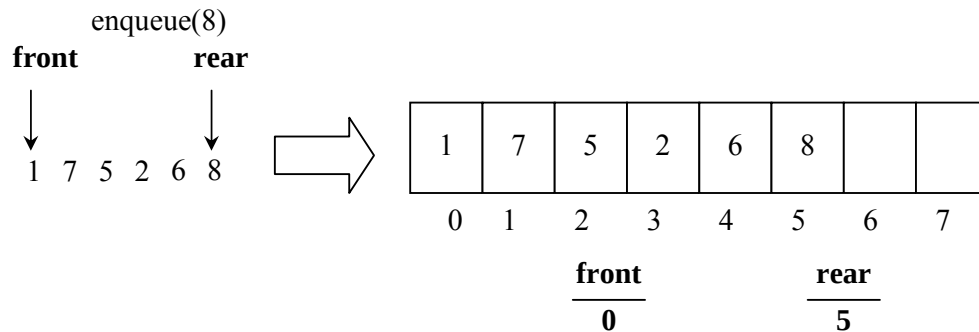


Fig 8. Insertion of another element 8

When an element is removed from the queue. It is removed from the *front* index.

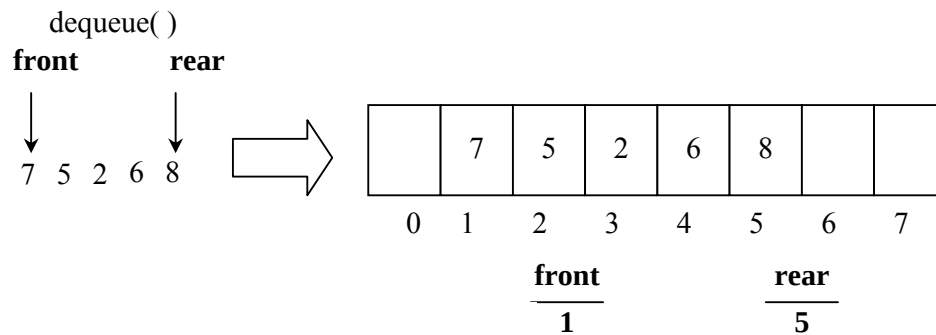


Fig 9. Removal of an element from front

After another call of *dequeue()* function:

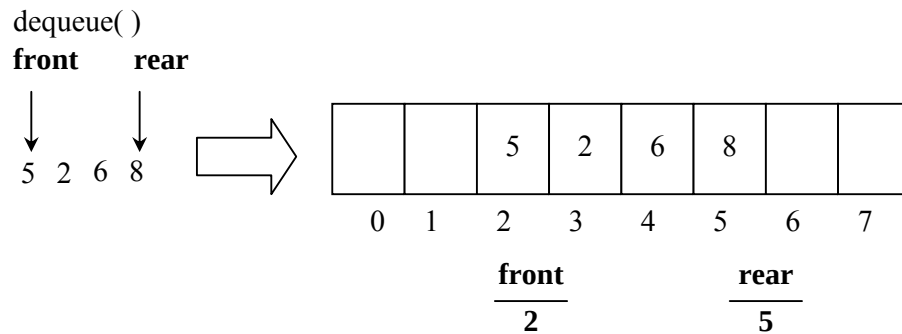


Fig 10. Removal of another element from front

With the removal of element from the queue, we are not shifting the array elements. The shifting of elements might be an expensive exercise to perform and the cost is increased with the increase in number of elements in the array. Therefore, we will leave them as it is.

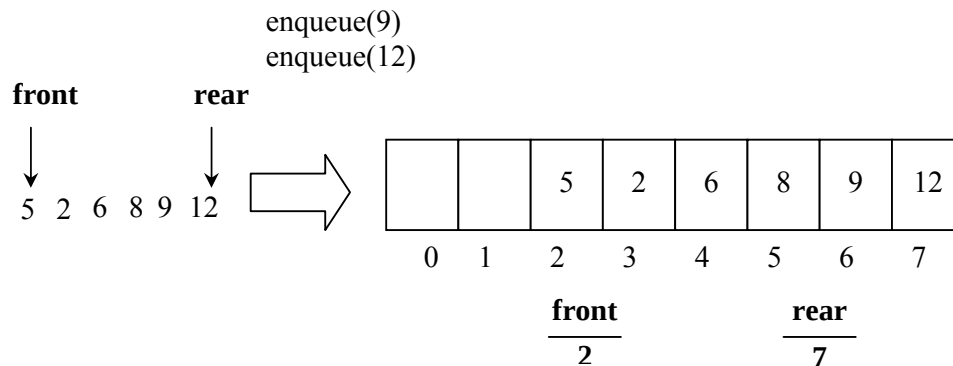


Fig 11. Insertion of elements in the queue

After insertion of two elements in the queue, the array that was used to implement it, has reached its limit as the last location of the array is in use now. We know that there is some problem with the array after it attained the size limit. We observed the similar problem while implementing a stack with the help of an array.

We can also see that two locations at the start of the array are vacant. Therefore, we should consider how to use those locations appropriately in to insert more elements in the array.

Although, we have insert and removal operations running in constantly, yet we created a new problem that we cannot insert new elements even though there are two places available at the start of the array. The solution to this problem lies in allowing the queue to *wrap around*.

How can we *wrap around*? We can use circular array to implement the queue. We know how to make a linked list circular using pointers. Now we will see how can we make a circular array.

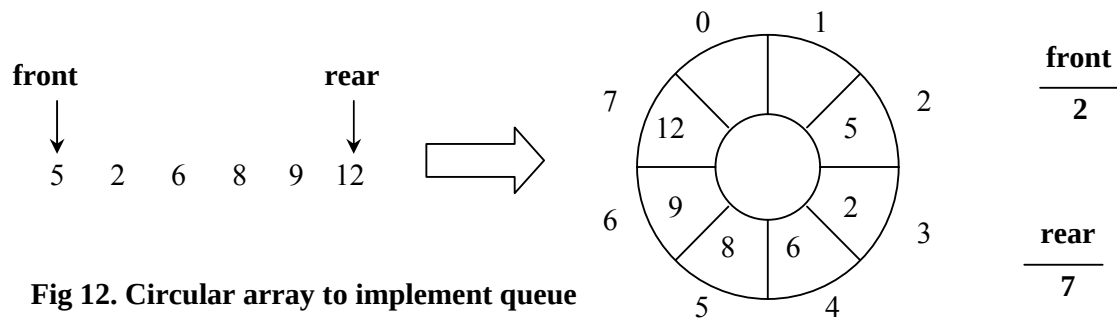


Fig 12. Circular array to implement queue

The number of locations in the above circular array are also eight, starting from index 0 to index 7. The index numbers are written outside the circle incremented in the clock-wise direction. To insert an element 21 in the array, we insert this element in the location, which is next to index 7.

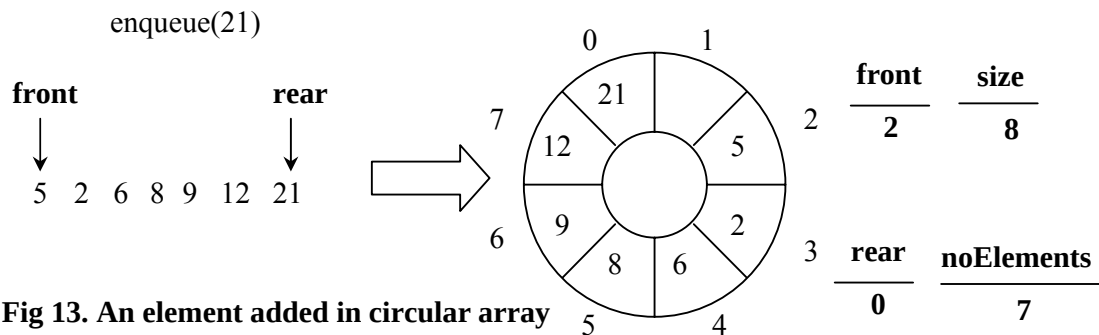


Fig 13. An element added in circular array

Now, we will have to maintain four variables. *front* has the same index 2 while the, *size* is 8. '*rear*' has moved to index 0 and *noElements* is 7. Now, we can see that *rear* index has decreased instead of increasing. It has moved from index 7 to 0. *front* is containing index 2 i.e. higher than the index in *rear*. Let's see, how do we implement the *enqueue()* method.

```
void enqueue( int x)
{
1.  rear = (rear + 1) % size;
2.  array[rear] = x;
3.  noElements = noElements + 1;
}
```

In line 1 of the code, 1 is added in *rear* and the mod operator (that results in remainder of the two operands) is applied with *size* variable. This expression on the right of assignment in line 1 can result from 0 to 7 as *size* is containing value 8. This operator ensures that value of this expression will always be from 0 to 7 and increase or decrease from this. This resultant is assigned to the *rear* variable.

In line 2, the *x* (the value passed to *enqueue()* method to insert in the queue) is inserted in the array at the *rear* index position. Therefore, in the above case, the new element 21 is inserted at index 0 in the array.

In line 3, *noElements* is added to accumulate another element in the queue.

Let's add another element in the queue.

enqueue(7)

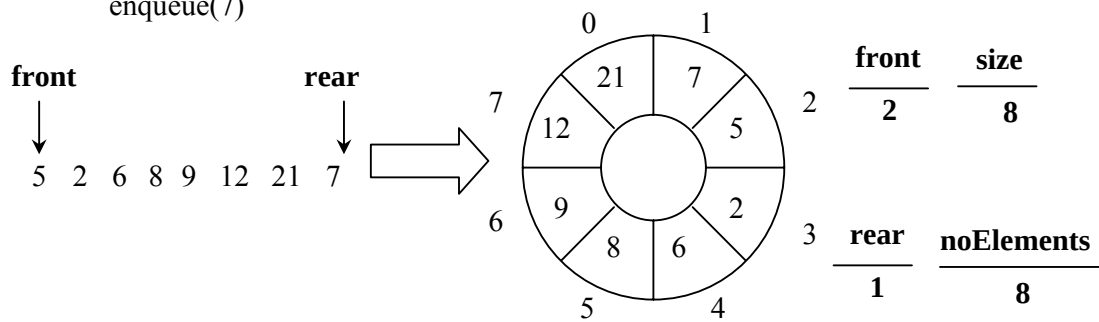


Fig 14. Another element added in circular array

Now, the queue, rather the array has become full. It is important to understand, that queue does not have such characteristic to become full. Only its implementation array has become full. To resolve this problem, we can use linked list to implement a queue. For the moment, while working with array, we will write the method *isFull()*, to determine the fullness of the array.

```
int isFull()
{
    return noElements == size;
}

int isEmpty()
{
    return noElements == 0;
}
```

isFull() returns true if the number of elements (*noElements*) in the array is equal to the *size* of the array. Otherwise, it returns false. It is the responsibility of the caller of the queue structure to call *isFull()* function to confirm that there is some space left in the queue to *enqueue()* more elements.

Similarly *isEmpty()* looks at the number of elements (*noElements*) in the queue. If there is no element, it returns true or vice versa..

Let's see the *dequeue()* method.

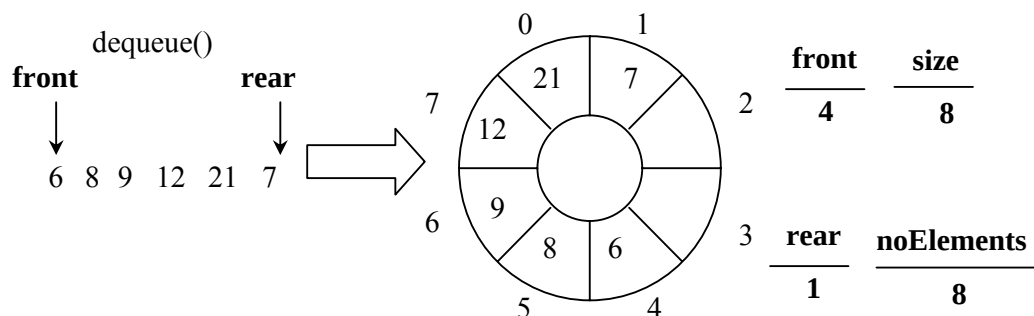


Fig 15. Element removed from the circular array


```
int dequeue()
{
    int x = array[front];
    front = (front + 1) % size;
    noElements = noElements - 1;
    return x;
}
```

In the first line, we take out an element from the array at *front* index position and store it in a variable *x*. In the second line, *front* is incremented by 1 but as the array is circular, the index is looped from 0 to 7. That is why the *mod (%)* is being used. In the third line, number of elements (*noElements*) is reduced by 1 and finally the saved array element is returned.

Use of Queues

We saw the uses of stack structure in *infix*, *prefix* and *postfix* expressions. Let's see the usage of queue now.

Out of the numerous uses of the queues, one of the most useful is *simulation*. A simulation program attempts to model a real-world phenomenon. Many popular video games are simulations, e.g., SimCity, Flight Simulator etc. Each object and action in the simulation has a counterpart in the real world. Computer simulation is very powerful tool and it is used in different high tech industries, especially in engineering projects. For example, it is used in aero plane manufacturing. Actually Computer Simulation is full-fledged subject of Computer Science and contains very complex Mathematics, sometimes. For example, simulation of computer networks, traffic networks etc.

If the simulation is accurate, the result of the program should mirror the results of the real-world event. Thus it is possible to understand what occurs in the real-world without actually observing its occurrence.

Let us look at an example. Suppose there is a bank with four tellers.

A customer enters the bank at a specific time (t_1) desiring to conduct a transaction.

Any one of the four tellers can attend to the customer. The transaction (withdraws, deposit) will take a certain period of time (t_2). If a teller is free, the teller can process the customer's transaction immediately and the customer leaves the bank at $t_1 + t_2$. It is possible that none of the four tellers is free in which case there is a line of customers at each teller. An arriving customer proceeds to the back of the shortest line and waits for his turn. The customer leaves the bank at t_2 time units after reaching the front of the line.

The time spent at the bank is t_2 plus time waiting in line.

So what we want to simulate is the working environment of the bank that there are specific number of queues of customers in the bank in front of the tellers. The tellers are serving customers one by one. A customer has to wait for a certain period of time before he gets served and by using simulation tool, we want to know the average waiting time of a bank customer. We will talk about this simulation in the next lecture and will do coding also in order to understand it well.

Data Structures

Lecture No. 10

Reading Material

Data Structures and Algorithm Analysis in C++
6

Chapter. 3,

3.4.3, 6.1

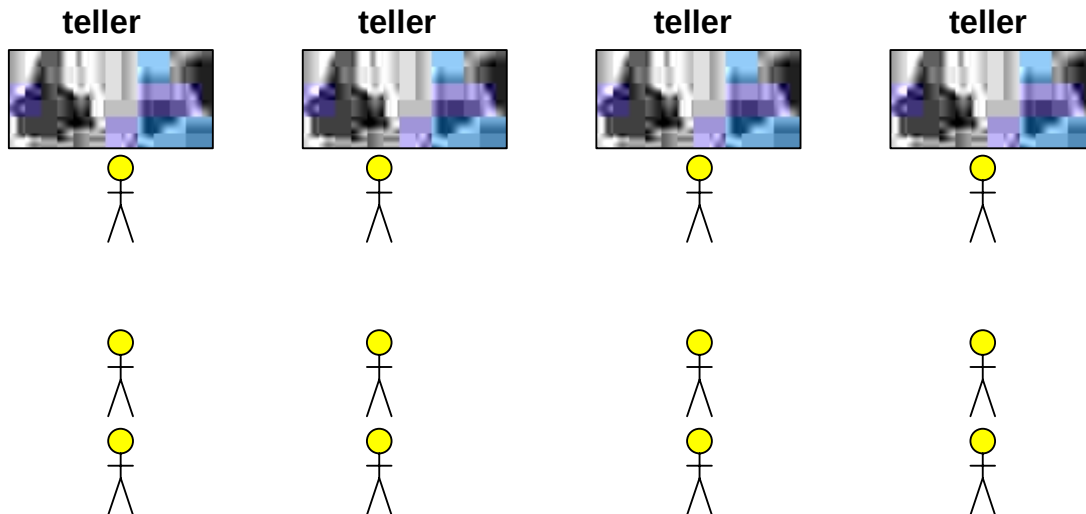
Summary

- 12) Queues
- 13) Simulation Models
- 14) Priority Queue
- 15) Code of the Bank simulation

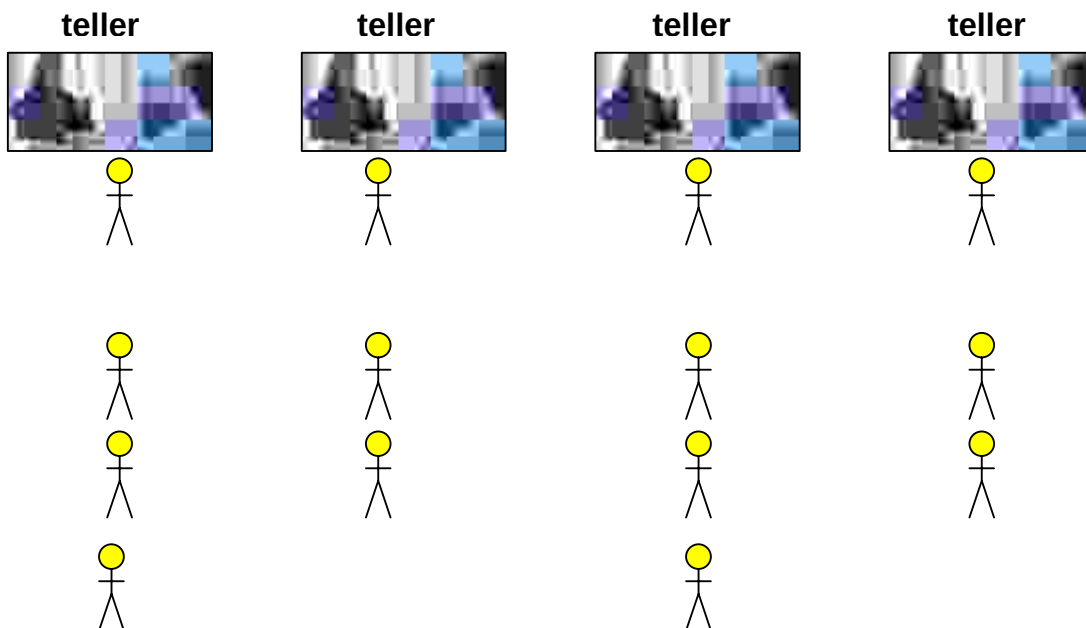
Queues

In the previous lecture, we discussed the queue data structure and demonstrated its implementation by using array and link list. We also witnessed the usefulness of queue as data structure in the simulation example. This concept can be further elaborated with a daily life example relating to banking sector. Suppose, customers want to deposit or withdraw money from a bank, having four cashiers or tellers. The teller helps you in depositing the money or withdrawing the money from the same window. This window is known as teller window. The customer needs some service from the bank like depositing the money, bill etc. This transaction needs some time that may be few minutes. A person enters the bank and goes to the teller who is free and requests him to do the job. After the completion of the transaction, the person goes out of the bank. Now we will discuss a scenario when there is a lot of rush of customers. The tellers are just four. Now the tellers are busy and the new customers will form a queue. In this example, we need a queue in front of each of the tellers. A new customer enters the bank and analyzes the four queues and wants to join the shortest queue. This person has to wait for the persons in front of him to be served. Another person may come behind him. In this simulation, we will restrict the person from changing the queue. A person comes into the bank at 10 O clock. His transaction time is 5 minutes. He has to wait for another fifteen minutes in the queue. After this, the teller serves him in 5 min. This person comes at 10 am and waits for fifteen minutes. As the transaction time is 5 minutes, so he will leave the bank at 1020. Now this is the situation of simulation and we have to write a program for this. We can go to some bank and analyze this situation and calculate the time. At the end of the day, we can calculate the average time for each of the customer. This time can be 30 minutes. Here we will simulate this situation with the help of a computer program. This is the real life example. Let's see the picture of simulations to understand what is happening in the bank.

In the picture below, we have four tellers and four queues, one for each of the tellers. Each teller is serving a customer. When the transaction of a customer is completed, he will leave the bank.



A person enters the bank. He sees that all the four tellers are busy and in each queue there are two persons waiting for their turn. This person chooses the queue no. 3. Another person enters the bank. He analyzed all the queues. The queue no 3 is the biggest and all other are having 2 persons in the queue. He chooses the queue no 1.



Now we have three persons waiting in queue no 1 and 3 and two persons waiting in queue no 2 and 4. The person in queue no.1 completes his transaction and leaves the bank. So the person in the front of the queue no. 1 goes to the teller and starts his transaction. Similarly the person at queue No. 3 finishes his transaction and leaves the premises. The person in front of queue number 3 goes to the teller.

Another person enters the bank and goes to the queue No. 1. This activity goes on. The queues become bigger and shorter. The persons coming in the bank have to wait. If the queues are shorter, people have to wait for less time. However, if the queues are

longer, people have to wait for more time. The transactions can also take much more time, keeping the people waiting. Suppose four persons come with big amount and their transaction takes too much time. These are all parameters which we can incorporate in our simulation making it real. For this, we have carry out more programming. With the introduction of these parameters in the simulation, it will be more close to the real life situation. Simulation, being a very powerful technique, can yield the results, very close to some real life phenomenon.

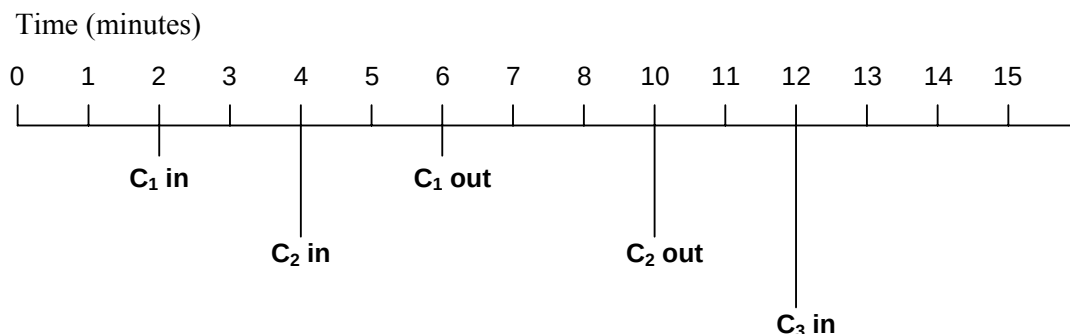
Simulation Models

Let's discuss little bit about the simulation models. Two common models of simulation are time-based simulation and event-based simulation. In time-based simulation, we maintain a timeline or a clock. The clock ticks and things happen when the time reaches the moment of an event.

Suppose we have a clock in the computer. The minute hand moves after every minute. We know the time of the customer's entry into the bank and are aware that his transaction takes 5 minutes. The clock is ticking and after 5 minutes, we will ask the customer to leave the bank. In the program, we will represent the person with some object. As the clock continues ticking, we will treat all the customers in this way. Note that when the customer goes to some teller, he will take 5 minutes for his transaction. During this time, the clock keeps on ticking. The program will do nothing during this time period. Although some other customer can enter the bank. In this model, the clock will be ticking during the transaction time and no other activity will take place during this time. If the program is in some loop, it will do nothing in that loop until the completion of the transaction time.

Now consider the bank example. All tellers are free. Customer C1 comes in 2 minutes after the opening of the bank. Suppose that bank opens at 9:00 am and the customer arrives at 9:02 am. His transaction (withdraw money) will require 4 minutes. Customer C2 arrives 4 minutes after the bank opens (9:04 am). He needs 6 minutes for transaction. Customer C3 arrives 12 minutes after the bank opens and needs 10 minutes for his transaction.

We have a time line and marks for every min.



C1 comes at 2 min, C2 enters at 4 min. As C1 needs 4 min for his transaction, so he leaves at 6 min. C2 requires 6 min for the processing so C2 leaves at 10 min. Then C3 enters at 12 min and so on. This way, the activity goes on. Therefore, we can write a

routine of the clock. We take a variable *clock* representing the clock. The clock will run for 24 hrs. Banks are not open for 24 hrs but we will run our loop for 24 hrs. The pseudo code is as under:

```
clock = 0;
while ( clock <= 24*60 ) { // one day
    read new customer;

    if customer.arrivaltime == clock
        insert into shortest queue;

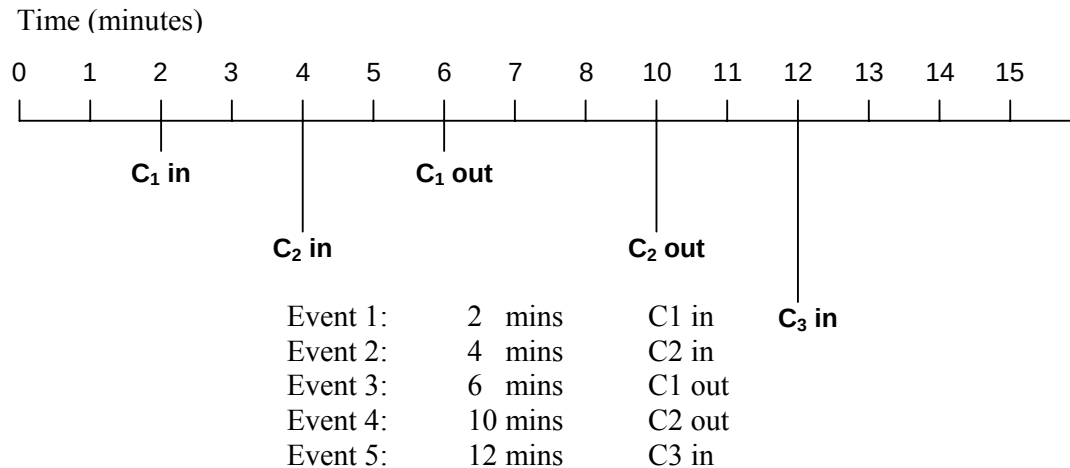
    check the customer at head of all four queues.

    if transaction is over
        remove from queue.

    clock = clock + 1;
}
```

The variable *clock* is initialized to zero. The while loop runs for 24 hrs. Then we read a new customer. This information may be coming from some file. The *if* statement is checking the arrival time of the customer. He comes 10 minutes after the opening of the bank. So when this time is equal to the clock, we insert the customer in the shortest queue. Then we will check the transaction time of all the four customers at each teller. If the transaction time of any customer ends, we will remove it from the queue and he will leave the bank. In the end, we increment the clock with one minute. As seen in the above statements, some activity takes place when the clock reaches at the event time (that is the time to enter the bank or leave the bank arrives). If the customer's arrival time has not come, the first *if* statement becomes false and we do nothing. Similarly if the transaction of customer is not finished, the second *if* statement becomes false. Then this while loop will simply add one min to the clock. This is the clock- based (time- based) simulation.

Let's discuss the other type of simulation i.e. the event-based simulation. Don't wait for the clock to tick until the next event. Compute the time of next event and maintain a list of events in increasing order of time. Remove an event from the list in a loop and process it. Let's see the time line again.



The customer C1 comes at 2 min and leaves at 6 min. Customer C2 comes at 4 min and leaves at 10 min and so on. We have written the events list in the above figure. Do not see the clock but see the events on time. Event 1 occurs at 2 min that is the customer C1 enters the bank 2 minutes after its opening. Event 2 is that C2 enters at 4 min. Event 3 is that the customer C1 leaves the bank at 6 min. Event 4 is that the C2 leaves the bank at 10 min and event 5 is that C3 enters the bank at 12 min. Here we have a list of events. How can we process them? We will make a queue of these events. Remove the event with the earliest time from the queue and process it. Insert the newly created events in the queue. A queue where the de-queue operation depends not on FIFO, is called a priority queue.

Priority Queue

As stated earlier, the queue is a FIFO (First in first out) structure. In daily life, you have also seen that it is not true that a person, who comes first, leaves first from the queue. Let's take the example of traffic. Traffic is stopped at the signal. The vehicles are in a queue. When the signal turns green, vehicles start moving. The vehicles which are at the front of the queue will cross the crossing first. Suppose an ambulance comes from behind. Here ambulance should be given priority. It will bypass the queue and cross the intersection. Sometimes, we have queues that are not FIFO i.e. the person who comes first may not leave first. We can develop such queues in which the condition for leaving the queue is not to enter first. There may be some priority. Here we will also see the events of future like the customer is coming at what time and leaving at what time. We will arrange all these events and insert them in a priority queue. We will develop the queue in such a way that we will get the event which is going to happen first of all in the future. This data structure is known as priority queue. In a sense, FIFO is a special case of priority queue in which priority is given to the time of arrival. That means the person who comes first has the higher priority while the one who comes later, has the low priority. You will see the priority queue being used at many places especially in the operating systems. In operating systems, we have queue of different processes. If some process comes with higher priority, it

will be processed first. Here we have seen a variation of queue. We will use the priority queue in the simulation. The events will be inserted in the queue and the event going to occur first in future, will be popped.

What are the requirements to develop this simulation? We need the C++ code for the simulation. There will be a need of the queue data structure and obviously, the priority queue. Information about the arrival of the customers will be placed in an input file. Each line of the file contains the items (arrival time, transaction duration).

Here are a few lines from the input file.

```
00 30 10 <- customer 1
00 35 05 <- customer 2
00 40 08
00 45 02
00 50 05
00 55 12
01 00 13
01 01 09
```

The first line shows the customer 1. “00 30 10” means Customer 1 arrives 30 minutes after the opening of the bank. He will need 10 minutes for his transaction. The last entry “01 01 09” means customer arrives one hour and one minute after the bank opened and his transaction will take 9 minutes and so on. The file contains similar information about the other customers. We will collect the events now. The first event to occur is the arrival of the first customer. This event is placed in the priority queue. Initially, the four teller queues are empty. The simulation proceeds as follows: when an arrival event is removed from the priority queue, a node representing the customer is placed on the shortest teller queue. Here we are trying to develop an algorithm while maintaining the events queue.

After the opening of the bank, the arrival of the first customer is the first event. When he enters the bank all the four tellers are free. Suppose he goes to the first teller and starts his transaction. After the conclusion of his transaction, he leaves the bank. With respect to events, we have only two events, one is at what time he enters the bank and other is at what time he leaves the bank. When other customers arrive, we have to maintain their events.

If the customer is the only one on a teller queue, an event for his departure is placed on the priority queue. At the same time, the next input line is read and an arrival event is placed in the priority queue. When a departure event is removed from the event priority queue, the customer node is removed from the teller queue. Here we are dealing with the events, not with the clock. When we come to know that a person is coming at say 9:20am, we make an event object and place it in the priority queue. Similarly if we know the time of leaving of the customer from the bank, we will make an event and insert it into the priority queue. When the next customer in the queue is served by the teller, a departure event is placed on the event priority queue. When the other customer arrives, we make an event object and insert it into the priority queue. Now the events are generated and inserted when the customer arrives. But the dequeue is not in the same fashion. When we dequeue, we will get the event which is

going to occur first.

When a customer leaves the bank, the total time is computed. The total time spent by the customer is the time spent in the queue waiting and the time taken for the transaction. This time is added to the total time spent by all customers. At the end of the simulation, this total time divided by the total customers served will be average time consumed by customers. Suppose that 300 customers were served, then we will divide the total time by 300 to get the average time. So with the help of simulation technique, we will get the result that x customers came today and spent y time in the bank and the average time spent by a customer is z .

Code of the Bank Simulation

Let's have a look on the C++ code of this simulation.

```
#include <iostream>
#include <string>
#include <strstream.h>

#include "Customer.cpp"
#include "Queue.h"
#include "PriorityQueue.cpp"
#include "Event.cpp"

Queue q[4];    // teller queues
PriorityQueue pq; //eventList;
int totalTime;
int count = 0;
int customerNo = 0;

main (int argc, char *argv[])
{
    Customer* c;
    Event* nextEvent;

    // open customer arrival file
    ifstream data("customer.dat", ios::in);

    // initialize with the first arriving customer.
    ReadNewCustomer(data);

    While( pq.length() > 0 )
    {
        nextEvent = pq.remove();
        c = nextEvent->getCustomer();
        if( c->getStatus() == -1 ){ // arrival event
            int arrTime = nextEvent->getEventTime();
            int duration = c->getTransactionDuration();
            int customerNo = c->getCustomerNumber();
            processArrival(data, customerNo,
```



```

                                arrTime, duration , nextEvent);
    }
    else { // departure event
        int qindex = c->getStatus();
        int departTime = nextEvent->getEventTime();
        processDeparture(qindex, departTime, nextEvent);
    }
}

```

We have included lot of files in the program. Other than the standard libraries, we have *Customer.cpp*, *Queue.h*, *PriorityQueue.cpp* and *Event.cpp*. With the help of these four files, we will create *Customer* object, *Queue* object, *PriorityQueue* object and *Event* object. You may think that these are four factories, creating objects for us.

As there are four tellers, so we will create equal number of queues (*Queue q[4]*). Then we create a priority queue object *pq* from the *PriorityQueue* factory. We declare *totalTime*, *count* and *customerNo* as int. These are global variables.

In the main method, we declare some local variables of customer and event. Afterwards, the customer.dat file for the input data is opened as:

```
ifstream data("customer.dat", ios::in);
```

We read the first customers data from this file as:

```
readNewCustomer(data);
```

Here *data* is the input file stream associated to *customer.dat*. We will read the arrival time and time of transaction from the file of the first customer. After reading it, we will process this information.

Now there is the *while loop* i.e. the main driver loop. It will run the simulation. First thing to note is that it is not clock-based which is that the loop will execute for 24 hours. Here we have the condition of priority queue's length. The variable *pq* represents the event queue. If there are some events to be processed, the queue *pq* will not be empty. Its length will not be zero. We get the next event from the priority queue, not from the queue. The method *pq.remove()* (de-queue method) will give us the event which is going to happen first in future. The priority of events is according the time. In the event object we have the *customerNo*. In the *if* statement, we check the status of the customer. If the status is -1, it will reflect that this is the new customer arrival event.

We know that when a new customer enters the bank, he will look at the four tellers and go to the teller where the queue is smallest. Therefore in the program, we will check which is the smallest queue and insert the customer in that queue. If the event is about the new customer, the *if* statement returns true. We will get its arrival time, duration and customer number and assign it to the variables *arrTime*, *duration* and *customerNo* respectively. We will call the method *processArrival()* and pass it the above information.

If the status of the customer is not equal to -1 , it means that the customer is in one of the four queues. The control will go to else part. We will get the status of the customer which can be 0, 1, 2 and 3. Assign this value to *qindex*. Later on, we will see how these values are assigned to the status. We will get the departure time of the customer and call the *processDeparture()* method.

In the main driver loop, we will get the next event from the event queue. In this case, events can be of two types i.e. arrival event and the departure event. When the person enters the bank, it is the arrival event. If any queue is empty, he will go to the teller. Otherwise, he will wait in the queue. After the completion of the transaction, the customer will leave the bank. It is the departure event.

Let's discuss the function *readNewCustomer()*. This function is used to read the data from the file.

```
void readNewCustomer(ifstream& data)
{
    int hour,min,duration;
    if (data >> hour >> min >> duration) {
        customerNo++;
        Customer* c = new Customer(customerNo,
                                    hour*60+min, duration);
        c->setStatus( -1 ); // new arrival
        Event* e = new Event(c, hour*60+min );
        pq.insert( e ); // insert the arrival event
    }
    else {
        data.close(); // close customer file
    }
}
```

Here, we have used the $>>$ to read the *hour*, *minute* and *duration* from the file. Then we create a customer object *c* from the customer factory with the *new* keyword. We pass the *customerNo*, *arrival time* and transaction *duration* to the constructor of the customer object. After the object creation, it is time to set its status to -1 . This means that it is an arriving customer. Then we create an event object *e* passing it the customer *c* and the *arrival time*. We insert this event into the priority queue *pq*. If there is no more data to read, we go into the else part and close the data file.

Let's see the function *processArrival()*. We have decided that when the customer arrives and no teller is available, he will go to the shortest queue.

```
int processArrival(ifstream &data, int customerNo, int arrTime, int duration,
                  Event* event)
{
    int i, small, j = 0;
    // find smallest teller queue
    small = q[0].length();
    for(i=1; i < 4; i++)
        if( q[i].length() < small ){
```

```

        small = q[i].length(); j = i;
    }

    // put arriving customer in smallest queue
    Customer* c = new Customer(customerNo, arrTime, duration );
    c->setStatus(j); // remember which queue the customer goes in
    q[j].enqueue(c);

    // check if this is the only customer in the.
    // queue. If so, the customer must be marked for
    // departure by placing him on the event queue.

    if( q[j].length() == 1 ) {
        c->setDepartureTime( arrTime+duration);
        Event* e = new Event(c, arrTime+duration );
        pq.insert(e);
    }

    // get another customer from the input
    readNewCustomer(data);
}

```

First of all, we will search for the smallest queue. For this purpose, there is a *for* loop in the method. We will check the length of all the four queues and get the smallest one. We store the index of the smallest queue in the variable *j*. Then we create a customer object. We set its status to *j*, which is the queue no. Then we insert the customer in the smallest queue of the four. The customer may be alone in the queue. In this case, he does not need to wait and goes directly to the teller. This is the real life scenario. When we go to bank, we also do the same. In the banks, there are queues and everyone has to enter in the queue. If the queue is empty, the customers go straight to the teller. Here we are trying to simulate the real life scenario. Therefore if the length of the queue is one, it will mean that the customer is alone in the queue and he can go to the teller. We calculate his departure time by adding the arrival time and transaction time. At this time, the person can leave the bank. We create a departure event and insert it into the priority queue. In the end, we read a new customer. This is the way; a programmer handles the new customers. Whenever a new person enters the bank, we create an event and insert it into the smallest queue. If he is alone in the queue, we create a departure event and insert it into the priority queue. In the main while loop, when we remove the event, in case of first future event, it will be processed. After the completion of the transaction, the person leaves the bank.

We may encounter another case. There may be a case that before leaving the bank, more persons arrive and they have to wait in the queue for their turn. We handle this scenario in the departure routine. The code is:

```

int processDeparture( int qindex, int departTime, Event* event)
{
    Customer* cinq = q[qindex].dequeue();
}

```

```

int waitTime = departTime - cinq->getArrivalTime();
totalTime = totalTime + waitTime;
count = count + 1;

// if there are any more customers on the queue, mark the
// next customer at the head of the queue for departure
// and place him on the eventList.
if( q[qindex].length() > 0 ) {
    cinq = q[qindex].front();
    int etime = departTime + cinq->getTransactionDuration();
    Event* e = new Event( cinq, etime);
    pq.insert( e );
}
}

```

In this method, we get the information about the *qindex*, *departTime* and *event* from the main method. We get the customer by using the *qindex*. Then we calculate the wait time of the customer. The wait time is the difference of departure time and the arrival time. The total time holds the time of all the customers. We added the wait time to the total time. We incremented the variable *count* by one. After the departure of this customer, next customer is ready for his transaction. The *if* statement is doing this. We check the length of the queue, in case of presence of any customer in the queue, we will check the customer with the *front()* method. We set its departure time (*etime*) by adding the depart time of the previous customer and his transaction time. Then we create an event and insert it in the priority queue.

In the end, we calculate the average time in the main loop and print it on the screen. Average time is calculated by dividing the total time to total customer.

```

// print the final average wait time.

double avgWait = (totalTime*1.0) / count;
cout << "Total time:  " << totalTime << endl;
cout << "Customer:    " << count << endl;
cout << "Average wait: " << avgWait << endl;

```

You may be thinking that the complete picture of simulation is not visible. How will we run this simulation? Another important tool in the simulation is animation. You have seen the animation of traffic. Cars are moving and stopping on the signals. Signals are turning into red, green and yellow. You can easily understand from the animation. If the animation is combined with the simulation, it is easily understood.

We have an animated tool here that shows the animation of the events. A programmer can see the animation of the bank simulation. With the help of this animation, you can better understand the simulation.

In this animation, you can see the Entrance of the customers, four tellers, priority queue and the Exit. The customers enter the queue and as the tellers are free. They go to the teller straight. Customer C1<30, 10> enters the bank. The customer C1 enters after 30 mins and he needs 10 mins for the transaction. He goes to the teller 1. Then

customer C2 enters the bank and goes to teller 2. When the transaction ends, the customer leaves the bank. When tellers are not free, customer will wait in the queue. In the event priority queue, we have different events. The entries in the priority queue are like *arr*, 76 (arrival event at 76 min) or *q1*, 80 (event in q1 at 80 min) etc. Let's see the statistics when a customer leaves the bank. At exit, you see the customer leaving the bank as C15<68, 3><77, 3>, it means that the customer C15 enters the bank at 68 mins and requires 3 mins for his transaction. He goes to the teller 4 but the teller is not free, so the customer has to wait in the queue. He leaves the bank at 77 mins.

This course is not about the animation or simulation. We will solve the problems, using different data structures. Although with the help of simulation and animation, you can have a real sketch of the problem.

Data Structures

Lecture No. 11

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4

4.1.1, 4.2.1

Summary

- Implementation of Priority Queue
- Tree
- Binary Tree
- Terminologies of a Binary Tree
- Strictly Binary Tree
- Level
- Complete Binary Tree
- Level of a Complete Binary Tree
- Tips

In the previous lecture, we demonstrated the technique of data structure priority queue with the help of the example from the banking system. Data structure priority queue is a variation of queue data structure to store the events. We have witnessed this simulation by animation. The simulation and animation help a programmer to understand the functionality of a program. As these are not the part of the current course, so we will study simulation and animation in detail in the coming courses.

A queue is a FIFO structure (i.e. first in first out). But there are its variations including the priority queue, discussed in the previous lecture. In priority queue, we were adding data to a queue but did not get the data from the queue by First In, First Out (FIFO) rule. We put events in the queue and got these from the queue with respect to the time of occurrence of the event. Thus we get the items from a priority queue in a particular order with respect to a priority given to the items (events). The priority queue is also an important data structure. Let's see its implementation.

Implementation of Priority Queue

In the priority queue, we put the elements in the queue to get them from the queue with a priority of the elements. Following is the C++ code of the priority queue.

```
#include "Event.cpp"
#define PQMAX 30

class PriorityQueue
{
```

```

public:
    PriorityQueue()
    {
        size = 0; rear = -1;
    };
    ~PriorityQueue() {};
    int full(void)
    {
        return ( size == PQMAX ) ? 1 : 0;
    };

    Event* remove()
    {
        if( size > 0 )
        {
            Event* e = nodes[0];
            for(int j=0; j < size-2; j++ )
                nodes[j] = nodes[j+1];
            size = size-1; rear=rear-1;
            if( size == 0 ) rear = -1;
            return e;
        }
        return (Event*)NULL;
        cout << "remove - queue is empty." << endl;
    };

    int insert(Event* e)
    {
        if( !full() )
        {
            rear = rear+1;
            nodes[rear] = e;
            size = size + 1;
            sortElements(); // in ascending order
            return 1;
        }
        cout << "insert queue is full." << endl;
        return 0;
    };

    int length() { return size; };
};

```

In this code, the file Events.cpp has been included. Here we use events to store in the queue. To cater to the need of storing other data types too, we can write the PriorityQueue class as a template class.

In the above code, we declare the class PriorityQueue. Then there is the public part of the class. In the public part, at first a programmer encounters the constructor of the class. In the constructor, we assign the value 0 to *size* and -1 to *rear* variables. A destructor, whose body is empty, follows this. Later, we employ the method *full()*

which checks the *size* equal to the PQMAX to see whether the queue is full. If the *size* is equal to PQMAX, the function returns 1 i.e. TRUE. Otherwise, it returns 0 i.e. FALSE. We are going to implement the priority queue with an array. We can also use linked list to implement the priority queue. However, in the example of simulation studied in the previous lecture, we implemented the queue by using an array. We have seen in the simulation example that there may be a maximum of five events. These events include one arrival event and four departure events. That means four queues from where the customers go to the teller and one to go out of the bank after the completion of the transaction. As we know that there is a need of only five queues, so it was decided to use the array instead of dynamic memory allocation and manipulating the pointers.

In the *remove()* method, there are some things which are the property of the priority queue. We don't have these in the queue. In this method, first of all we check the size of the priority queue to see whether there is something in the queue or not. If size is greater than 0 i.e. there are items in the queue then we get the event pointer (pointer to the object Event) *e* from the first position (i.e. at index 0) of the array, which we are using internally to implement the queue. At the end of the method, we return this event object *e*. This means that we are removing the first object from the internal array. We already know that the removal of an item from the start of an array is very time consuming as we have to shift all the remaining items one position to the left. Thus the *remove()* method of the queue will execute slowly. We solved this problem by removing the item from the position where the *front* pointer is pointing. As the *front* and *rear* went ahead and we got empty spaces in the beginning, the circular array was used. Here, the *remove()* method is not removing the element from the *front*. Rather, it is removing element from the first position (i.e. index 0). Then we execute a *for* loop. This *for* loop starts from 0 and executes to *size-2*. We can notice that in this *for* loop, we are shifting the elements of the array one position left to fill the space that has been created by removing the element from the first position. Thus the element of index 1 becomes at index 0 and element of index 2 becomes at index 1 and so on. Afterwards, we decrease *size* and *rear* by 1. By decreasing the size 1 if it becomes zero, we will set *rear* to -1. Now by the statement

```
return e ;
```

We return the element (object *e*), got from the array. The outer part of the *if* block

```
return (Event*)NULL;
cout << "remove - queue is empty." << endl;
```

is executed if there is nothing in the queue i.e. *size* is less than 0. Here we return NULL pointer and display a message on the screen to show that the queue is empty.

Now let's look at the *insert()* method. In this method, first of all we check whether the array (we are using internally) is full or not. In case, it is not full, we increase the value of *rear* by 1. Then we insert the object *e* in the nodes array at the position *rear*. Then the *size* is increased by 1 as we have inserted (added) one element to the queue. Now we call a method *sortElements()* that sorts the elements of the array in an order. We will read different algorithms of sorting later in this course.

We have said that when we remove an element from a priority queue, it is not according to the FIFO rule. We will remove elements by some other rule. In the simulation, we had decided to remove the element from the priority queue with

respect to the time of occurrence of an event (arrival or departure). We will remove the element (event) whose time of occurrence is before other events. This can be understood from the example of the traffic over a bridge or crossing. We will give higher priority to an ambulance as compared to a bus. The cars will have the lower priority. Thus when a vehicle has gone across then after it we will see if there is any ambulance in the queue. If it is there, we will remove it from the queue and let go across the bridge. Afterwards, we will allow a bus to go and then the cars. In our simulation example, we put a number i.e. time of occurrence, with the object when we add it to the queue. Then after each insertion, we sort the queue with these numbers in ascending order. Thus the objects in the nodes array get into an order with respect to the time of their occurrence. After sorting, the first element in the array is the event, going to be occurring earliest in the future. Now after sorting the queue we return 1 that shows the insert operation has been successful. If the queue is full, we display a message to show that the queue is full and return 0, indicating that the insert operation had failed.

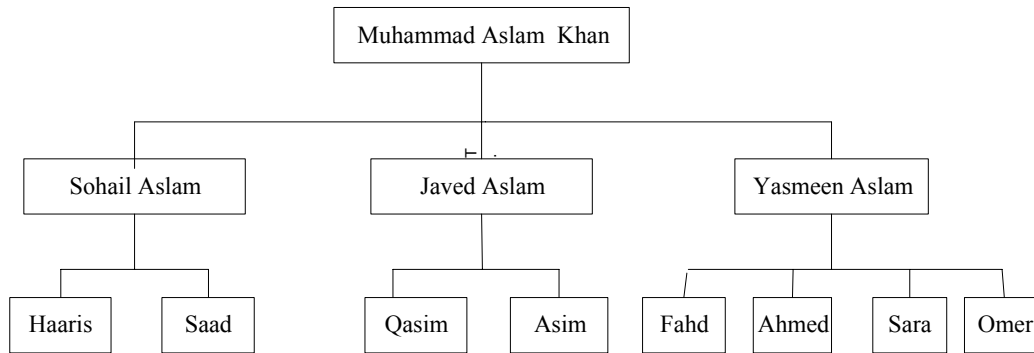
Then there comes the *length()* method, having a single statement i.e.

```
return size ;
```

This method returns the size of the queue, reflecting the number of elements in the queue. It is not the size of the array used internally to store the elements of the queue. We have seen the implementation of the priority queue by using an array. We will use the priority queue in some other algorithms later. Now, we will see another implementation of the priority queue using some thing other than array, which is much better than using an array. This will be more efficient. Its *remove* and *insert* methods will be faster than the ones in array technique. Here in the simulation, we were making only five queues for the events. Suppose, if these go to hundreds or thousands, a lot of time will be spent to remove an element from the queue. Similarly, when an element is added, after adding the element, to sort the whole array will be a time consuming process. Thus the application, with the use of the priority queue, will not be more efficient with respect to the time.

Tree

Now let's talk about a data structure called tree. This is an important data structure. This data structure is used in many algorithms. We will use it in most of our assignments. The data structures that we have discussed in previous lectures are linear data structures. The linked list and stack are linear data structures. In these structures, the elements are in a line. We put and get elements in and from a stack in linear order. Queue is also a linear data structure as a line is developed in it. There are a number of applications where linear data structures are not appropriate. In such cases, there is need of some non-linear data structure. Some examples will show us that why non-linear data structures are important. Tree is one of the non-linear data structures. Look at the following figure. This figure (11.1) is showing a genealogy tree of a family.



In this genealogy tree, the node at the top of the tree is *Muhammad Aslam Khan* i.e. the head of the family. There are three nodes under this one. These are *Sohail Aslam*, *Javed Aslam* and *Yasmeen Aslam*. Then there are nodes under these three nodes i.e. the sons of these three family members. You may have seen the tree like this of some other family. You can make the tree of your family too. The thing to be noted in this genealogical tree is that it is not a linear structure like linked list, in which we have to tell that who is the father and who is the son. We make it like a tree. We develop the tree top-down, in which the father of the family is on the top with their children downward. We can see the similar tree structure of a company. In the tree of a company, the CEO of the company is on the top, followed downwardly by the managers and assistant managers. This process goes downward according to the administrative hierarchy of the company. Our tree structure is different from the actual tree in the way that the actual tree grows from down to up. It has root downside and the leaves upside. The data structure tree is opposite to the real tree as it goes upside down. This top-down structure in computer science makes it easy to understand the tree data structure.

There may be situations where the data, in our programs or applications, is not in the linear order. There is a relationship between the data that cannot be captured by a linked list or other linear data structure. Here we need a data structure like tree.

In some applications, the searching in linear data structures is very tedious. Suppose we want to search a name in a telephone directory having 100000 entries. If this directory is in a linked list manner, we will have to traverse the list from the starting position. We have to traverse on average half of the directory if we find a name. We may not find the required name in the entire directory despite traversing the whole list. Thus it would be better to use a data structure so that the search operation does not take a lot of time. Taking into account such applications, we will now talk about a special tree data structure, known as binary tree.

Binary Tree

The mathematical definition of a binary tree is

“A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right sub-trees”. Each element of a binary tree is called a node of the tree.

Following figure shows a binary tree.

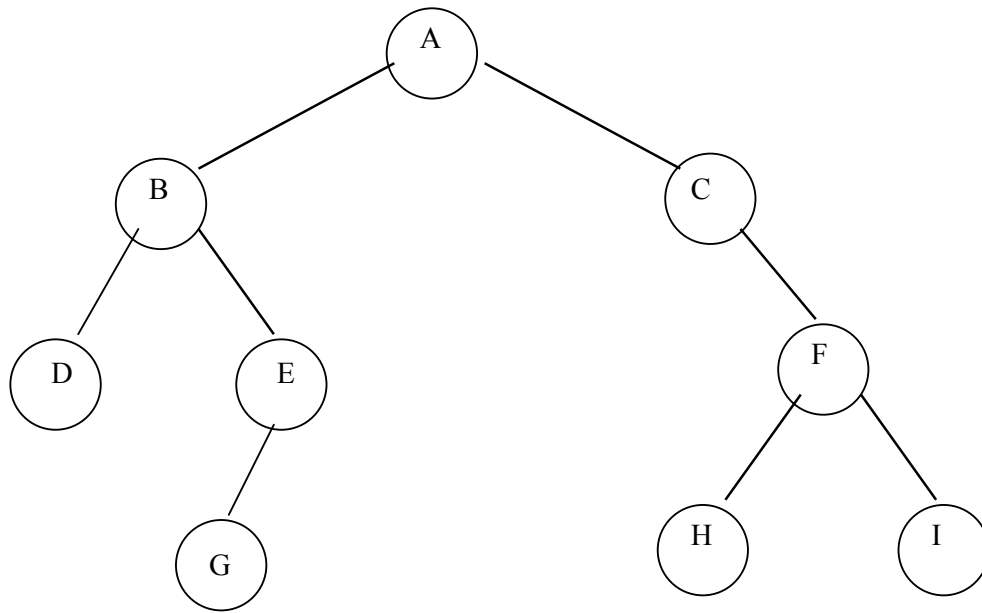


Fig 11.2: A Binary Tree

There are nine nodes in the above figure of the binary tree. The nine nodes are A, B, C, D, E, F, G, H, and I. The node A is at the top of the tree. There are two lines from the node A to left and right sides towards node B and C respectively. Let's have a look at the left side of the node A. There are also two lines from node B to left and right, leading to the nodes D and E. Then from node, E there is only one line that is to the left of the node and leads to node G. Similarly there is a line from node C towards the node F on right side of the node C. Then there are two lines from node F that leads to the nodes H and I on left and right side respectively.

Now we analyze this tree according to the mathematical definition of the binary tree. The node A is the root of the tree. And tree structure (i.e. the nodes B, D, E, G and their relation) on the left side of the node A is a sub-tree, called the *Left subtree*. Similarly the nodes (C, F, H and I) on the right side of the node A comprise the sub-tree termed as *Right subtree*. Thus we made three parts of the tree. One part contains only one node i.e. A while the second part has all the nodes on left side of A, called as *Left subtree*. Similarly the third part is the *Right subtree*, containing the nodes on right side of A. The following figure depicts this scenario.

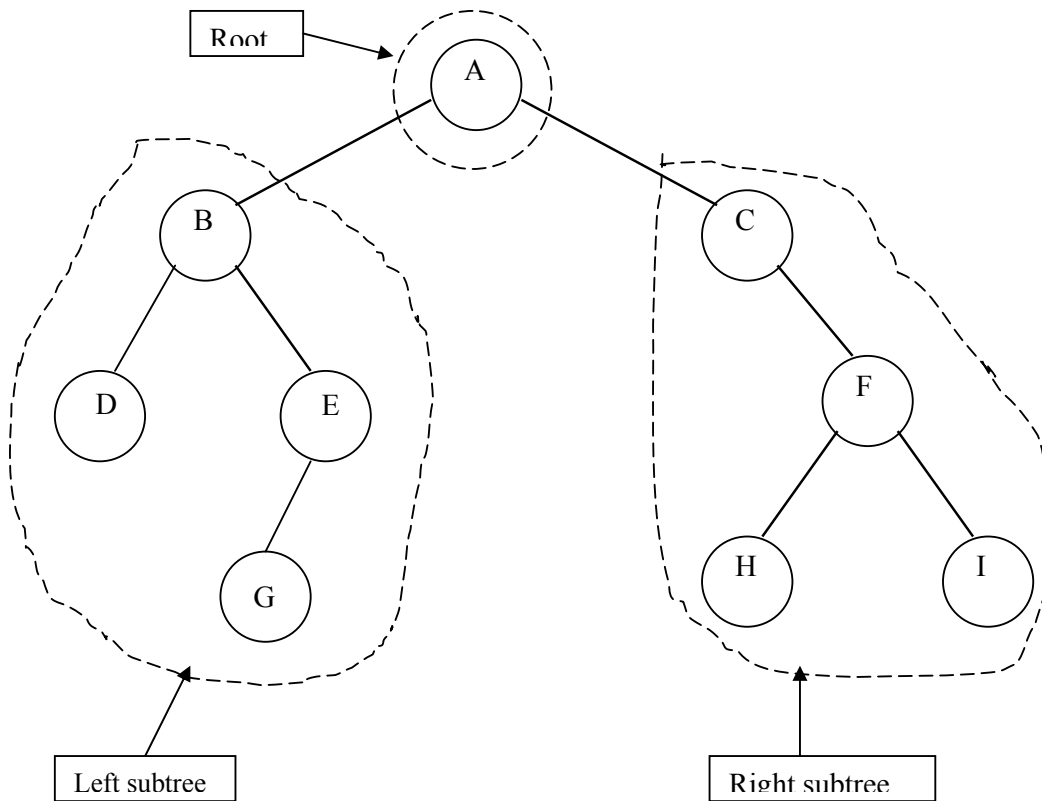
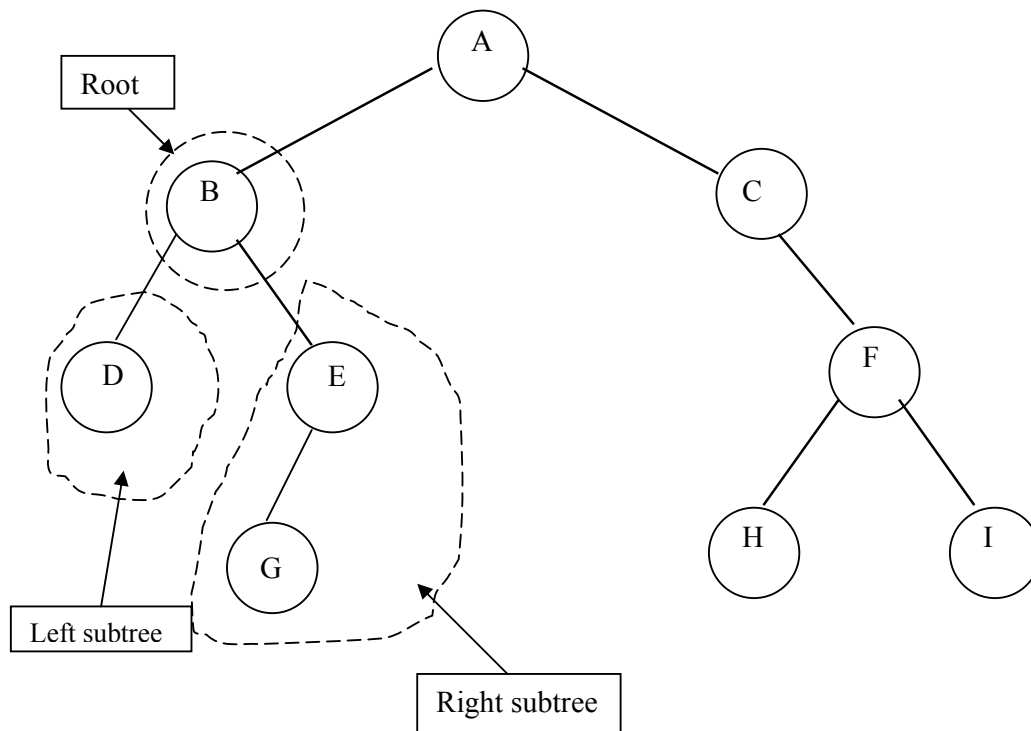
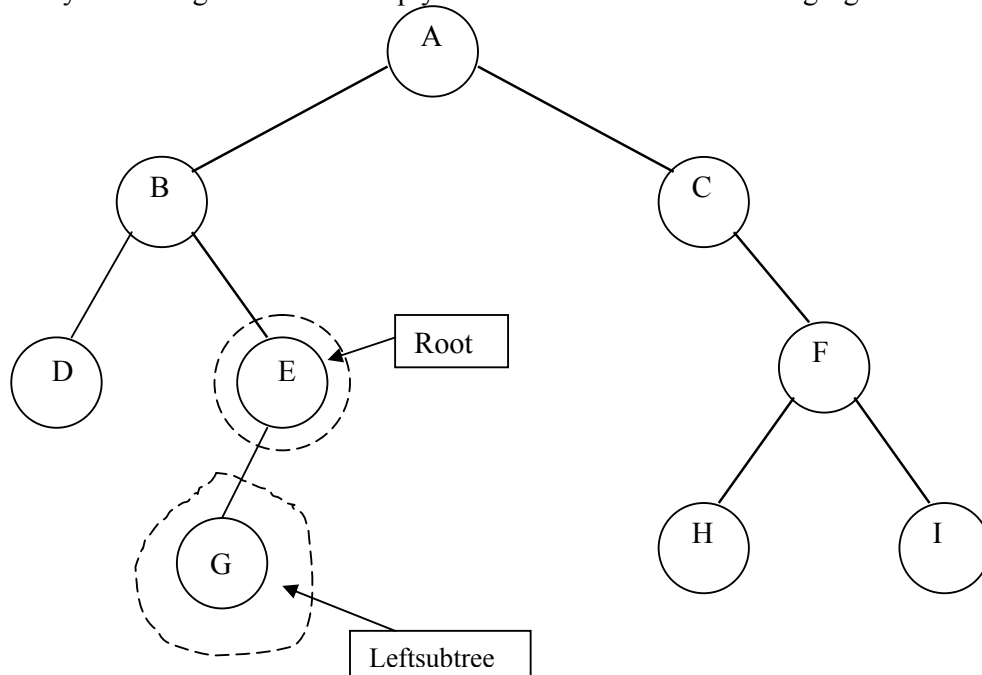


Fig 11.3: Analysis of a binary tree

The same process of sub classing the tree can be done at the second level. That means the left and right subtrees can also be divided into three parts similarly. Now consider the left subtree of node A. This left subtree is also a tree. The node B is the root of this tree and its left subtree is the node D. There is only one node in this left subtree. The right subtree of the node B consists of two nodes i.e. E and G. The following figure shows the analysis of the tree with root B.

**Fig 11.4:** Analysis of a binary tree

On going one step down and considering right subtree of node B, we see that E is the root and its left subtree is the single node G. There is no right subtree of node E or we can say that its right subtree is empty. This is shown in the following figure.

**Fig 11.5:** Analysis of a binary tree

Now the left sub tree of E is the single node G. This node G can be considered as the root node with empty left and right sub trees.

The definition of tree is of recursive nature. This is due to the fact that we have seen that which definition has been applied to the tree having node A as root, is applied to its subtrees one level downward. Similarly as long as we go down ward in the tree the same definition is used at each level of the tree. And we see three parts i.e. root, left subtree and right subtree at each level.

Now if we carry out the same process on the right subtree of root node A, the node C will become the root of this right subtree of A. The left subtree of this root node C is empty. The right subtree of C is made up of three nodes F, H and I. This is shown in the figure below.

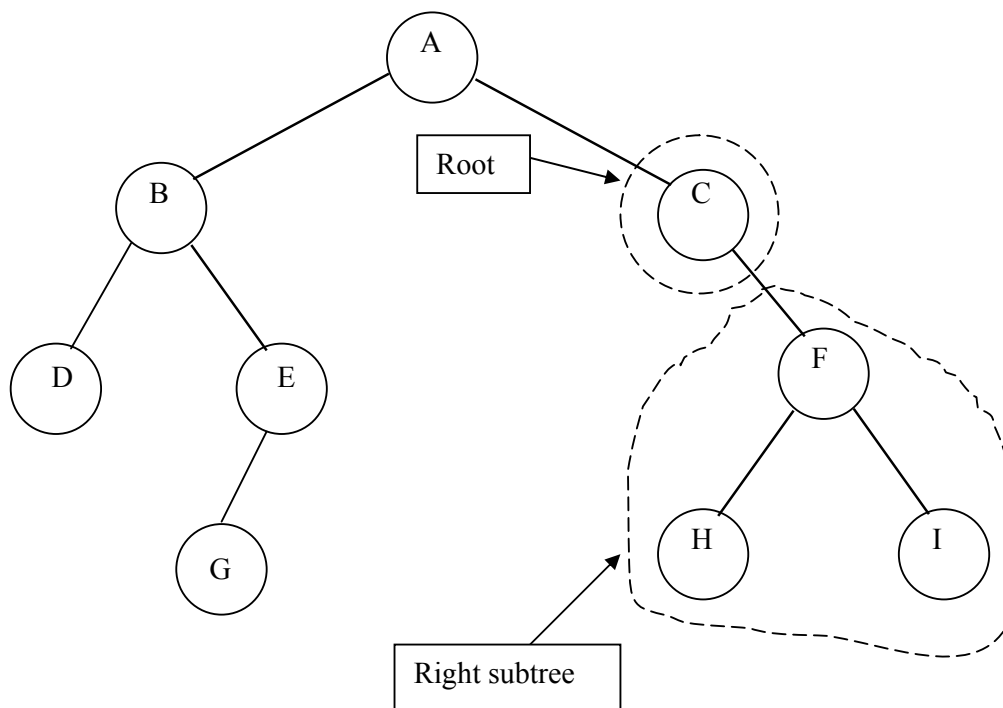


Fig 11.6: Analysis of a binary tree

Now we apply the same recursive definition on the level below the node C. Now the right subtree of the node C will be considered as a tree. The root of this tree is the node F. The left and right subtrees of this root F are the nodes H and I respectively. The following figure depicts this.

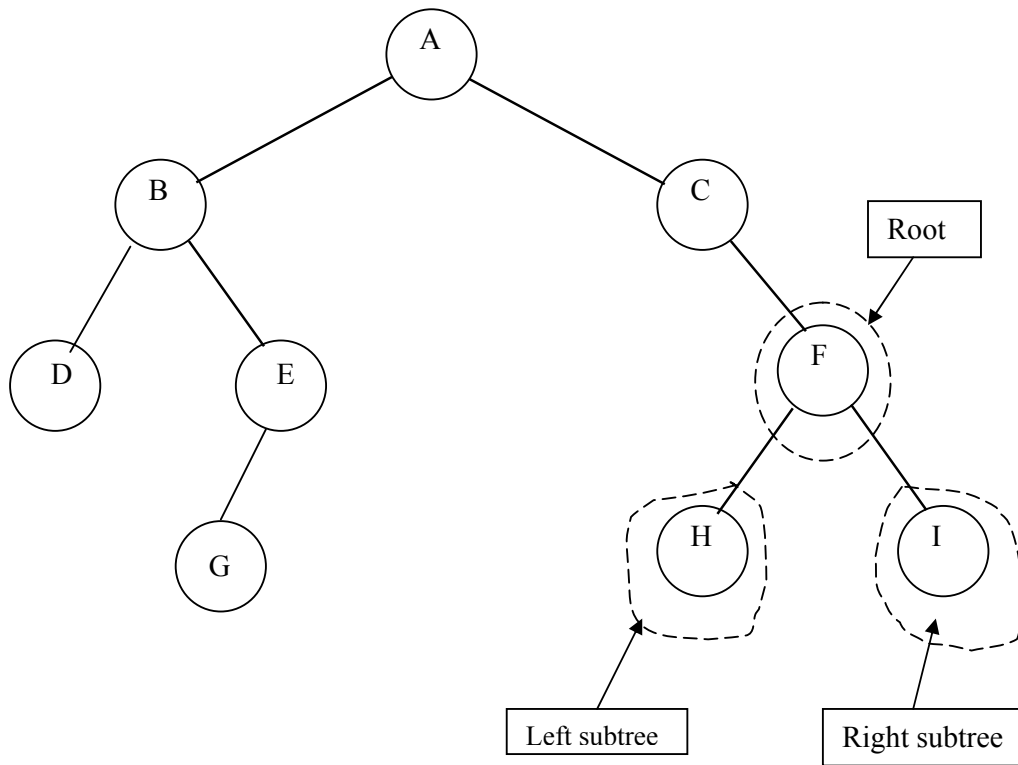


Fig 11.7: Analysis of a binary tree

We make (draw) a tree by joining different nodes with lines. But we cannot join any nodes whichever we want to each other. Consider the following figure.

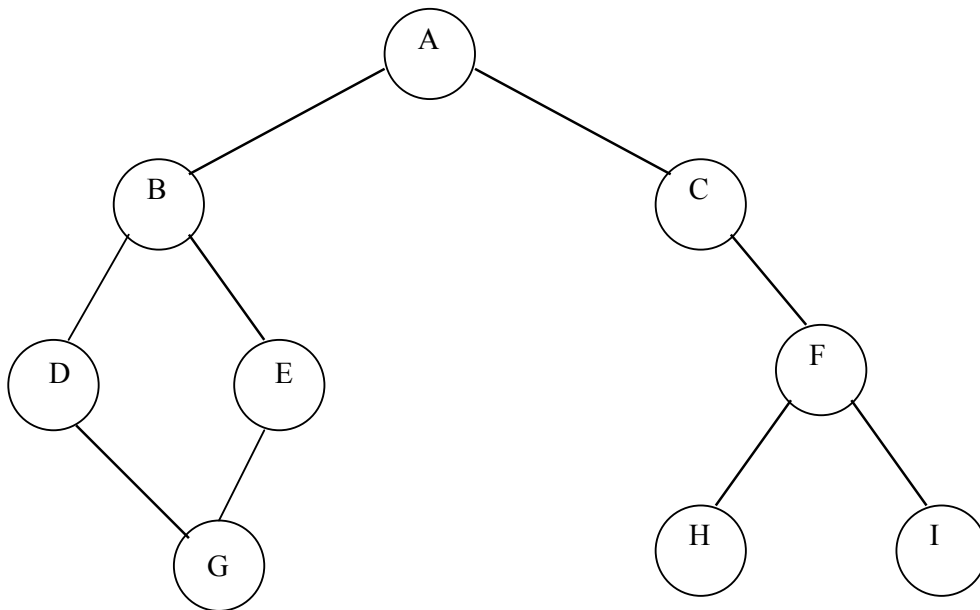


Fig 11.8: A non-tree structure

It is the same tree, made earlier with a little modification. In this figure we join the node G with D. Now this is not a tree. It is due to the reason that in a tree there is always one path to go (from root) to a node. But in this figure, there are two paths (tracks) to go to node G. One path is from node A to B, then B to D and D to G. That means the path is A-B-D-G. We can also reach to node G by the other path that is the path A-B-E-G. If we have such a figure, then it is not a tree. Rather, it may be a graph. We will discuss about graphs at the end of this course.

Similarly if we put an extra link between the nodes A and B, as in the figure below, then it is also no more a tree. This is a multiple graph as there are multiple (more than 1) links between node A and B.

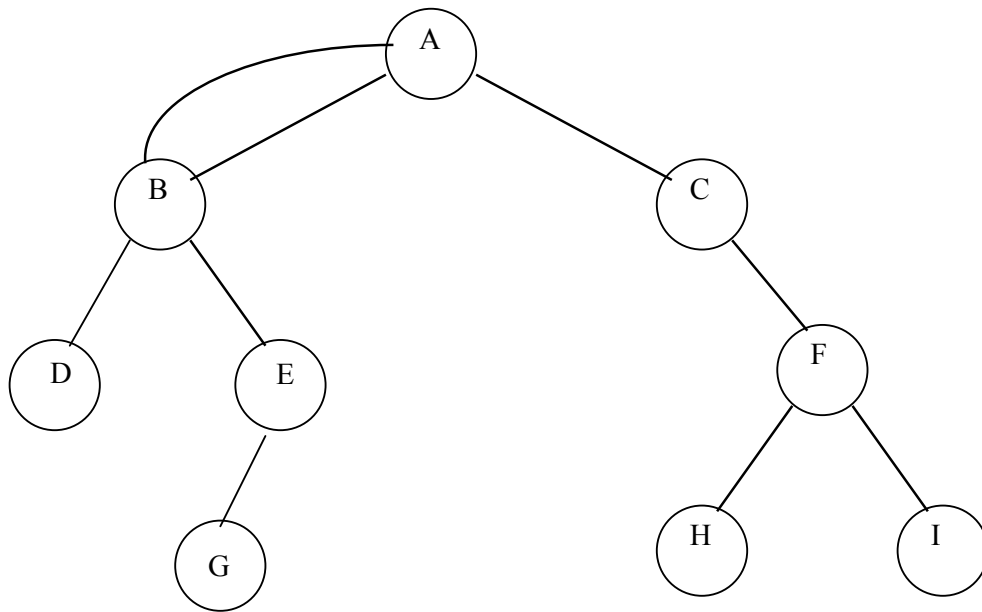


Fig 11.9: A non-tree structure

Similarly if we put other links between different nodes, then the structure thus developed will not be a tree. The following figure is also an example of a structure that is not a tree as there are multiple links between the different nodes.

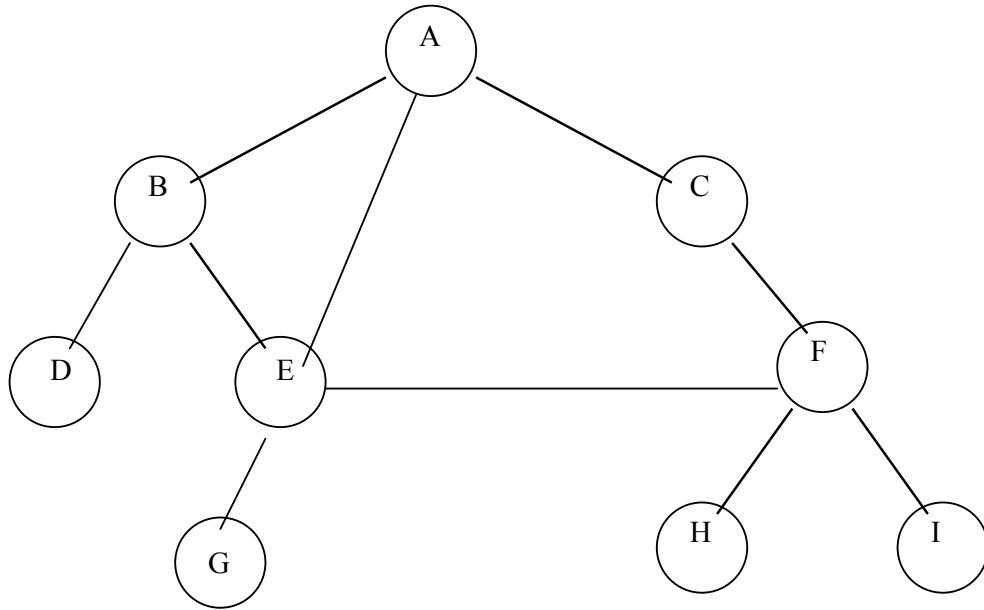


Fig 11.10: A non-tree structure

Terminologies of a binary tree

Now let's discuss different terminologies of the binary tree. We will use these terminologies in our different algorithms. Consider the following figure.

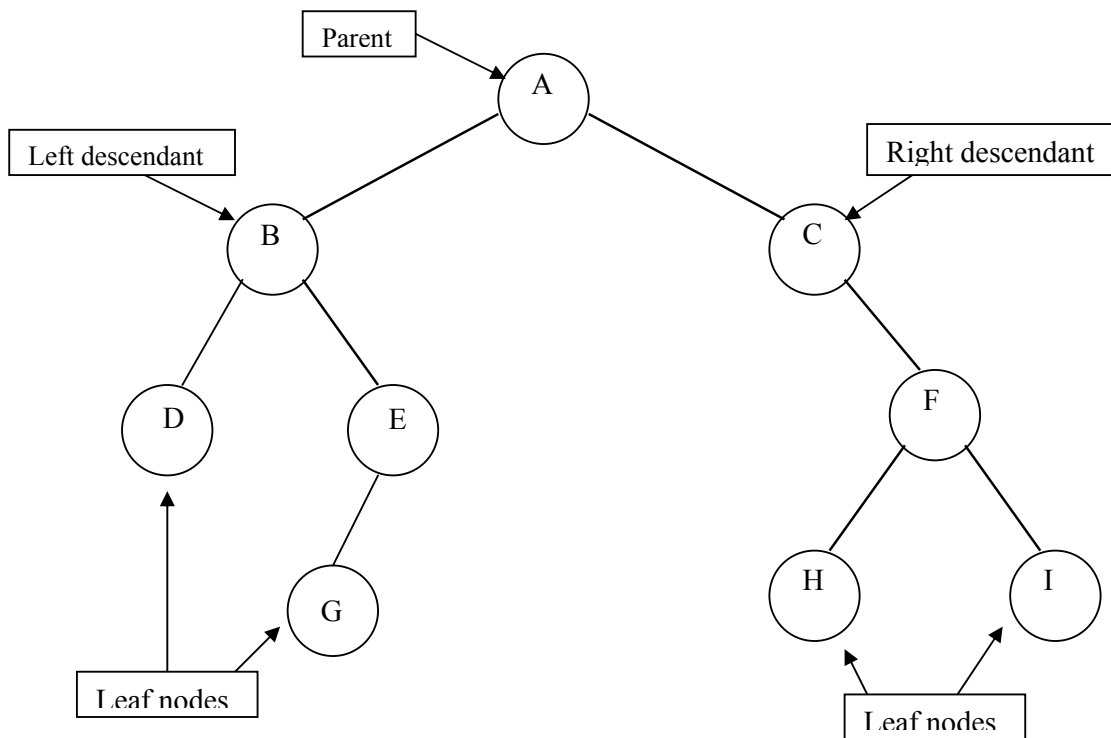


Fig 11.11: Terminologies used in a binary tree

We have discussed that the node on the top is said root of the tree. If we consider the relationship of these nodes, A is the parent node with B and C as the left and right descendants respectively. C is the right descendant of A. Afterwards, we look at the node B where the node D is its left descendant and E is its right descendant. We can use the words descendant and child interchangeably. Now look at the nodes D, G, H and I. These nodes are said leaf nodes as there is no descendant of these nodes. We are just introducing the terminology here. In the algorithms, we can use the words root or parent, child or descendant. So the names never matter.

Strictly Binary Tree

There is a version of the binary tree, called strictly binary tree. A binary tree is said to be a strictly binary tree if every non-leaf node in a binary tree has non-empty left and right subtrees.

Now consider the previous figure 11.11. We know that a leaf node has no left or right child. Thus nodes D, G, H and I are the leaf nodes. The non-leaf nodes in this tree are A, B, C, E and F. Now according to the definition of strictly binary tree, these non-leaf nodes (i.e. A, B, C, E and F) must have left and right subtrees (Childs). The node A has left child B and right child C. The node B also has its left and right children that are D and E respectively. The non-leaf node C has right child F but not a left one. Now we add a left child of C that is node J. Here C also has its left and right children. The node F also has its left and right descendants, H and I respectively. Now the last non-leaf node E has its left child, G but no right one. We add a node K as the right child of the node E. Thus the tree becomes as shown below in the figure 11.12.

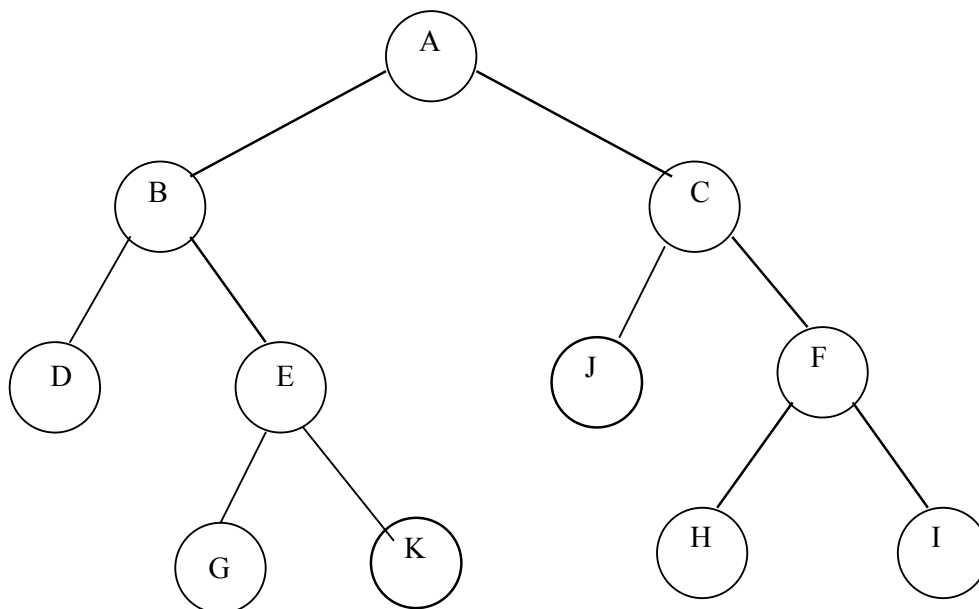


Fig 11.12: A Strictly binary tree

Now all the non-leaf nodes (A, B, C, E and F) have their left and right children so according to the definition, it is a strictly binary tree.

Level

The level of a node in a binary tree is defined as follows:

- Root has level 0,
- Level of any other node is one more than the level its parent (father).
- The *depth* of a binary tree is the maximum level of any leaf in the tree.

To understand level of a node, consider the following figure 11.13. This figure shows the same tree of figure 11.2, discussed so far.

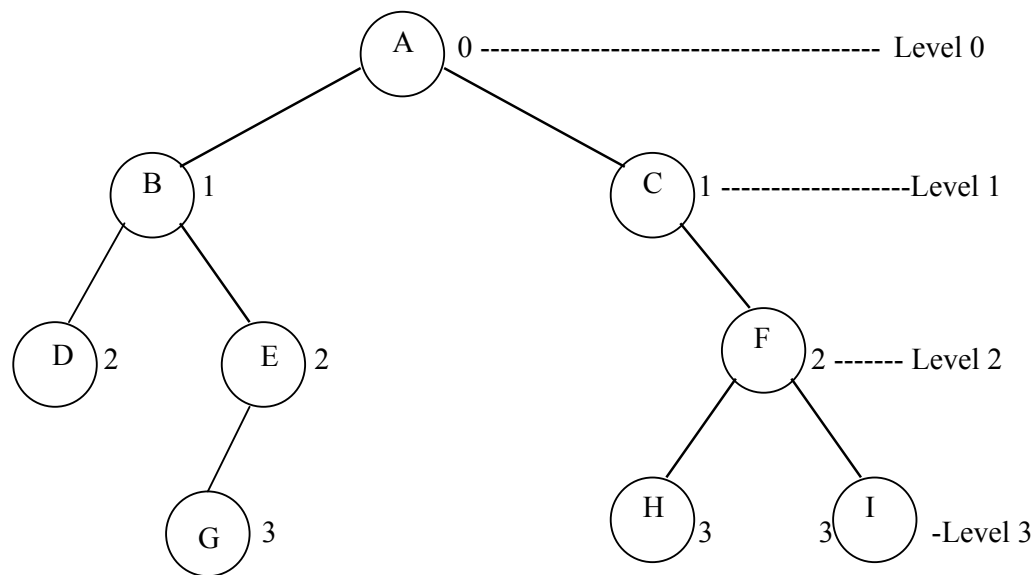


Fig 11.13: Level of nodes of a tree

In the above figure, we have mentioned the level of each node. The figure also shows that the root node has level 0. We have talked in the definition of the level that the level of a node, other than root, is one more than the level of its parent. Now in the tree, the parent of nodes B and C is node A with level 0. Now by definition, the level of B and C will be 1 (i.e. level of A + 1). Now if go downward in the tree and see the nodes D, E and F, the parent node of D and E is B the level of which is 1. So the level of D and E is 2 (i.e. 1 + 1). Similarly the level of F is 2 as the level of its parent (i.e. C) is 1. In the same way, we can easily understand that the level of G, H and I is 3 as the parent nodes of these (E and F) have level 2. Thus we see that tree has multi levels in the downward direction. The levels of the tree increase, as the tree grows downward more. The number of nodes also increases.

Now let's talk why we call it a binary tree. We have seen that each node has a maximum of two subtrees. There might be two, one or no subtree of a node. But it cannot have more than two. So, we call such a tree a binary one due to the fact that a node of it can have a maximum of two subtrees. There are trees whose nodes can have more than two subtrees. These are not binary trees. We will talk about these trees later.

By seeing the level of a tree, we can tell the depth of the tree. If we put level with each node of the binary tree, the depth of a binary tree is the maximum level. In the

figure 11.13, the deepest node is at level 3 i.e. the maximum level is 3. So the depth of this binary tree is 3.

Complete Binary Tree

Now, if there are left and right subtrees for each node in the binary tree as shown below in figure 11.14, it becomes a complete binary tree.

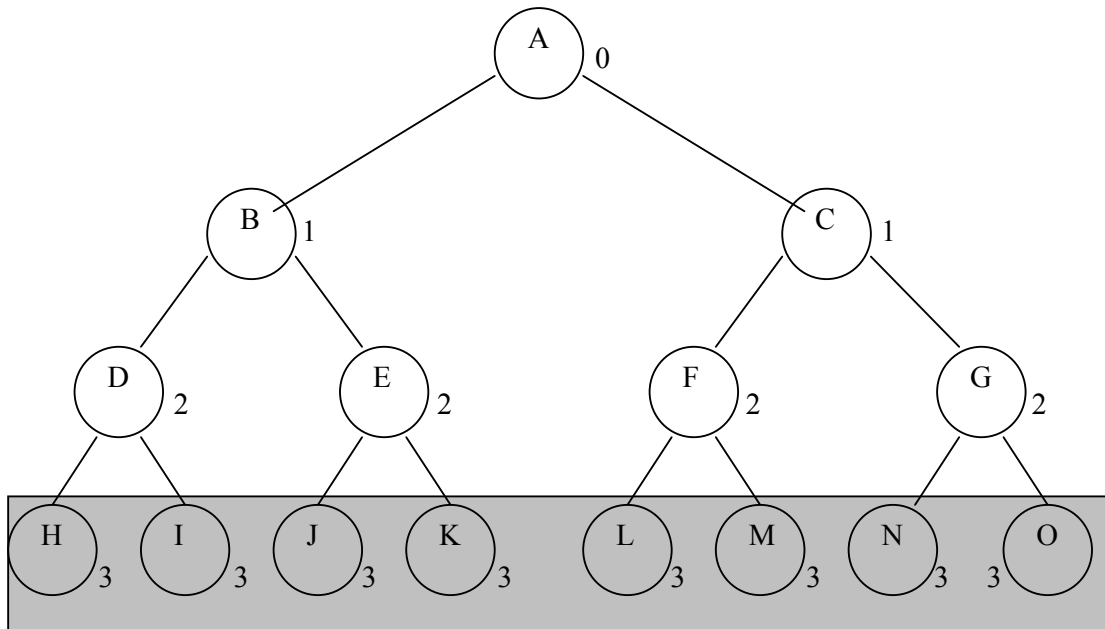


Fig 11.14: A Complete Binary Tree

The definition of the complete binary tree is

“A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d ”.

Now look at the tree, the leaf nodes of the tree are at level 3 and are H, I, J, K, L, M, N and O. There is no such a leaf node that is at some level other than the depth level d i.e. 3. All the leaf nodes of this tree are at level 3 (which is the depth of the tree i.e. d). So this is a complete binary tree. In the figure, all the nodes at level 3 are highlighted.

We know that the property of the binary tree is that its node can have a maximum of two subtrees, called as left and right subtrees. The nodes increase at each level, as the tree grows downward. In a complete binary tree, the nodes increase in a particular order. If we reconsider the previous tree (figure 11.14), we note that the number of node at level 0 is 1. We can say it 2^0 , as 2^0 is equal to 1. Down to this, the node is the level 1 where the number of nodes is 2, which we can say 2^1 , equal to 2. Now at the next level (i.e. level 2), the number of nodes is 4 that mean 2^2 . Finally the number of nodes at level 3 is 8, which can be called as 2^3 . This process is shown pictorially in the following figure 11.15.

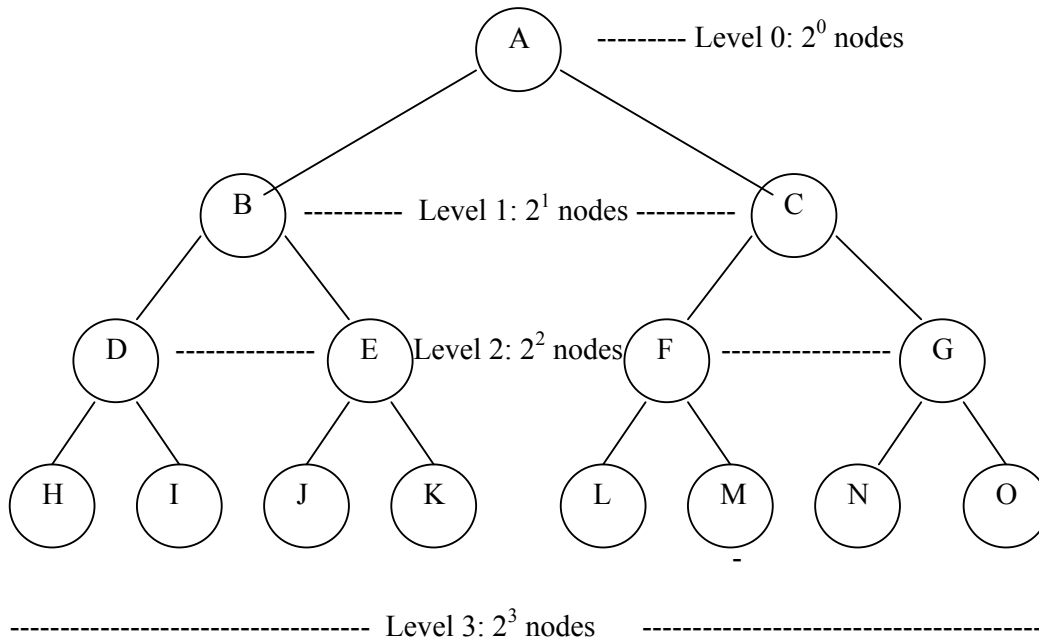


Fig 11.15: Number of nodes at each level in a complete binary tree

By observing the number of nodes at a particular level, we come to the conclusion that the number of nodes at a level is equal to the level number raising to the power of two. Thus we can say that in a complete binary tree at a particular level k , the number of nodes is equal to 2^k . Note that this formula for the number of nodes is for a complete binary tree only. It is not necessary that every binary tree fulfill this criterion. Applying this formula to each level while going to the depth of the tree (i.e. d), we can calculate the total number of nodes in a complete binary tree of depth d by adding the number of nodes at each level. This can be written as the following summation.

$$2^0 + 2^1 + 2^2 + \dots + 2^d = \sum_{j=0}^d 2^j = 2^{d+1} - 1$$

Thus according to this summation, the total number of nodes in a complete binary tree of depth d will be $2^{d+1} - 1$. Thus if there is a complete binary tree of depth 4, the total number of nodes in it will be calculated by putting the value of d equal to 4. It will be calculated as under.

$$2^{4+1} - 1 = 2^5 - 1 = 32 - 1 = 31$$

Thus the total number of nodes in the complete binary tree of depth 4 is 31.

We know that the total number of nodes (leaf and non-leaf) of a complete binary tree of depth d is equal to $2^{d+1} - 1$. In a complete binary tree, all the leaf nodes are at the depth level d . So the number of nodes at level d will be 2^d . These are the leaf nodes. Thus the difference of total number of nodes and number of leaf nodes will give us the number of non-leaf (inner) nodes. It will be $(2^{d+1} - 1) - 2^d$ i.e. $2^d - 1$. Thus we conclude that in a complete binary tree, there are 2^d leaf nodes and $2^d - 1$ non-leaf (inner) nodes.

Level of a Complete Binary Tree

We can conclude some more results from the above mathematical expression. We can find the depth of a complete binary tree if we know the total number of nodes. If we have a complete binary tree with n total nodes, then by the equation of the total number of nodes we can write

$$\text{Total number of nodes} = 2^{d+1} - 1 = n$$

To find the value of d , we solve the above equation as under

$$2^{d+1} - 1 = n$$

$$2^{d+1} = n + 1$$

$$d + 1 = \log_2 (n + 1)$$

$$d = \log_2 (n + 1) - 1$$

After having n total nodes, we can find the depth d of the tree by the above equation. Suppose we have 100,000 nodes. It means that the value of n is 100,000, reflecting a depth i.e. d of the tree will be $\log_2 (100000 + 1) - 1$, which evaluates to 20. So the depth of the tree will be 20. In other words, the tree will be 20 levels deep. The significance of all this discussion about the properties of complete binary tree will become evident later.

Operations on Binary Tree

We can define different operations on binary trees.

If p is pointing to a node in an existing tree, then

- $\text{left}(p)$ returns pointer to the left subtree
- $\text{right}(p)$ returns pointer to right subtree
- $\text{parent}(p)$ returns the father of p
- $\text{brother}(p)$ returns brother of p .
- $\text{info}(p)$ returns content of the node.

We will discuss these methods in detail in next lecture.

Tips

- A priority queue is a variation of queue that does not follow FIFO rule.
- Tree is a non-linear data structure.

- The maximum level of any leaf in a binary tree is called the depth of the tree.
- Other than the root node, the level of each node is one more than the level of its parent node.
- A complete binary tree is necessarily a strictly binary tree but not vice versa.
- At any level k , there are 2^k nodes at that level in a complete binary tree.
- The total number of nodes in a complete binary tree of depth d is $2^{d+1} - 1$.
- In a complete binary tree there are 2^d leaf nodes and $2^d - 1$ non-leaf nodes.

Data Structures

Lecture No. 12

Reading Material

Data Structures And Algorithm analysis in C++

Chapter 4

4.2, 4.3(4.3.2, 4.3.4)

Summary

- Operations on Binary Tree
- Applications of Binary Tree
- Searching for Duplicates
- C++ Implementation of Binary Tree
- Trace of insert

Operations on Binary Tree

In the last lecture, we talked about the uses of binary tree, which is an abstract data type. We discussed about the functions to find the information inside a node, the parent, the siblings(brothers), the left and right children of a node. In this lecture, we will talk about the algorithms and implementation of those functions.

When we discuss about an abstract data type, firstly we focus *what* it can do for us and don't bother about the *how* part. The how part or implementation is thought out later.

While implementing these abstract data types, the implementation is hidden in a class so it is abstract to the user. The user only needs to be aware of the interface. But there can be situations when the users may like to know about the implementation detail, for example, when a data type is performing quite slower than promised.

For now, we start our discussion from the methods of tree data type. Consider a tree has been formed already, following methods will be used to perform different operations on a node of this tree:

Operation	Description
left(p)	Returns a pointer to the left sub-tree
right(p)	Returns a pointer to the right sub-tree
parent(p)	Returns the father node of p
brother(p)	Returns the brother node of p
info(p)	Returns the contents of node p

These methods have already been discussed at the end of the previous lecture, however, few more methods are required to construct a binary tree:

Operation	Description
setLeft(p, x)	Creates the left child node of p and set the value x into it.
setRight(p, x)	Creates the right child node of p , the child node contains the info x .

All these methods are required to build and to retrieve values from a tree.

Applications of Binary Tree

Let's take few examples to understand how the tree data type is used and what are its benefits. We will also develop some algorithms that may be useful in future while working with this data type.

Binary tree is useful structure when two-way decisions are made at each point. Suppose we want to find all duplicates in a list of the following numbers:

14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

This list may comprise numbers of any nature. For example, roll numbers, telephone numbers or voter's list. In addition to the presence of duplicate number, we may also require the frequency of numbers in the list. As it is a small list, so only a cursory view may reveal that there are some duplicate numbers present in this list. Practically, this list can be of very huge size ranging to thousands or millions.

Searching for Duplicates

One way of finding duplicates is to compare each number with all those that precede it. Let's see it in detail.

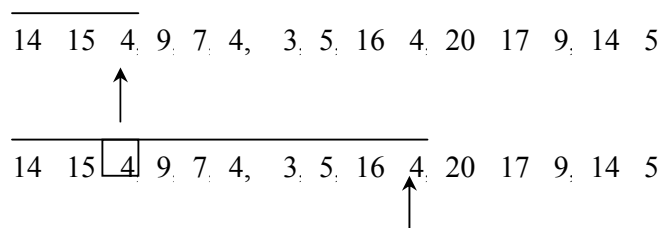


Fig 12.1: Search for Duplicates

Suppose, we are looking for duplicates for the number 4, we will *start* scanning from the first number 14. While scanning, whenever we find the number 4 inside, we remember the position and increment its frequency counter by 1. This comparison will go on till the end of the list to get the duplicates or sequence of the number 4. You might have understood already that we will have to perform this whole scanning of list every time for each number to find duplicates. This is a long and time consuming process.

So this procedure involves a large number of comparisons, if the list of numbers is large and is growing.

A linked list can handle the growth. But it can be used where a programmer has no idea about the size of the data before hand. The number of comparisons may still be large. The comparisons are not reduced after using linked list as it is a linear data

structure. To search a number in a linked list, we have to begin from the start of the list to the end in the linear fashion, traversing each node in it. For optimizing search operation in a list, there is no real benefit of using linked list. On the contrary, the search operation becomes slower even while searching in an array because the linked list is not contiguous like an array and traversing is done in the linked list with the help of pointers.

So, the solution lies in reducing the number of comparisons. The number of comparisons can be drastically reduced with the help of a binary tree. The benefits of linked list are there, also the tree grows dynamically like the linked list.

The binary tree is built in a special way. The first number in the list is placed in a node, designated as the *root* of the binary tree. Initially, both left and right sub-trees of the *root* are empty. We take the next number and compare it with the number placed in the *root*. If it is the same, this means the presence of a duplicate. Otherwise, we create a new tree node and put the new number in it. The new node is turned into the left child of the *root* node if the second number is less than the one in the *root*. The new node is turned into the right child if the number is greater than the one in the *root*.

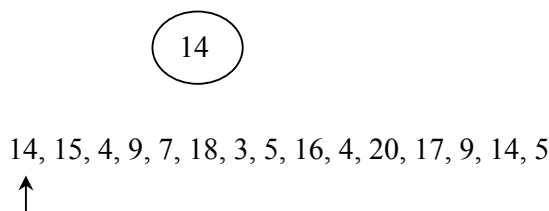


Fig 12.2: First number in the list became the *root*

In the above figure, the first number in the list *14* is placed in a node, making it the *root* of the binary tree. You can see that it is not pointing to any further node. Therefore, its left and right pointers are *NULL* at this point of tree construction time. Now, let's see, how do we insert the next element into it.

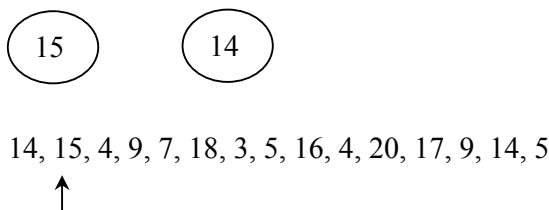


Fig 12.3: A new node is created to insert it in the binary tree

As a first step to add a new node in the tree, we take the next number *15* in the list and compare it with *14*, the number in the *root* node. As they are different, so a new node is created and the number *15* is set into it.

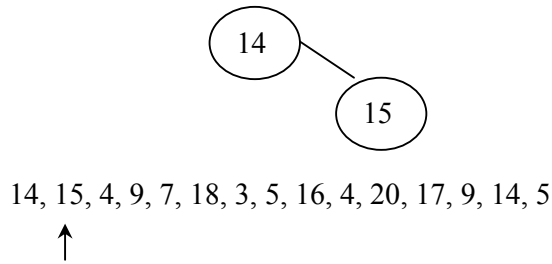


Fig 12.4: The second node is added into the tree

The next step is to add this node in the tree. We compare *15*, the number in the new node with *14*, the number in the *root* node. As number *15* is greater than number *14*, therefore, it is placed as right child of the *root* node.

The next number in the list i.e. *4*, is compared with *14*, the number in the *root* node. As the number *4* is less than number *14*, so we see if there is a left child of the *root* node to compare the number *4* with that. At this point of time in the tree, there is no further left child of the *root* node. Therefore, a new node is created and the number *4* is put into it.

The below figure shows the newly created node.

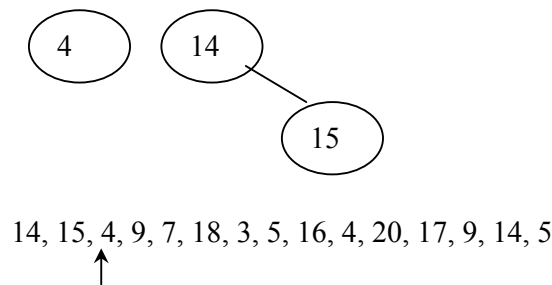


Fig 12.5: A new node is created and number 4 put into it

Next, the newly created node is added as the left child of the *root* node. It is shown in the figure below.

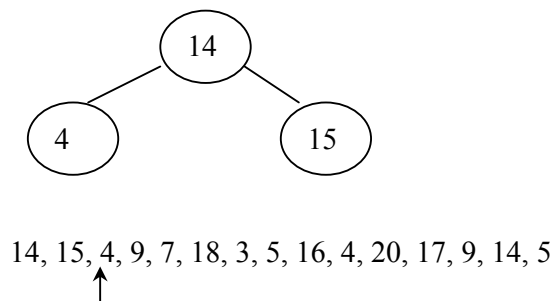


Fig 12.6: The node is added as the left child of the *root* node

The next number in the list is 9. To add this number in the tree, we will follow the already defined and experimented procedure. We compare this number first with the number in the *root* node of the tree. This number is found to be smaller than the number in the *root* node. Therefore, left sub-tree of the *root* node is sought. The left child of the *root* node is the one with number 4. On comparison, number 9 is found greater than the number 4. Therefore, we go to the right child of the node with number 4. At the moment, there is no further node to the right of it, necessitating the need of creating a new node. The number 9 is put into it and the new node is added as the right child of the node with number 4. The same is shown in the figure given below.

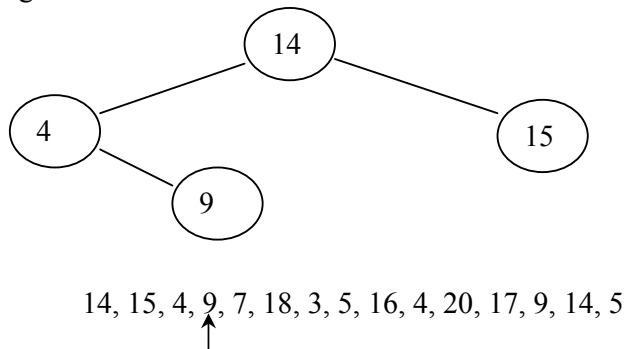


Fig 12.7: A new node is added in the tree

We keep on adding new nodes in the tree in line with the above practiced rule and eventually, we have the tree shown in the below figure.

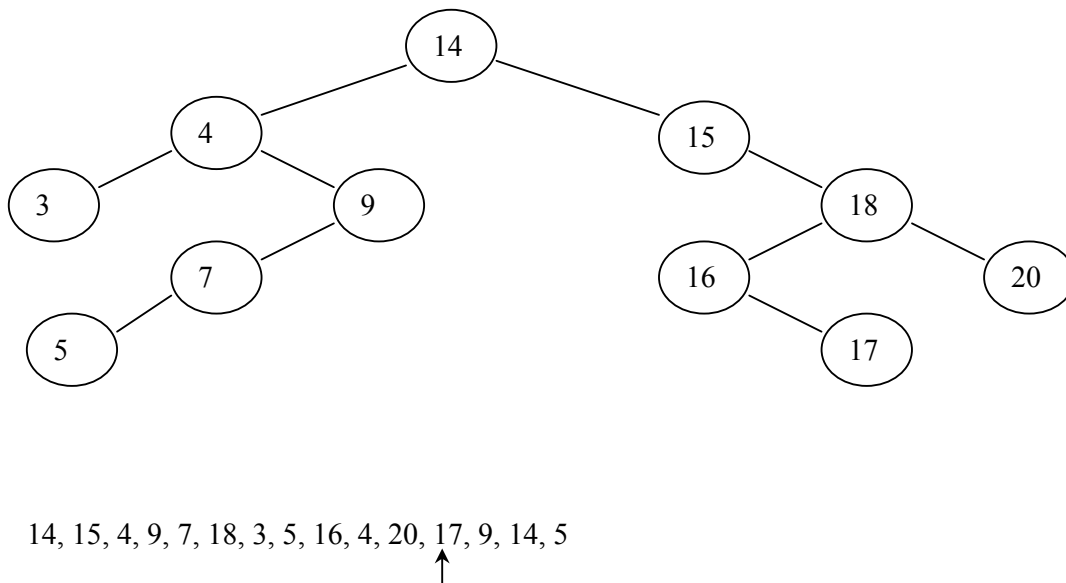


Fig 12.8: Binary tree of whole list of numbers

It is pertinent to note that this is a binary tree with two sub-nodes or children of each node. We have not seen the advantage of binary tree, the one we were earlier talking about i.e. it will reduce the number of comparisons. Previously, we found that search operation becomes troublesome and slower as the size of list grows. We will see the benefit of using binary tree over linked list later. Firstly, we will see how the tree is

implemented.

C++ Implementation of Binary Tree

See the code below for the file *treenode.cpp*.

```
/* This file contains the TreeNode class declaration. TreeNode contains the
functionality for a binary tree node */
1. #include <stdlib.h>
2.
3. template <class Object>
4.
5. class TreeNode
6. {
7.     public:
8.         // constructors
9.         TreeNode()
10.        {
11.            this->object = NULL;
12.            this->left = this->right = NULL;
13.        };
14.
15.        TreeNode( Object * object )
16.        {
17.            this->object = object;
18.            this->left = this->right = NULL;
19.        };
20.
21.        Object * getInfo()
22.        {
23.            return this->object;
24.        };
25.
26.        void setInfo(Object * object)
27.        {
28.            this->object = object;
29.        };
30.
31.        TreeNode * getLeft()
32.        {
33.            return left;
34.        };
35.
36.        void setLeft(TreeNode * left)
37.        {
38.            this->left = left;
39.        };
40.
41.        TreeNode * getRight()
42.        {
43.            return right;
```

```

44.     };
45.
46.     void setRight(TreeNode * right)
47.     {
48.         this->right = right;
49.     };
50.
51.     int isLeaf( )
52.     {
53.         if( this->left == NULL && this->right == NULL )
54.             return 1;
55.         return 0;
56.     };
57.
58.     private:
59.         Object * object;
60.         TreeNode * left;
61.         TreeNode * right;
62. }; // end class TreeNode

```

For implementation, we normally write a class that becomes a factory for objects of that type. In this case too, we have created a class *TreeNode* to be used to create nodes of the binary tree. As we want to use this class for different data types, therefore, we will make it a *template class*, the line 2 is doing the same thing. Inside the class body, we start with the *private* data members given at the bottom of the class declaration. At line 59, the *object* is a *private* data element of *Object **, used to store the tree element (value) inside the node of the tree. *left* is a *private* data member of type *TreeNode**, it is used to store a pointer to the left sub-tree. *right* is a *private* data member of type *TreeNode**, employed to store a pointer to the right sub-tree.

Now, we go to the top of the class declaration to see the *public* functions. At line 9, a public parameter-less constructor is declared. The data members have been initialized in the constructor. At line 11, the *object* data member is initialized to *NULL*. Similarly *left* and *right* data members are initialized to *NULL* at line 12.

There is another constructor declared at line 15- that takes *object* value as a parameter to construct a *TreeNode* object with that object value. While the pointers for *right* and *left* sub-trees are pointing to *NULL*.

At line 21, there is method *getInfo()*, which returns the object i.e. the element of the *TreeNode* object.

At line 26, the method *setInfo(Object *)* sets the value of the *object* data member to the value passed to it as the argument.

The method *getLeft()* returns the pointer to the left sub-tree. Similarly, the *getRight()* returns the right sub-tree. Note that both of these methods return a pointer to the object of type *TreeNode*.

The *setLeft(TreeNode *)* method is used to set the pointer *left* to left sub-tree. Similarly, *setRight(TreeNode *)* is used to set the pointer *right* to right sub-tree. Both of these methods accept a pointer of type *TreeNode*.

The *isLeaf()* method at line 51, is to see whether the current node is a *leaf* node or not. The method returns 1 if it is leaf node. Otherwise, it returns 0.

Using the above *TreeNode*, nodes for the binary tree can be created and linked up together to form a binary tree. We can write a separate method or a class to carry out the node creation and insertion into tree.

Let's use this class by writing couple of functions. Below is the code of main program file containing the *main()* and *insert()* functions.

```

1. #include <iostream>
2. #include <stdlib.h>
3. #include "TreeNode.cpp"
4.
5. int main(int argc, char * argv[])
6. {
7.     int x[] = {14,15,4,9,7,18,3,5,16,4,20,17,9,14,5,-1};
8.     TreeNode<int> * root = new TreeNode<int>();
9.     root->setInfo( &x[0] );
10.    for(int i = 1; x[i] > 0; i++)
11.    {
12.        insert( root, &x[i] );
13.    }
14. }
15.
16. void insert (TreeNode<int> * root, int * info)
17. {
18.     TreeNode<int> * node = new TreeNode<int> (info);
19.     TreeNode<int> * p, * q;
20.     p = q = root;
21.     while( *info != *(p->getInfo()) && q != NULL )
22.     {
23.         p = q;
24.         if( *info < *(p->getInfo()) )
25.             q = p->getLeft();
26.         else
27.             q = p->getRight();
28.     }
29.
30.     if( *info == *(p->getInfo()) )
31.     {
32.         cout << "attempt to insert duplicate: " << *info << endl;
33.         delete node;
34.     }
35.     else if( *info < *(p->getInfo()) )
36.         p->setLeft( node );
37.     else
38.         p->setRight( node );
39. } // end of insert

```

We have used the same list of numbers for discussion in this lecture. It is given at line 7 in the code. It is the same, only the last number is *-1*. This is used as delimiter or marker to indicate that the list has finished.

At line 8, we are creating a new *TreeNode* object i.e. a *root* node as the name implies.

This node will contain an *int* type element as evident from the syntax.

At line 9, the first number in the list is set into the *root* node. At line 10, the *for* loop is started, which is inserting all the elements of the list one by one in the tree with the use of the *insert()* function. Most of the time, this loop will be reading the input numbers from the users interactively or from a file. If it is a Windows application then a form can be used to take input from the user. In this implementation, our objective is to insert numbers in the tree and see the duplicates, so hard coding the numbers within our program will serve the purpose.

The *insert()* method starts from line 16. It is accepting two parameters. The first parameter is pointer to a *TreeNode* object, containing a value of *int* type while second is the *info* that is an *int **.

In the first line of the function, line 18, a new node has been created by calling the parameterized constructor of the *TreeNode* class.

Then at line 19, two pointer variables *p* and *q* are declared.

In line 20, we are initializing variables *p* and *q* to the *root* node.

In line 21, the *while* loop is being started, inside the *while* loop we are checking the equality of the number inside the node (being pointed to by pointer *p*) with the number being passed. The control is entered into the loop, if both the numbers are not equal and *q* is not pointing to *NULL*. This *while* loop will be terminated if the numbers in the two nodes are equal or end of a sub-tree is reached.

At line 23 inside the loop, *p* is assigned the value of *q*.

At line 24 inside the loop, if the number inside the node is smaller than the number in the node pointed to by the pointer *p*. Then at line 25, we are getting the left sub-tree address (*left*) and assigning it to the pointer *q*.

Otherwise, if this is not the case, means the number in the node to be inserted is not smaller than the number in the node pointed to by the pointer *p*. Then at line 27, the right sub-tree address is retrieved and assigned to pointer *q*.

At line 30, the comparison is made to see if the both the values are equal (number in the node to be inserted and the number inside the node pointed to by the pointer *p*). In case they are equal, it displays a message with the number informing that a duplicate number has been found. This also means that the upper *while* loop was terminated because of equality of the two numbers. In line 33, the newly created node is deleted afterwards.

At line 35, we are checking if the number in the newly constructed node is less than the number in the node at the end of a sub-tree. If this is so then the newly constructed node is inserted to the left of the tree node. The insertion code is at line 36.

If the number in the newly constructed node is greater than the number in the tree node, the newly constructed node will be inserted to the right of the tree node as shown on line 38. To make it clearer, let's see the same thing, the *insert()* method pictorially.

Trace of insert

We will take the tree and the figure, we constructed above. At this time, we want to insert some new numbers in the tree as given in the figure below:

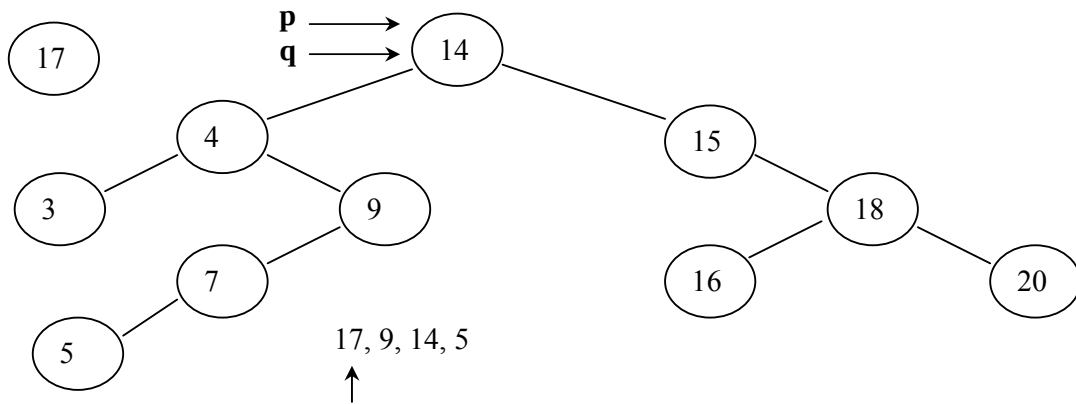


Fig 12.9: Start of insertion of new node in the tree

Initially the pointers p and q are pointing to the start (*root*) of the tree. We want to insert the number 17 in the tree. We compare the number in the root node (14) with number 17. Because number 17 is greater than the number 14, so as we did in the while loop in the function above, we will move toward the right sub-tree. In the next picture below, we can see that the pointer q has moved to the right sub-tree of the *root*.

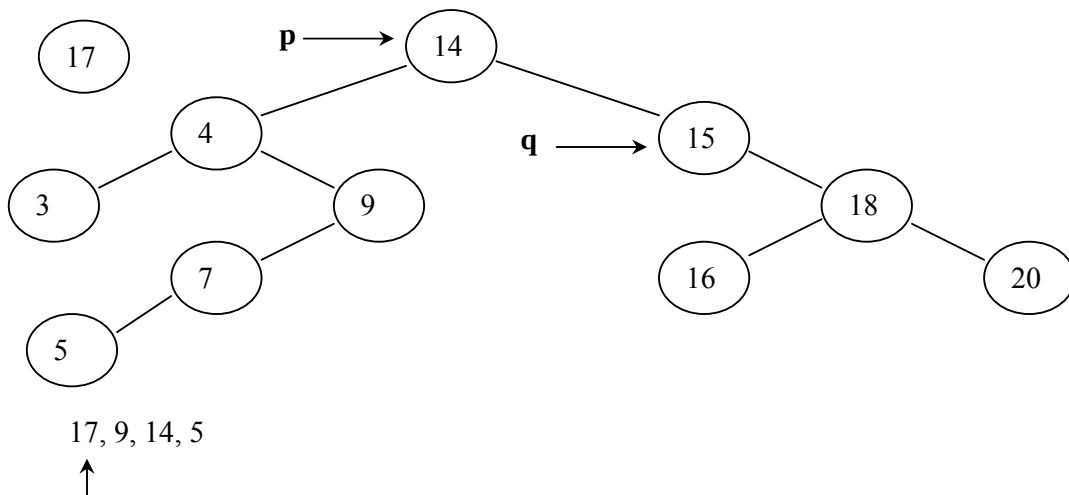


Fig 12.10: Insertion of a new node in progress

After moving the pointer q forward, we make the pointer p point to the same node. We do this operation as the first step inside the while loop. It can be seen at line 23 above. So following will be the latest position of pointers in the tree.

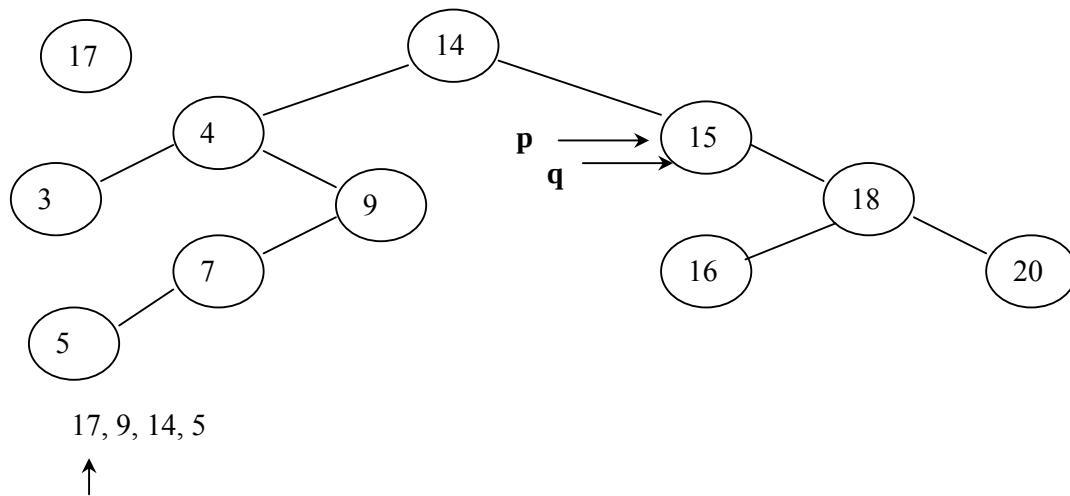


Fig 12.11: Insertion of a new node in progress

Now, the number 15 in the tree node is compared with new number 17. Therefore, the pointer q is again moved forward to the right child node i.e. the node with number 18. In the next step, we move forward the p pointer also. The following figure depicts the current picture of the tree:

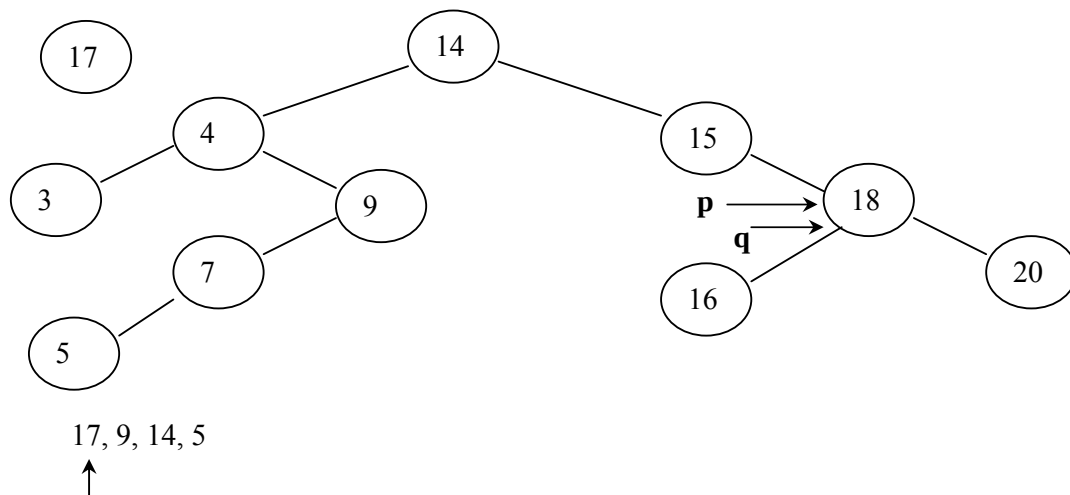


Fig 12.12: Insertion of a new node in progress

The previous process is repeated again (*while* loop is the repeating construct) that the number 17 is compared with the number 18. This time the left child node is traversed and q pointer starts pointing the node with number 16 inside. In the next step, p is moved forward to point to the same node as q .

The same comparison process starts again, number 17 is found to be greater than the number 16, therefore, the right child is seek. But we see that the current tree node does not have right child node, so it returns NULL. Following figure depicts it.

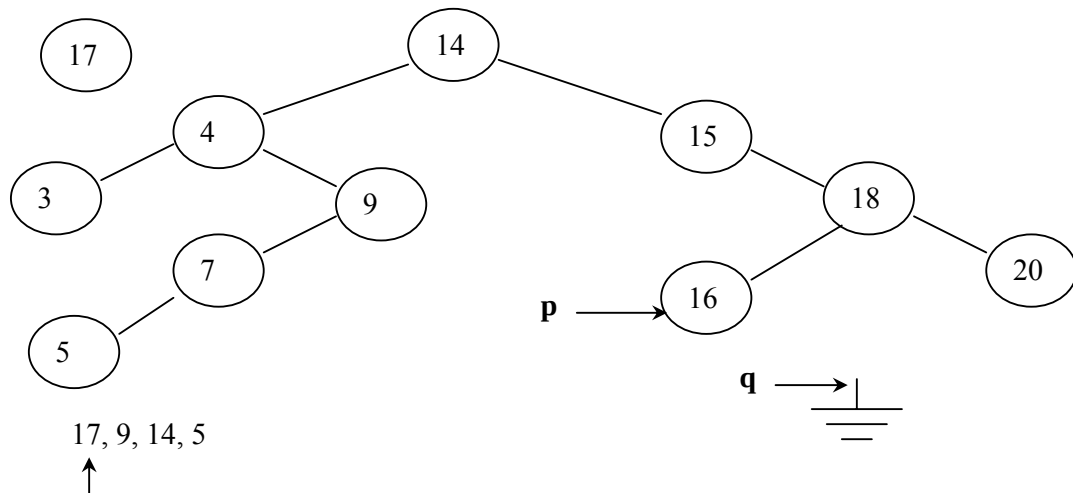


Fig 12.13: Insertion of a new node in progress

Above shown (q is pointing to *NULL*) is the condition that causes the *while* loop in the code above to terminate. Later we insert the new node as the right child of the current node.

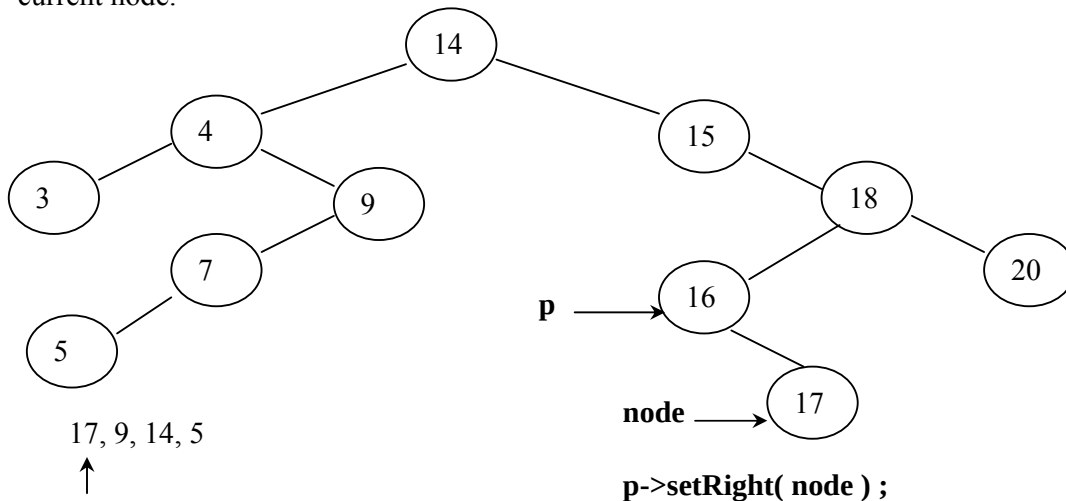


Fig 12.14: A new node is inserted in the tree

It is recommended to execute this algorithm manually to insert the remaining numbers in the tree. This will make the understanding more clear. This tree algorithm is going to be used rigorously in the future and complete understanding of it will make the complex future implementations easier to comprehend. The specific advantages of using binary tree over linked list will be discussed in more detail in the next lecture.

Data Structures

Lecture No. 13

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4

4.3, 4.6

Summary

- Cost of Search
- Binary Search Tree (BST)
- Traversing a Binary Tree
- C++ code
- Example
- Exercise

In the previous lecture, we had written and demonstrated the use of C++ code of *insert* routine for a tree through an example. We also saw how a new node is inserted into a binary tree. If the to-be-inserted number (node) is already in the tree i.e. it matches a number already present in the tree, we display a message that the number is already in the tree. In the last lecture, the advantages of the tree data structure vis-à-vis linked list data structure were also discussed. In a linked list, a programmer has to search the whole list to find out a duplicate of a number to be inserted. It is very tedious job as the number of stored items in a linked list is very large. But in case of tree data structure, we get a dynamic structure in which any number of items as long as memory is available, can be stored. By using tree data structure, the search operation can be carried out very fast. Now we will see how the use of binary tree can help in searching the duplicate number in a very fast manner.

Cost of Search

Consider the previous example where we inserted the number 17 in the tree. We executed a *while* loop in the *insert* method and carried out a comparison in *while* loop. If the comparison is true, it will reflect that in this case, the number in the node where the pointer *p* is pointing is not equal to 17 and also *q* is not NULL. Then we move *p* actually *q* to the left or right side. This means that if the condition of the *while* loop is true then we go one level down in the tree. Thus we can understand it easily that if there is a tree of 6 levels, the while loop will execute maximum 6 times. We conclude from it that in a given binary tree of depth *d*, the maximum number of executions of the *while* loop will be equal to *d*. The code after the *while* loop will do the process depending upon the result of the *while* loop. It will insert the new number or display a message if the number was already there in the tree.

Now suppose we have another method *find*. This method does not insert a new number in the tree. Rather, it traverses a tree to find if a given number is already present in the tree or not. The tree which the *find* method traverses was made in such an order that all the numbers less than the number at the root are in the left sub-tree of the root while the right sub-tree of the root contains the numbers greater than the number at the root. Now the *find* method takes a number x and searches out its duplicate in the given tree. The *find* method will return true if x is present in the tree. Otherwise, it will return false. This *find* method does the same process that a part of the *insert* method performs. The difference is that the *insert* method checks for a duplicate number before putting a number in the tree whereas the *find* method only finds a number in the tree. Here in the *find* method, the *while* loop is also executed at maximum equal to the number of levels of the tree. We do a comparison at each level of the tree until either x is found or q becomes NULL. The loop terminates in case, the number is found or it executes to its maximum number, i.e. equal to the number of levels of the tree.

In the discussion on binary tree, we talked about the level and number of nodes of a binary tree. It was witnessed that if we have a complete binary tree with n numbers of nodes, the depth d of the tree can be found by the following equation:

$$d = \log_2 (n + 1) - 1$$

Suppose we have a complete binary tree in which there are 100,000 nodes, then its depth d will be calculated in the following fashion.

$$d = \log_2 (100000 + 1) - 1 = \log_2 (100001) - 1 = 20$$

The statement shows that if there are 100,000 unique numbers (nodes) stored in a complete binary tree, the tree will have 20 levels. Now if we want to find a number x in this tree (in other words, the number is not in the tree), we have to make maximum 20 comparisons i.e. one comparison at each level. Now we can understand the benefit of tree as compared to the linked list. If we have a linked list of 100,000 numbers, then there may be 100,000 comparisons (supposing the number is not there) to find a number x in the list.

Thus in a tree, the search is very fast as compared to the linked list. If the tree is complete binary or near-to-complete, searching through 100,000 numbers will require a maximum of 20 comparisons or in general, approximately $\log_2(n)$. Whereas in a linked list, the comparisons required could be a maximum of n .

Tree with the linked structure, is not a difficult data structure. We have used it to allocate memory, link and set pointers to it. It is not much difficult process. In a tree, we link the nodes in such a way that it does not remain a linear structure. If instead of 100,000, we have 1 million or 10 million or say, 1 billion numbers and want to build a complete binary tree of these numbers, the depth (number of levels) of the tree will be $\log_2$ of these numbers. The $\log_2$ of these numbers will be a small number, suppose 25, 30 or 40. Thus we see that the number of level does not increase in such a ratio as the number of nodes increase. So we can search a number x in a complete binary tree of 1 billion nodes only in 30-40 comparisons. As the linked list of such a large number grows large, the search of a number in such a case will also get time consuming

process. The usage of memory space does not cause any effect in the linked list and tree data structures. We use the memory dynamically in both structures. However, time is a major factor. Suppose one comparison takes one micro second, then one billion seconds are required to find a number from a linked list (we are supposing the worst case of search where traversing of the whole list may be needed). This time will be in hours. On the other hand, in case of building a complete binary tree of these one billion numbers, we have to make 30-40 comparisons (as the levels of the tree will be 30-40), taking only 30-40 microseconds. We can clearly see the difference between hours and microseconds. Thus it is better to prefer the process of building a tree of the data to storing it in a linked list to make the search process faster.

Binary Search Tree

While discussing the search procedure, the tree for search was built in a specific order. The order was such that on the addition of a number in the tree, we compare it with a node. If it is less than this, it can be added to the left sub-tree of the node. Otherwise, it will be added on the right sub-tree. This way, the tree built by us has numbers less than the root in the left sub-tree and the numbers greater than the root in the right sub-tree. A binary tree with such a property that items in the left sub-tree are smaller than the root and items in the right sub-tree are larger than the root is called a *binary search tree* (BST). The searching and sorting operations are very common in computer science. We will be discussing them many times during this course. In most of the cases, we sort the data before a search operation. The building process of a binary search tree is actually a process of storing the data in a sorted form. The BST has many variations, which will be discussed later. The BST and its variations play an important role in searching algorithms. As data in a BST is in an order, it may also be termed as ordered tree.

Traversing a Binary Tree

Now let's discuss the ways to print the numbers present in a BST. In a linked list, the printing of stored values is easy. It is due to the fact that we know wherefrom, a programmer needs to start and where the next element is. Equally is true about printing of the elements in an array. We execute a *for* loop starting from the first element (i.e. index 0) to the last element of the array. Now let's see how we can traverse a tree to print (display) the numbers (or any data items) of the tree.

We can explain this process with the help of the following example in which we traverse a binary search tree. Suppose there are three nodes tree with three numbers stored in it as shown below.

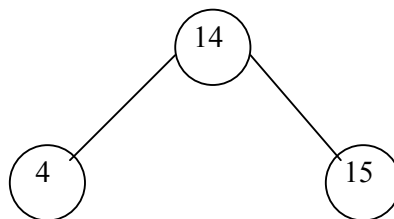


Fig 13.1: A three node binary tree

Here we see that the root node has the number 14. The left sub-tree has only one node i.e. number 4. Similarly the right sub-tree consists of a single node with the number 15. If we apply the permutations combinations on these three nodes to print them, there may be the following six possibilities.

- 1: (4, 14, 15)
- 2: (14, 4, 15)
- 3: (15, 4, 14)
- 4: (4, 15, 14)
- 5: (14, 15, 4)
- 6: (15, 14, 4)

Look at these six combinations of printing the nodes of the tree. In the first combination, the order of printing the nodes is 4-14-15. It means that left subtree-root-right subtree. In the second combination the order is root-left subtree-right subtree. In the third combination, the order of printing the nodes is right subtree-root-left subtree. The fourth combination has left subtree-right subtree-root. The fifth combination has the order root-right subtree-left subtree. Finally the sixth combination has the order of printing the nodes right subtree-root-left subtree. These six possibilities are for the three nodes only. If we have a tree having a large number of nodes, then there may increase number of permutations for printing the nodes.

Let's see the general procedure of traversing a binary tree. We know by definition that a binary tree consists of three sets i.e. root, left subtree and right subtree. The following figure depicts a general binary tree.

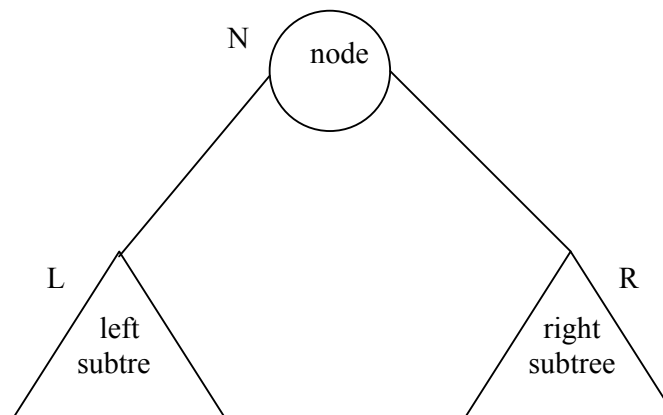


Fig 13.2: A generic binary tree

In this figure, we label the root node with N . The left subtree in the figure is in a triangle labeled as L . This left subtree may consist of any number of nodes. Similarly the right subtree is enclosed in a triangle having the label R . This triangle of right subtree may contain any number of nodes. Following are the six permutations, which we have made for the three nodes previously. To generalize these permutations, we use N , L and R as abbreviations for root, left subtree and right subtree respectively.

- 1: (L, N, R)
- 2: (N, L, R)
- 3: (R, L, N)
- 4: (L, R, N)
- 5: (N, R, L)
- 6: (R, N, L)

In these permutations, the left and right subtrees are not single nodes. These may consist of several nodes. Thus where we see *L* in the permutations, it means traversing the left subtree. Similarly *R* means traversing the right subtree. In the previous tree of three nodes, these left and right subtrees are of single nodes. However, they can consist of any number of nodes. We select the following three permutations from the above six. The first of these three is (N, L, R), also called as preorder traversal. The second permutation is (L, N, R) which we called inorder traversal. Finally the third permutation, also termed as postorder traversal is (L, R, N). Now we will discuss these preorder, inorder and postorder traversal in detail besides having a look on their working. We will also see the order in which the numbers in the tree are displayed by these traversing methods.

C++ code

Let's write the C++ code for it. Following is the code of the *preorder* method.

```
void preorder(TreeNode<int>* treeNode)
{
    if( treeNode != NULL )
    {
        cout << *(treeNode->getInfo())<<" ";
        preorder(treeNode->getLeft());
        preorder(treeNode->getRight());
    }
}
```

In the arguments, there is a pointer to a *TreeNode*. We may start from any node and the pointer of the node will be provided as argument to the *preorder* method. In this method, first of all we check whether the pointer provided is *NULL* or not. If it is not *NULL*, we print the information stored in that node with the help of the *getInfo()* method. Then we call the *getLeft()* method that returns a pointer of left node, which may be a complete subtree. With the help of this method, we get the root of that subtree. We call the *preorder* method again passing that pointer. When we return from that, the *preorder* method is called for the right node. Let's see what is happening in this method. We are calling the *preorder* method within the *preorder* method. This is actually a recursive call. Recursion is supported in C++ and other languages. Recursion means that a function can call itself. We may want to know why we are doing this recursive call. We will see some more examples in this regard and understand the benefits of recursive calls. For the time being, just think that we are provided with a tree with a root pointer. In the preorder, we have to print the value of the root node. Don't think that you are in the preorder method. Rather keep in mind

that you have a preorder function. Suppose you want to print out the left subtree in the preorder way. For this purpose, we will call the preorder function. When we come back, the right subtree will be printed. In the right subtree, we will again call the preorder function that will print the information. Then call the preorder function for the left subtree and after that its right subtree. It will be difficult if you try to do this incursion in the mind. Write the code and execute it. You must be knowing that the definition of the binary tree is recursive. We have a node and left and right subtrees. What is left subtree? It is also a node with a left subtree and right subtree. We have shown you how the left subtree and right subtree are combined to become a tree. The definition of tree is itself recursive. You have already seen the recursive functions. You have seen the factorial example. What is factorial of N? It is N multiplied by N-1 factorial. What is N-1 factorial? N-1 factorial is N-1 multiplied by N-2 factorial. This is the recursive definition. For having an answer, it is good to calculate the factorial of one less number till the time you reach at the number 2. You will see these recursions or recursive relations here and also in mathematic courses. In the course of discrete mathematics, recursion method is used. Here we are talking about the recursive calls. We will now see an example to understand how this recursive call works and how can we traverse the tree using the recursion. One of the benefits of recursion that it prints out the information of all the nodes without caring for the size of the tree. If the tree has one lakh nodes, this simple four lines routine will print all the nodes. When compared with the array printing, we have a simple loop there. In the link list also, we have a small loop that executes till the time we have a next pointer as NULL. For tree, we use recursive calls to print it.

Here is the code of the *inorder* function.

```
void inorder(TreeNode<int>* treeNode)
{
    if( treeNode != NULL )
    {
        inorder(treeNode->getLeft());
        cout << *(treeNode->getInfo())<<" ";
        inorder(treeNode->getRight());
    }
}
```

The argument is the same as in the preorder i.e. a pointer to the *TreeNode*. If this node is not NULL, we call *getLeft()* to get the left node and call the *inorder* function. We did not print the node first here. In the *inorder* method, there is no need to print the root tree first of all. We start with the left tree. After completely traversing the complete left tree, we print the value of the node. In the end, we traverse the right subtree in recursion.

Hopefully, you have now a fair knowledge about the postorder mechanism too. Here is the code of the postorder method.

```
void postorder(TreeNode<int>* treeNode)
{
    if( treeNode != NULL )
    {
        postorder(treeNode->getLeft());
        postorder(treeNode->getRight());
        cout << *(treeNode->getInfo())<<" ";
    }
}
```

In the postorder, the input argument is a pointer to the *TreeNode*. If the node is not NULL, we traverse the left tree first. After that we will traverse the right tree and print the node value from where we started.

As all of these above routines are function so we will call them as:

```
cout << "inorder: ";
preorder( root);

cout << "inorder: ";
inorder( root );

cout << "postorder: ";
postorder( root );
```

Here the root represents the root node of the tree. The size of the tree does not matter as the complete tree will be printed in *preorder*, *inorder* and *postorder*. Let's discuss an example to see the working of these routines.

Example

Let's have a look on the following tree.

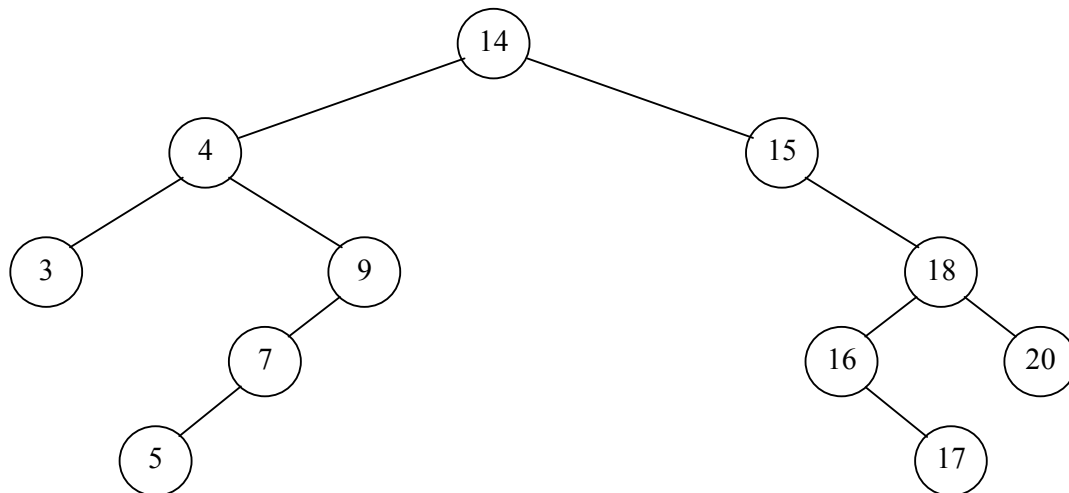


Fig 13.3: Preorder: 14 4 3 9 7 5 15 18 16 17 20

This is the same tree we have been using previously. Here we want to traverse the tree. In the bottom of the figure, the numbers are printed with the help of preorder method. These numbers are as 14 4 3 9 7 5 15 18 16 17 20. Now take these numbers and traverse the tree. In the preorder method, we print the root, followed by traversing of the left subtree and the right subtree respectively. As the value of the root node is 14, so it will be printed first of all. After printing the value of the root node, we call the preorder for the left node which is 4. Forget the node 14 as the root is 4 now. So the value 4 is printed and we call the preorder again with the left sub tree i.e. the node with value 3. Now the root is the node with value 3. We will print its value before going to its left. The left side of node with value 3 is NULL. Preorder will be called if condition is false. In this case, no action will be taken. Now the preorder of the left subtree is finished. As the right subtree of this node is also NULL, so there is no need of any action. Now the left subtree of the node with value 4 is complete. The method preorder will be called for the right subtree of the node with value 4. So we call the preorder with the right node of the node with value 4. Here, the root is the node with value 9 that is printed. We will call its left subtree where the node value is 7. It will be followed by its left subtree i.e. node 5 which will be printed.

In the preorder method, we take the root i.e. 14 in this case. Its value is printed, followed by its left subtree and so on. This activity takes us to the extreme left node. Then we back track and print the right subtrees.

Let's try to understand the inorder method from the following statement.

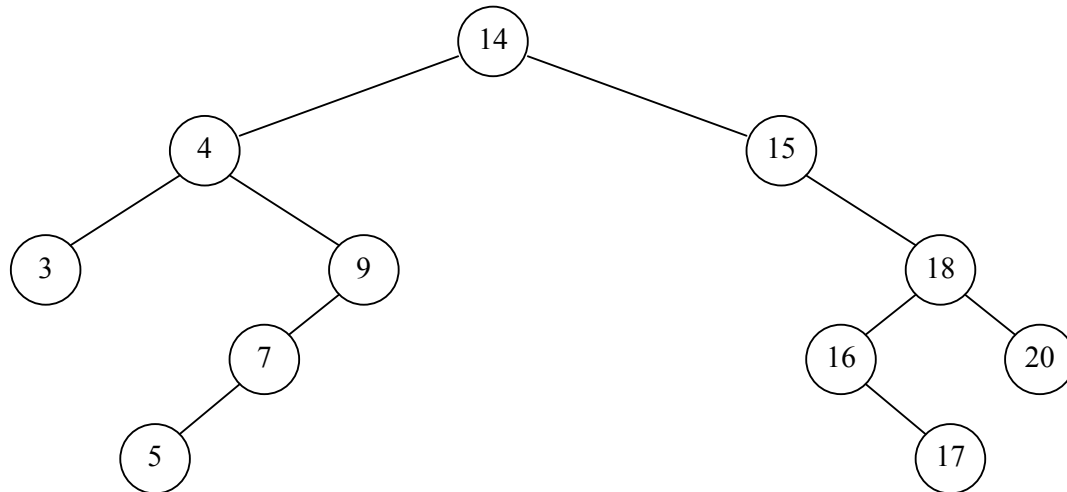


Fig 13.4: Inorder: 3 4 5 7 9 14 15 16 17 18 20

When we call the inorder function, the numbers will be printed as 3 4 5 7 9 14 15 16 17 18 20. You might have noted that these numbers are in sorted order. When we build the tree, there was no need of sorting the numbers. But here we have the sorted numbers in the inorder method. While inserting a new node, we compare it with the existing ones and place the new node on the left or right side. So during the process of running the inorder on the binary tree, a programmer gets the numbers sorted. Therefore we have a sorting mechanism. If we are given some number, it will not be difficult to get them sorted. This is a new sorting algorithm for you. It is very simple. Just make BST with these numbers and run the inorder traversal. The numbers obtained from the inorder traversal are sorted.

In the inorder method, we do not print the value of node first. We go to traverse its left subtree. When we come back from the left subtree and print the value of this node. Afterwards, we go to the right subtree. Now consider the above tree of **figure 13.4**. If we start from number 14, it will not be printed. Then we will go to the left subtree. The left subtree is itself a tree. The number 4 is its root. Now being at the node 4, we again look if there is any left subtree of this node. If it has a left subtree, we will not print 4 and call the inorder method on its left subtree. As there is a left subtree of 4 that consists a single node i.e. 3, we go to that node. Now we call the inorder of node 3 to see if there is a subtree of it. As there is no left subtree of 3, the *if* statement that checks if the node is not NULL will become false. Here the recursive calls will not be executed. We will come back from the call and print the number 3. Thus the first number that is printed in the inorder traversal is 3. After printing 3, we go to its right subtree that is also NULL. So we come back to node 3. Now as we have traversed the left and right subtrees of 3 which itself is a left subtree of 4, thus we have traversed the left subtree of 4. Now we print the value 4 and go to the right subtree of 4. We come at node 9. Before printing 9 we go to its left subtree that leads us to node 7. In this subtree with root 7, we will not print 7. Now we will go to its left subtree which is node 5. We look for the left subtree of 5 which is NULL. Now we print the value 5. Thus, we have so far printed the numbers 3, 4 and 5. Now we come back to 7. After traversing its left subtree, we print the number 7. The right subtree of

7 is NULL. Thus finally, we have traversed the whole tree whose root is 7. It is a left subtree of 9. Now, we will come back to 9 and print it (as its left subtree has been traversed). The right subtree of 9 is NULL. So there is no need to traverse it. Thus the whole tree having 9 as its root has been traversed. So we come back to the root 4, the right subtree of which is the tree with root 9. Resultantly, the left and right subtree of 4 have been traversed now. Thus we have printed the numbers 3, 4, 5, 7 and 9. Now from here (i.e. node 4) we go to the node 14 whose left subtree has been traversed. Now we print 14 and go to its right subtree. The right subtree of 14 has 15 as the root. From here, we repeat the same process earlier carried out in case of the left subtree of 14. As a result of this process, the numbers 15, 16, 17, 18 and 20 are printed. Thus we get the numbers stored in the tree in a sorted order. And we get the numbers printed as 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20. This sorted order is only in the inorder traversal.

When we apply the post order traversal on the same tree, the numbers we get are not in sorted order. The numbers got through postorder traversal are in the following order:

3 5 7 9 4 17 16 20 18 15 14

We see that these are not in a sorted order.

We know that every function has a signature. There is return type, function name and its parameter list in the signature of a function. Then there comes the body of the function. In the body of a function, there may be local variables and calls to some other functions. Now if a statement in the body of a function is calling the same function in which it exists, it will be termed as a recursive call. For example if a statement in function *A* is calling the function *A*, it is a recursive call and the process is called recursion.

Now let's see how recursion is carried out. We know that the function calls are implemented through stacks at run time. Let's see what happens when there is a recursive call. In a recursive call that means calling a function itself makes no difference as far as the call stack is concerned. It is the same as a function *F* calls a function *G*, only with the difference that now function *F* is calling to function *F* instead of *G*. The following figure shows the stack layout when function *F* calls function *F* recursively.

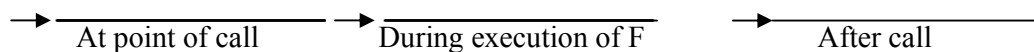
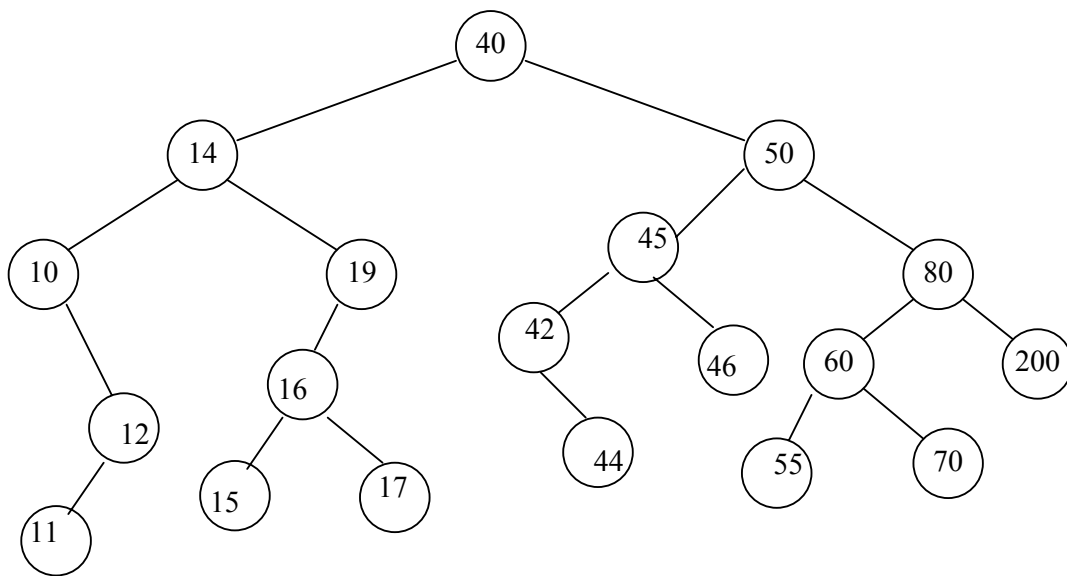


Fig 13.5

When F is called first time, the parameters, local variables and its return address are put in a stack, as some function has called it. Now when F calls F , the stack will increase the parameters, local variables and return address of F comes in the stack again. It is the second call to F . After coming back from this second call, we go to the state of first call of F . The stack becomes as it was at first call. In the next lecture, we will see the functionality of recursive calls by an example.

Exercise

Please find out the preorder, inorder and postorder traversal of the tree given below:

**Fig 13.6**

Data Structures

Lecture No. 14

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4

4.3, 4.6

Summary

- Recursive Calls
- Preorder Recursion
- Inorder Recursion
- Non-Recursive Traversal
- Traversal Trace

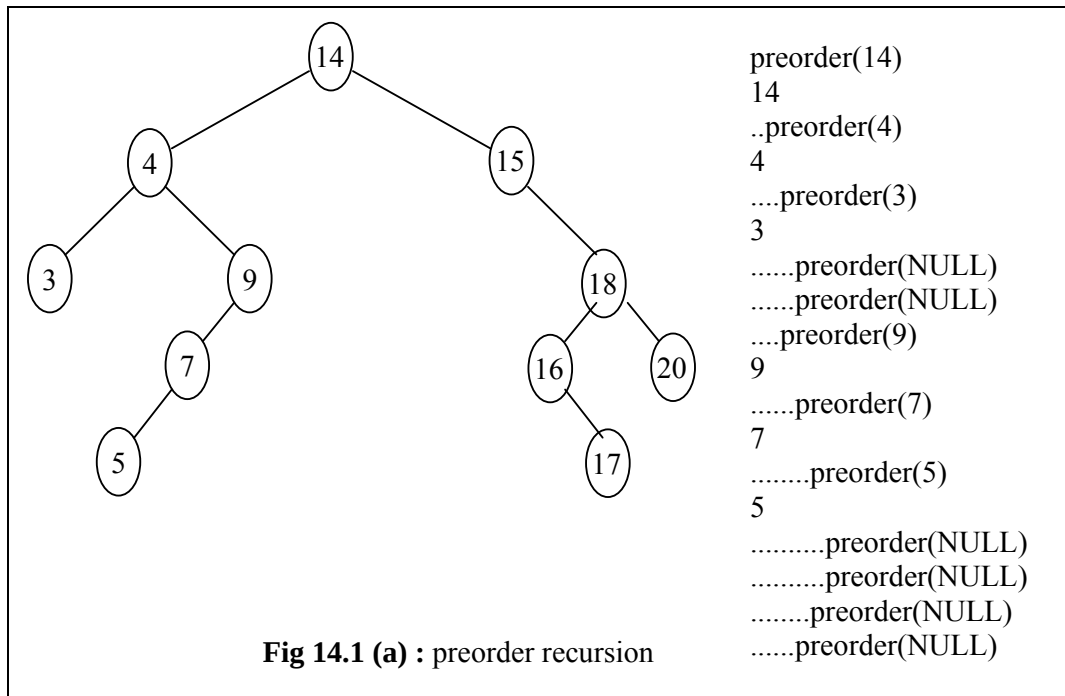
We discussed the methods of traversal of the binary tree in the previous lecture. These methods are- preorder, inorder and postorder. It was witnessed that in the C++ code that the coding of these methods becomes very short with the use of recursive calls. The whole tree can be traversed with a few lines of code. We also demonstrated the benefits of the methods with the help of an example in which a tree was traversed by preorder, inorder and postorder methods. Implementation of the recursion also came under discussion in the previous lecture.

Recursive Calls

We know that function calls are made with the help of stacks. When a function calls some other function, the parameters and return address of the function is put in a stack. The local variables of the function are also located at the stack and the control is passed to the called function. When the called function starts execution, it performs its job by using these parameters and local variables. When there in the function there comes a return statement or when the function ends then the return address, already stored in the stack, is used and control goes back to the next statement after the calling function. Similarly when a function is calling to itself, there is no problem regarding the stack. We know, in recursion a function calls to itself instead of calling some other function. Thus the recursion is implemented in the way as other function calls are implemented.

Preorder Recursion

Now let's see the preorder traversal with respect to the recursion. We will see what changes happen when preorder function calls itself recursively. Consider the following binary search tree used in our previous example.



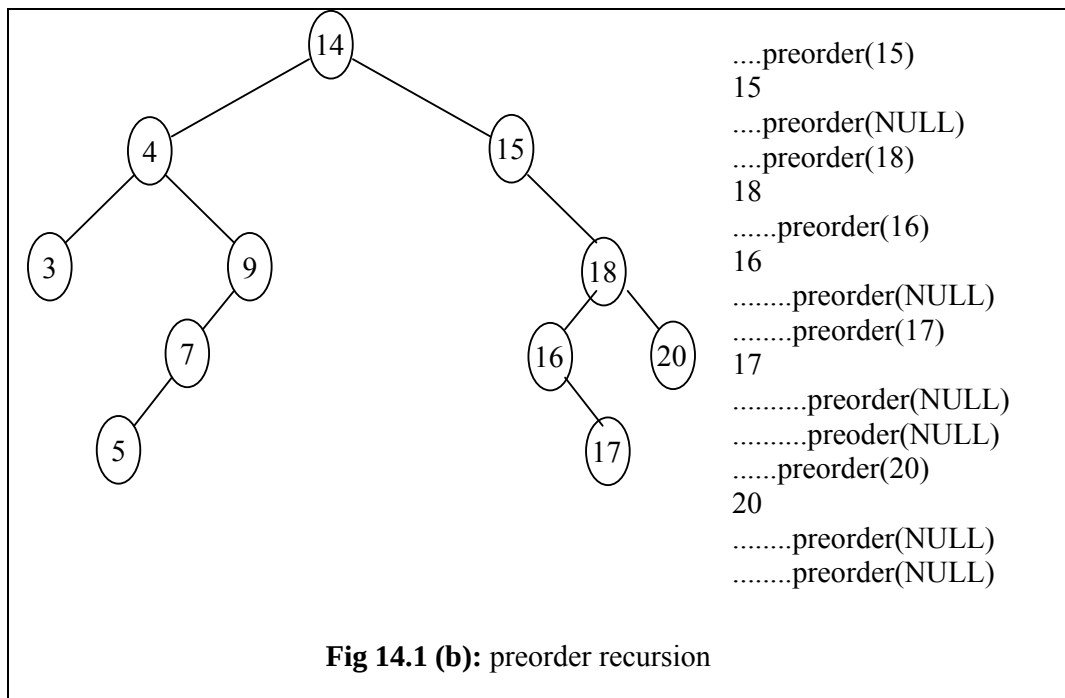
For the preorder traversal, we call the preorder function and pass it the root node of the tree (i.e. 14) as a parameter. We have discussed in the preorder traversing that preorder method first prints the value of the node that is passed to it. So number 14 is printed. After it, the preorder method calls itself recursively. Thus we will call the preorder and give it the left subtree of 14 (actually the root node of left subtree) as an argument. So this call will be *preorder (4)*. As it is preorder traversing, the value of the root node i.e. 4 will be printed. Now the preorder method looks if there is a left subtree of the node (i.e.4). If it has a left subtree, the preorder function will call itself again. The root node of this left subtree will be passed as an argument to the new function call. Here it is 3, so the number 3 is printed. Thus by the previous three calls (i.e. *preorder(14)*, *preorder(4)* and *preorder(3)*) we have printed the values 14, 4 and 3. Now the left subtree of 3 is NULL. We know that in preorder method, first of all we check whether the node is NULL or not. If it is not NULL, the recursion process continues. It means that we print the value of the node and call the preorder function again. If the node is NULL, the process ends.

There is always a terminating condition in recursive call or recursive procedure. In the previous lecture, while trying to end the factorial recursion, we defined the condition that the factorial of zero is 1. When the factorial of n starts, the factorial is called again and again by decreasing the value of n by 1. So when we reach at 0, there is no further recursive call for the factorial as we have set the condition for it i.e. the factorial of 0 is equal to 1. Thus the recursion ends. Equally is true about the preorder process. When a node is NULL, then 'if statement' becomes false and we exit the function (see code of preorder function in previous lecture).

In the above figure, the preorder process that started from node 14, came to an end at node 3 as the left subtree of 3 is NULL. Now this preorder process will make no further recursive call. We will come back from here by using the return address of the

call *preorder (NULL)* from the stack. If we see the stack, this fourth call will come back to the third call i.e. *preorder (3)*. The left subtree of 3 has ended. Now we have to traverse the right subtree of 3. Here the right subtree of 3 is also NULL. So the traversing of tree with node 3 as root has been completed. Now we return from the call *preorder (3)* and reach one level backward at the node 4. The left subtree of node 4 has been traversed. Now we go to traverse the right subtree of 4. The preorder call for the right subtree has started. This started from the call *preorder (9)*. Note in the figure that this call has made at the same level as was for node 3. It is because we are traversing the right subtree of the same tree that has node 4 as its root and 3 was its left subtree. After printing the number 9 we go to call the preorder process for the left subtree of 9. That is *preorder (7)*. When we go to 7, we print it and call the preorder process for its left subtree i.e. *preorder (5)*. Now after printing the number 5, the preorder is applied to its left subtree i.e. NULL. So the preorder traversal of left subtree of 5 ends this way. Now the preorder is applied to the right subtree of 5 which is also NULL. Thus the next two preorder calls (one for left subtree of 5 and one for right subtree of 5) after the call *preorder (5)* are *preorder (NULL)*. Here the returning address takes the control back to the call *preorder (7)* through the use of stack. The left subtree of 7 has been traversed and now we go to the right subtree of it i.e. NULL. Thus there is a call *preorder (NULL)*. So we come back to node 9 completing the traversal of its left subtree. Now we traverse the right subtree of 9 which again happens to be NULL. So there is one more call of preorder with NULL. The number of dots with the preorder calls in the figure describes the state of the stack. When we go ahead to traverse a node the dots increase in figure as the call made is put on the stack. When we come back to a node, the returning address is popped from the stack and we mention it by decreasing a dot in the figure.

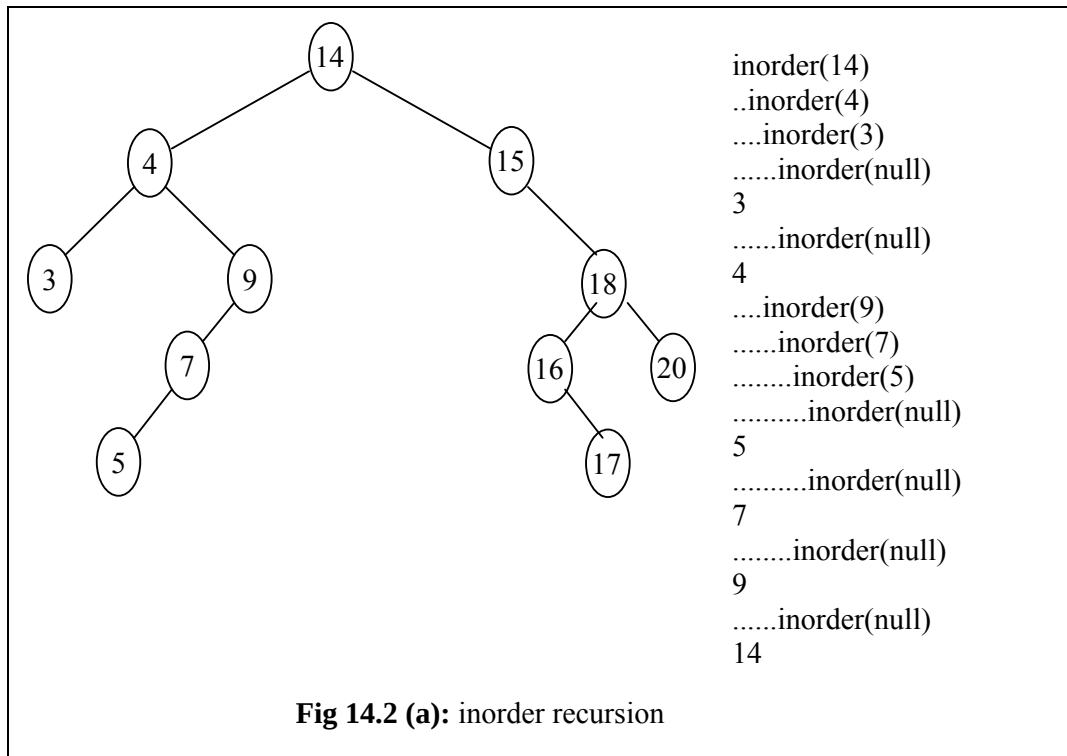
After these four calls of preorder with NULL as argument, we return to the node 4. We have completed the tree (left and right subtree) with node 4 as root. As seen in the figure, this tree (having node 4 as root) is a left subtree of node 14. The numbers that we got printed upto now are 14, 4, 3, 9, 7 and 5. So after traversing the left subtree of 14, we come back to node 14 and start to traverse its right subtree. We traverse the right subtree by applying the recursive calls of preorder in the way as done while traversing the left subtree of 14. The recursive calls involved in it are shown in the following figure.



Now we are at node 14 and know that in preorder, the value of the node is printed first before going to its left subtree. And after traversing the left subtree, a programmer comes back and goes to traverse the right subtree. As we have traversed the left subtree of 14 and have come back to 14, now we will traverse the right subtree of 14. In the right subtree, we reach at node 15. We print the value 15 and look at its left subtree. The left subtree is NULL so the call to the preorder method is *preorder(NULL)*. This call makes the condition of 'if statement' false and the recursive calls ends on this side. Afterwards, we go to the right subtree of 15. The call *preorder(18)* is made, followed by printing of the value 18. Then we go to the left subtree of it that is node 16. After calling *preorder(16)* and printing it, we go to the left subtree of 16 which is NULL. Now we will move to right subtree of 16 and the call *preorder(17)* prints the number 17. After it, the two next calls are *preorder(NULL)* as the left and right subtree of 17 are NULL. Thus after traversing the left and right subtree of 16 (which itself is a left subtree of 18), we return to node 18. Then we go to the right subtree of 18 and reach at node 20. So *preorder(20)* prints the value 20. There are again two *preorder(NULL)* calls as the left and right subtree of 20 are NULL. The preorder recursive call ends here. Now we go back and reach at node 14 whose left and right subtrees have been traversed. That means the whole tree having 14 as the root has traversed.

Inorder Recursion

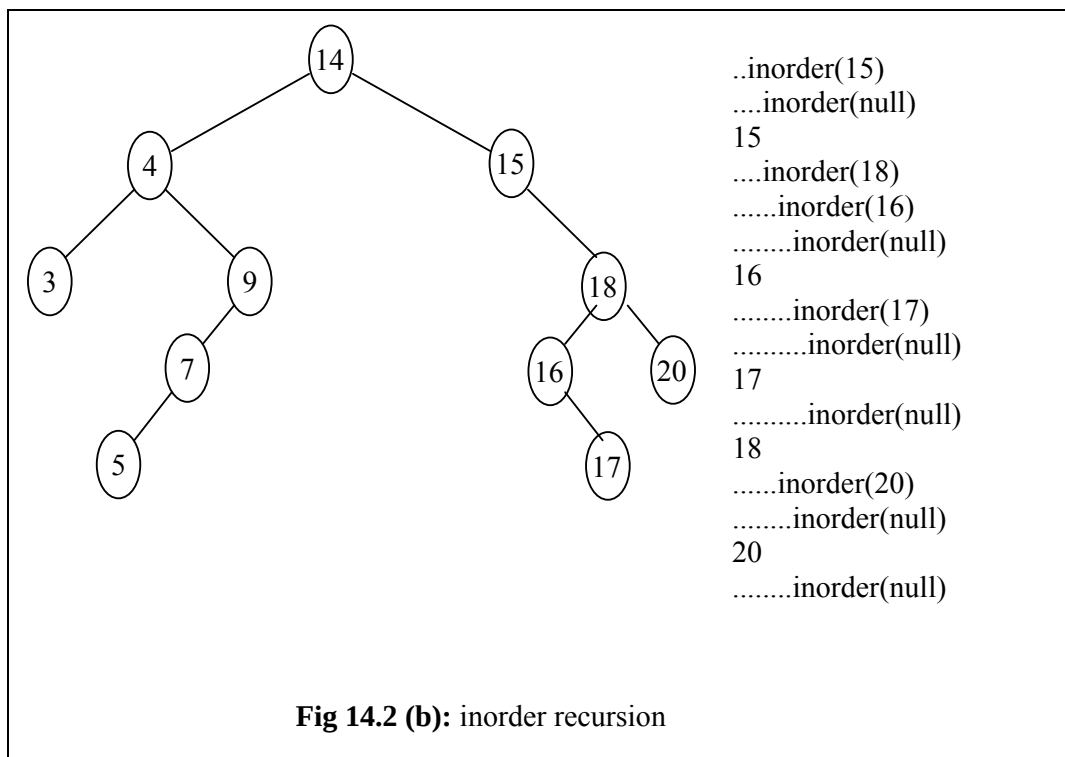
Now let's see the inorder recursive calls for traversing a tree. It is not different from the preorder. The pattern of recursive calls will be changed as in the inorder we traverse the left subtree first before printing the root. Afterwards, the right subtree is traversed. The following figures (Fig 14.2(a) and 14.2(b)) explains this phenomenon of inorder recursion by showing the recursive calls.



We start the inorder with the call `inorder (14)` as the root of the tree is 14. Now in the inorder traversing, a programmer has to traverse the left subtree before printing the value of the root node and then going to the right subtree. So the call to the left subtree of 14 i.e. `inorder (4)` will lead to the node 4. At the node 4, the same process of calling its left subtree, before printing its value, will be applied and we reach at node 3 that is the left subtree of 4. From the node 3, we go to its left subtree. This left subtree is NULL. Here in the inorder method, we have the same terminating condition as that seen in the preorder i.e. we will not make further recursive calls if there becomes a NULL node in the method call. We end the recursion at that node and come back. Now when we come back from the NULL left subtree of 3 this shows that we have visited the left subtree of 3. So the value of the node i.e. 3 is printed. Now we go to the right subtree of 3. Here the inorder call is `inorder (NULL)`. This call will make the *if* statement, in the inorder method, false, leading to the end of the recursion. Thus we have visited the whole tree whose root is node 3. So we come back to node 4. As the left subtree of node 4 has been visited, we print the value of node i.e. 4. Thus we have printed the numbers 3 and 4. Now we go to the right subtree of 4. The right subtree of 4 is the node 9. Now we send 9 as an argument to inorder method. So there is a call `inorder (9)`. As it is inorder traversing, we go to traverse its left subtree before printing the number 9. We reach the node 7 and make a call `inorder (7)`. Later, we go to the left subtree of 7 i.e. the node 5. Now we visit the left subtree of 5 which is NULL. So here the inorder recursion ends and we come back to node 5, print it and go to the right subtree of 5. The right subtree of 5 is also NULL. So ending recursion here, we have finally traversed the tree with 5 as its root. After traversing it, we come back to the node 7. After visiting the left subtree of 7, we print 7 and go to the right subtree of 7 which is NULL.

After the call *inorder (NULL)* we have traversed the tree having 7 as the root and it is a left subtree of 9. Thus we have visited the left subtree of 9. We print the number 9 before going to traverse its right subtree. The right subtree of 9 is NULL so the recursion ends here and we come back to the node 4. The tree having node 4 as root has been traversed now. This tree is a left subtree of node 14. Thus after traversing it we come back to node 14 and print it (as we have traversed its left subtree i.e. tree with 4 as root). It is evident here that before going to traverse the right subtree of 14, we have printed the numbers 3, 4, 5, 7, 9 and 14.

Now after printing the number 14, we go to traverse the right subtree of 14. The traversal of this right subtree with 15 as a root, is being carried out in the way, we traversed the left subtree. In the right subtree, we reach at node 15 and look at its left subtree before printing it. This left subtree is NULL. The call *inorder (NULL)* makes the condition of *if* statement false and we come back to the root node i.e. 15. We print this number and go to the right subtree of 15. The right subtree of 15 is also a tree the root node of which is 18. From this root node i.e. 18, we go to its left subtree and reach at node 16. NULL being the left subtree of 16, makes us to return to 16 and print it (as its left subtree has been traversed, which is NULL). After printing 16, we go to the right subtree of 16. This leads us to node 17. As we are traversing the tree by inorder process, so before printing the value 17, it will be necessary to go to traverse its left subtree. Its left subtree is NULL. So after the call *inorder (NULL)* we return to 17 and print it. Now we look at the right subtree of 17 that is also NULL. So again there is a call *inorder (NULL)* which ends the recursive calls for this tree. Thus we have traversed the whole tree having node 17 as root and it is a right subtree of 16. We have already traversed the left subtree of 16.



So we have traversed the tree with node 16 as its root. This tree is a left subtree of node 18. After traversing the left subtree of 18, we print the number 18 and go to its right subtree and reach at node 20. Now we go to the left subtree of 20 i.e. NULL and come back to 20 and print it. Then go to its right subtree, which is also NULL that ends the recursion. Thus we have traversed the tree having node 20 as its root. This tree (having 20 as root) is a right subtree of 18 which itself is a right subtree of 15. We have already traversed the left subtree of 15 and gone through the whole tree having 15 as root. This tree is a right subtree of the main tree with 14 as the root. We have already traversed its left subtree. So we have traversed the whole tree by the inorder process. From the figure 14.3 we see that by the inorder traversal we have printed the numbers in the order as below

3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20

We have successfully demonstrated the preorder and inorder traversal of a tree by using recursion method. The postorder traversal has been left for you as an exercise. You can do it with the same above tree or build a tree on your own and traverse it.

The tree data structure by nature is a recursive data structure. In the coming lecture, we will see that most of the methods we write for tree operations are recursive. From the programming point of view, the recursion is implemented internally by using call stack. This stack is maintained by the run time environment. The recursion is there in the machine code generated by the compiler. It is the programmer's responsibility to provide a terminating condition to stop the recursion. Otherwise, it will become an infinite recursion. If we do not put a terminating condition, the recursive calls will be continued and the call stack will go on increasing. We know that when a program executes, it becomes a process and some memory is allocated to it by the system for its call stack. This memory has a limit. Thus when the recursive calls do not end, there is no memory left to be used for increasing call stack after a certain time. This will crash the program or the program will halt. So we should be very careful while using the recursive calls and ensure the provision of a terminating condition in the recursive calls.

Non Recursive Traversal

We can also carry out these traversing with the help of non-recursive calls. Think for a few moments that how can we keep track of going left or going right or coming back to the node without using the recursion. In the preorder, inorder and postorder traversal, we visit nodes at different level and then return. We know the way to comeback or go further deep. Now if we are not using the recursion, how these things can be managed? We have to manage all this in our methods. Let's see how we can write the non-recursive tree traversal methods. We can implement non-recursive versions of the preorder, inorder and postorder traversal by using an explicit stack. We cannot avoid stack. The stack will be used to store the tree nodes in the appropriate order. Let's try to write methods without using recursion. For this purpose, we have to create stack.

Here, for example, is the routine for inorder traversal that uses a stack.

```
void inorder(TreeNode<int>* root)
{
```

```

Stack<TreeNode<int>*> stack;
TreeNode<int>* p;
p = root;
do
{
    while( p != NULL )
    {
        stack.push( p );
        p = p->getLeft();
    }
    // at this point, left tree is empty
    if( !stack.empty() )
    {
        p = stack.pop();
        cout << *(p->getInfo()) << " ";
        // go back & traverse right subtree
        p = p->getRight();
    }
} while ( !stack.empty() || p != NULL );
}

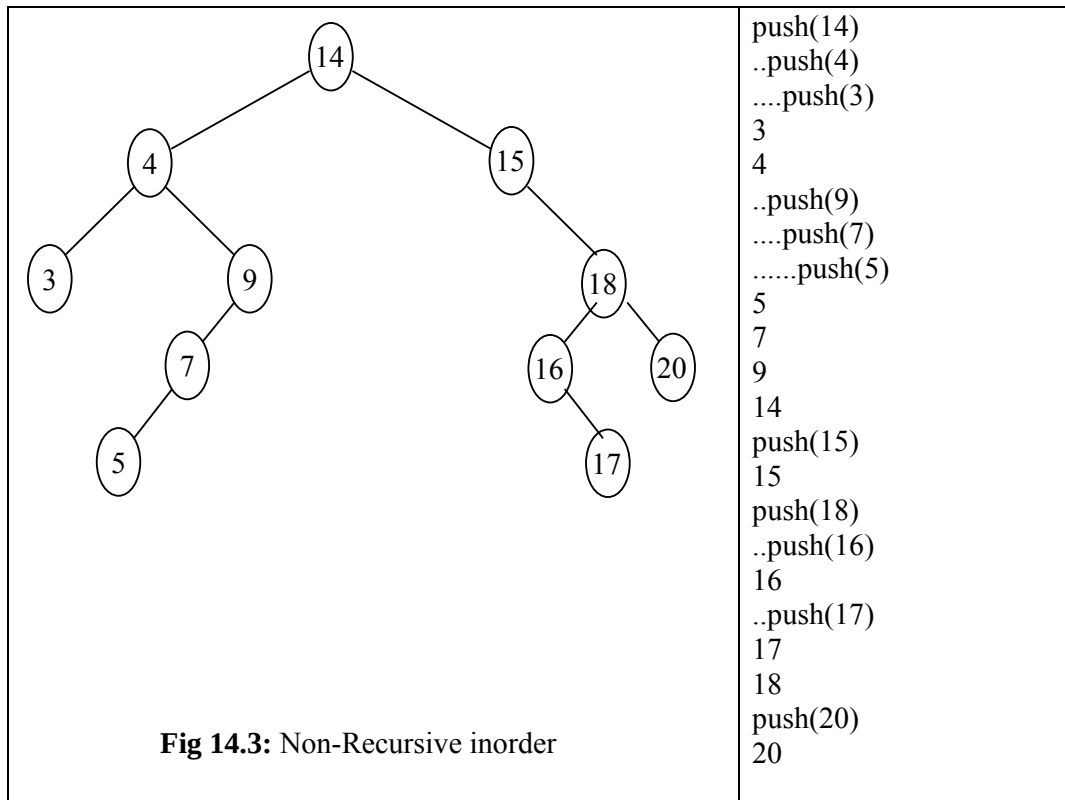
```

The signature is same. The name of the function that is *inorder*, its return type is *void* and the argument is pointer to *TreeNode*. Then there is stack abstract data type. We can store int, float or other data type in the stack. Here we want to store the *TreeNode* in the stack and then in the nodes we want to store integers. The statement is:

```
Stack<TreeNode<int>*> stack;
```

We send a message to the *Stack* factory that creates a stack to store *TreeNode*. *The integers will be stored in the TreeNode*. We have almost two levels of template. Then we declare a *TreeNode* pointer as *p* to traverse the tree up and down. We point this pointer *p* to the *root* node. Then we have a do-while loop. The loop checks the value of *p* is not NULL or *stack* is not empty. Inside the outer loop, we have an inner ‘while loop’ which checks that *p* is not NULL. If *p* is not NULL, we push *p* on the stack i.e. we push the root on the stack. Then we replace the *p* with *p->getLeft()*. This shows that *p* is now pointing to the left subtree node. In case of having a left subtree of *p*, we come back in the while loop as *p* is not NULL. Again we push *p* on the stack. Now *p* is pointing to the left subtree. You have presumed that this while loop will take us to the left subtree of the root. On the exit from the while loop, we have pushed the node in the stack whose left subtree is NULL. Here, *p* is pointing to NULL. Now we check that stack is empty or not. If we have pushed some nodes in the stack, it will not be empty. We pop a value from the stack and assign it to *p* before printing the info stored in that node and assign *p* the right subtree of that node. We come back to the outer do-while loop. It will continue executing till stack is empty or *p* is NULL.

To understand this code, we will trace the code on the same tree, earlier used in the recursive *inorder* function.



This is the same tree earlier discussed with reference to node as 14. We create a stack and assign the root to the pointer p and have an inner while loop. In the while loop, we pushed the root node in the stack i.e. push(14). After this, we assign the pointer p to the left subtree of the root and return to while loop again. Now p is pointing to the node with value 4, so we push 4 and then 3 on the stack. As the left subtree of 3 is null so we exit from the inner loop. Afterwards, we pop the stack in the inner loop and print it. As we know that stack is a LIFO structure (last in, first out). Finally, we have pushed the node 3. So this node will be popped and the number 3 will be printed. Now we will assign the right subtree of the node 3 to the p. As this is NULL, the inner loop will not be executed and again come to the if statement. We will pop the stack in the if statement and as a result, the node 4 will be popped and printed. Then we will assign p to the right subtree of node 4 i.e. node 9. The control comes to the inner loop in which we will push the node 9 and all of its left nodes one by one i.e. push(9), push(7) and push(5). As the left subtree of the node 5 is NULL, the inner while loop will be terminated. We will pop the stack and print 5. As the right subtree of node 5 is NULL, the node 7 will be popped and printed. Here the right subtree of node 7 is NULL. The stack is again popped and the number 9 will be printed. Here, the right subtree of node 9 is also NULL. The stack will be popped resulting in popping and printing of the node 14. Then we go the right subtree of the node 14 which is the node 15. The same rules of left subtree are applied here. You can understand it from the above table.

Traversal Trace

Let's compare the recursive inorder and non-recursive inorder and see whether there is any difference between them. We find no difference between them. Function call takes place with the help of stack. Explicit stack is also a stack and its behavior is also

the same. Let's compare these. In the left column, we have recursive inorder and on the right column there is non-recursive inorder.

recursive inorder	nonrecursive inorder
inorder(14)	push(14)
..inorder(4)	..push(4)
....inorder(3)	push(3)
3	3
4	4
..inorder(9)	..push(9)
....inorder(7)	push(7)
.....inorder(5)	push(5)
5	5
7	7
9	9
14	14
inorder(15)	push(15)
15	15
inorder(18)	push(18)
..inorder(16)	..push(16)
16	16
..inorder(17)	..push(17)
17	17
18	18
inorder(20)	push(20)
20	20

In recursive inorder, the function name is *inorder()*. We are printing the values. In case of non-recursive, *push()* method is employed. You can see the order in which tree nodes are being pushed explicitly on the stack. Then we print the values and use the *pop()* method in a special order to see the values of the nodes.

In the recursive inorder we have *inorder(14)* while in the non-recursive inorder, there is *push(14)*. In the recursive inorder, we have *inorder(4)*, in the non-recursive inorder we have *push(4)*. In the recursive inorder we have *inorder(3)*. Whereas in the non-recursive inorder, you may have *push(3)*. It is pertinent to note that the only difference in these two cases is that *inorder()* is in the left column while *push()* will be found in the right column. You should not be surprised over this as stack is being used in both recursive calls and non-recursive calls.

We can use recursive calls in which stack is used internally. However, in case of non-recursive calls, the stack is used explicitly. Now the question arises which one of these methods is better. Let's see some other aspects of recursion. When should we use the recursion with respect to efficiency and implementation? How many statements were there in the inorder recursive function. The first statement was an if statement which is the terminating condition. After this, we have only three statements, two calls and one cout statement. The order in the inorder is *inorder(left)*, *cout* and then *inorder(right)*. That's the complete function. Now see the non recursive function. There are a lot of statements in this function so that there should be more code in the non-recursive function. The second thing is the readability. The readability and understanding is better in the recursive function. One can easily understand what is happening. To understand the non-recursive code, one has to read the code with

care. He has to go through every statement to see what you are doing. It will also help ascertain if you are doing it right or not.

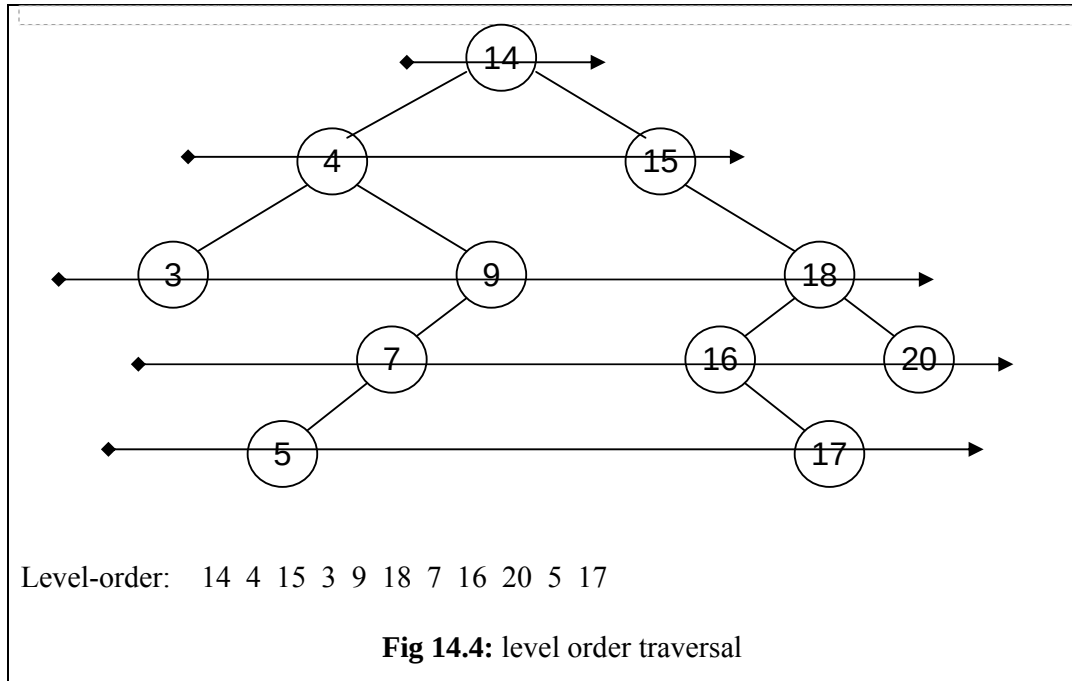
This is also an important aspect of programming. Program readability is also an issue. Suppose you have written some program. Will you understand it after some months that why you have written this and how? The first thing may be that what you are doing in the program and how you do it? After going through the program, you will remember it and recall that you have used this data structure for some purpose. Always comment the code. Comment the data structure, logic and algorithm of the program. Recursive procedure is an elegant procedure. Both the data structure and procedure are recursive. We have traversed a tree with only three four statements. No matter whatever is the size of the tree.

When the recursion happens with the help of function calls and stack.. There are some other values also included. It has return address, local variables and parameters. When a function calls another function irrespective of recursive or non recursive like function F is calling function G. It will take time to put the values in the stack. If you create your own stack, it takes time for creation and then push and pop will also consume time. Now think that which one of these will take more time. The situation is this that function calls always takes place using stack irrespective of the language. The implementation of using stack is implemented very efficiently in the Assembly language. In the computer architecture or Assembly language program, you will study that the manipulation of stack calls that is push and pop and other methods are very efficiently coded. Now you may think that there is a lot of work needed for the recursive calls and non- recursive calls are faster. If you think that the non-recursive function will work fast, it is wrong. The experience shows that if recursive calls are applied on recursive data structures, it will be more efficient in comparison with the non-recursive calls written by you. We will see more data structures which are inherently recursive. If the data structures are recursive, then try to write recursive methods. We will further discuss binary tree, binary search tree and see more examples. Whenever we write a method dealing with these data structures and use recursion. With the use of recursion, our program will be small, efficient and less error prone. While doing programming, we should avoid errors. We don't want to see there are errors while writing a program and executing it.

Try to do these traversals by hands on trees. These traversals are very important in trees. We will write more methods while dealing with tree, used internally in the inorder or postorder traversal.

There is yet another way of traversing a binary tree that is not related to recursive traversal procedures discussed previously. In level-order traversal, we visit the nodes at each level before proceeding to the next level. At each level, we visit the nodes in a left-to-right order. The traversal methods discussed earlier are inorder, preorder and post order. We print the node or to the left subtree or to the right subtree. In the preorder, we traverse the complete left subtree before coming to the right subtree. These procedures are in one way , depth-oriented. Now we may like to traverse the tree in a way that we go at one level and print all the nodes at that level. There are some algorithm in which we need level order traversal.

In the following figure, the level order traversal is represented using the arrows.



We started from the root with 14 as a root node. Later, we printed its value. Then we move to the next level. This is the binary tree so that we may have two elements at next level i.e. 4 and 15. So we printed 4 and 15 while moving from left to right. Now we are not moving to left or right subtree of any node. Rather, we move to the next level. At next level, a programmer will have three nodes that are from left to right as 3, 9 and 18. We printed these values. If we take root node as zero level, it will be at level 2. Now we move to the level 3. Here at level 3, we have 7, 16 and 20. At next level, we have 5 and 17. So the numbers we are having are- 14 4 15 3 9 18 7 16 20 5 17. How can we programmatically traverse the tree in level order? We will discuss it in the next lecture.

Data Structures

Lecture No. 15

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4
4.3, 4.3.5, 4.6

Summary

- Level-order Traversal of a Binary Tree
- Storing Other Types of Data in Binary Tree
- Binary Search Tree (BST) with Strings
- Deleting a Node From BST

Level-order Traversal of a Binary Tree

In the last lecture, we implemented the tree traversal in preorder, postorder and inorder using recursive and non-recursive methods. We left one thing to explore further that how can we print a binary tree level by level by employing recursive or non-recursive method.

See the figure below:

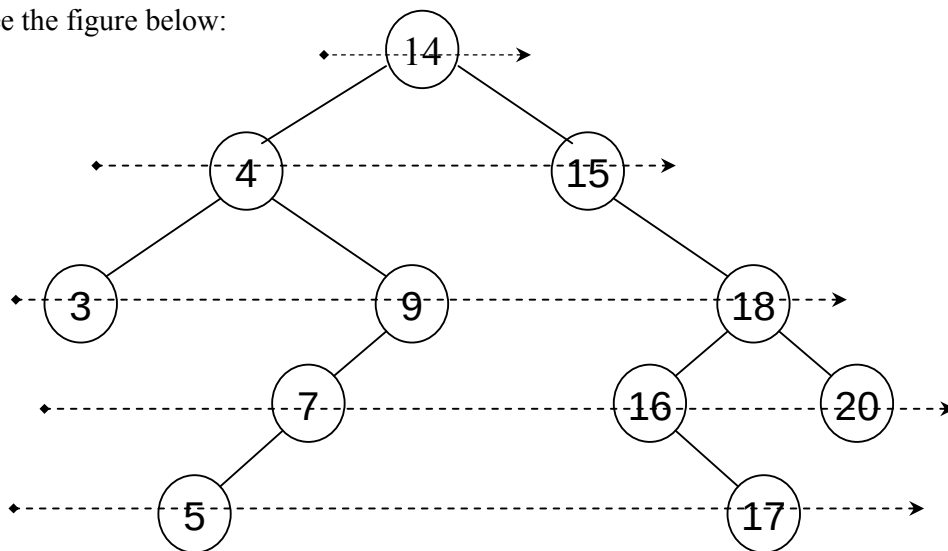


Fig 15.1: Level-order Traversal

In the above figure, levels have been shown using arrows:

At the first level, there is only one number 14.

At second level, the numbers are 4 and 15.

At third level, 3, 9 and 18 numbers are present.

At fourth level the numbers are 7, 16 and 20.

While on fifth and final level of this tree, we have numbers 5 and 17.

This will also be the order of the elements in the output if the level-order traversal is done for this tree. Surprisingly, its implementation is simple using non-recursive method and by employing queue instead of stack. A queue is a FIFO structure, which can make the level-order traversal easier.

The code for the level-order traversal is given below:

```

1. void levelorder( TreeNode <int> * treeNode )
2. {
3.     Queue <TreeNode<int> *> q;
4.
5.     if( treeNode == NULL ) return;
6.     q.enqueue(treeNode);
7.     while( !q.empty() )
8.     {
9.         treeNode = q.dequeue();
10.        cout << *(treeNode->getInfo()) << " ";
11.        if(treeNode->getLeft() != NULL )
12.            q.enqueue( treeNode->getLeft());
13.        if(treeNode->getRight() != NULL )
14.            q.enqueue( treeNode->getRight());
15.    }
16.    cout << endl;
17. }
```

The name of the method is *levelorder* and it is accepting a pointer of type *TreeNode <int>*. This method will start traversing tree from the node being pointed to by this pointer. The first line (line 3) in this method is creating a queue *q* by using the *Queue* class factory interface. This queue will be containing the *TreeNode<int> ** type of objects. Which means the queue will be containing nodes of the tree and within each node the element is of type *int*.

The line 5 is checking to see, if the *treeNode* pointer passed to the function is pointing to NULL. In case it is NULL, there is no node to traverse and the method will return immediately.

Otherwise at line 6, the very first node (the node pointed to by the *treeNode* pointer) is added in the queue *q*.

Next is the *while* loop (at line 7), which runs until the queue *q* does not become empty. As we have recently added one element (at line 6), so this loop is entered.

At line 9, *dequeue()* method is called to remove one node from the queue, the element at *front* is taken out. The return value will be a pointer to *TreeNode*. In the next line (line 10), the *int* value inside this node is taken out and printed. At line 11, we check to see if the left subtree of the tree node (we've taken out in at line 9) is present. In case the left subtree exists, it is inserted into the queue in the next statement at line 12. Next, we see if the right subtree of the node is there, it is inserted into the queue also. Next statement at line 15 closes the *while* loop. The control goes back to the line 7, where it checks to see if there is some element left in the queue. If it is not empty, the loop is entered again until it becomes empty.

Let's execute this method *levelorder(TreeNode <int> *)* by hand on a tree shown in

the *fig 15.1* to understand the level order traversal better. We can see that the *root* node of the tree is containing element *14*. This node is inserted in the queue first, it is shown in the below figure in gray shade.

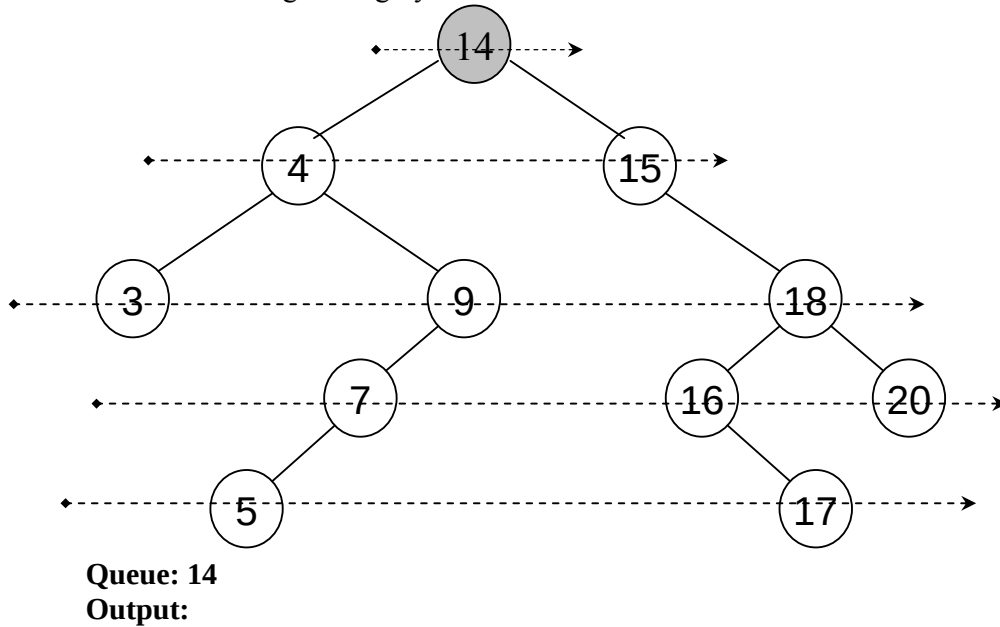


Fig 15.2: First node is inserted in the queue

After insertion of node containing number 14, we reach the *while* loop statement, where this is taken out from the queue and element *14* is printed. Next the left and right subtrees are inserted in the queue. The following figure represents the current stage of the tree.

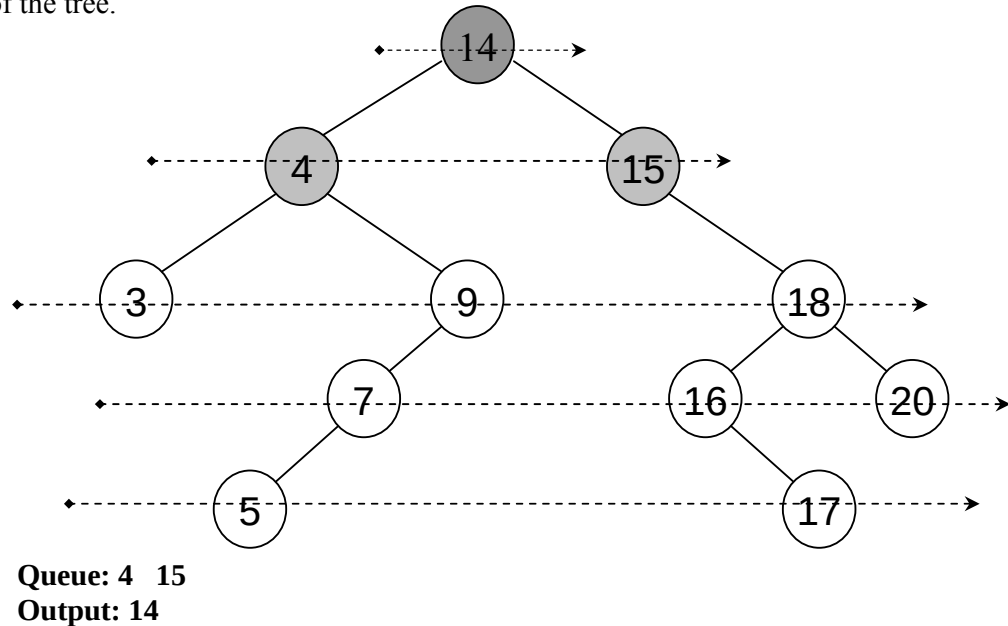


Fig 15.3: 14 is printed and two more elements are added in the queue

The element that has been printed on the output is shown with dark gray shade while the elements that have been inserted are shown in the gray shade. After this the control is transferred again to the start of the *while* loop. This time, the node containing number 14 is taken out (because it was inserted first and then the right node) and printed. Further, the left and right nodes (3 and 9) are inserted in the queue. The following figure depicts the latest picture of the queue.

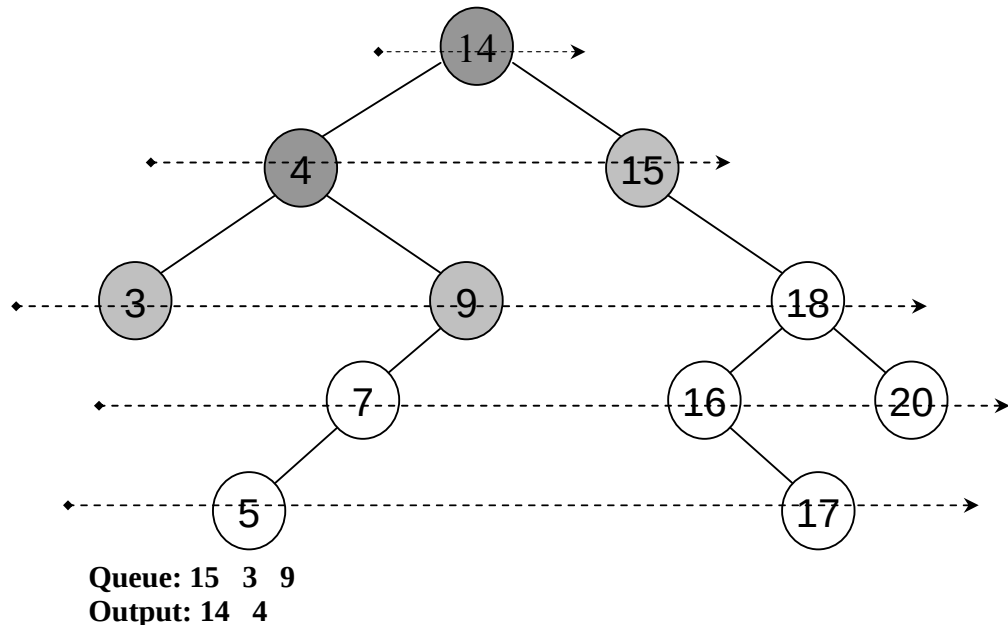


Fig 15.4

In the above *Queue*, numbers 15, 3 and 9 have been inserted in the queue until now. We came back to the *while* loop starting point again and dequeued a node from the queue. This time a node containing number 15 came out. Firstly, the number 15 is printed and then we checked for its left and right child. There is no left child of the node (with number 15) but the right child is there. Right child is a node containing number 18, so this number is inserted in the queue.

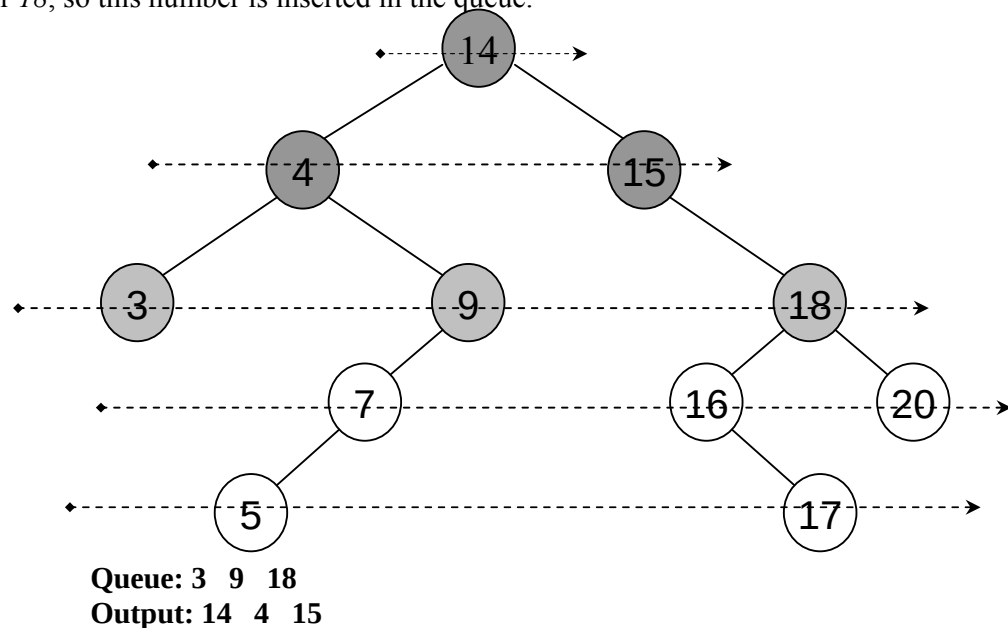


Fig 15.5

This time the node that is taken out is containing number 3. It is printed on the output. Now, node containing number 3 does not have any *left* and *right* children, therefore, no new nodes are inserted in the queue in this iteration.

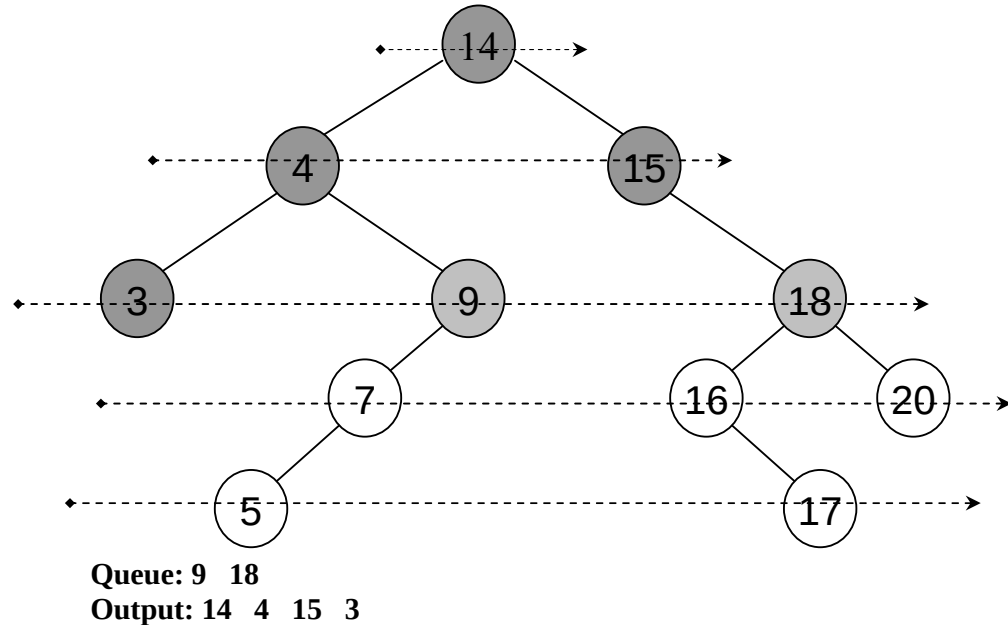


Fig 15.6

In the next iteration, we take out the node containing number 9. Print it on the output. Look for its left subtree, it is there, hence inserted in the queue. There is no right subtree of 9.

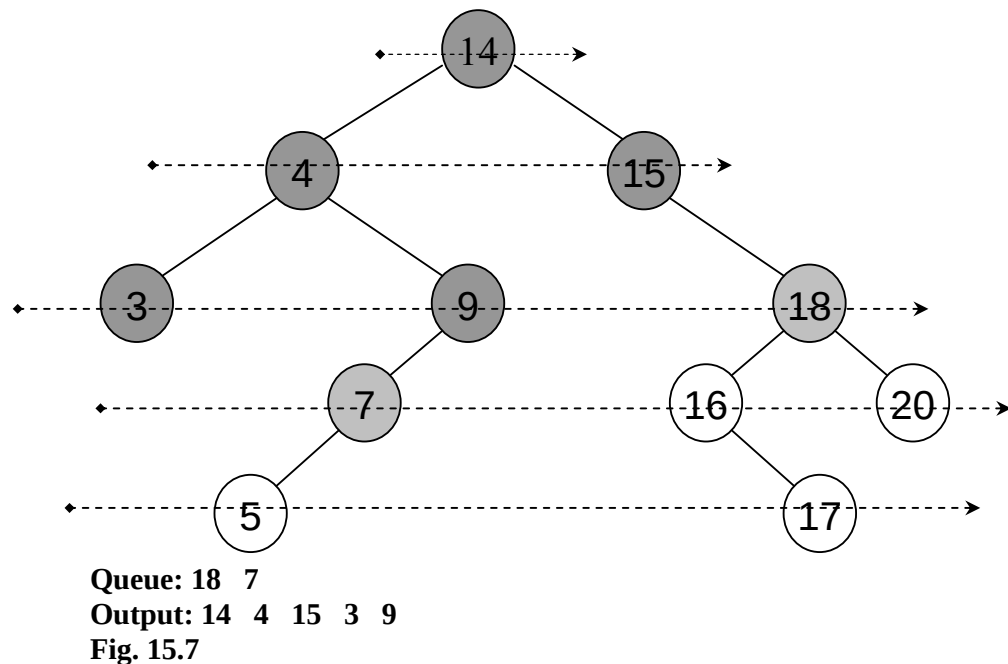


Fig. 15.7

Similarly, if we keep on executing this function. The nodes are inserted in the queue and printed. At the end, we have the following picture of the tree:

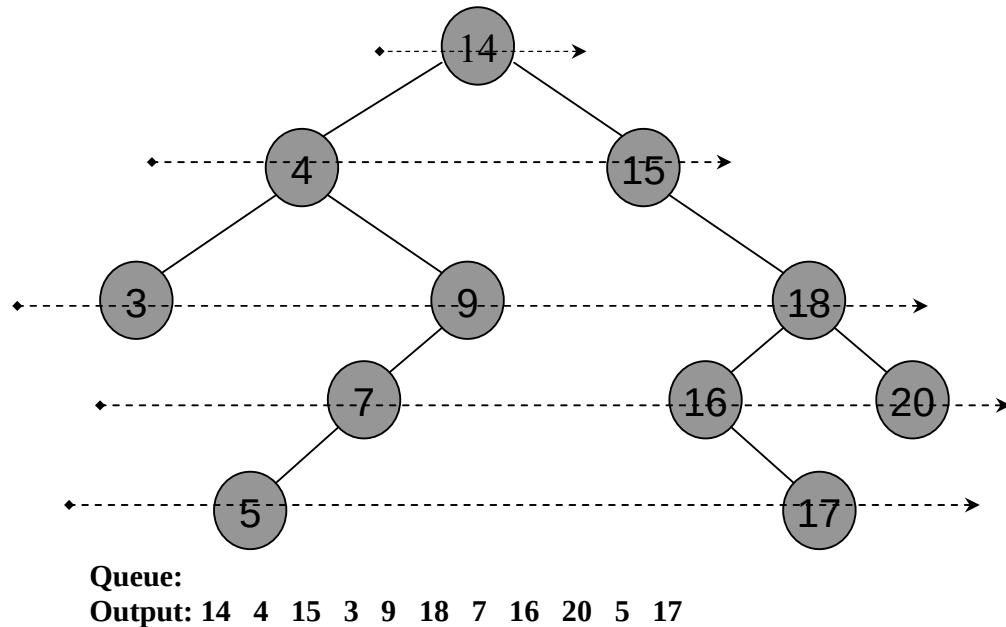


Fig 15.8

As shown in the figure above, the *Queue* has gone empty and in the output we can see that we have printed the tree node elements in the level order.

In the above algorithm, we have used *queue* data structure. We selected to use queue data structure after we analyzed the problem and sought most suitable data structure. The selection of appropriate data structure is done before we think about the programming constructs of *if* and *while* loops. The selection of appropriate data structure is very critical for a successful solution. For example, without using the queue data structure our problem of *levelorder* tree traversal would not have been so easier.

What is the reason of choosing queue data structure for this problem. We see that for this levelorder (level by level) tree traversal, the levels come turn by turn. So this turn by turn guides us of thinking of using queue data structure.

Always remember that we don't start writing code after collecting requirements, before that we select the appropriate data structures to be used as part of the design phase. It is very important that as we are getting along on our course of data structures, you should take care why, how and when these structures are employed.

Storing Other Types of Data in Binary Tree

Until now, we have used to place *int* numbers in the tree nodes. We were using *int* numbers because they were easier to understand for sorting or comparison problems. We can put any data type in tree nodes depending upon the problem where tree is employed. For example, if we want to enter the names of the people in the telephone

directory in the binary tree, we build a binary tree of strings.

Binary Search Tree (BST) with Strings

Let's write C++ code to insert non-integer data in a binary search tree.

```
void wordTree()
{
    TreeNode<char> * root = new TreeNode<char>();
    static char * word[] = "babble", "fable", "jacket",
        "backup", "eagle", "daily", "gain", "bandit", "abandon",
        "abash", "accuse", "economy", "adhere", "advise", "cease",
        "debunk", "feeder", "genius", "fetch", "chain", NULL};
    root->setInfo( word[0] );

    for(i = 1; word[i]; i++ )
        insert(root, word[i] );
    inorder( root );
    cout << endl;
}
```

The function name is *wordTree()*. In the first line, a *root* tree node is constructed, the node will be containing *char* type of data. After that is a *static* character array *word*, which is containing words like *babble*, *fable* etc. The last word or string of the array is *NULL*. Next, we are putting the first word (*babble*) of the *word* array in the *root* node. Further is a *for* loop, which keeps on executing and inserting words in the tree using *insert (TreeNode<char> *, char *)* method until there is a word in the *word* array (*word* is not *NULL*). You might have noticed that we worked in the same manner when we were inserting *ints* in the tree. Although, in the *int* array, we used *-1* as the ending number of the array and here for words we are using *NULL*. After inserting the whole array, we use the *inorder()* method to print the tree node elements.

Now, we see the code for the *insert* method.

```
void insert(TreeNode<char> * root, char * info)
{
    TreeNode<char> * node = new TreeNode<char>(info);
    TreeNode<char> *p, *q;
    p = q = root;
    while( strcmp(info, p->getInfo()) != 0 && q != NULL )
    {
        p = q;
        if( strcmp(info, p->getInfo()) < 0 )
            q = p->getLeft();
        else
            q = p->getRight();
    }
    if( strcmp(info, p->getInfo()) == 0 )
    {
        cout << "attempt to insert duplicate: " << * info << endl;
        delete node;
    }
}
```

```
        else if( strcmp(info, p->getInfo()) < 0 )  
            p->setLeft( node );  
        else  
            p->setRight( node );  
    }
```

The `insert(TreeNode<char> * root, char * info)` is accepting two parameters. First parameter `root` is a pointer to a `TreeNode`, where the node is containing element `char` type. The second parameter `info` is pointer to `char`.

In the first line of the `insert` method, we are creating a new `TreeNode` containing the `char` value passed in the `info` parameter. In the next statement, two pointers `p` and `q` of type `TreeNode<char>` are declared. Further, both pointers `p` and `q` start pointing to the tree node passed in the first parameter `root`.

You must remember while constructing binary tree of numbers, we were incrementing its count instead of inserting the new node again if the same number is present in the node. On the other hand, if the same number was not there but the new number was less than the number in the node, we used to traverse to the left subtree of the node. In case, the new number was greater than the number in the node, we used to seek its right subtree.

Similarly, in case of strings (words) here, we will increment the counter if it is already present, and will seek the left and right subtrees accordingly, if required.

In case of `int`'s we could easily compare them and see which one is greater but what will happen in case of strings. You must remember, every character of a string has an associated ASCII value. We can make a lexicographic order of characters based on their ASCII values. For example, the ASCII value of `B (66)` is greater than `A (65)`, therefore, character `B` is greater than character `A`. Similarly if a word is starting with the letter `A`, it is smaller than the words starting from `B` or any other character up to `Z`. This is called lexicographic order.

C++ provides us overloading facility to define the same operators (`<`, `>`, `<=`, `>=`, `==` etc) that were used for `ints` to be used for other data types for example strings. We can also write functions instead of these operators if we desire. `strcmp` is similar kind of function, part of the standard C library, which compares two strings.

In the code above inside the `while` loop, the `strcmp` function is used. It is comparing the parameter `info` with the value inside the node pointed to by the pointer `p`. As `info` is the first parameter of `strcmp`, it will return a negative number if `info` is smaller, `0` if both are equal and a positive number if `info` is greater. The `while` loop will be terminated if the same numbers are found. There is another condition, which can cause the termination of loop that pointer `q` is pointing to `NULL`.

First statement inside the loop is the assignment of pointer `q` to `p`. In the second insider statement, the same comparison is done again by using `strcmp`. If the new word pointed to by `info` is smaller, we seek the left subtree otherwise we go to the right subtree of the node.

Next, we check inside the `if-statement`, if the reason of termination of loop is duplication. If it is, a message is displayed on the output and the newly constructed node (that was to be inserted) is deleted (deallocated).

If the reason of termination is not duplication, which means we have reached to the node where insertion of the new node is made. We check if the `info` is smaller than the word in the current tree node. If this is the case, the newly constructed node is inserted to the left of the current tree node. Otherwise, it is inserted to the right.

This *insert()* method was called from inside the *for* loop in the *wordTree()* method. That loop is terminated when the NULL is reached at the end of the array *word*. At the end, we printed the inserted elements of the tree using the *inorder()* method.

Following is the output of *inorder()*:

Output:

```
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket
```

Notice that the words have been printed in the sorted order. Sorting is in increasing order when the tree is traversed in *inorder* manner. This should not come as a surprise if you consider how we built the binary search tree. For a given node, values less than the *info* in the node were all in the left subtree and values greater or equal were in the right. *Inorder* prints the left subtree first, then the node itself and at the end the right subtree.

Building a binary search tree and doing an *inorder* traversal leads to a sorting algorithm.

We have found one way of sorting data. We build a binary tree of the data, traverse the tree in *inorder* fashion and have the output sorted in increasing order. Although, this is one way of sorting, it may not be the efficient one. When we will study sorting algorithms, will prove Mathematically that which method is the fastest.

Deleting a Node From BST

Until now, we have been discussing about adding data elements in a binary tree but we may also require to delete some data (nodes) from a binary tree. Consider the case where we used binary tree to implement the telephone directory, when a person leaves a city, its telephone number from the directory is deleted.

It is common with many data structures that the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

For case 1, If the node is a *leaf*, it can be deleted quite easily.

See the tree figure below.

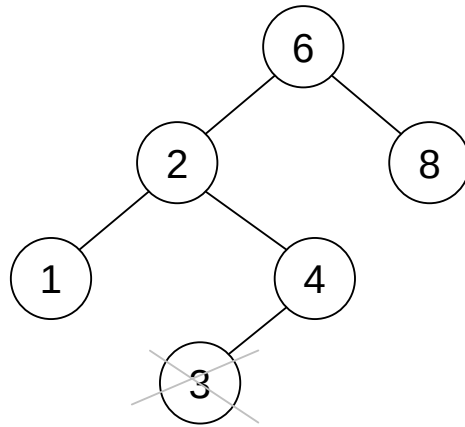


Fig 15.9: BST

Suppose we want to delete the node containing number 3, as it is a leaf node, it is pretty straight forward. We delete the leaf node containing value 3 and point the right subtree pointer to NULL.

For case 2, if the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to *inorder* successor.

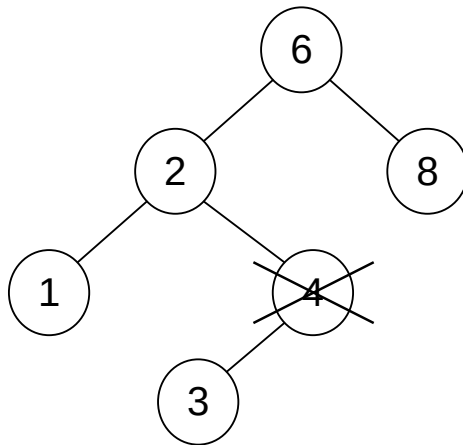


Fig 15.10: Deleting a Node From BST

If we want to delete the node containing number 4 then we have to adjust the right subtree pointer in the node containing value 2 to the *inorder* successor of 4. The important point is that the *inorder* traversal order has to be maintained after the delete.

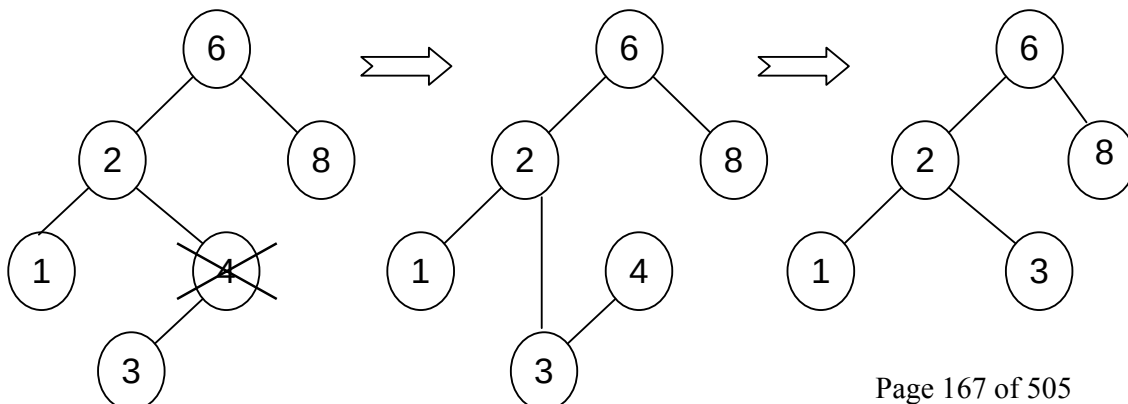
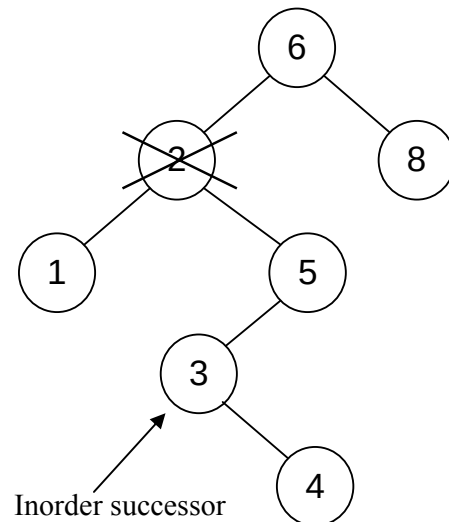


Fig 15.11: Deletion in steps

The case 3 is bit complicated, when the node to be deleted has both left and right subtrees.

The strategy is to replace the data of this node containing the smallest data of the right subtree and recursively delete that node.

Let's see this strategy in action. See the tree below:



In this tree, we want to delete the node containing number 2. Let's do *inorder* traversal of this tree first. The *inorder* traversal give us the numbers: 1, 2, 3, 4, 5, 6 and 8.

In order to delete the node containing number 2, firstly we will see its right subtree and find the left most node of it.

The left most node in the right subtree is the node containing number 3. Pay attention to the nodes of the tree where these numbers are present. You will notice that node containing number 3 is not right child node of the node containing number 2 instead it is left child node of the right child node of number 2. Also the left child pointer of node containing number 3 is NULL.

After we have found the left most node in the right subtree of the node containing number 2, we copy the contents of this left most node i.e. 3 to the node to be deleted with number 2.

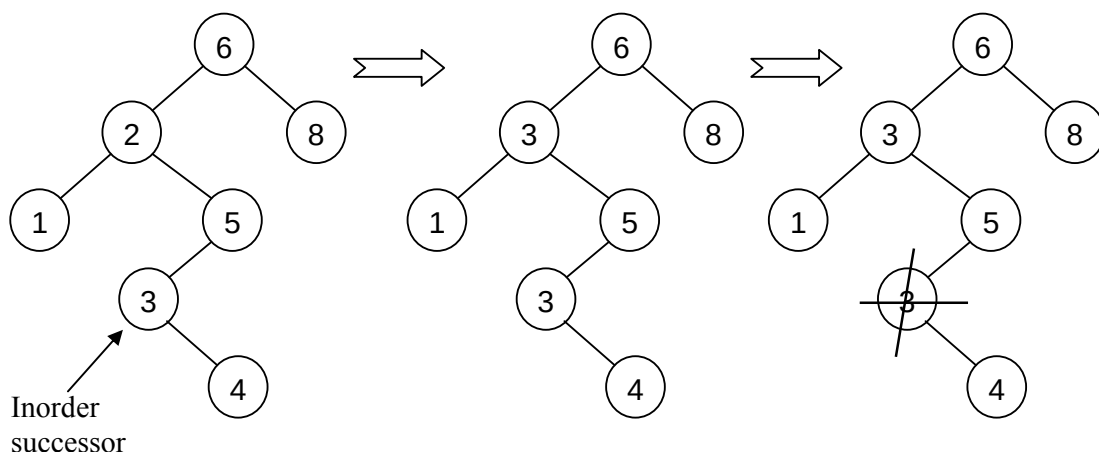


Fig 15.13: delete (2) - remove the inorder successor

Next step is to delete the left most node containing value 3. Now being the left most node, there will be no left subtree of it, it might have a right subtree. Therefore, the deletion of this node is the case 2 deletion and the delete operation can be called recursively to delete the node.

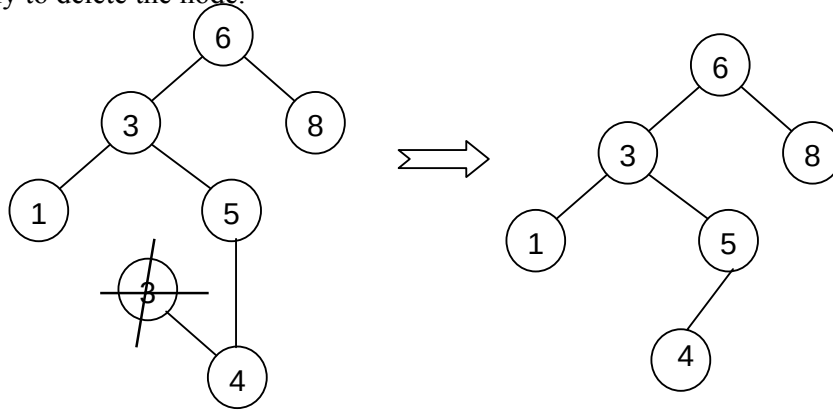


Fig 15.14: delete (2)

Now if we traverse the tree in *inorder*, we get the numbers as: 1, 3, 4, 5, 6 and 8. Notice that these numbers are still sorted. In the next lecture, we will also see the C++ implementation code for this deletion operation.

Data Structures

Lecture No. 16

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4

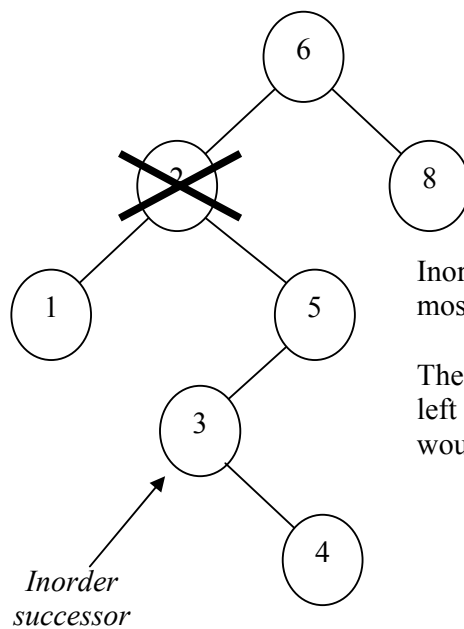
4.3 (all sub sections)

Summary

- Deleting a node in BST
- C++ code for remove
- Binary Search Tree Class (BST)

Deleting a node in BST

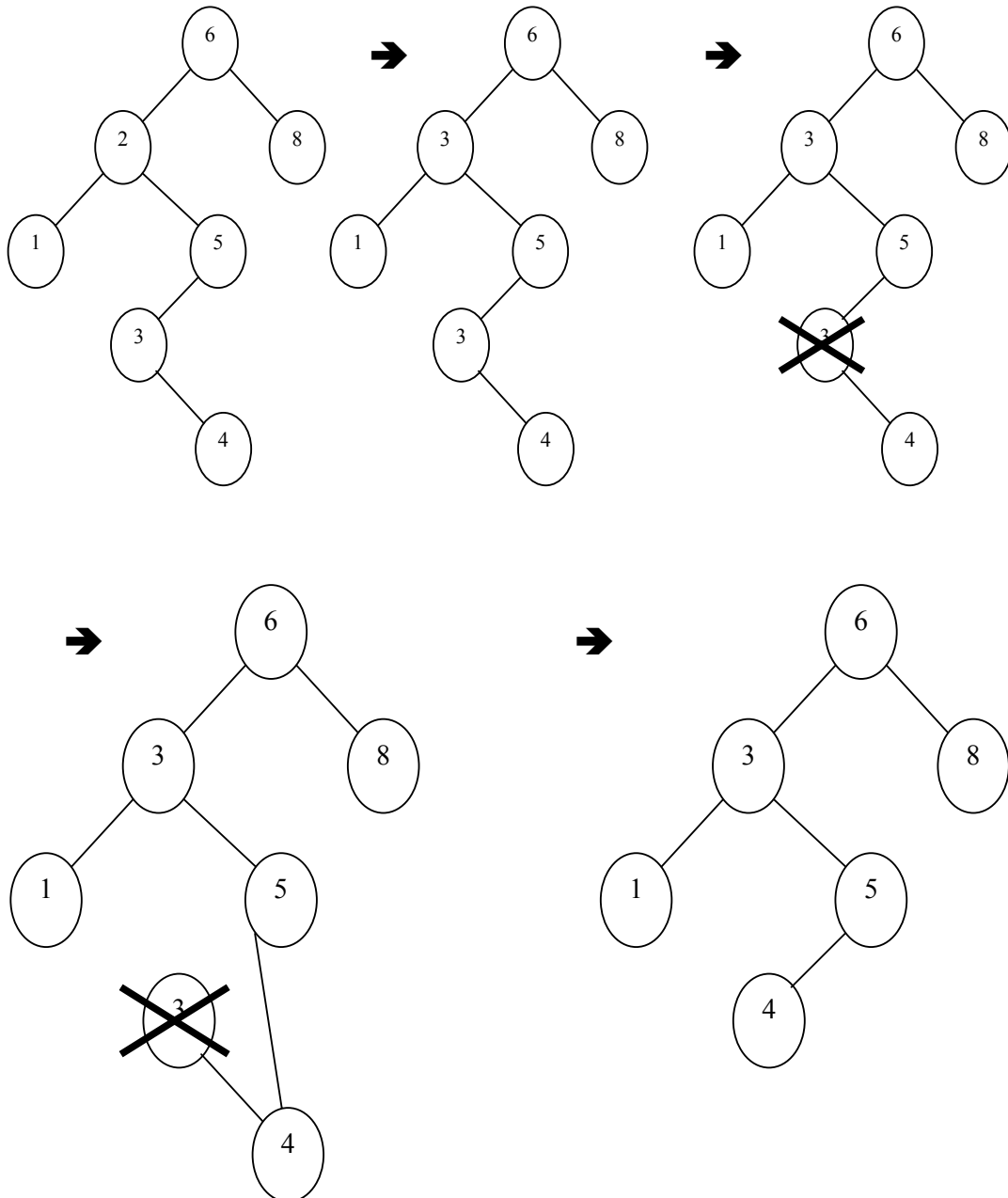
In the previous lecture, we talked about deleting a node. Now we will discuss the ways to delete nodes of a BST (*Binary Search Tree*). Suppose you want to delete one of the nodes of BST. There are three cases for deleting a node from the BST. Case I: The node to be deleted is the leaf node i.e. it has no right or left child. It is very simple to delete such node. We make the pointer in the parent node pointing to this node as NULL. If the memory for the node has been dynamically allocated, we will release it. Case II: The node to be deleted has either left child (subtree) or right child (subtree). Case III: The node to be deleted has both the left and right children (subtree). This is the most difficult case. Let's recap this case. The following figure shows the tree and the node to be deleted.



Inorder successor will be the left-most node in the right subtree of 2.

The inorder successor will not have a left child because if it did, that child would be the left-most node.

In the above example, the node to be deleted is 2. It has both right and left subtrees. As this is the BST, so while traversing the tree in inorder, we will have sorted data. The strategy is to find the inorder successor of the node to be deleted. To search the inorder successor, we have to go to its right subtree and find the smallest element. Afterwards we will copy this value into the node to be deleted. In the above example, the number 3 is the smallest in the right subtree of the node 2. A view of data of inorder traversal shows that number 3 is the inorder successor of number 2. We copy the number 3 in the node to be deleted. Now in this transition tree, the number 3 is at two places. Now we have to delete the inorder successor node i.e. node 3. The node 3 has only right child. So this is the Case II of deletion.



We will get the inorder successor of node 3 i.e. number 4. Therefore we will connect

the node 5 with node 4 that is the left pointer of 5 which was earlier pointing to node 3. But now it is pointing to node 4. We delete the node 3. Remember that number 3 has been copied in the node 2. This way, we will delete the node having right and left both subtrees.

C++ code for remove

Now we will see the C++ code to carry out this deletion. In the above figures, we have named the method as *delete*, but *delete* is a keyword of C++. So we cannot write a method named *delete*. We have to name it something else, say *remove*. Keep in mind all the three cases of delete and the figures.

Here is the code of the *remove* method.

```
/* This method is used to remove a node from the BST */

TreeNode<int>* remove(TreeNode<int>* tree, int info)
{
    TreeNode<int>* t;
    int cmp = info - *(tree->getInfo());

    if( cmp < 0 ){                // node to delete is in left subtree
        t = remove(tree->getLeft(), info);
        tree->setLeft( t );
    }
    else if( cmp > 0 ){           // node to delete is in right subtree
        t = remove(tree->getRight(), info);
        tree->setRight( t );
    }
    //two children, replace with inorder successor
    else if( tree->getLeft() != NULL && tree->getRight() != NULL ){
        TreeNode<int>* minNode;
        minNode = findMin(tree->getRight());
        tree->setInfo( minNode->getInfo() );
        t = remove(tree->getRight(), *(minNode->getInfo()));
        tree->setRight( t );
    }
    else { // case 1
        TreeNode<int>* nodeToDelete = tree;
        if( tree->getLeft() == NULL ) //will handle 0 children
            tree = tree->getRight();
        else if( tree->getRight() == NULL )
            tree = tree->getLeft();
        else tree = NULL;

        delete nodeToDelete; // release the memory
    }
    return tree;
}
```

The return type of the *remove* method is a pointer to *TreeNode*. The argument list contains two arguments, the pointer to *TreeNode* pointing the root of the tree and an integer *info* containing the value of the node to be deleted. A thorough analysis of the *delete* method reveals that you need these two things.

Here we are dealing with the templates and the nodes containing the *int* values. Inside the function, we create a temporary variable *t* of type *TreeNode* pointer. Then we have *int* type variable *cmp*. The value of the *TreeNode* provided in the argument is subtracted from the *info* (this is the value of the node to be deleted) and assigned to the *cmp*. Then we have conditional *if* statement which checks if the value of *cmp* is less than 0, we call *remove* method recursively. As the *cmp* is less than zero, it means that the *info* is not the node provided in the argument. The value of *cmp* is less than zero, depicting that the value of the current node is greater than *info*. Therefore if the node with value *info* exists, it should be in the left subtree. So we call the *remove* method on the left subtree. When we return from this call, the left subtree pointer of *tree* is assigned to our temporary variable *t*.

If the first *if* statement evaluates to true, it means that our required node is in the left subtree of the current node. We call the *remove* method recursively and remove that node. On coming back, we assign *t* to the left subtree pointer of the *tree*. Suppose that the node to be deleted is the immediate left child of the node. It will be deleted by the *remove* method. When the node is deleted, some other node will come to its place, necessitating the readjustment of the left pointer of the parent node. We can do it by pointing the left subtree pointer to *t*.

If the value of *cmp* is greater than zero, the node with *info* value should be in the right subtree of the current node. Therefore we call the *remove* method recursively, providing it the pointer to the right subtree and the *info*. When we return from that call, the right subtree is readjusted by assigning it the *t*.

In the third case, we are at the node, which is to be deleted. Here we have to deal with the three delete cases- 1) the node to be deleted is the leaf node 2) node to be deleted has either right or left child and 3) the node to be deleted has both left and right subtree. We will begin with the third case. We will check in the *else-if* statement that the left and right subtrees are not NULL. It means that the node has both left and right subtrees. We create a temporary variable *minNode* of type *TreeNode* pointer. This variable *minNode* is local to this *if-else* block. We assign it value by calling the *findMin* method. We call the *findMin* method as:

```
findMin(tree->getRight())
```

In this delete case, we need to find the inorder successor of the node to be deleted which is the minimum value in the right subtree. Then we will put the value of this node to the node to be deleted. This node is saved in the variable *minNode*. The node to be deleted is now pointed by *tree*. Then we get the value of the inorder successor, stored in the *minNode* and assign it to the node to be deleted. Now we have to delete the *minNode*. For this purpose, we call the *remove* method recursively providing it the right subtree and the value of the *minNode*. We have also changed the value of the current node and can also send this value to the *remove* method. For clarity purposes, we provide it the *minNode* value. The node returned by the *remove* method, is stored

in the temporary variable *t*. Then we assign *t* to the right subtree of *tree*. We do it due to the fact that the tree has to re-adjusted after the deletion of a node in the right subtree.

Now we will talk about other two cases i.e. the node to be deleted is the leaf node or has left or right child. We will deal with these two cases together. The *else* statement is being employed in these cases. We create a temporary variable *nodeToDelete* and assign it to the *tree*. Now we have to check whether this is the leaf node or has either left or right child. First we check that the left subtree of the tree is NULL before assigning the right subtree to the *tree*. Then we check that the right subtree of the tree is NULL. If it is NULL, the *tree* will be replaced by the left subtree. In the end, we make the tree NULL and delete the *nodeToDelete*. Here the memory is released.

While dealing with this case of delete, we find the inorder successor from the right subtree i.e. the minimum value in the right subtree. That will also be the left most node of the right subtree. We use the *findMin* method for this purpose. Let's have a look on the code of this method.

```
/* This method is used to find the minimum node in a tree
*/

TreeNode<int>* findMin(TreeNode<int>* tree)
{
    if( tree == NULL )
        return NULL;
    if( tree->getLeft() == NULL )
        return tree; // this is it.
    return findMin( tree->getLeft() );
}
```

The return type is a pointer to *TreeNode* which will be the minimum node. The input argument is also a pointer to *TreeNode*. This input argument is a root of some subtree in which we have to find the minimum node. Inside the function, we have a couple of *if* statements. If the *tree* is NULL, it will result in the return of the NULL. If the left child of the tree is NULL, the *tree* will be returned. In the end, we are calling *findMin* recursively providing it the left subtree as argument. When we were talking about the inorder successor or the left most node, it was stated that this node will not contain a left child because if it has a left child then it will not be a left most child. The left-most node is the node which either does not have a left child or its left pointer is NULL. The above method works the same way. It looks for the left node having left pointer NULL. This is the node with the minimum value. You must have noted that in this function, we are making a recursive call in the end of the function. If a function has recursive call as the last statement, it is known as tail recursion. We can replace the tail recursion with the loop. How can we do that? This is an exercise for you. In case of tail recursion, you have either to use the tail recursion or loop. While opting for loop, you will get rid of the recursion stack.

You can trace this function on the sample tree we have been using in the previous lecture. It can also be hand-traced. Provide the root of the BST to the method and try to find the *minNode* while employing the above method. You will see that the value

returned by this function is the minimum node. Also try to write the *FindMin* in a non-recursive way i.e. use of the loop. You will get the same result.

Binary Search Tree Class (BST)

Let's see the code of the binary search tree (BST). We put the interface in the .h file. We also have the state variable in addition to the public and private methods of the class in it. In the public methods, the user of the class calls with their objects. Let's create a .h file for binary search tree and the objects of binary search tree obtained from the factory will use the attributes defined in the .h file. Here is the .h file.

```

/* binarysearchtree.h file contains the interface of binary search tree (BST) */

#ifndef _BINARY_SEARCH_TREE_H_
#define _BINARY_SEARCH_TREE_H_

#include <iostream.h>    // For NULL

// Binary node and forward declaration
template <class EType>
class BinarySearchTree;

template <class EType>
class BinaryNode
{
    EType element;
    BinaryNode *left;
    BinaryNode *right;

public:
    // constructor
    BinaryNode( const EType & theElement, BinaryNode *lt, BinaryNode *rt )
        : element( theElement ), left( lt ), right( rt ) { }

    friend class BinarySearchTree<EType>;
};
// continued

```

At the start of the file, we have conditional statement using the *ifndef* that checks a constant `_BINARY_SEARCH_TREE_H_`. If this constant is not defined, we define it with the help of *define* keyword. This is a programming trick. Sometimes, we include the same .h file more than once. If we do not use this trick, the compiler will give error on the inclusion of the file for the second time. Here, we have a statement:

```
class BinarySearchTree;
```

This is the forward declaration of the class *BinarySearchTree*. We have not defined this class so far. First we define the class *BinaryNode* and then by combining these nodes, a binary tree is created. This class is a template class. The nodes can contain data other than integers. Then we define the *BinaryNode* class and define three state

variables. The data of the node is stored in the *element* which is of type *EType*. Being a binary node, it may have a left and right child. We have declared two pointers for this node. Then we have its constructor that takes three arguments, used to initialize the *element*, *left* and *right* variables. We wrote the colon(:) and the initialization list. There are three state variables in the *BinaryNode*. These are *element*, *left* and *right*. When we create an object of *BinaryNode*, all of these variables are created. The *EType* can also be some class so its object will be created and a call to its constructor is made. Whenever an object is created, its constructor will be called. We have to provide the arguments. If we do not call the constructor explicitly, the default constructor will be called. In the constructor of *BinaryNode*, after the colon, we have written *element(theElement)*. (Here *theElement* is the argument in the constructor of *BinaryNode*). It seems to be a function call but actually it is a call to the constructor for *element*. If *element* was of type *String*, constructor of *String* will be called and *theElement* is sent as argument. Through this way of writing the variables after the colon, we are actually calling the explicit constructor for the state variables. While calling the constructor, we are not using the name of the class but the variable names. There should be no ambiguity. Then we have *left(lt)* and *right(rt)*, as *left* and *right* are of type *BinaryNode* so the constructor for *BinaryNode* will be called. The body of the constructor of the *BinaryNode* is empty.

You might not be familiar with this type of coding but it is commonly used in C++ coding. You can get further information on this from the C++ book or from the internet. In some cases, a programmer is forced to use this syntax for having no other option. The last statement of the class definition has a friend class:

```
friend class BinarySearchTree<EType>;
```

You are familiar with the *friend* keyword. The class *BinaryNode* has defined the *BinarySearchTree* class as its friend class. It means that the *BinarySearchTree* class can access the state variables of *BinaryNode* class. We can also declare friend method of some class which can access the state variable of that class. We have not defined the *BinarySearchTree* class yet. We have just declared the class with the name *BinarySearchTree* as the forward declaration. This forward declaration is for compilers. When the compiler reads the file from top to bottom and encounters the line defining the *BinarySearchTree* as friend, it gives an error. The compiler does not know about the *BinarySearchTree*. To overcome this error we use forward declaration. With this forward declaration, the compiler is communicated that we want to use the *BinarySearchTree* as a class. Therefore, when the compiler reads the line, which defines the *BinarySearchTree* as a friend of *BinaryNode*, it knows that it is a class, which used *EType* template and will be defined later. Now there is no ambiguity for the compiler. We must have to define this class later on.

Let's see the code of *BinarySearchTree*. We are giving it *EType* template parameter. We want to make our *BinarySearchTree* generic and it can be used with integers, strings, characters or with some other data type. Therefore, we have defined it as a template class. The user of this class will create a *BinarySearchTree* of its specific data type. The code is as follows:

```

/* binarysearchtree.h file also contains the definition of the
BinarySearchTree */

template <class EType>
class BinarySearchTree
{
public:
    BinarySearchTree( const EType& notFound );
    BinarySearchTree( const BinarySearchTree& rhs );
    ~BinarySearchTree( );

    const EType& findMin( ) const;
    const EType& findMax( ) const;
    const EType& find( const EType & x ) const;
    bool isEmpty( ) const;
    void printInorder( ) const;
    void insert( const EType& x );
    void remove( const EType& x );

    const BinarySearchTree & operator = ( const BinarySearchTree & rhs
);

private:
    BinaryNode<EType>* root;
    // ITEM_NOT_FOUND object used to signal failed finds
    const EType ITEM_NOT_FOUND;

    const EType& elementAt( BinaryNode<EType>* t );
    void insert(const EType& x, BinaryNode<EType>* & t);
    void remove(const EType& x, BinaryNode<EType>* & t);
    BinaryNode<EType>* findMin(BinaryNode<EType>* t);
    BinaryNode<EType>* findMax(BinaryNode<EType>* t);
    BinaryNode<EType>* find(const EType& x, BinaryNode<EType>* t
);
    void makeEmpty(BinaryNode<EType>* & t);
    void printInorder(BinaryNode<EType>* t);
};
#endif

```

We start the definition of our class with the public interface. The user of this class is interested in the public interface of the class. These are the methods, which the user can employ. We have two constructors for this class. One constructor takes *EType* reference parameter while other takes *BinarySearchTree* as a parameter. The types of constructors depend on the usage. We can have more constructors if needed. Then we have a destructor of *BinarySearchTree*. Besides, we have interface methods for *BinarySearchTree*. There are also *findMin* and *findMax* methods, which will return the minimum and maximum value of *EType* in the tree respectively. We have used the *const* keyword with these functions. The reference variables in functions are also being used. Next thing we are going to discuss is the *find* method. Its signature is as

under:

```
const EType& find( const EType & x ) const;
```

This method will takes an argument *x* of *EType*. It will search the tree whether *x* exists in the tree or not. Then we have *isEmpty* method that will ascertain if the tree is empty or not. There is also the *printInorder* method, which will print the tree in inorder traversal. If the tree has integers, we have sorted integers as a result of inorder traversal. Next thing we have is the *insert (const EType& x)* method, which inserts the *x* as a new node in the tree. After this, there is the *remove* method i.e.delete method. We have renamed *delete* as *remove* because *delete* is a keyword of C++. We can also name it as *deleteNode* or some other name which you think is suitable. The interface of the class is almost complete. Now think as a user of *BinarySearchTree* and decide if you need more methods. With the help of methods defined above, we can do a lot of work on *BinarySearchTree*. If you feel to add more methods due to the usage need, these can be added later.

We also need some private variables and methods for this class. At First, we have defined a pointer to *BinaryNode* as root before defining an *EType* variable as *ITEM_NOT_FOUND*. Then there are *elementAt*, *insert* and *remove* methods. But the method signatures of these methods are different than public methods. The signature of insert method is:

```
void insert(const EType& x, BinaryNode<EType>* & t);
```

In the public *insert* method, we have only one argument of type *EType*. Similarly there are *findMin*, *findMax* and *find* methods with different signatures. Then we have *makeEmpty* and *printInorder* methods. Why we are talking about these methods especially when these are the private ones. In C++, we have to do like this. In *.h* file we define the class and in *.cpp* file, the implementation of the class is given. The user of this class is advised not to look at the private part of the class. He should be interested only in the public part of the class. We are not the users of this class, but only the developers. In the *BinarySeachTree.h* file, we have defined two classes i.e. *BinaryNode* and *BinarySearchTree*. Are these classes private or public? Can the user of *BinarySearchTree* benefit from the *BinaryNode* class? Is it possible or not? This is an exercise for you. Study about the reference data types before coming to next lecture.

Sample Program

Here is the code of the program. *BinarySearchTree.h* file.

```
/* This file contains the declaration of binary node and the binary search tree */

#ifndef BINARY_SEARCH_TREE_H_
#define BINARY_SEARCH_TREE_H_

#include <iostream.h>    // For NULL

// Binary node and forward declaration
```

```

template <class EType>
class BinarySearchTree;

template <class EType>
class BinaryNode
{
    EType element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const EType & theElement, BinaryNode *lt, BinaryNode *rt )
        : element( theElement ), left( lt ), right( rt ) { }
    friend class BinarySearchTree<EType>;
};

// BinarySearchTree class
//
// CONSTRUCTION: with ITEM_NOT_FOUND object used to signal failed
finds
//
// *****PUBLIC OPERATIONS*****
// void insert( x )    --> Insert x
// void remove( x )    --> Remove x
// EType find( x )    --> Return item that matches x
// EType findMin( )    --> Return smallest item
// EType findMax( )    --> Return largest item
// boolean isEmpty( )    --> Return true if empty; else false
// void makeEmpty( )    --> Remove all items
// void printTree( )    --> Print tree in sorted order

template <class EType>
class BinarySearchTree
{
public:
    BinarySearchTree( const EType & notFound );
    BinarySearchTree( const BinarySearchTree & rhs );
    ~BinarySearchTree( );

    const EType & findMin( ) const;
    const EType & findMax( ) const;
    const EType & find( const EType & x ) const;
    bool isEmpty( ) const;
    void printTree( ) const;

    void makeEmpty( );
    void insert( const EType & x );
    void remove( const EType & x );

    const BinarySearchTree & operator=( const BinarySearchTree & rhs );

```



```

private:
    BinaryNode<EType> *root;
    const EType ITEM_NOT_FOUND;

    const EType & elementAt( BinaryNode<EType> *t ) const;

    void insert( const EType & x, BinaryNode<EType> * & t ) const;
    void remove( const EType & x, BinaryNode<EType> * & t ) const;
    BinaryNode<EType> * findMin( BinaryNode<EType> *t ) const;
    BinaryNode<EType> * findMax( BinaryNode<EType> *t ) const;
    BinaryNode<EType> * find( const EType & x, BinaryNode<EType> *t )
const;
    void makeEmpty( BinaryNode<EType> * & t ) const;
    void printTree( BinaryNode<EType> *t ) const;
    BinaryNode<EType> * clone( BinaryNode<EType> *t ) const;
};
#include "BinarySearchTree.cpp"
#endif

```

BinarySearchTree.cpp file.

```

/* This file contains the implementation of the binary search tree */

#include <iostream.h>
#include "BinarySearchTree.h"

/**
 * Construct the tree.
 */
template <class EType>
BinarySearchTree<EType>::BinarySearchTree( const EType & notFound ) :
    ITEM_NOT_FOUND( notFound ), root( NULL )
{
}

/**
 * Copy constructor.
 */

template <class EType>
BinarySearchTree<EType>::
BinarySearchTree( const BinarySearchTree<EType> & rhs ) :
    root( NULL ), ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND )
{
    *this = rhs;
}

/**

```

```
* Destructor for the tree.
*/
template <class EType>
BinarySearchTree<EType>::~~BinarySearchTree( )
{
    makeEmpty( );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
template <class EType>
void BinarySearchTree<EType>::insert( const EType & x )
{
    insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
template <class EType>
void BinarySearchTree<EType>::remove( const EType & x )
{
    remove( x, root );
}

/**
 * Find the smallest item in the tree.
 * Return smallest item or ITEM_NOT_FOUND if empty.
 */
template <class EType>
const EType & BinarySearchTree<EType>::findMin( ) const
{
    return elementAt( findMin( root ) );
}

/**
 * Find the largest item in the tree.
 * Return the largest item of ITEM_NOT_FOUND if empty.
 */
template <class EType>
const EType & BinarySearchTree<EType>::findMax( ) const
{
    return elementAt( findMax( root ) );
}

/**
 * Find item x in the tree.
 * Return the matching item or ITEM_NOT_FOUND if not found.
```

```
*/
template <class EType>
const EType & BinarySearchTree<EType>::
    find( const EType & x ) const
{
    return elementAt( find( x, root ) );
}

/**
 * Make the tree logically empty.
 */
template <class EType>
void BinarySearchTree<EType>::makeEmpty( )
{
    makeEmpty( root );
}

/**
 * Test if the tree is logically empty.
 * Return true if empty, false otherwise.
 */
template <class EType>
bool BinarySearchTree<EType>::isEmpty( ) const
{
    return root == NULL;
}

/**
 * Print the tree contents in sorted order.
 */
template <class EType>
void BinarySearchTree<EType>::printTree( ) const
{
    if( isEmpty( ) )
        cout << "Empty tree" << endl;
    else
        printTree( root );
}

/**
 * Deep copy.
 */
template <class EType>
const BinarySearchTree<EType> &
BinarySearchTree<EType>::
operator=( const BinarySearchTree<EType> & rhs )
{
    if( this != &rhs )
    {
        makeEmpty( );
    }
}
```

```

        root = clone( rhs.root );
    }
    return *this;
}

/**
 * Internal method to get element field in node t.
 * Return the element field or ITEM_NOT_FOUND if t is NULL.
 */
template <class EType>
const EType & BinarySearchTree<EType>::
elementAt( BinaryNode<EType> *t ) const
{
    if( t == NULL )
        return ITEM_NOT_FOUND;
    else
        return t->element;
}

/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 * Set the new root.
 */
template <class EType>
void BinarySearchTree<EType>::
insert( const EType & x, BinaryNode<EType> * & t ) const
{
    if( t == NULL )
        t = new BinaryNode<EType>( x, NULL, NULL );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ; // Duplicate; do nothing
}

/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the tree.
 * Set the new root.
 */
template <class EType>
void BinarySearchTree<EType>::
remove( const EType & x, BinaryNode<EType> * & t ) const
{
    if( t == NULL )

```

```

        return; // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != NULL && t->right != NULL ) // Two children
    {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else
    {
        BinaryNode<EType> *nodeToDelete = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete nodeToDelete;
    }
}

/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
template <class EType>
BinaryNode<EType> *
BinarySearchTree<EType>::findMin( BinaryNode<EType> *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}

/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
template <class EType>
BinaryNode<EType> *
BinarySearchTree<EType>::findMax( BinaryNode<EType> *t ) const
{
    if( t != NULL )
        while( t->right != NULL )
            t = t->right;
    return t;
}

/**
 * Internal method to find an item in a subtree.
 * x is item to search for.

```

```

    * t is the node that roots the tree.
    * Return node containing the matched item.
    */
template <class EType>
BinaryNode<EType> *
BinarySearchTree<EType>::
find( const EType & x, BinaryNode<EType> *t ) const
{
    if( t == NULL )
        return NULL;
    else if( x < t->element )
        return find( x, t->left );
    else if( t->element < x )
        return find( x, t->right );
    else
        return t;    // Match
}

/***** NONRECURSIVE VERSION *****/
template <class EType>
BinaryNode<EType> *
BinarySearchTree<EType>::
find( const EType & x, BinaryNode<EType> *t ) const
{
    while( t != NULL )
        if( x < t->element )
            t = t->left;
        else if( t->element < x )
            t = t->right;
        else
            return t;    // Match

    return NULL;    // No match
}

*****/

/**
 * Internal method to make subtree empty.
 */
template <class EType>
void BinarySearchTree<EType>::
makeEmpty( BinaryNode<EType> * & t ) const
{
    if( t != NULL )
    {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
    }
    t = NULL;
}

```

```

/**
 * Internal method to print a subtree rooted at t in sorted order.
 */
template <class EType>
void BinarySearchTree<EType>::printTree( BinaryNode<EType> *t ) const
{
    if( t != NULL )
    {
        printTree( t->left );
        cout << t->element << endl;
        printTree( t->right );
    }
}

/**
 * Internal method to clone subtree.
 */
template <class EType>
BinaryNode<EType> *
BinarySearchTree<EType>::clone( BinaryNode<EType> * t ) const
{
    if( t == NULL )
        return NULL;
    else
        return new BinaryNode<EType>( t->element, clone( t->left ), clone( t-
>right ) );
}

```

TestBinarySearchTree.cpp file. This file contains the main program.

```

/* This file contains the test program for the binary search tree */

#include <iostream.h>
#include "BinarySearchTree.h"

// Test program
int main( )
{
    const int ITEM_NOT_FOUND = -9999;
    BinarySearchTree<int> t( ITEM_NOT_FOUND );
    int NUMS = 30;
    int i;

    cout << "Inserting elements (1 to 30) in the tree ....." << endl;
    for( i = 0; i <= NUMS; i++ )
        t.insert( i );

    cout << "Printing the values of the nodes in tree ....." << endl;
    t.printTree();
}

```

```
cout << "Removing the even number elements in the tree .....)" << endl;
for( i = 0; i <= NUMS; i+= 2 )
    t.remove( i );

cout << "Printing the values of the nodes in tree .....)" << endl;
t.printTree();

int abc;
cin >> i;
return 0;
}
```

When we run the TestBinarySearch program the following output is obtained.

```
Inserting elements (1 to 30) in the tree .....)
Printing the values of the nodes in tree .....)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Removing the even number elements in the tree .....)
Printing the values of the nodes in tree .....)
1
```


3
5
7
9
11
13
15
17
19
21
23
25
27
29

Data Structures

Lecture No. 17

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4

Summary

- Reference Variables
- Sample Program

After demonstrating the use of `const` and reference variables in the interface class for a binary search tree in the previous lectures, we will now see what these reference variables are and how does internal process go on during the use of reference variables.

Reference Variables

Before proceeding ahead, there is need to know why the reference variables are used; and what benefits a programmer can get while employing them. We will discuss these all things in detail with the help of examples.

The symbol `&`, used for reference variable has a few different purposes with respect to its occurrence in the code. In C++ programming, we have seen that when the ampersand sign i.e. `&` appears in front of a variable name, it is the address operator. It returns the address of the variable in front of which it is written. Thus for example, if `x` is a variable name, then `&x` ; will return the address of the variable `x`. In general we can say that

`&variablename ;`

will return the address of the variable. We know that an address can be stored in a pointer. To further understand this concept, let's suppose that there are following lines in our code.

```
int x ;  
int* ptr = &x;
```

The first line declares a variable `x` of type `int` while the second one declares a pointer to `int` and assigns it the address of `x`. This address of variable `x` is returned by the `&` sign written in front of the variable `x`. Thus a pointer variable is initialized by assigning it the address of a variable. This address of the variable is gotten by using the `&` sign with the variable name.

The other place where `&` sign can appear is the signature of the function where it

appears after the type of the parameter. Consider the insert and remove methods from BinarySearchTree class that were declared as the following.

```
void insert( const EType& x );  
void remove( const EType& x );
```

Notice that the & sign is after the type. Here we define the class and functions as templates. Whenever, we use these methods, the type EType will be replaced with a proper data type.

Suppose that we have designed the BinarySearchTree class to hold the integers only. This means that there are no templates and the class can be defined only for integers. So the insert and remove methods of this class will be as follows.

```
void insert( const int& x );  
void remove( const int& x );
```

Here, in the function signature, & sign after the type of the parameter indicates that the parameter is a reference variable. We can use this & sign with any data type i.e. built-in or user defined, of an argument to show that this is a reference variable. This is the syntax for the reference variable. Now let's see why and how we use it and what is its advantage? Let's look at an example that will explain the reference variable. Suppose we have the following three different functions.

The name of the first function is *intMinus1*, written as under.

```
//Function 1  
  
int intMinus1( int oldVal)  
{  
    oldVal = oldVal - 1;  
    return oldVal;  
}
```

This function takes an integer as an argument. The local name of this argument is *oldVal*. The first statement in the body of the function decreases the value of *oldVal* by 1 then the next statement returns this value to the calling function.

The second function is *intMinus2*. This function takes a pointer to integer as an argument. In the statement

```
*oldVal = *oldVal - 2 ;
```

**oldVal* means the value at the location where *oldVal* is pointing to. To get the value this way is called dereferencing. After experiencing a decrease by 2, this value remains at the same location i.e. where *oldVal* is pointing to. Then the pointer to integer is returned. This function is defined as follows.

```
// Function 2  
  
int intMinus2( int* oldVal)  
{
```

```
*oldVal = *oldVal - 2;  
return *oldVal;  
}
```

The third function *intMinus3* takes a reference variable as an argument. Its definition is as under.

```
// Function 3  
  
int intMinus3( int& oldVal)  
{  
    oldVal = oldVal - 3;  
    return oldVal;  
}
```

The & sign after the type in the signature (declaration line) indicates that this argument is a reference variable. Notice that & sign is used only in the function declaration, leaving no need for its use in the body of the function. The statements in the body of the function decrease the value of *oldVal* by 3 and return it to the calling function.

We can see that these three functions have the same return type and their body performs the task of decreasing the passed value. However it is clear from the declaration of these functions that the way of passing argument to these functions is different. We will now see what changes occur while passing the argument in different ways. We will write calling functions those call the above three functions one by one. We will see how these calls are made and what differences happen during the process of passing same value (in different ways) to these functions. The use of call stack will help in ascertaining what happens to these local variables and parameters.

Following is the function that calls the first function i.e. *intMinus1*.

```
void caller()  
{  
    int myInt = 31;  
    int retVal;  
    retVal = intMinus1( myInt );  
    cout << myInt << retVal;  
}
```

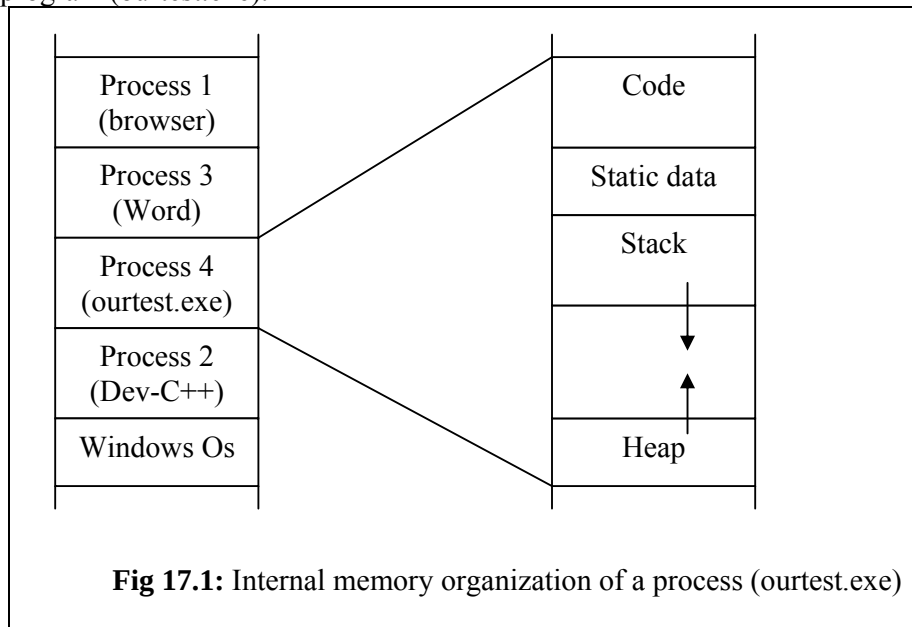
In this function (caller), we declare an integer variable *myInt* and assign it a value 31. We also declare a variable *retVal* of type *int*. In the statement

```
retVal = intMinus1( myInt );
```

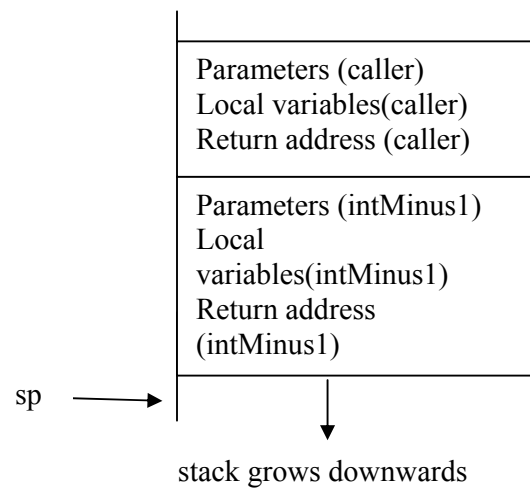
We call the function *intMinus1* by passing it the variable *myInt* (the value of which is 31) and assign the return value of this function call to *retVal*. In the next statement, we print the value of these two variables.

While talking about runtime environment, we noticed that an executable program

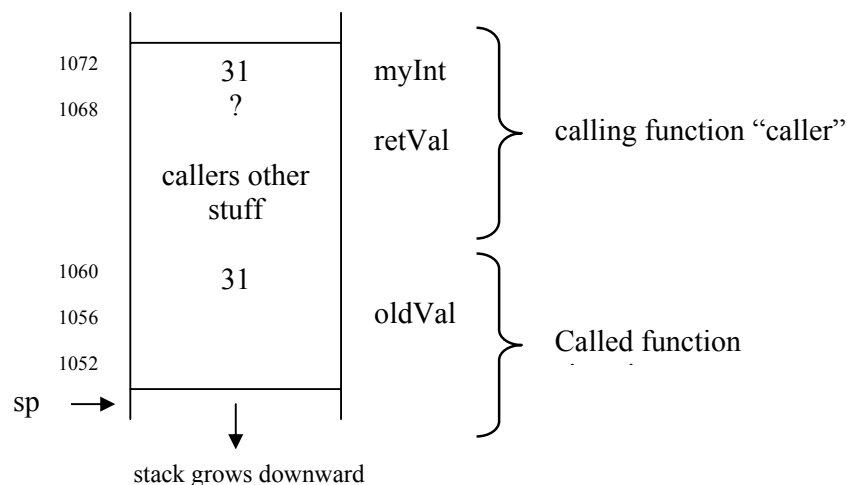
while in run, is loaded in the memory. It later becomes a process. This process is given a block of memory, which it uses during its execution. Suppose we are running many programs simultaneously i.e. browser, MS Word, Excel and dev-C++. We can also run programs written by us. The figure 17.1 shows that process4 is a program written by us. Every program, which we run takes a block of memory and becomes a process. The memory occupied by a process is further partitioned into different parts. The first part of the memory is for the code of the program. This code is in binary form i.e. it is the compiled code. Afterwards, there is some area of memory for static data that holds static and global variables. Then in the memory, there becomes the stack. This stack is used in function calls. At the end, there is some area, called heap. When we allocate memory dynamically by using *new* operator, this memory is allocated from the heap. The following figure shows the internal memory organization of a program (ourtest.exe).



We have discussed in detail the call stack that is used in function calls. We know that when a function call is made, it's all the arguments go on the stack. The return address of the function also goes to the stack. The local variables are placed on the stack. When a function calls some other function, an activation record is made. It has much information including all these things (parameters, return address etc). The detail of activation record relates to the compiler construction course. Here we are concerned with only the call stack. Following figure shows the call stack layout.

**Fig 17.2:** call stack layout

In the stack layout figure, there are the entries of the caller in the upper portion. This is due to the fact that the caller function itself was called (executed) by some one else. The lower portion of the stack describes the stack status when the function *intMinus1* is called. The stack contains parameters, local variables and return address of the function *intMinus1*. There is a stack pointer *sp* that points to the top of the stack (the figure shows the stack downward; it should not be confused with the top). As the caller has called *intMinus1*, this function is on the top of the stack. The following figure shows the contents of the stack when the function *intMinus1* is called.

**Fig 17.3:** call stack layout when *intMinus1* is called

The figure shows the stack in two parts. The first part that is in the upper curly bracket, shows the contents of the calling function i.e. caller. The lower part shows the contents of the called function i.e. *intMinus1*. Recall that we declared two

variables in the caller function. These variables were *myInt* and *retVal*. We also assigned a value 31 to *myInt*. This value (31) is in the memory location of *myInt* in the stack. When we declared the variable *retVal*, there was no value assigned to it. So its memory location contains nothing yet. After it, there is the other stuff of the caller function. We have also shown the memory addresses of the locations on the left hand side. We can see that the memory address of *myInt* is 1072 while that of *retVal* is 1068. Similarly, we see the addresses 1060, 1056 and 1052 further down. We note that the addresses are with a difference of four. This is because of the use of the integer variable, as an integer requires four bytes in the memory for storing. Similarly the pointer variables are also of four bytes. The memory addresses are in 32 bits that is four bytes (eight bits make a byte). Generally we start addressing the memory from down. The lowest byte is addressed 0, followed by 1, 2, 3 and so on to upward direction. If we have larger variables, for example, objects created by our defined classes, these will require more memory. In that case, the memory chunks will not be of four bytes. Rather, it will be equal to the size of the object. In C++, we have an operator *size* or *sizeof* by which the size of a type or variable, acquired in the memory, can be found. There are some machines that do not use byte addressing. These machines use word addressing. Every word is of 48 bits. All these things (memory addressing etc) relate to computer architecture course.

Now in the stack, there are the entries of the called function. There is the parameter *oldVal* of the called function in the stack. It has the value 31. This is the same value as that of *myInt* in the caller function. As studied earlier, in call by value phenomenon, when arguments (whatever type they have) are sent to a function, a copy of these arguments is made before sending. That copy is sent to the called function. The called function uses this copy of the argument. Thus a copy of *myInt* is passed to the called function *intMinus1*. Thus its parameter *oldVal* has value 31. Now *intMinus1* function uses this value (i.e. *oldVal*) to do its functionality. It makes alternations to this copy. The original value that is *myInt* does not change. This phenomenon is shown in the following figure. Here the function *intMinus1* decreases the value of *oldVal* by 1 and becomes 30. The original value (*myInt*) remains unchanged.

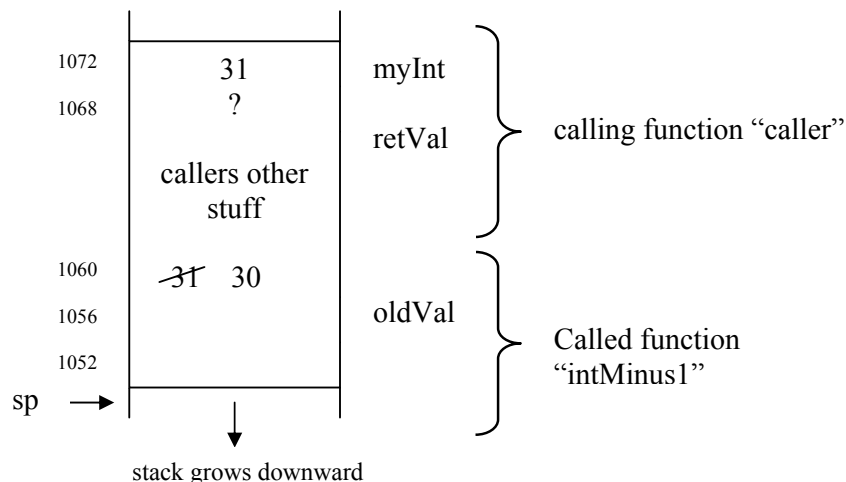


Fig 17.4: call stack layout after subtraction in *intMinus1*

When the return call of *intMinus1* is executed, the control comes back to the calling function. The stuff of the called function *intMinus1* is popped from the stack and the

pointer *sp* moves up accordingly. In other words, the activation record of the function *intMinus1* has been removed from the stack. The *oldval* (value of which is 30 now) is returned to the caller function. We have written in the code that this value is assigned to *retVal* variable. So in the call stack layout, the memory location of *retVal* has the value 30. The figure 17.5 shows the stack layout after return from the called function i.e. *intMinus1*.

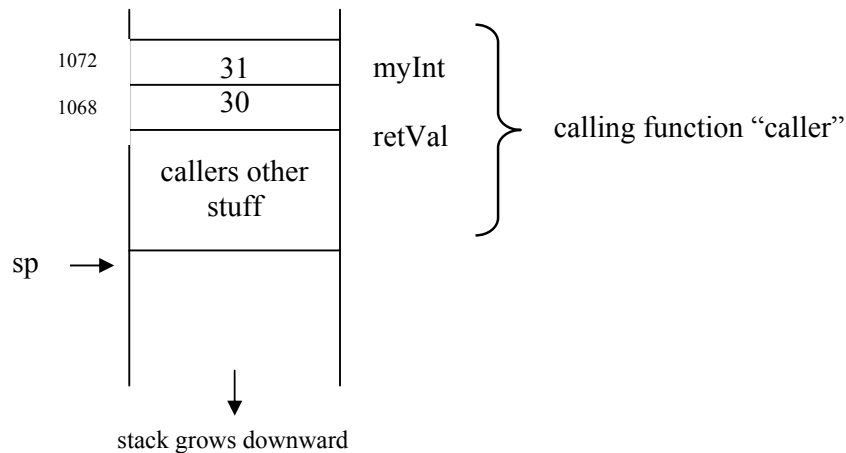


Fig 17.5 call stack layout after return from *intMinus1*

In the call by value phenomenon of function calling, we can pass a literal or even an expression. Thus in the previous function *caller*, we could have called the function *intMinus1* as follows.

```
void caller()
{
    int retVal;
    retVal = intMinus1( 31 ); // a literal is passed
    cout << retVal;
}
```

Here we did not put the value 31 in an integer variable. Rather, it was directly passed as an argument in the function call i.e. *intMinus1(31)*. We are not passing a variable as it was in the previous code of the function *caller*. The calling statement also be written as under:

```
retVal = intMinus1( 30 + 1 );
```

In this statement, first the expression $30 + 1$ is evaluated and then the copy of the result goes to the call stack as a value of *oldVal* of called function.

In the previous calling function (i.e. *caller*), it was witnessed that the value of the passed variable (*myInt*) remained same when we passed the variable by value. Now there may be situations where we want to actually change the value of a variable of the caller function from within the called function. In such a situation, we send the pointer to the variable (i.e. the address of the variable) as an argument. Look at the following code of the function *caller*. In this code, we call the function *intMinus2* by

passing it the address of the variable *myInt*.

```
void caller()
{
    int retVal;
    int myInt = 31;
    retVal = intMinus2( &myInt );
    cout << myInt << retVal;
}
```

Following is the function calling statement

```
retVal = intMinus2( &myInt );
```

The ‘&’ sign before the name of the variable *myInt* means that the address of the variable is being passed. The called function will take it as a pointer. Remember that we have declared the argument of the function *intMinus2* as *int* oldVal* that means this function takes a pointer as an argument.

Now, keeping in mind that a pointer is being sent to the function, let’s see what is happening in the call stack. The stack portion which has the stuff of the calling function i.e. *caller* is the same as seen in the call by value process. The variable *myInt* has value 31 and value of *retVal* will be written after returning from the called function. Its stack layout is shown in the figure 17.6 below.

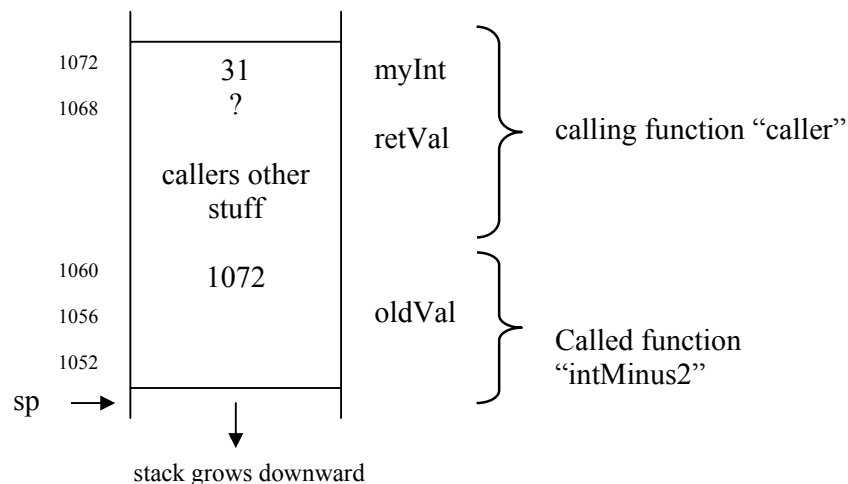


Fig 17.6 call stack layout when *intMinus2* is called

In the stuff of the called function i.e. *intMinus2*, the memory location of the *oldVal* holds the value 1072. We can see on the left side of the stack in the figure above that this 1072 is the address of the variable *myInt*. Look at the statement

```
*oldVal = *oldVal - 2 ;
```

of *intMinus2*. Here **oldVal* can be expressed as ‘the value at the memory address that

is in *oldVal*'. In the stack layout, we see that the memory address in *oldVal* is 1072. The value stored at the address 1072 is 31. The following statement

$$*oldVal = *oldVal - 2;$$

decreases this value 2 and thus brings the value at the memory address 1072 down to 29. As 1072 is the address of *myInt*, thus actually the value of *myInt* is changed. The following figure of stack layout depicts this process.

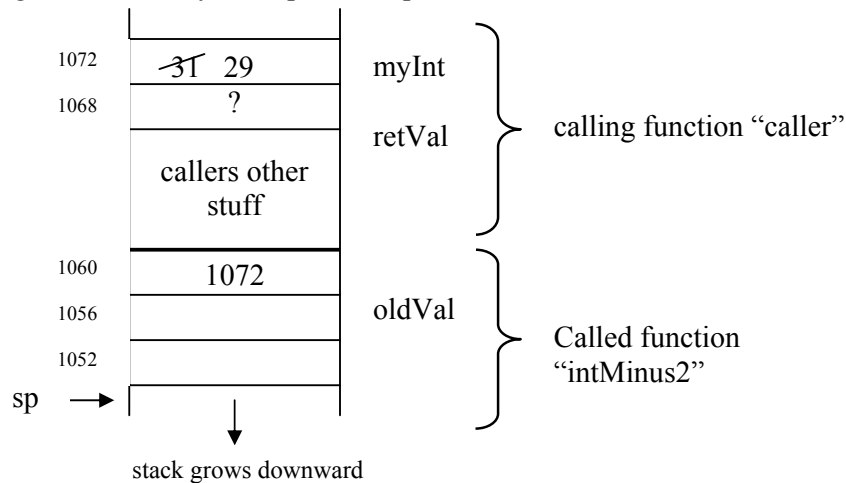


Fig 17.7: call stack layout after `*oldVal = *oldVal - 2;`

Now when the execution of the called function ends after returning **oldVal*, the activation record of this function is removed from the stack. The pointer of the stack comes up. The value 29 is put in the variable *retVal* of the calling function. Following is the stack layout after return from the function *intMinus2*.

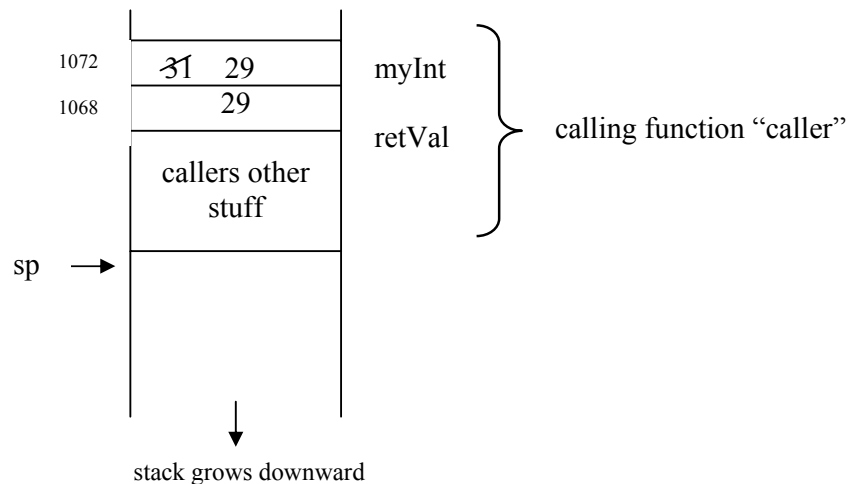


Fig 17.8: call stack after return from *intMinus2*

Notice that the value of *myInt* of the caller function has also been changed.

We have seen that in call by value a copy of the argument is passed and used by the called function. In this method, the original value of the argument in the calling function remains unchanged. In the other method, we saw that when we pass the address of the variable, the pointer is used to manipulate the variable data. In this case, the original value of the variable in the calling function is changed. Suppose that we want a function to change an object (variable/argument) but don't want to send the copy of the object to the function. The reason of not sending a copy is that the object may be large (as we have seen the object *Customer* in bank simulation example) and making a copy of it costs time and memory space. Say, the object *Customer* is of 500 bytes. Now when a calling function will call a function using call by value method, a copy of this object will be made on the call stack same as in our previous example a copy of *myInt* was made. The copy constructor will make a copy of this whole object that costs time. Moreover, the copy of this object will take 500 bytes on the stack that is a lot of memory consuming. There will be as many copies as the number of calls. In case of a large number of calls, there may be inefficient memory as call stack has a limited memory. So we do not use call by value methodology. Furthermore, we want to avoid the massive syntax of pointers. For this purpose, we do not use pointers in function calling. Now the question arises is there any way through which we can fulfill our requirement. (i.e. we don't want to make a copy and want to change the objective without using the pointers). The use of reference variables may be a suitable answer to this very ticklish situation. The phenomenon of function calls using reference variables is termed as call by reference. Following is the code of the caller function that involves call by reference.

```
Void caller()
{
    int retVal;
    int myInt = 31;
    retVal = intMinus3( myInt );
    cout << myInt << retVal;
}
```

Note that this is almost the same function, we wrote at first to call *intMinus1*. The only difference is that here we are calling the function *intMinus3* instead of *intMinus1*. We did not use & sign with *myInt* as used in the call for *intMinus2*. This is only due to the fact that we are not sending an address. It is pertinent to note that in the definition of the function *intMinus3*, we have used & sign with the argument variable. We have written it as

```
int intMinus3( int& oldVal)
```

This statement means that the function *intMinus3* will take a reference as an argument. In other words, a reference of the argument variable will be passed to this function. Thus the idea of reference variable is that an object will be used exactly as it exists in the caller function. The called function simply accesses it through a different name i.e. other than the name in the caller function. Here in our example, the called function *intMinus3* accesses (and uses) the passed argument i.e. *myInt* with the name *oldVal*. The function *intMinus3* cannot use the name *myInt* as it is in the caller's scope. Now the both variable names (*myInt* and *oldVal*) refer to the same object. It

means that the same memory cell where the object lies. We have read the scope of variables that the local variables can be used only by the function in which they are declared. The other functions cannot use the local variables of a function. But in reference variables, we can access the memory location of a local variable in another function by using another name. Thus in our example, we access the local variable *myInt* (actually the memory location) of the function *caller* in the other function *intMinus3* with the name *oldVal*. The following figure explains this phenomenon of reference variables with the help of call stack layout.

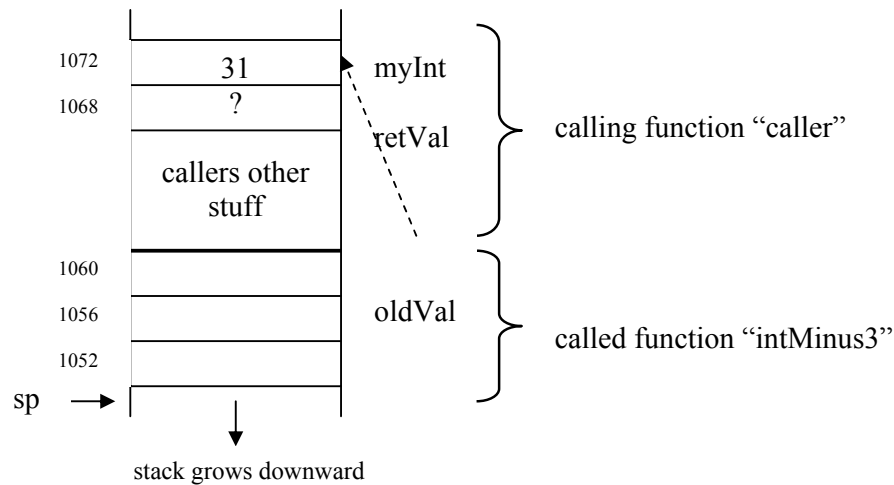


Fig 17.9: call stack when *intMinus3* is called

The caller function part of the stack contains *myInt*, *retVal* and other stuff of the caller function. In the stack part *intMinus3*, the name *oldVal* shows that it has nothing in this portion but it is the other name of the memory location 1072, named as *myInt* in the caller function. The dotted arrow shows this.

In the following figure 17.10, we show the *oldVal* along with the *myInt* at same memory location. Here we want to show that the *oldVal* of called function and *myInt* of caller function are the two names of the same memory location.

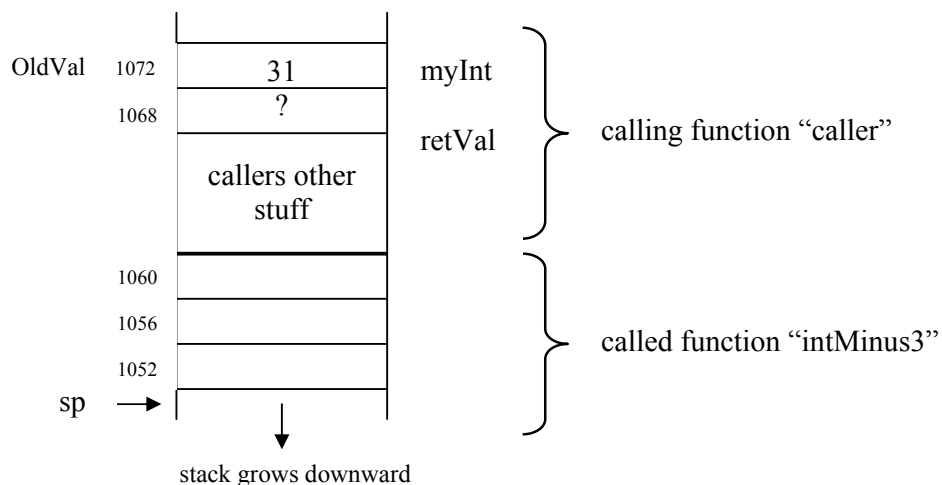


Fig 17.10: call stack when *intMinus3* is called

Now when the body of the *intMinus3* executes the statement i.e.

$$\text{oldVal} = \text{oldVal} - 3 ;$$

It changes the value at the memory location in the caller function as it is referring to that memory location. This value now becomes 28. Actually, the value of *myInt* is changed as it also refers to the same memory location. The following figure explains the stack lay out after the execution of the above statement.

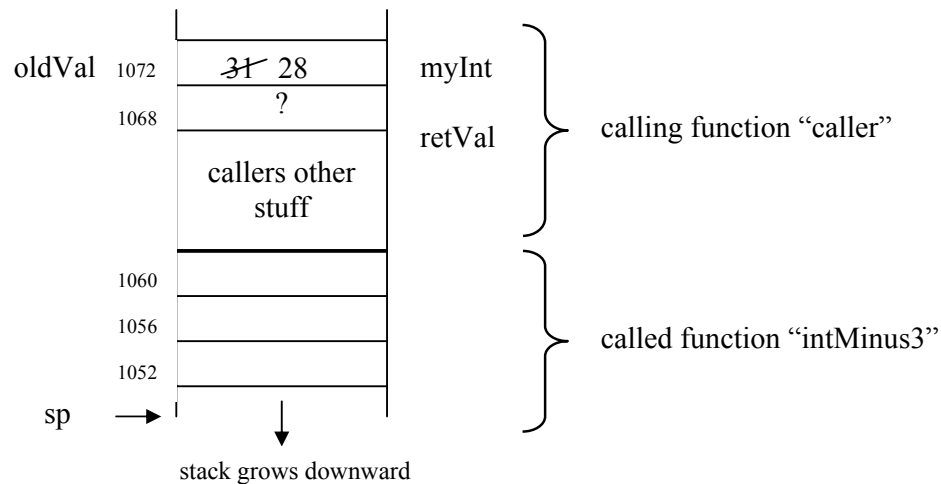


Fig 17.11: call stack layout after $\text{oldVal} = \text{oldVal} - 3 ;$

Now when the function *intMinus3* returns, the returned value (that is 28) is written in *retVal* and the stuff (activation record) of *intMinus3* is removed. The stack layout becomes as shown in the figure below.

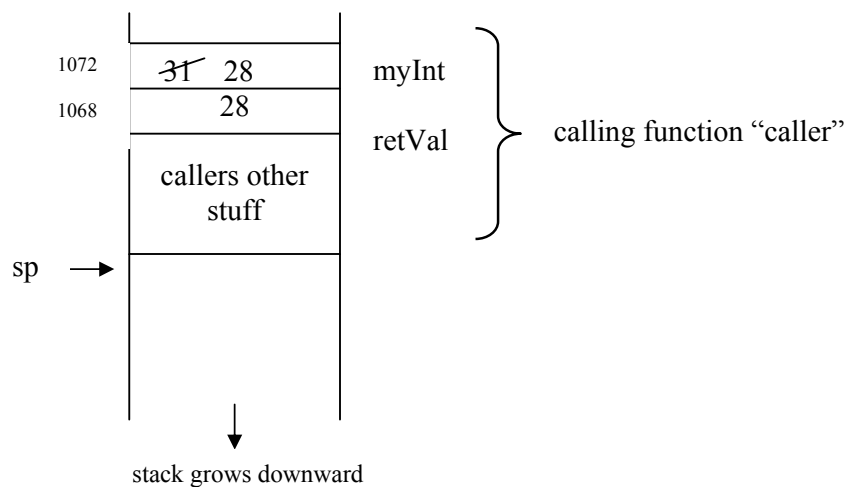


Fig 17.12: call stack layout after return from *intMinus3*

This phenomenon of call by reference is actually implemented by the compiler by using pointers. The obtaining of address and de-referencing will be done behind the scene. We have no concern to do this. For a programmer, it is simply a renaming abstraction, in which we rename the argument variable of the caller function and use it in the called function.

Sample Program

Following is the program, which demonstrate the above three function calls that we discussed in the previous example. We define the three functions `intMinus1`, `intMinus2` and `intMinus3`. These functions accept the argument as value, pointer and reference variable respectively. The *endl* puts a new line and is used in the program for the output clarity. Here is the code of the program followed by the output of the program.

```
/*This program demonstrate tha how the value in a caller function is effected when it
is passed to a function by using call by value, by using pointers and by using call by
reference methods.
*/
#include <iostream.h>

//Function 1, call by value

int intMinus1( int oldVal)
{
    oldVal = oldVal - 1;
    return oldVal;
}

// Function 2, call by using pointers

int intMinus2( int* oldVal)
{
    *oldVal = *oldVal - 2;
    return *oldVal;
}

// Function 3, call by reference

int intMinus3( int& oldVal)
{
    oldVal = oldVal - 3;
    return oldVal;
}

void main ()
{
    int myInt = 31;
```

```
int retVal;
retVal = intMinus1( myInt ); //call by value
cout << "After returning from the called function intMinus1" << endl ;
cout << "The value returned by the called function (retVal) is : " << retVal ;
cout << endl ;
cout << "The value of the calling function's variable (myInt) is : " << myInt ;
cout << endl << endl;

// now pass the argument by using pointer, also initialize the value of myInt

myInt = 31 ;
retVal = intMinus2( &myInt ); //call by passing a pointer
cout << "After returning from the called function intMinus2" << endl;
cout << "The value returned by the called function (retVal) is : " << retVal ;
cout << endl;
cout << "The value of the calling function's variable (myInt) is : " << myInt ;
cout << endl << endl;

// now pass the argument by as reference, also initialize the value of myInt

myInt = 31 ;
retVal = intMinus3( myInt ); //call by passing a reference
cout << "After returning from the called function intMinus3" << endl;
cout << "The value returned by the called function (retVal) is : " << retVal ;
cout << endl;
cout << "The value of the calling function's variable (myInt) is : " << myInt ;
}
```

Following is the output of the program.

```
After returning from the called function intMinus1
The value returned by the called function (retVal) is : 30
The value of the calling function's variable (myInt) is : 31

After returning from the called function intMinus2
The value returned by the called function (retVal) is : 29
The value of the calling function's variable (myInt) is : 29

After returning from the called function intMinus3
The value returned by the called function (retVal) is : 28
The value of the calling function's variable (myInt) is : 28
```

We can see from the output of the program how the passed variable of the caller function is affected by these different ways of function calling. Note that the values of the variables used are the same as we have discussed with the help of call stack layout.

Data Structures

Lecture No. 18

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4
4.3.3

Summary

- Reference Variables
- *const* keyword
- Tips

Reference Variables

In the last lecture we were discussing about reference variables, we saw three examples; call by value, call by reference and call by pointer. We saw the use of stack when a function is called by value, by reference or by pointer. The arguments passed to the function and local variables are pushed on to the stack.

There is one important point to note that in this course, we are using C/C++ but the usage of stack is similar in most of the computer languages like FORTRAN and Java . The syntax we are using here is C++ specific, like we are sending a parameter by pointer using & sign. In Java, the native data types like *int*, *float* are passed by value and the objects are passed by reference. In FORTRAN, every parameter is passed by reference. In PASCAL, you can pass a parameter by value or by reference like C++. You might have heard of ALGOL, this language had provided another way of passing parameter called *call by name*. These kinds of topics are covered in subjects like *Study of Computer Languages* or *Compiler's Theory*.

It is recommended while you are doing your degree, you study other computer languages and compare them from different aspects. Java is quite popular now a day, quite similar in syntax to C++. May be as a next language, you can study that and compare its different aspects with C/C++. The concepts like how a program is loaded into memory to become a process, how the functions are called and the role of stack etc are similar in all major languages.

we have discussed when the variables are passed by reference then behind the scene what goes on inside the stack. There are few important things to take care of while using reference variables:

One should be careful about transient objects that are stored by reference in data structures.

We know that the local variables of a function are created on call stack. Those variables are created inside the function, remains in memory until the control is inside the function and destroyed when the function exits. Activation record comprise of function call parameters, return address and local variables. The activation record remains inside stack

until the function is executing and it is destroyed once the control is returned from the function.

Let's see the following code that stores and retrieves objects in a queue:

```
void loadCustomer( Queue & q)
{
    Customer c1("irfan");
    Customer c2("sohail");
    q.enqueue( c1 );
    q.enqueue( c2 );
}
```

Above given is a small function *loadCustomer(Queue &)*, which accepts a parameter of type *Queue* by reference. Inside the function body, firstly, we are creating *c1* and *c2* *Customer* objects. *c1* and *c2* both are initialized to string values *irfan* and *sohail* respectively. Then we queue up these objects *c1* and *c2* in the queue *q* using the *enqueue()* method and finally the function returns.

Now, the objects created inside the above function are *c1* and *c2*. As local variables are created on stack, therefore, objects are also created on stack, no matter how big is the size of the data members of the object. In the *Bank* example, in previous lecture, we saw that for each customer we have the name (32 characters maximum), arrival time (*int* type, 4 bytes), transaction time (*int* type) and departure time (*int* type) of the customer. So the size of the *Customer* object is 44 bytes. Our *c1* and *c2* objects are created on stack and have 44 bytes occupied. It is important to mention here that we are referring each 44 bytes of allocation with the name of the object. The allocated 44 bytes are bound with the name of the object *c1* or *c2*. Another significant point here is that the function *enqueue()* accepts the object *Customer* by reference. See the code below of *serviceCustomer()* method, which is executed after the *loadCustomer()*.

```
void serviceCustomer( Queue & q)
{
    Customer c = q.dequeue();
    cout << c.getName() << endl;
}
```

The *serviceCustomer(Queue &)* also accepts one parameter of type *Queue* by reference. In the first statement, it is taking out one element from the queue and assigning to newly created object *c*. Before assignment of address of *c1* object (*c1* because it was inserted first), the object *c* is constructed by calling the default (parameter less) constructor. In the next statement, *c.getName()* function call is to get the name of the customer and then to print it. What do you think about it? Will this name be printed or not? Do you see any problem in its execution? In short, this statement will not work.

To see the problem in this statement, we have to understand the mechanism; where the object was created, what was pushed on the stack, when the function *loadCustomer()* returned and what had happened to the objects pushed on to the stack.

The objects *c1* and *c2*, which were created locally in *loadCustomer()* function, therefore,

they were created on stack. After creating the objects, we had added their addresses, not the objects themselves in the queue *q*. When the function *loadCustomer()* returned, the local objects *c1* and *c2* were destroyed but their addresses were there in the queue *q*. After some time, the *serviceCustomer()* is called. The address of the object is retrieved from the queue and assigned to another newly created local object *c* but when we wanted to call a method *getName()* of *c1* object using its retrieved address, we encountered the problem.

This shows that this is true that use of reference alleviate the burden of copying of object but storing of references of transient objects can create problems because the transient object (object created on stack) is destroyed when the function execution finishes.

The question arises, what can we do, if we do not want the objects created in a function to be destroyed. The answer to this is dynamic memory allocation. All the variables or objects created in a function that we want to access later are created on memory heap (sometimes called free store) using the dynamic memory allocation functions or operators like *new*. Heap is an area in computer memory that is allocated dynamically. You should remember that all the objects created using *new* operator have to be explicitly destroyed using the *delete* operator.

Let's see the modified code of *loadCustomer()* function, where the objects created in a function are not transient, means they are created on heap to be used later in the program outside the body of the function *loadCustomer()*.

```
void loadCustomer( Queue & q)
{
    Customer * c1 = new Customer("irfan");
    Customer * c2 = new Customer("sohail");
    q.enqueue( c1 ); // enqueue takes pointers
    q.enqueue( c2 );
}
```

This time, we are creating the same two objects using the *new* operator and assigning the starting addresses of those objects to *c1* and *c2* pointers. Nameless objects (objects accessed by pointers) are called *anonymous* objects.

Here *c1* and *c2* are pointers to the objects not the actual objects themselves, as it was previously. These starting addresses *c1* and *c2* of the objects are then queued using the *enqueue()* method. As the objects lie on the heap, so there will not be any problem and the objects will be accessible after the function *loadCustomer()* returns.

There is a bit tricky point to understand here. Although, the objects are created on heap but the pointer variables *c1* and *c2* are created on stack and they will be definitely destroyed after the *loadCustomer()* activation record is destroyed. Importantly, you should understand the difference between the pointer variables and the actual objects created on heap. The pointer variables *c1* and *c2* were just used to store the starting addresses of the objects inside the function *loadCustomer()*, once the function is returned the pointer variables will not be there. But as the starting addresses of the objects are put in the queue, they will be available to use later after retrieving them from queue using the

dequeue() operation. These dynamic objects will live in memory (one heap) unless explicitly deleted.

By the way, there is another heap, heap data structure that we are going to cover later in this course.

At the moment, see the layout of computer memory and heap as we previously saw in this course. Heap is an area in memory given to a process from operating system when the process does dynamic memory allocation.

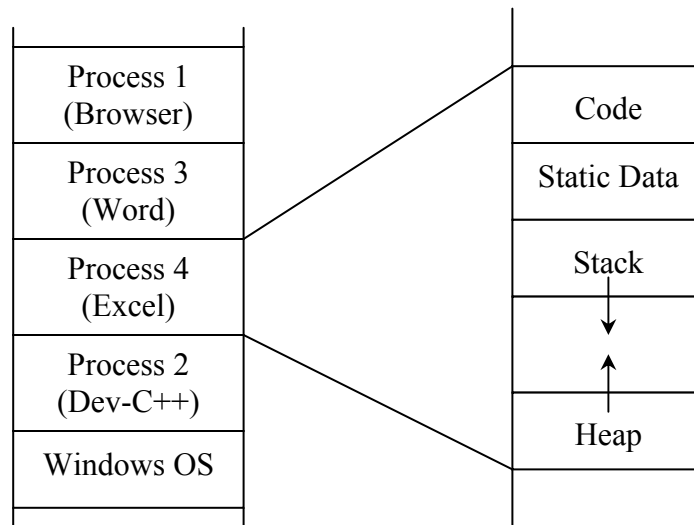


Fig 18.1: Memory Organization

On the left of the picture, we can see different processes in the computer memory. When we zoomed into one of the processes, we saw the picture on the right. That firstly, there is a section for *code*, then for *static data* and for *stack*. Stack grows in the downward section. You can see the heap section given at the end, which grows upward. An interesting question arises here is that why the stack grows downward and heap in the upward direction. Think about an endless recursive call of a function to itself. For every invocation, there will be an *activation record* on stack. So the stack keeps on growing and growing even it overwrites the heap section. On the other hand, if your program is performing dynamic memory allocation endlessly, the heap grows in the upward direction such that it overwrites the stack section and destroys it.

You might have already understood the idea that if a process has some destructive code then it will not harm any other process, only its own destruction is caused. By the way, lot of viruses exploit the stack overflow to change the memory contents and cause further destruction to the system.

Consider that we allocate an array of 100 elements of *Customer* objects dynamically. As each object is 44 bytes, therefore, the size of memory allocated on heap will be 4400 bytes ($44 * 100$). To explain the allocation mechanism on stack and heap, let's see the figure below where the objects are created dynamically.

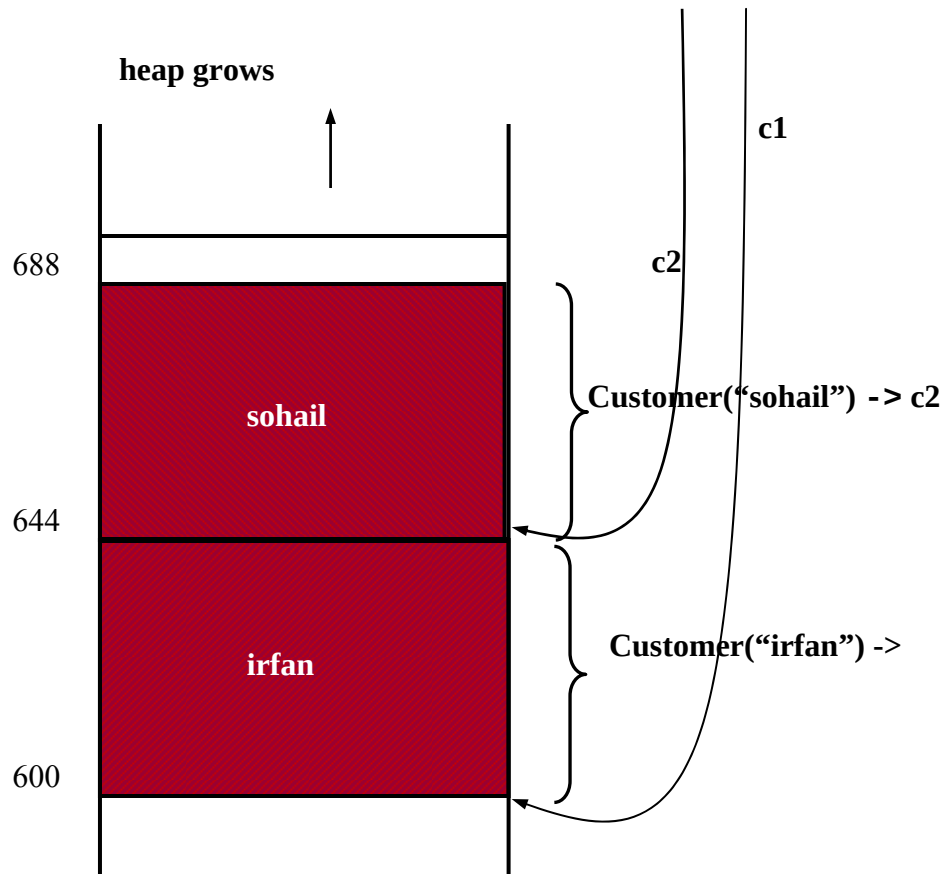


Fig 18.2: Heap layout during call to `loadCustomer`

The objects are shown in this figure by using the names of the customers inside them. Actually, there are three more `int` type variables inside each object. You can see that the object with string `irfan` is from memory address 600 to 643 and object with name customer name as `sohail` is from address 644 to 687. Now when these objects are inserted in the queue, only their starting addresses are inserted as shown in the below figure.

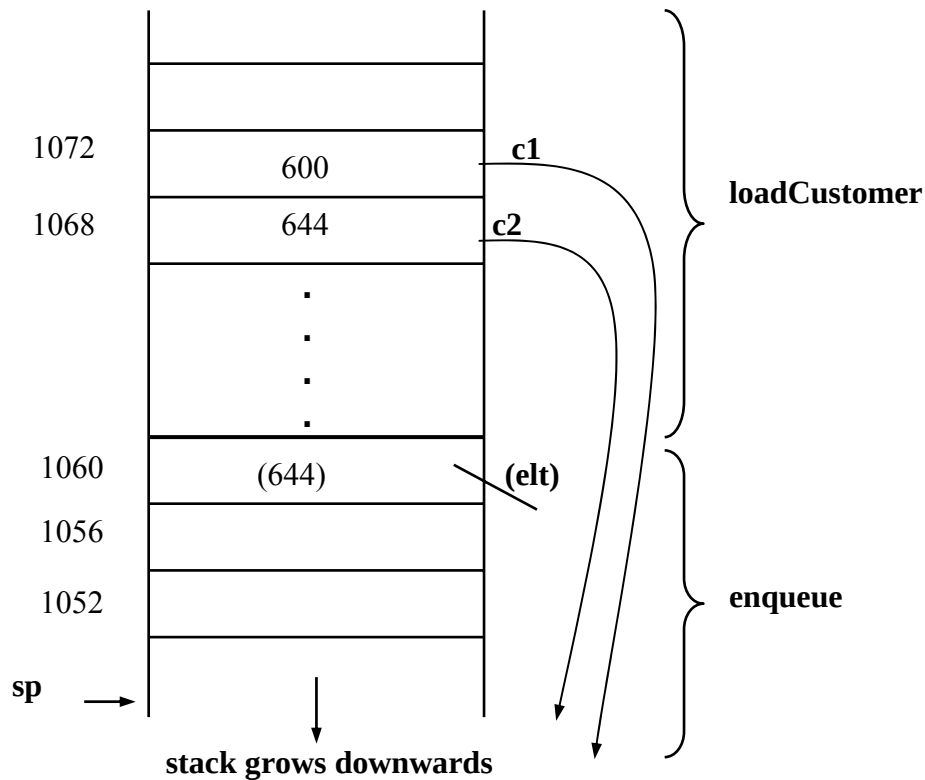


Fig 18.3: Stack layout when *q.enqueue(2)* called from *loadCustomer*

The *loadCustomer()* is being executed. It is containing two pointers *c1* and *c2* containing the addresses 600 and 643 respectively. *enqueue(elt)* method is called and the parameter values (which actually are addresses) 600 and 643 are inserted in the queue.

Because the objects have been allocated on heap, therefore, there will no issue with them. The pointer variables *c1* and *c2*, which we used to store addresses, are destroyed. But the queue *q*, which is passed by reference to *loadCustomer* will be there and it is containing the starting addresses of the *Customer* objects. Those are valid addresses of valid objects, so they can be used in the program later to access the customer objects. See the function below:

```
void serviceCustomer( Queue & q)
{
    Customer* c = q.dequeue();
    cout << c->getName() << endl;
    delete c; // the object in heap dies
}
```

You can see that we are taking one pointer out of the queue and in the second line calling the method of the *Customer* object *getName()* with *c->*. We are using *->* operator because we are taking out pointer from the queue.

Now, we should be sure that this method will be executed successfully because the object

was created dynamically inside the *loadCustomer()* method. The last statement inside the method is *delete*, which has been used to deallocate the object.

So now, we understand that we cannot pass references to transient objects. If we want to use the objects later we create them on heap and keep the address. There is another point to mention here that in case, the object has already been deallocated and we are accessing it (calling any of its member), it may cause the program to crash. The pointer of the object (when object has already been deallocated or released) is called *dangling* pointer.

The *const* Keyword

The *const* keyword is used for something to be constant. The actual meanings depend on where it occurs but it generally means something is to be held constant. There can be constant functions, constant variables or parameters etc.

The references are pointers internally, actually they are constant pointers. You cannot perform any kind of arithmetic manipulation with references that you normally do with pointers. You must be remembering when we wrote header file for binary tree class, we had used *const* keyword many times. The *const* keyword is often used in function signatures. The function signature is also called the function prototype where we mention the function name, its parameters and return type etc.

Here are some common uses of *const* keyword.

1. The *const* keyword appears before a function parameter. E.g., in a chess program:

```
int movePiece(const Piece & currentPiece)
```

The function *movePiece()* above is passed one parameter, which is passed by reference. By writing *const*, we are saying that parameter must remain constant for the life of the function. If we try to change value, for example, the parameter appears on the left side of the assignment, the compiler will generate an error. This also means that if the parameter is passed to another function, that function must not change it either.

Use of *const* with reference parameters is very common. This is puzzling; why are we passing something by reference and then make it constant, i.e., don't change it? Doesn't passing by reference mean we want to change it?

Think about it, consult your C++ book and from the internet. We will discuss about the answer in the next lecture.

Tips

- The arithmetic operations we perform on pointers, cannot be performed on references
- Reference variables must be declared and initialized in one statement.
- To avoid dangling reference, don't return the reference of a local variable (transient) from a function.
- In functions that return reference, return global, static or dynamically allocated variables.
- The reference data types are used as ordinary variables without any dereference operator. We normally use arrow operator (->) with pointers.
- *const* objects cannot be assigned any other value.
- If an object is declared as *const* in a function then any further functions called from this function cannot change the value of the *const* object.

Data Structures

Lecture No. 19

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4

4.4

Summary

- Usage of const keyword
- Degenerate Binary Search Tree
- AVL tree

Usage of const keyword

In the previous lecture, we dealt with a puzzle of constant keyword. We send a parameter to a function by using call by reference and put *const* with it. With the help of the reference variable, a function can change the value of the variable. But at the same time, we have used the *const* keyword so that it does not effect this change. With the reference parameter, we need not to make the copy of the object to send it to the calling function. In case of call by value, a copy of object is made and placed at the time of function calling in the activation record. Here the copy constructor is used to make a copy of the object. If we don't want the function to change the parameter without going for the use of time, memory creating and storing an entire copy of, it is advisable to use the reference parameter as *const*. By using the references, we are not making the copy. Moreover, with the *const* keyword, the function cannot change the object. The calling function has read only access to this object. It can use this object in the computation but can not change it. As we have marked it as constant, the function cannot alter it, even by mistake. The language is supportive in averting the mistakes.

There is another use of keyword *const*. The *const* keyword appears at the end of class member's function signature as:

```
EType& findMin( ) const;
```

This method is used to find the minimum data value in the binary tree. As you have noted in the method signature, we had written *const* at the end. Such a function cannot change or write to member variables of that class. Member variables are those which appear in the *public* or *private* part of the class. For example in the *BinaryTree*, we have *root* as a member variable. Also the *item* variable in the node class is the member variable. These are also called state variables of the class. When we create an object from the factory, it has these member variables and the methods of this class which manipulate the member variables. You will also use *set* and *get* methods, generally employed to set and get the values of the member variables. The member function can access and change the *public* and *private* member variables of a class. Suppose, we want that a member function can access the member variable but cannot change it. It means that we want to make the variables read only for that member

function. To impose that constraint on the member function, a programmer can put the keyword *const* in the end of the function. This is the way in the C++ language. In other languages, there may be alternative methods to carry out it. These features are also available in other object oriented languages. This type of usage often appears in functions that are supposed to read and return member variables. In the *Customer* example, we have used a method *getName* that returns the name of the customer. This member function just returns the value of member variable *name* which is a *private* data member. This function does not need to change the value of the variable. Now we have written a class and its functions. Why we are imposing such restrictions on it? This is the question of discipline. As a programmer when we write programs, sometimes there are unintentional mistakes. On viewing the code, it seems unbelievable that we have written like this. If these codes contain mistakes, the user will get errors. At that time, it was thought that we have imposed restrictions on the function and can avoid such mistakes at compile time or runtime. The discipline in programming is a must practice in the software engineering. We should not think that our programs are error-free. Therefore, the programming languages help in averting the common errors. One of the examples of such support is the use of *const* keyword.

There is another use of *const*. The *const* keyword appears at the beginning of the return type in function signature:

```
const EType& findMin( ) const;
```

The return type of the *findMin()* function is *EType&* that means a reference is returned. At the start of the return type, we have *const* keyword. How is this implemented internally? There are two ways to achieve this. Firstly, the function puts the value in a register that is taken by the caller. Secondly, the function puts the value in the stack that is a part of activation record and the caller functions gets the value at that point from the stack and use it. In the above example, we have return value as a reference as *EType&*. Can a function return a reference of its local variable? When the function ends, the local variables are destroyed. So returning the reference of local variable is a programming mistake. Therefore, a function returns the reference of some member variable of the class. By not writing the *&* with the return type, we are actually returning the value of the variable. In this case, a copy of the returning variable is made and returned. The copy constructor is also used here to create the copy of the object. When we are returning by value, a copy is created to ascertain whether it is a local variable or member variable. To avoid this, we use return by reference. Now we want that the variable being returned, does not get changed by the calling function especially if it is the member variable.

When we create an object from the factory, the member variable has some values. We do not want that the user of this object has direct access to these member variables. So *get* and *set* methods are used to obtain and change the value of these member variables. This is a programming practice that the values of the object should be changed while using these methods. This way, we have a clean interface. These methods are in a way sending messages to the object like give me the name of the customer or change the name of the customer. The presence of a queue object can help us send a message to it that gets an object and returns it. In these function-calling mechanisms, there are chances that we start copying the objects that is a time consuming process. If you want that the function returns the reference of the member

variable without changing the value of the member variable using this reference, a construct is put at the start of the function. It makes the reference as a *const* reference. Now the value of this member variable cannot be changed while using this reference. The compiler will give error or at the runtime, you will get the error. When we return an object from some function, a copy is created and returned. If the object is very big, it will take time. To avoid this, we return this through the reference. At this point, a programmer has to be very careful. If you do not use the *const* with the reference, your object is not safe and the caller can change the values in it.

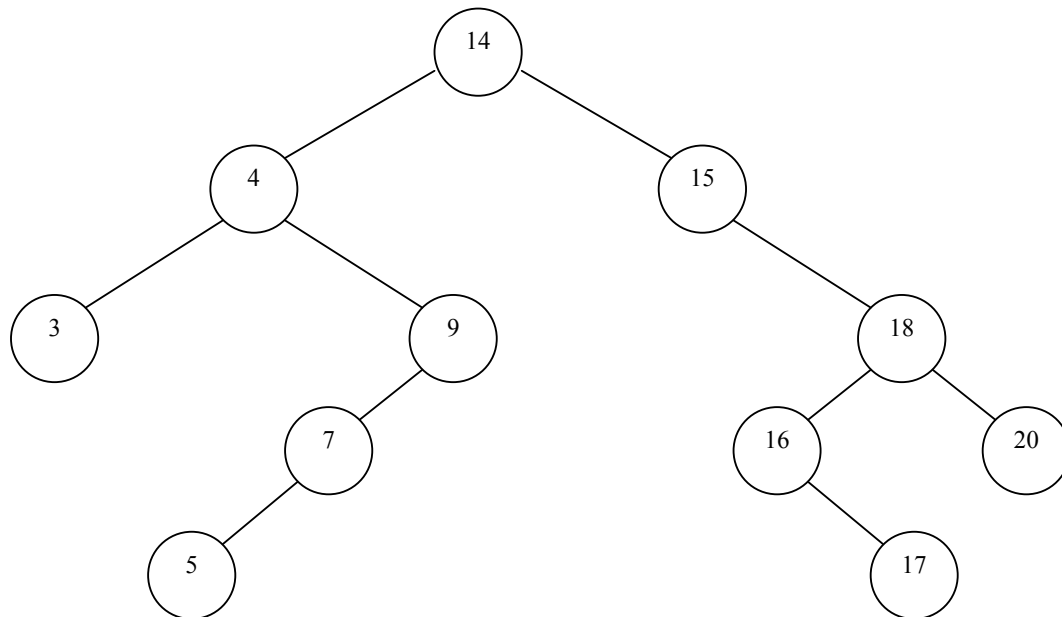
These are the common usage of *const*. It is mostly used with the member function. It is just due to the fact that we avoid creating copy of the object and secondly we get our programming disciplined. When we send a reference to some function or get a reference from some function, in both cases while using the *const*, we guard our objects. Now these objects cannot be changed. If the user of these objects needs to change the object, he should use the *set* methods of the object.

We have used such methods in the *BinarySearchTree.h* file. However, the implementation of this class has not been discussed so far. We advise you to try to write its code yourself and experiment with it.

Degenerate Binary Search Tree

Consider the tree as shown below:

BST for 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17

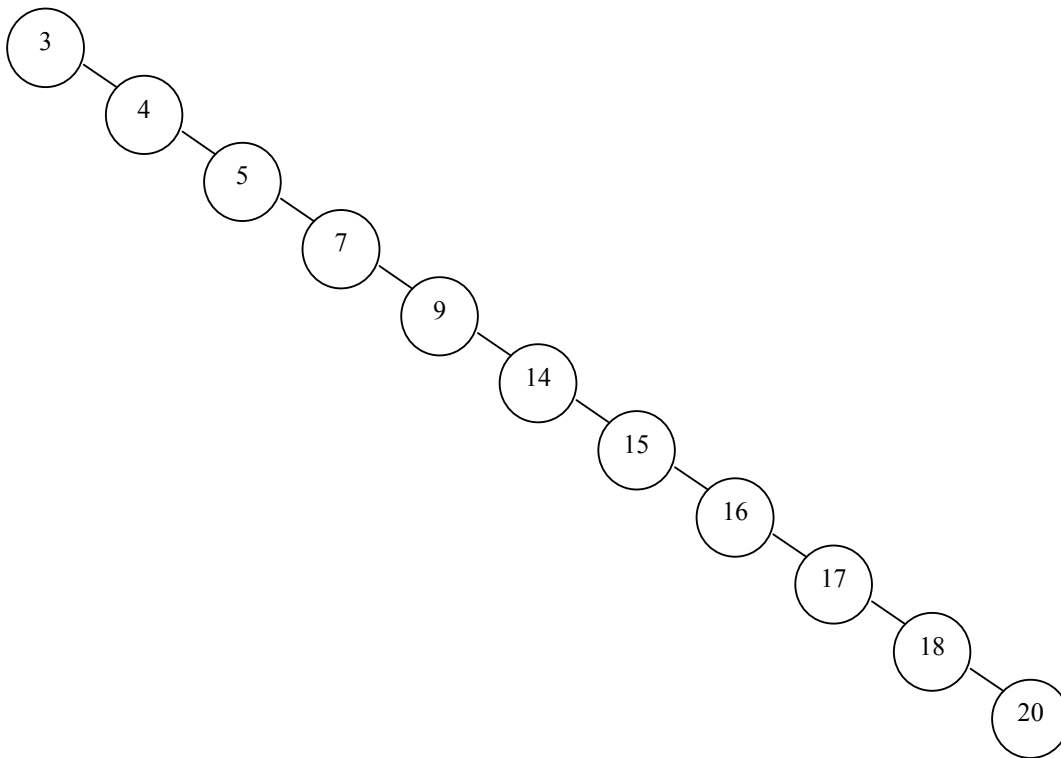


The above tree contains nodes with values as 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17 respectively. The root node is 14. The right subtree contains the numbers greater than 14 and the left subtree contains the numbers smaller than 14. This is the property of the binary search tree that at any node, the left subtree contains the numbers smaller

than this node and the right subtree contains the numbers greater than this node.

Now suppose that we are given data as 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20 to create a tree containing these numbers. Now if our *insert* method takes the data in the order as given above, what will be the shape of our tree? Try to draw a sketch of the tree with some initial numbers in your mind. The tree will look like:

BST for 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20



It does not seem to be a binary tree. Rather, it gives a look of a linked list, as there is a link from 3 to 4, a link from 4 to 5, a link from 5 to 7. Similarly while traversing the right link of the nodes, we reached at the node 20. There is no left child of any node. That's why, it looks like a link list. What is the characteristic of the link list? In link list, every node has a pointer that points to the next node. While following this pointer, we can go to the next node. The root of this tree is 3. Now we have to find the node with value 20 in this tree. Remember that it is a tree, not a link list. We will use *find* method to search the number 20 in this tree. Now we will start from the root. As 20 is greater than 3, the recursive call to the method *find* will be made and we come to the next node i.e. 4. As 20 is greater than 4, so again a recursive call is generated. Similarly we will come to 5, then 7, 9 and so on. In the end, we will reach at 20 and the recursion will stop here.

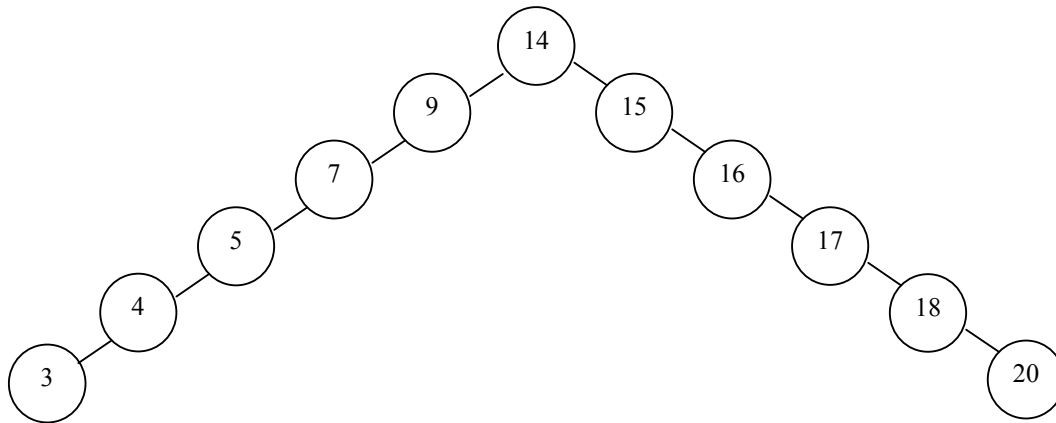
Now if we search the above tree through the method in which we started from 3, then 4, 5 and so on, this will be the same technique as adopted in the link list. How much time it will take to find the number? We have seen in the link list that if the number to be searched is at the last node, a programmer will have to traverse all the nodes. This means that in case of nodes having strength of n , the loop will execute n times.

Similarly as shown in the above tree, our find method will be called recursively equal to number of nodes in the tree. We have designed binary search tree in such a fashion that the search process is very short. You must be remembering the example of previous lecture that if we have one lakh numbers, it is possible to find the desired number in 20 iterations. If we have link list for one lakh elements, the required results can be obtained only after executing the loop for one lakh times if the element to be searched is the last element. However, in case of BST, there are only 20 steps. The BST technique, as witnessed earlier, is quite different as compared to this tree. They have both left and right subtrees. What happened with this tree? The benefit we have due to BST is not applicable here. It seems that it is a link list. This is only due to the fact that the data of the tree was given in the sorted order.

If you want to create a tree out of a sorted data with the *insert* method, it will look like the above tree. It means that you do not want to have sorted data. But it is not easy, as you might not have control over this process. Consider the example of polling. It is not possible that all the voters come to the polling station in some specific order. But in another example, if you are given a list of sorted data and asked to create a BST with this data. If you create a BST with data that is in an ascending order, it will look like a link list. In the link list, the search takes a lot of time. You have created a BST but the operations on it are working as it is a singly link list. How can we avoid that?

We know that the BST is very beneficial. One way to avoid this is that some how we get the sorted data unsorted. How this can be done. It is not possible, as data is not always provided as a complete set. Data is provided in chunks most of the times. Now what should we do? We will apply a technique here so that we can get the benefits of the BST. We should keep the tree balanced. In the above tree, nodes have left child and no right child. So this tree is not balanced. One way to achieve it is that both the left and right subtrees have the same height. While talking about the binary search tree, we discussed the height, depth and level of BST. Every node has some level. As we go down to the tree from the root, the levels of the tree increased and also the number of nodes, if all the left and right subtrees are present. You have earlier seen different examples of tree. The complete binary tree is such a tree that has all the left and right subtrees and all the leaf nodes in the end. In the complete binary tree, we can say that the number of nodes in the left subtree and right subtree are equal. If we weigh that tree on the balance, from the root, both of its sides will be equal as the number of nodes in the right subtree and left subtree are equal. If you have such a balanced binary search tree with one lakh nodes, there will need of only 20 comparisons to find a number. The levels of this tree are 20. We have also used the formula $\log_2(100,000)$. The property of such a tree is that the search comparison can be computed with the help of *log* because subtrees are switched at every comparison.

Now let's see the above tree which is like a singly link list. We will try to convert it into a balanced tree. Have a look on the following figure.



This tree seems to be a balanced tree. We have made 14 as the root. The nodes at the left side occur at the left of all the nodes i.e. left subtree of 14 is 9, the left subtree of 9 is 7, the left subtree of 7 is 5 and so on. Similarly the right subtree contains the nodes 15, 16, 17, 18, 20. This tree seems to be a balanced tree. Let's see its level. The node 14 i.e. the root is at level zero. Then at level one, we have 9 and 15. At level two, there are 7 and 16. Then 5 and 17, followed by 4 and 18. In the end, we have 3 and 20. It seems that we have twisted the tree in the middle, taking 14 as a root node. If we take other nodes like 9 or 7, these have only left subtree. Similarly if we take 15 or 16, these have right subtrees only. These nodes do not have both right and left subtree. In the earlier example, we have seen that the nodes have right and left subtrees. In that example, the data was not sorted. Here the tree is not shallow. Still we can not get the required BST. What should we do? With the sorted data, the tree can not become complete binary search tree and the search is not optimized. We want the data in unsorted form that may not be available.

We want to make a balanced tree, keeping in mind that it should not be shallow one. We could insist that every node must have left and right subtrees of same height. But this requires that the tree be a complete binary tree. To achieve it, there must be $(2^{d+1} - 1)$ data items, where d is the depth of the tree. Here we are not pleading to have unsorted data. Rather, we need as much data which could help make a balanced binary tree. If we have a tree of depth d , there will be need of $(2^{d+1} - 1)$ data items i.e. we will have left and right subtrees of every node with the same height. Now think yourself that is it possible that whenever you build a tree or someone uses your BST class can fulfill this condition. This is not possible that whenever we are going to create a tree, there will be $(2^{d+1} - 1)$ data items for a tree of depth d . The reason is that most of the time you do not have control over the data. Therefore this is too rigid condition. So this is also not a practical solution.

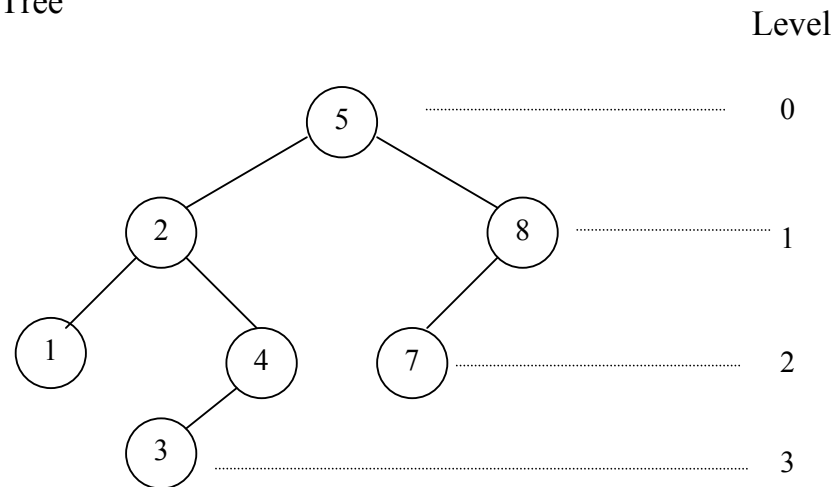
AVL Tree

AVL tree has been named after two persons Adelson-Velskii and Landis. These two had devised a technique to make the tree balanced. According to them, an AVL tree is identical to a BST, barring the following possible differences:

- Height of the left and right subtrees may differ by at most 1.
- Height of an empty tree is defined to be (-1) .

We can calculate the height of a subtree by counting its levels from the bottom. At some node, we calculate the height of its left subtree and right subtree and get the difference between them. Let's understand this with the help of following fig.

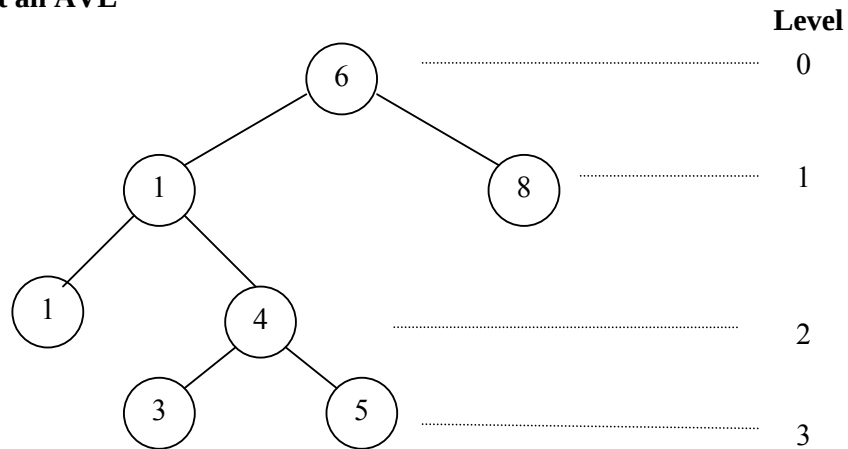
An AVL Tree



This is an AVL tree. The root of the tree is 5. At next level, we have 2 and 8, followed by 1, 4 and 7 at next level where 1, 4 are left and right subtrees of node 2 and 7 is the left subtree of node 8. At the level three, we have 3. We have shown the levels in the figure at the right side. The root is at level 0, followed by the levels 1, 2 and 3. Now see the height of the left subtree of 5. It is 3. Similarly the height of the right subtree is 2. Now we have to calculate the difference of the height of left subtree and right subtree of 5. The height of left subtree of 5 is 3 and height of right subtree of 5 is 2. So the difference is 1. Similarly, we can have a tree in which right subtree is deeper than left subtree. The condition in the AVL tree is that at any node the height of left subtree can be one more or one less than the height of right subtree. These heights, of course, can be equal. The difference of heights can not be more than 1. This difference can be -1 if we subtract the height of left subtree from right subtree where the height of left subtree is one less than the height of right subtree. Remember that this condition is not at the root. It should satisfy at any level at any node. Let's analyze the height of left subtree and right subtree of node 2. This should be -1, 0 or 1. The height of left subtree of node 2 is 1 while that of right subtree of the node 2 is 2. Therefore the absolute difference between them is 1. Similarly at node 8, the height of left subtree is 1 and right subtree does not exist so its height is zero. Therefore the difference is 1. At leaves, the height is zero, as there is no left or right subtree. In the above figure, the balanced condition is satisfactory at every level and node. Such trees have a special structure.

Let's see another example. Here is the diagram of the tree.

- **Not an AVL**



The height of the left subtree of node 6 is three whereas the height of the right subtree is one. Therefore the difference is 2. The balanced condition is not satisfactory. Therefore, it is not an AVL tree.

Let's give this condition a formal shape that will become a guiding principle for us while creating a tree. We will try to satisfy this condition during the insertion of a node in the tree or a deletion of a node from the tree. We will also see later how we can enforce this condition satisfactorily on our tree. As a result, we will get a tree whose structure will not be like a singly linked list.

The definition of height of a tree is:

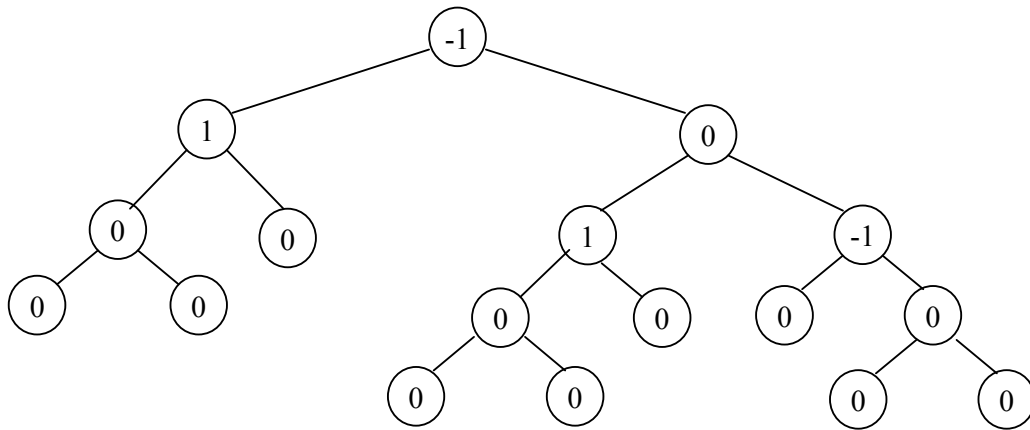
- The *height* of a binary tree is the maximum level of its leaves (also called the depth).

The height of a tree is the longest path from the root to the leaf. This can also be calculated as the maximum level of the tree. If we have to calculate the height of some node, we should start counting the levels from that node.

The balance of a node is defined as:

- The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

Here, for example, is a balanced tree whose each node has an indicated balance of 1, 0, or -1.



In this example, we have shown the balance of each node instead of the data item. In the root node, there is the value -1. With this information, you know that the height of the right subtree at this node is one greater than that of the left subtree. In the left subtree of the root, we have node with value 1. You can understand from this example that the height of the right subtree at this node is one less than the height of the left subtree. In this tree, some nodes have balance -1, 0 or 1. You have been thinking that we have to calculate the balance of each node. How can we do that? When we create a tree, there will be a need of some information on the balance factor of each node. With the help of this information, we will try to balance the tree. So after getting this balance factor for each node, we will be able to create a balance tree even with the sorted data. There are other cases, which we will discuss, in the next lecture. In short, a balance tree with any kind of data facilitates the search process.

Data Structures

Lecture No. 20

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4

4.4

Summary

- AVL Tree
- Insertion in AVL Tree
- Example (AVL Tree Building)

We will continue the discussion on AVL tree in this lecture. Before going ahead, it will be better to recap things talked about in the previous lecture. We built a balanced search tree (BST) with sorted data. The numbers put in that tree were in increasing sorted order. The tree built in this way was like a linked list. It was witnessed that the use of the tree data structure can help make the process of searches faster. We have seen that in linked list or array, the searches are very time consuming. A loop is executed from start of the list up to the end. Due to this fact, we started using tree data structure. It was evident that in case, both the left and right sub-trees of a tree are almost equal, a tree of n nodes will have $\log_2 n$ levels. If we want to search an item in this tree, the required result can be achieved, whether the item is found or not, at the maximum in the $\log n$ comparisons. Suppose we have 100,000 items (number or names) and have built a balanced search tree of these items. In 20 (i.e. $\log 100000$) comparisons, it will be possible to tell whether an item is there or not in these 100,000 items.

AVL Tree

In the year 1962, two Russian scientists, Adelson-Velskii and Landis, proposed the criteria to save the binary search tree (BST) from its degenerate form. This was an effort to propose the development of a balanced search tree by considering the height as a standard. This tree is known as AVL tree. The name AVL is an acronym of the names of these two scientists.

An AVL tree is identical to a BST, barring one difference i.e. the height of the left and right sub-trees can differ by at most 1. Moreover, the height of an empty tree is defined to be (-1) .

Keeping in mind the idea of the level of a tree, we can understand that if the root of a tree is at level zero, its two children (subtrees) i.e. nodes will be at level 1. At level 2, there will be 4 nodes in case of a complete binary tree. Similarly at level 3, the number of nodes will be 8 and so on. As discussed earlier, in a complete binary tree, the number of nodes at any level k will be 2^k . We have also seen the level order

traversal of a tree. The term height is identical to the level of a tree. Following is the figure of a tree in which level/height of nodes is shown.

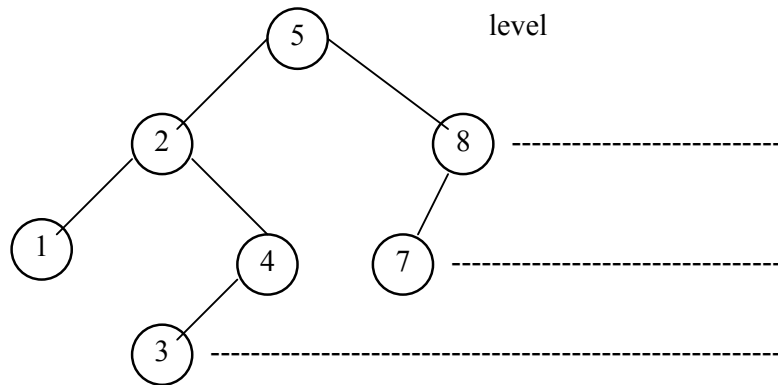


Fig 20.1: levels of nodes in a tree

Here in the figure, the root node i.e. 5 is at the height zero. The next two nodes 2 and 8 are at height (or level) 1. Then the nodes 1, 4 and 7 are at height 2 i.e. two levels below the root. At the last, the single node 3 is at level (height) 3. Looking at the figure, we can say that the maximum height of the tree is 3. AVL states that a tree should be formed in such a form that the difference of the heights (maximum no of levels i.e. depth) of left and right sub-trees of a node should not be greater than 1. The difference between the height of left subtree and height of right subtree is called the balance of the node. In an AVL tree, the balance (also called balance factor) of a node will be 1, 0 or -1 depending on whether the height of its left subtree is greater than, equal to or less than the height of its right subtree.

Now consider the tree in the figure 20.1. Its root node is 5. Now go to its left subtree and find the deepest node in this subtree. We see that node 3 is at the deepest level. The level of this deepest node is 3, which means the height of this left subtree is 3. Now from node 5, go to its right subtree and find the deepest level of a node. The node 7 is the deepest node in this right subtree and its level is 2. This means that the height of right subtree is 2. Thus the difference of height of left subtree (i.e. 3) and height of right subtree (i.e. 2) is 1. So according to the AVL definition, this tree is balanced one. But we know that the AVL definition does not apply only to the root node of the tree. Every node (non-leaf or leaf) should fulfill this definition. This means that the balance of every node should be 1, 0 or -1. Otherwise, it will not be an AVL tree.

Now consider the node 2 and apply the definition on it. Let's see the result. The left subtree of node 2 has the node 1 at deepest level i.e. level 2. The node 2, itself, is at level 1, so the height of the left subtree of node 2 is $2-1$ i.e. 1. Now look at the right subtree of node 2. The deepest level of this right subtree is 3 where the node 3 exists. The height of this right subtree of node 2 will be $3-1 = 2$ as the level of node 2 is 1. Now the difference of the height of left subtree (i.e. 1) and height of the right subtree (i.e. 2) is -1. We subtract the height of left subtree from the height of the right subtree and see that node 2 also fulfills the AVL definition. Similarly we can see that all other nodes of the tree (figure 20.1) fulfill the AVL definition. This means that the balance

of each node is 1, 0 or -1 . Thus it is an AVL tree, also called the balanced tree. The following figure shows the tree with the balance of each node.

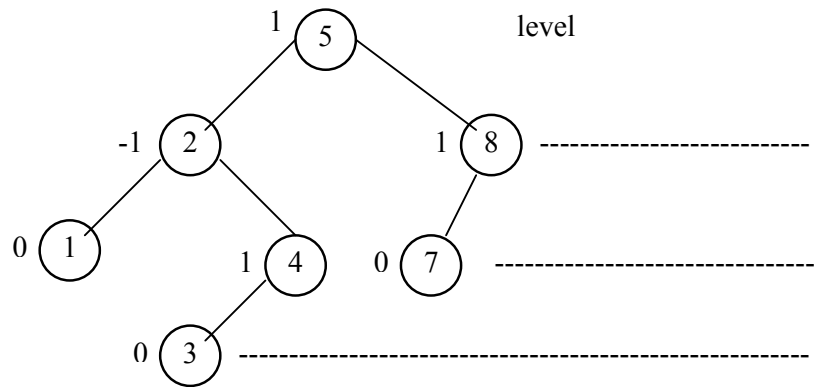


Fig 20.2: balance of nodes in an AVL

Let's consider a tree where the condition of an AVL tree is not being fulfilled. The following figure shows such a tree in which the balance of a node (that is root node 6) is greater than 1. In this case, we see that the left subtree of node 6 has height 3 as its deepest nodes 3 and 5 are at level 3. Whereas the height of its right subtree is 1 as the deepest node of right subtree is 8 i.e. level 1. Thus the difference of heights (i.e. balance) is 2. But according to AVL definition, the balance should be 1, 0 or -1 . As shown in the figure, this node 6 is only the node that violates the AVL definition (as its balance is other than 1, 0 and -1). The other nodes fulfill the AVL definition. We know that to be an AVL tree, each node of the tree should fulfill the definition. Here in this tree, the node 6 violates this definition so this is not an AVL tree.

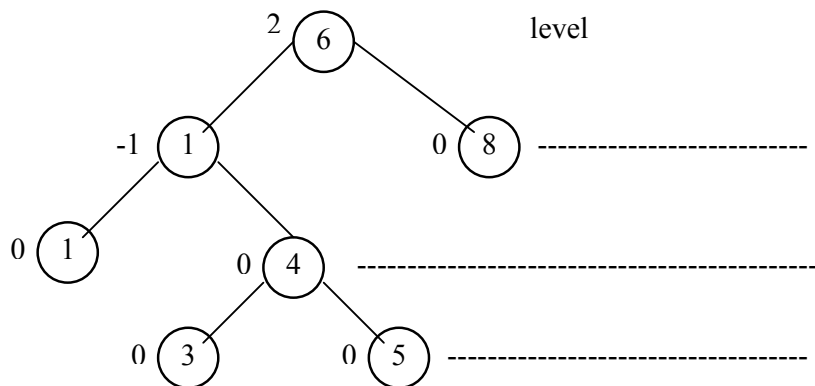


Fig 20.3: not an AVL tree

From the above discussion, we encounter two terms i.e. height and balance which can be defined as under.

Height

The height of a binary tree is the maximum level of its leaves. This is the same definition as of *depth* of a tree.

Balance

The balance of a node in a binary search tree is defined as the height of its left subtree minus height of its right subtree. In other words, at a particular node, the difference in heights of its left and right subtree gives the balance of the node.

The following figure shows a balanced tree. In this figure the balance of each node is shown along with. We can see that each node has a balance 1, 0 or -1.

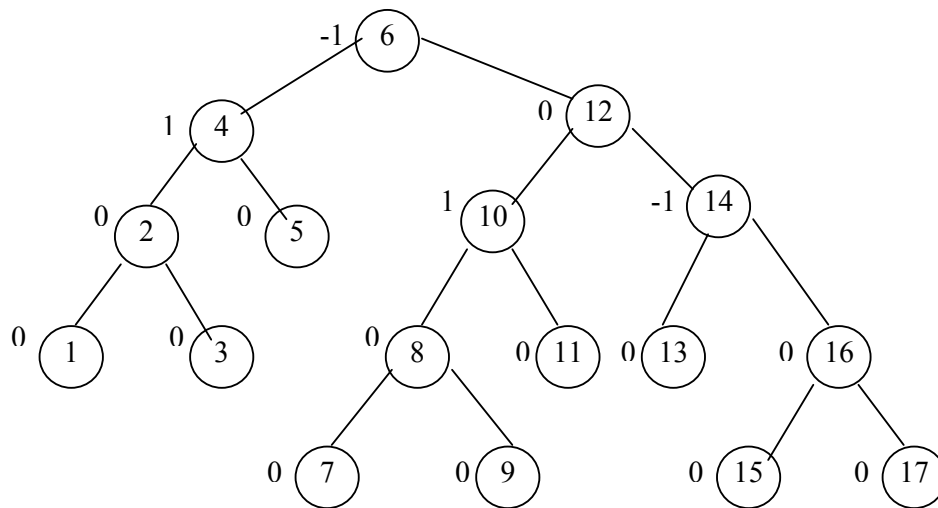


Fig 20.4: A balanced binary tree

Here in the figure, we see that the balance of the root (i.e. node 6) is -1. We can find out this balance. The deepest level of the left subtree is 3 where the nodes 1 and 3 are located. Thus the height of left subtree is 3. In the right subtree, we see some leaf nodes at level 3 while some are found at level 4. But we know that the height of the tree is the maximum level. So 4 is the height of the right subtree. Now we know that the balance of the root node will be the result of height of left subtree minus the height of right subtree. Thus the balance of the root node is $3 - 4 = -1$. Similarly we can confirm the balance of other nodes. The confirmation of balance of the other nodes of the tree can be done. You should do it as an exercise. The process of height computation should be understood as it is used for the insertion and deletion of nodes in an AVL tree. We may come across a situation, when the tree does not remain balanced due to insertion or deletion. For making it a balanced one, we have to carry out the height computations.

While dealing with AVL trees, we have to keep the information of balance factor of the nodes along with the data of nodes. Similarly, a programmer has to have additional information (i.e. balance) of the nodes while writing code for AVL tree.

Insertion of Node in an AVL Tree

Now let's see the process of insertion in an AVL tree. We have to take care that the tree should remain AVL tree after the insertion of new node(s) in it. We will now see how an AVL tree is affected by the insertion of nodes.

We have discussed the process of inserting a new node in a binary search tree in previous lectures. To insert a node in a BST, we compare its data with the root node. If the new data item is less than the root node item in a particular order, this data item will hold its place in the left subtree of the root. Now we compare the new data item with the root of this left subtree and decide its place. Thus at last, the new data item becomes a leaf node at a proper place. After inserting the new data item, if we traverse the tree with the inorder traversal, then that data item will become at its appropriate position in the data items. To further understand the insertion process, let's consider the tree of figure 20.4. The following figure (Fig 20.5) shows the same tree with the difference that each node shows the balance along with the data item. We know that a new node will be inserted as a leaf node. This will be inserted where the facility of adding a node is available. In the figure, we have indicated the positions where a new node can be added. We have used two labels *B* and *U* for different positions where a node can be added. The label *B* indicates that if we add a node at this position, the tree will remain balanced tree. On the other hand, the addition of a node at the position labeled as *U*1, *U*2*U*12, the tree will become unbalanced. That means that at some node the difference of heights of left and right subtree will become greater than 1.

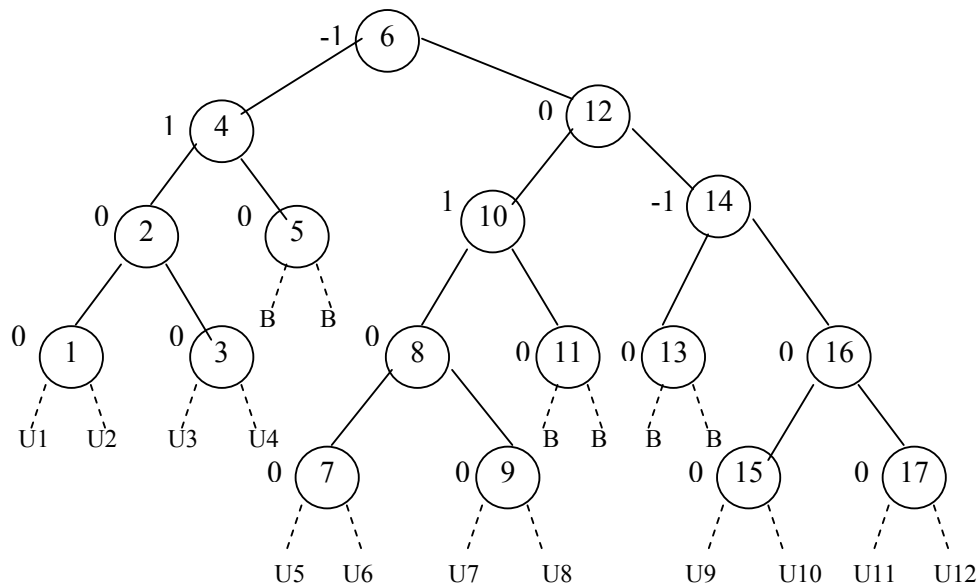


Fig 20.5: Insertions and effect in a balanced tree

By looking at the labels B, U1, U2 U12, we conclude some conditions that will be implemented while writing the code for insert method of a balanced tree.

We may conclude that the tree becomes unbalanced only if the newly inserted node

- Is a left descendent of a node that previously had a balance of 1
(in the figure 20.5 these positions are U1, U2 U8)
- Or is a descendent of a node that previously had a balance of -1
(in the tree in fig 20.5 these positions are U9, U10, U11 and U12)

The above conditions are obvious. The balance 1 of a node indicates that the height of its left subtree is 1 more than the height of its right subtree. Now if we add a node to this left subtree, it will increase the level of the tree by 1. Thus the difference of heights will become 2. It violates the AVL rule, making the tree unbalanced.

Similarly the balance -1 of a node indicates that the right subtree of this node is one level deep than the left subtree of the node. Now if the new node is added in the right subtree, this right subtree will become deeper. Its depth/height will increase as a new node is added at a new level that will increase the level of the tree and the height. Thus the balance of the node, that previously has a balance -1, will become -2.

The following figure (Fig 20.6) depicts this rule. In this figure, we have associated the new positions with their grand parent. The figure shows that U1, U2, U3 and U4 are the left descendents of the node that has a balance 1. So according to the condition, the insertion of new node at these positions will unbalance the tree. Similarly the positions U5, U6, U7 and U8 are the left descendents of the node that has a balance 1. Moreover we see that the positions U9, U10, U11 and U12 are the right descendents of the node that has balance -1. So according to the second condition as stated earlier, the insertion of a new node at these positions would unbalance the tree.

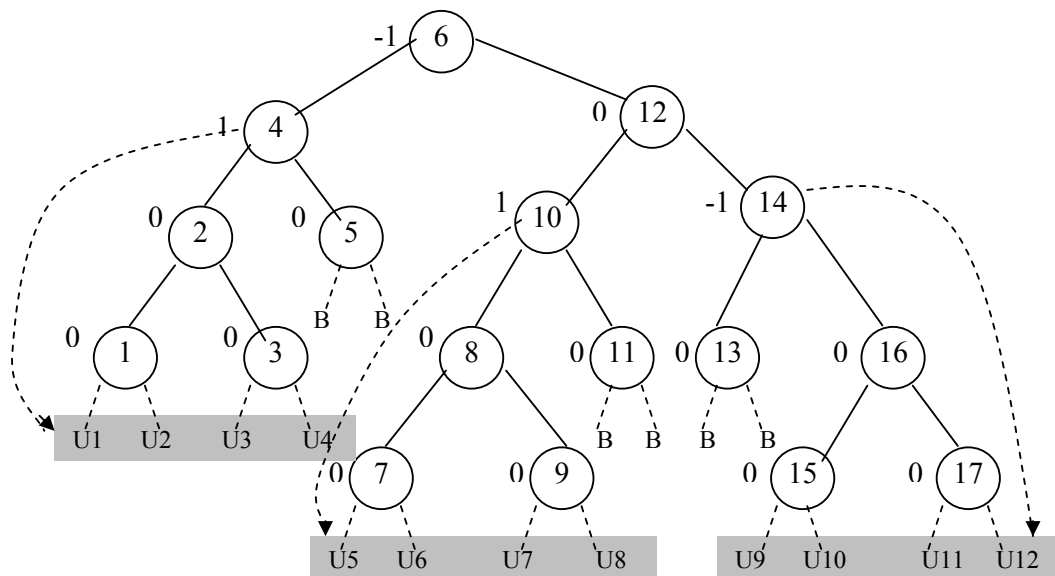


Fig 20.6: Insertions and effect in a balanced tree

Now let's discuss what should we do when the insertion of a node makes the tree unbalanced. For this purpose, consider the node that has a balance 1 in the previous tree. This is the root of the left subtree of the previous tree. This tree is shown as shaded in the following figure.

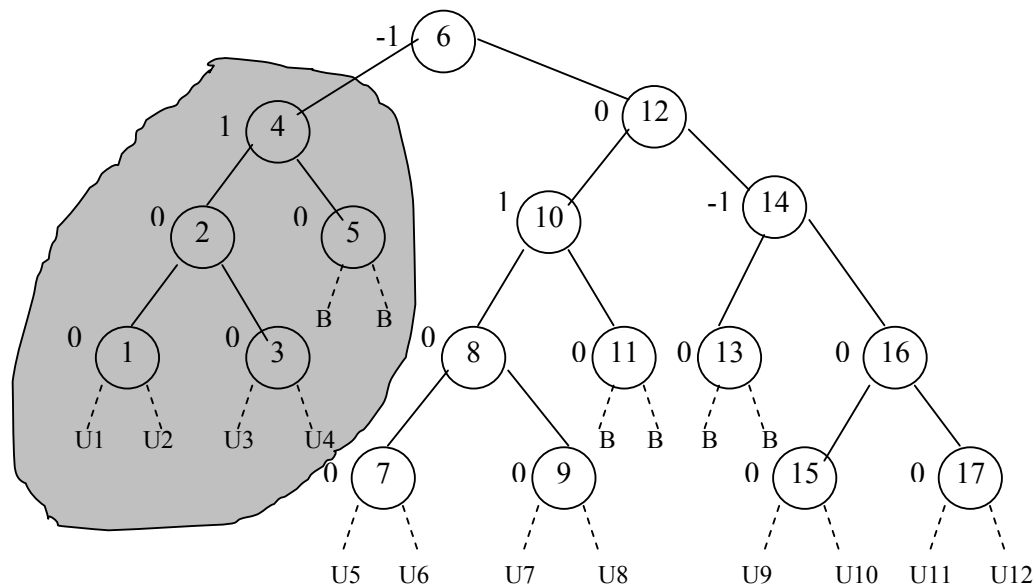
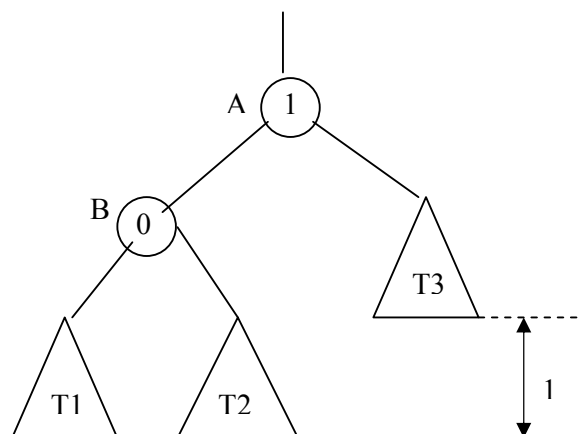


Fig 20.7: The node that has balance 1 under consideration

We will now focus our discussion on this left subtree of node having balance 1 before applying it to other nodes. Look at the following figure (Fig 20.8). Here we are talking about the tree that has a node with balance 1 as the root. We did not mention the other part of the tree. We indicate the root node of this left subtree with label A. It has balance 1. The label B mentions the first node of its left subtree. Here we did not mention other nodes individually. Rather, we show a triangle that depicts all the nodes in subtrees. The triangle T3 encloses the right subtree of the node A. We are not concerned about the number of nodes in it. The triangles T1 and T2 mention the left and right subtree of the B node respectively. The balance of node B is 0 that describes that its left and right subtrees are at same height. This is also shown in the figure. Similarly we see that the balance of node A is 1 i.e. its left subtree is one level deep than its right subtree. The dotted lines in the figure show that the difference of depth/height of left and right subtree of node A is 1 and that is the balance of node A.



Now considering the notations of figure 20.8, let's insert a new node in this tree and observe the effect of this insertion in the tree. The new node can be inserted in the tree T1, T2 or T3. We suppose that the new node goes to the tree T1. We know that this new node will not replace any node in the tree. Rather, it will be added as a leaf node at the next level in this tree (T1). The following figure (fig 20.9) shows this phenomenon.

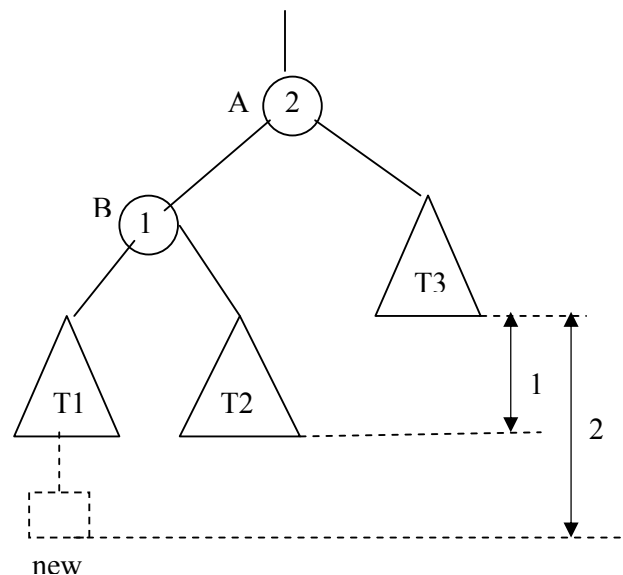


Fig 20.9: Inserting new node in AVL tree

Due to the increase of level in T1, its difference with the right subtree of node A (i.e. T3) will become 2. This is shown with the help of dotted line in the above figure. This difference will affect the balances of node A and B. Now the balance of node A becomes 2 while balance of node B becomes 1. These new balances are also shown in the figure. Now due to the balance of node A (that is 2), the AVL condition has been violated. This condition states that in an AVL tree the balance of a node cannot be other than 1, 0 or -1. Thus the tree in fig 20.9 is not a balanced (AVL) tree.

Now the question arises what a programmer should do in case of violation of AVL condition. In case of a binary search tree, we insert the data in a particular order. So that at any time if we traverse the tree with inorder traversal, only sorted data could be obtained. The order of the data depends on its nature. For example, if the data is numbers, these may be in ascending order. If we are storing letters, then A is less than B and B is less than C. Thus the letters are generally in the order A, B, C This order of letters is called lexographic order. Our dictionaries and lists of names follow this order.

If we want that the inorder traversal of the tree should give us the sorted data, it will not be necessary that the nodes of these data items in the tree should be at particular positions. While building a tree, two things should be kept in mind. Firstly, the tree should be a binary tree. Secondly, its inorder traversal should give the data in a sorted order. Adelson-Velskii and Landis considered these two points. They said that if we see that after insertion, the tree is going to be unbalanced. Then the things should be reorganized in such a way that the balance of nodes should fulfill the AVL condition. But the inorder traversal should remain the same.

Now let's see the example of tree in figure 20.9 and look what we should do to balance the tree in such a way that the inorder traversal of the tree remains the same. We have seen in figure 20.9 that the new node is inserted in the tree T1 as a new leaf node. Thus T1 has been modified and its level is increased by 1. Now due to this, the difference of T1 and T3 is 2. This difference is the balance of node A as T1 and T3 are its left and right subtrees respectively. The inorder traversal of this tree gives us the result as given below.

T1 B T2 A T3

Now we rearrange the tree and it is shown in the following figure i.e. Fig 20.10.

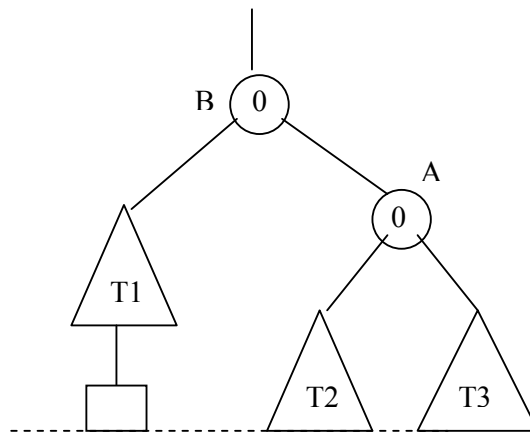


Fig 20.10: Rearranged tree after inserting a new

By observing the tree in the above figure we notice at first that node A is no longer the root of the tree. Now Node B is the root. Secondly, we see that the tree T2 that was the right subtree of B has become the left subtree of A. However, tree T3 is still the right subtree of A. The node A has become the right subtree of B. This tree is balanced with respect to node A and B. The balance of A is 0 as T2 and T3 are at the same level. The level of T1 has increased due to the insertion of new node. It is now at the same level as that of T2 and T3. Thus the balance of B is also 0. The important thing in this modified tree is that the inorder traversal of it is the same as in the previous tree (fig 10.9) and is

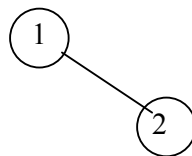
T1 B T2 A T3

We see that the above two trees give us data items in the same order by inorder traversal. So it is not necessary that data items in a tree should be in a particular node at a particular position. This process of tree modification is called rotation.

Example (AVL Tree Building)

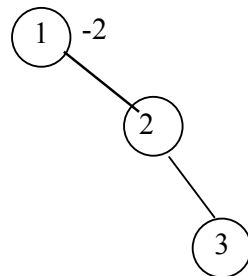
Let's build an AVL tree as an example. We will insert the numbers and take care of the balance of nodes after each insertion. While inserting a node, if the balance of a node becomes greater than 1 (that means tree becomes unbalance), we will rearrange the tree so that it should become balanced again. Let's see this process.

Assume that we have *insert* routine (we will write its code later) that takes a data item as an argument and inserts it as a new node in the tree. Now for the first node, let's say we call *insert (1)*. So there is one node in the tree i.e. 1. Next, we call *insert (2)*. We know that while inserting a new data item in a binary search tree, if the new data item is greater than the existing node, it will go to the right subtree. Otherwise, it will go to the left subtree. In the call to insert method, we are passing 2 to it. This data item i.e. 2 is greater than 1. So it will become the right subtree of 1 as shown below.



As there are only two nodes in the tree, there is no problem of balance yet.

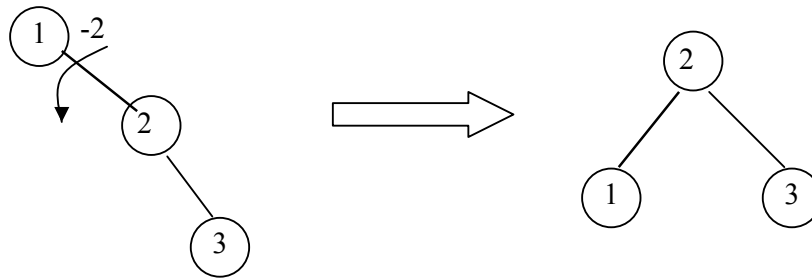
Now insert the number 3 in the tree by calling *insert (3)*. We compare the number 3 with the root i.e.1. This comparison results that 3 will go to the right subtree of 1. In the right subtree of 1 there becomes 2. The comparison of 3 with it results that 3 will go to the right subtree of 2. There is no subtree of 2, so 3 will become the right subtree of 2. This is shown in the following figure.



Let's see the balance of nodes at this stage. We see that node 1 is at level 0 (as it is the root node). The nodes 2 and 3 are at level 1 and 2 respectively. So with respect to the node 1, the deepest level (height) of its right subtree is 2. As there is no left subtree of node 1 the level of left subtree of 1 is 0. The difference of the heights of left and right subtree of 1 is -2 and that is its balance. So here at node 1, the AVL condition has been violated. We will not insert a new node at this time. First we will do the rotation to make the tree (up to this step) balanced. In the process of inserting nodes, we will do the rotation before inserting next node at the points where the AVL condition is being violated. We have to identify some things for doing rotation. We have to see that on what nodes this rotation will be applied. That means what nodes will be

rearranged. Some times, it is obvious that at what nodes the rotation should be done. But there may situations, when the things will not be that clear. We will see these things with the help of some examples.

In the example under consideration, we apply the rotation at nodes 1 and 2. We rotate these nodes to the left and thus the node 1 (along with any tree if were associated with it) becomes down and node 2 gets up. The node 3 (and trees associated with it, here is no tree as it is leaf node) goes one level upward. Now 2 is the root node of the tree and 1 and 3 are its left and right subtrees respectively as shown in the following figure.



Non AVL Tree

AVL tree after applying rotation

We see that after the rotation, the tree has become balanced. The figure reflects that the balance of node 1, 2 and 3 is 0. We see that the inorder traversal of the above tree before rotation (tree on left hand side) is 1 2 3. Now if we traverse the tree after rotation (tree on right hand side) by inorder traversal, it is also 1 2 3. With respect to the inorder traversal, both the traversals are same. So we observe that the position of nodes in a tree does not matter as long as the inorder traversal remains the same. We have seen this in the above figure where two different trees give the same inorder traversal. In the same way we can insert more nodes to the tree. After inserting a node we will check the balance of nodes whether it violates the AVL condition. If the tree, after inserting a node, becomes unbalance then we will apply rotation to make it balance. In this way we can build a tree of any number of nodes.

Data Structures

Lecture No. 21

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4

4.4, 4.4.1

Summary

- AVL Tree Building Example
- Cases for Rotation

AVL Tree Building Example

This lecture is a sequel of the previous one in which we had briefly discussed about building an AVL tree. We had inserted three elements in the tree before coming to the end of the lecture. The discussion on the same example will continue in this lecture. Let's see the tree's figures below:

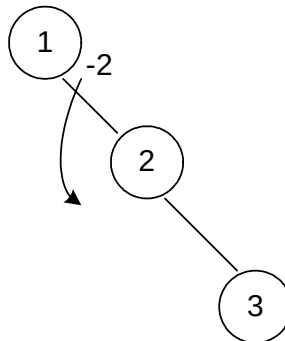


Fig 21.1: insert(3) single left rotation

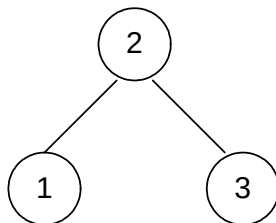


Fig 21.2: insert(3)

Node containing number 2 became the root node after the rotation of the node having number 1. Note the direction of rotation here.

Let's insert few more nodes in the tree. We will build an AVL tree and rotate the node when required to fulfill the conditions of an AVL tree.

To insert a node containing number 4, we will, at first, compare the number inside the

root node. The current *root* node is containing number 2. As 4 is greater than 2, it will take the right side of the root. In the right subtree of the *root*, there is the node containing number 3. As 4 is also greater than 3, it will become the right child of the node containing number 3.

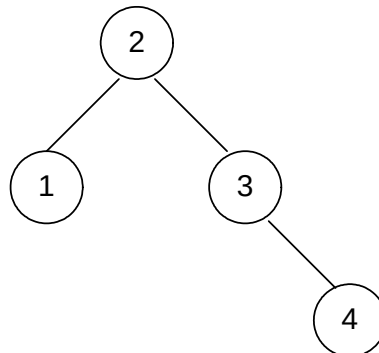


Fig 21.3: insert(4)

Once we insert a node in the tree, it is necessary to check its balance to see whether it is within AVL defined balance. If it is not so, then we have to rotate a node. The balance factor of the node containing number 4 is zero due to the absence of any left or right subtrees. Now, we see the *balance factor* of the node containing number 3. As it has no left child, but only right subtree, the *balance factor* is -1 . The *balance factor* of the node containing number 1 is 0. For the node containing number 2, the height of the left subtree is 1 while that of the right subtree is 2. Therefore, the *balance factor* of the node containing number 2 is $1 - 2 = -1$. So every node in the tree in fig. 21.3 has *balance factor* either 1 or less than that. You must be remembering that the condition for a tree to be an AVL tree, every node's balance needs not to be zero necessarily. Rather, the tree will be called AVL tree, if the *balance factor* of each node in a tree is 0, 1 or -1 . By the way, if the *balance factor* of each node inside the tree is 0, it will be a perfectly balanced tree.

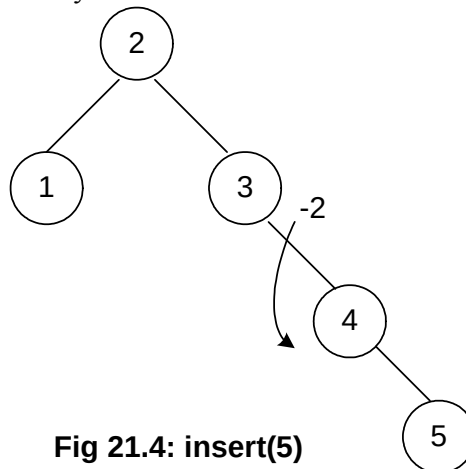
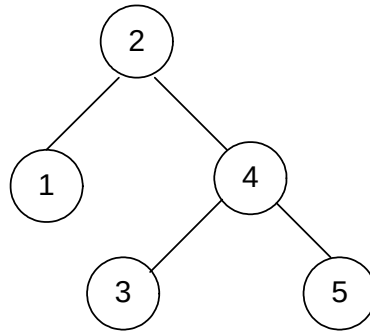


Fig 21.4: insert(5)

Next, we insert a node containing number 5 and see the *balance factor* of each node. The *balance factor* for the node containing 5 is 0. The *balance factor* for node containing 4 is -1 and for the node containing 3 is -2 . The condition for AVL is not satisfied here for the node containing number 3, as its *balance factor* is -2 . The rotation operation will be performed here as with the help of an arrow as shown in the above Fig 21.4. After rotating the node 3, the new tree will be as under:

**Fig 21.5: insert(5)**

You see in the above figure that the node containing number 4 has become the right child of the node containing number 2. The node with number 3 has been rotated. It has become the left child of the node containing number 4. Now, the *balance factor* for different nodes containing numbers 5, 3 and 4 is 0. To get the *balance factor* for the node containing number 2, we see that the height of the left subtree containing number 2 is 1 while height of the right subtree is 2. So the *balance factor* of the node containing number 2 is -1 . We saw that all the nodes in the tree above in Fig 21.5 fulfill the AVL tree condition.

If we traverse the tree Fig 21.5, in inorder tree traversal, we get:

1 2 3 4 5

Similarly, if we traverse the tree in inorder given in Fig 21.4 (the tree before we had rotated the node containing number 3), following will be the output.

1 2 3 4 5

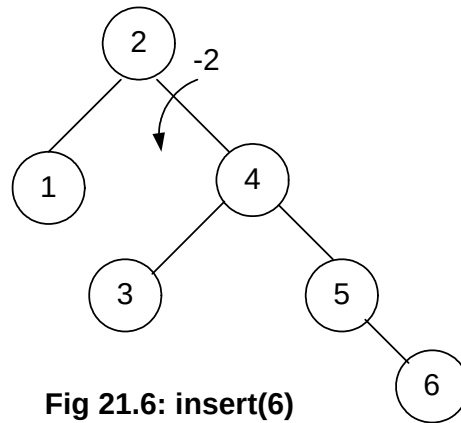
In both the cases above, before and after rotation, we saw that the inorder traversal of trees gives the same result. Also the *root* (node containing number 2) remained the same.

See the Fig 21.4 above. Considering the inorder traversal, we could arrange the tree in such a manner that node 3 becomes the *root* of the tree, node 2 as the left child of node 3 and node 1 as the left child of the node 2. The output after traversing the changed tree in inorder will still produce the same result:

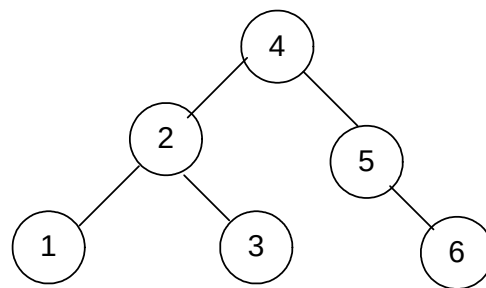
1 2 3 4 5

While building an AVL tree, we rotate a node immediately after finding that that the node is going out of balance. This ensures that tree does not become shallow and remains within the defined limit for an AVL tree.

Let's insert another element 6 in the tree. The figure of the tree becomes:

**Fig 21.6: insert(6)**

The newly inserted node 6 becomes the right child of the node 5. Usually, after the insertion of a node, we will find out the node factor for each node and rotate it immediately. This is carried out after finding the difference out of limit. The *balance factor* for the node 6 is 0, for node 5 is -1 and 0 for node 3. Node 4 has -1 balance factor and node 1 has 0. Finally, we check the *balance factor* of the root node, node 2, the left subtree's height is 1 and the right subtree's height is 3. Therefore, the *balance factor* for node 2 is -2 , which necessitates the rotation of the root node 2. Have a look on the following figure to see how we have rotated the node 2.

**Fig 21.7: insert(6)**

Now the node 4 has become the *root* of the tree. Node 2, which was the *root* node, has become the left child of node 4. Nodes 5 and 6 are still on their earlier places while remaining the right child and sub-child of node 4 respectively. However, the node 3, which was left child of node 4, has become the right child of node 2.

Now, let's see the inorder traversal of this tree:

1 2 3 4 5 6

You are required to practice this inorder traversal. It is very important and the basic point of performing the rotation operation is to preserve the inorder traversal of the tree. There is another point to note here that in Binary Search Tree (BST), the *root* node remains the same (the node that is inserted first). But in an AVL tree, the *root* node keeps on changing.

In Fig 21.6: we had to traverse three links (node 2 to node 4 and then node 5) to reach the node 6. While after rotation, (in Fig 21.7), we have to traverse the two links (node

4 and 5) to reach the node 6. You can prove it mathematically that inside an AVL tree built of n items; you can search up to $1.44\log_2 n$ levels to find a node inside. After this maximum number of links traversal, a programmer will have success or failure, as $1.44\log_2 n$ is the maximum height of the AVL tree. Consider the BST case, where we had constructed a linked list. If we want to build a BST of these six numbers, a linked list structure is formed. In order to reach the node 6, we will have to traverse five links. In case of AVL tree, we had to traverse two links only.

Let's add few more items in the AVL tree and see the rotations performed to maintain AVL characteristics of the tree.

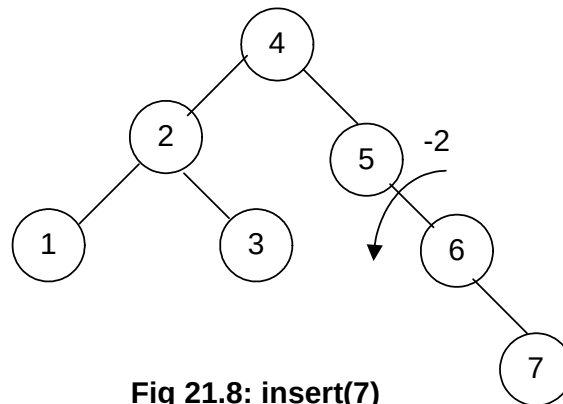


Fig 21.8: insert(7)

Node 7 is inserted as the right child of node 6. We start to see the balance factors of the nodes. The balance factors for node 7, 6 are 0 and -1 respectively. As the balance factor for node 5 is -2, the rotation will be performed on this node. After rotation, we get the tree as shown in the following figure.

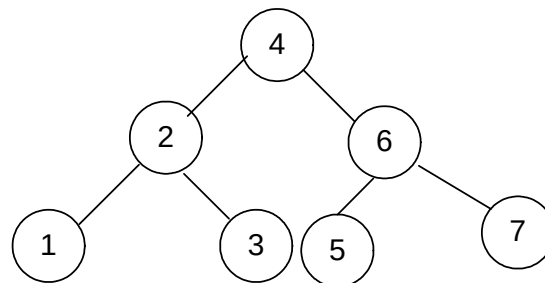
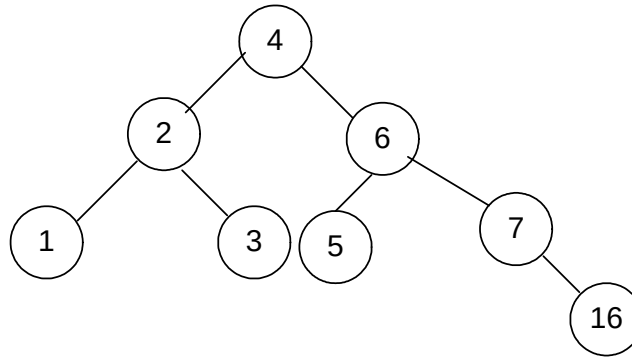


Fig 21.9: insert(7)

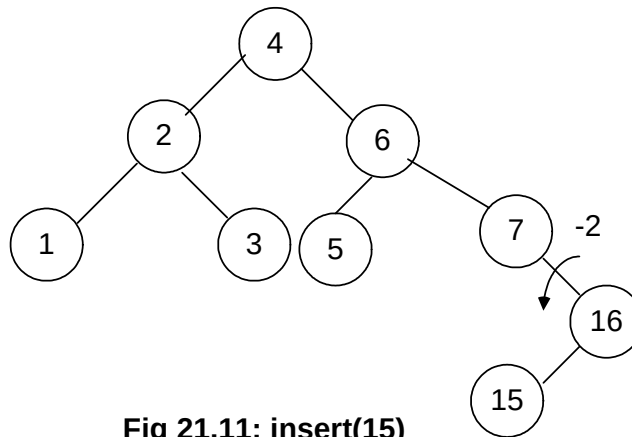
After the rotation, node 5 has become the left child of node 6. We can see in the Fig 21.9 that the tree has become the perfect binary tree. While writing our program, we will have to compute the balance factors of each node to know that the tree is a perfectly balanced binary tree. We find that balance factor for all nodes 7, 5, 3, 1, 6, 2 and 4 is 0. Therefore, we know that the tree is a perfect balanced tree. Let's see the inorder traversal output here:

1 2 3 4 5 6 7

It is still in the same sequence and the number 7 has been added at the end.

**Fig 21.10: insert(16)**

We have inserted a new node 16 in the tree as shown in the above Fig 21.10. This node has been added as the right child of the node 7. Now, let's compute the balance factors for the nodes. The balance factor for nodes 16, 7, 5, 3, 1, 6, 2 and 4 is either 0 or -1. So this fulfills the condition of a tree to be an AVL. Let's insert another node containing number 15 in this tree. The tree becomes as given in the figure below:

**Fig 21.11: insert(15)**

Next step is to find out the balance factor of each node. The factors for nodes 5 and 16 are 0 and 1 respectively. This is within limits of an AVL tree but the balance factor for node 7 is -2. As this is out of the limits of AVL, we will perform the rotation operation here. In the above diagram, you see the direction of rotation. After rotation, we have the following tree:

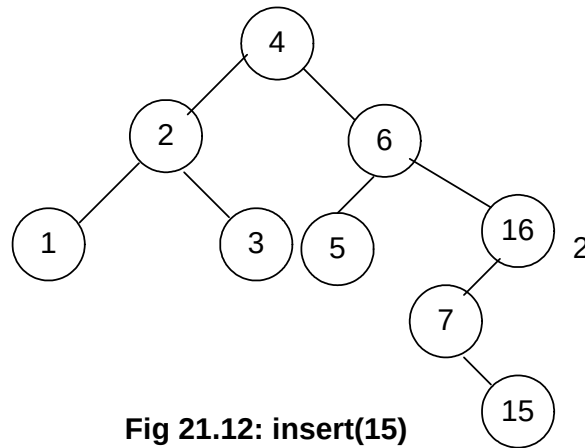


Fig 21.12: insert(15)

Node 7 has become the left child of node 16 while node 15 has attained the form of the right child of node 7. Now the balance factors for node 15, 7 and 16 are 0, -1 and 2 respectively. Note that the single rotation above when we rotated node 7 is not enough as our tree is still not an AVL one. This is a complex case that we had not encountered before in this example.

Cases of Rotation

The single rotation does not seem to restore the balance. We will re-visit the tree and rotations to identify the problem area. We will call the node that is to be rotated as α (node requires to be re-balanced). Since any node has at the most two children, and a height imbalance requires that α 's two sub-trees differ by two (or -2), the violation will occur in four cases:

1. An insertion into left subtree of the left child of α .
2. An insertion into right subtree of the left child of α .
3. An insertion into left subtree of the right child of α .
4. An insertion into right subtree of the right child of α .

The insertion occurs on the *outside* (i.e., left-left or right-right) in *cases 1* and *4*. Single rotation can fix the balance in *cases 1* and *4*.

Insertion occurs on the *inside* in *cases 2* and *3* which a single rotation cannot fix.

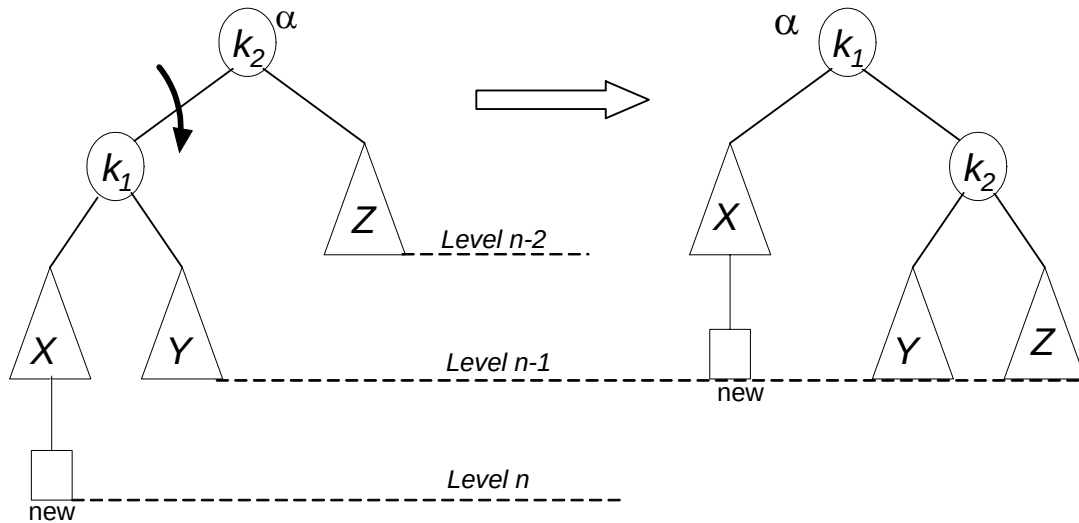


Fig 21.13: Single right rotation to fix case 1

We have shown, single right rotation to fix case 1. Two nodes k_2 and k_1 are shown in the figure, here k_2 is the *root* node (and also the α node, k_1 is its left child and Z shown in the triangle is its right child. The nodes X and Y are the left and right subtrees of the node k_1 . A new node is also shown below to the triangle of the node X , the exact position (whether this node will be right or left child of the node X) is not mentioned here. As the new node is inserted as a child of X that is why it is called an *outside* insertion, the insertion is called *inside* if the new node is inserted as a child of the node Y . This insertion falls in *case 1* mentioned above, so by our definition above, single rotation should fix the balance. The k_2 node has been rotated single time towards right to become the right child of k_1 and Y has become the left child of k_2 . If we traverse the tree in inorder fashion, we will see that the output is same:

X k1 Y k2 Z

Consider the the figure below:

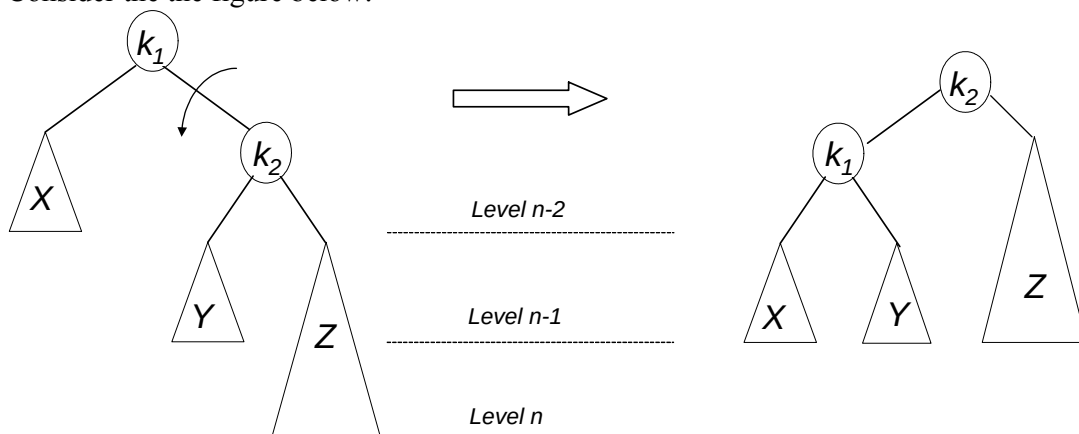


Fig 21.14: Single left rotation to fix case 4

In this figure (Fig 21.14), the new node has been inserted as a child node of Z , that is why it is shown in bigger size covering the next level. Now this is an example of *case 4* because the new node is inserted below the right subtree of the right child of the

root node (α). One rotation towards should make it balanced within limits of AVL tree. The figure on the right is after rotation the node k_1 one time towards left. This time node Y has become the right child node of the node k_1 .

In our function of insertion in our code, we will do insertion, will compute the balance factors for nodes and make rotations.

Now, we will see the *cases 2 and 3*, these are not resolved by a single rotation.

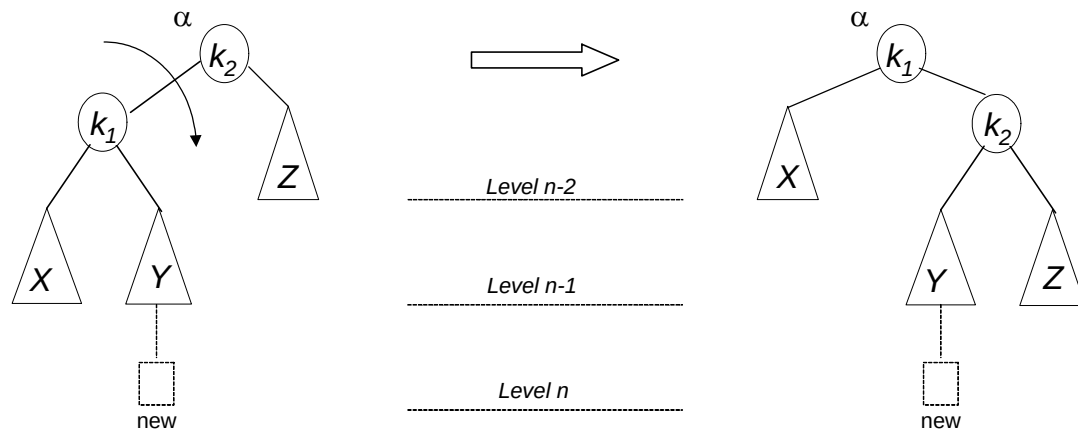


Fig 21.15: Single right rotation fails to fix case 2

We see here that the new node is inserted below the node Y . This is an *inside* insertion. The balance factor for the node k_2 became 2. We make single rotation by making right rotation on the node k_2 as shown in the figure on the right. We compute the balance factor for k_1 , which is -2 . So the tree is still not within the limits of AVL tree. Primarily the reason for this failure is the node Y subtree, which is unchanged even after making one rotation. It changes its parent node but its subtree remains intact. We will cover the double rotation in the next lecture.

It is very important that you study the examples given in your text book and try to practice the concepts rigorously.

Data Structures

Lecture No. 22

Reading Material

Data Structures and Algorithm Analysis in C++

Chapter. 4
4.4.2

Summary

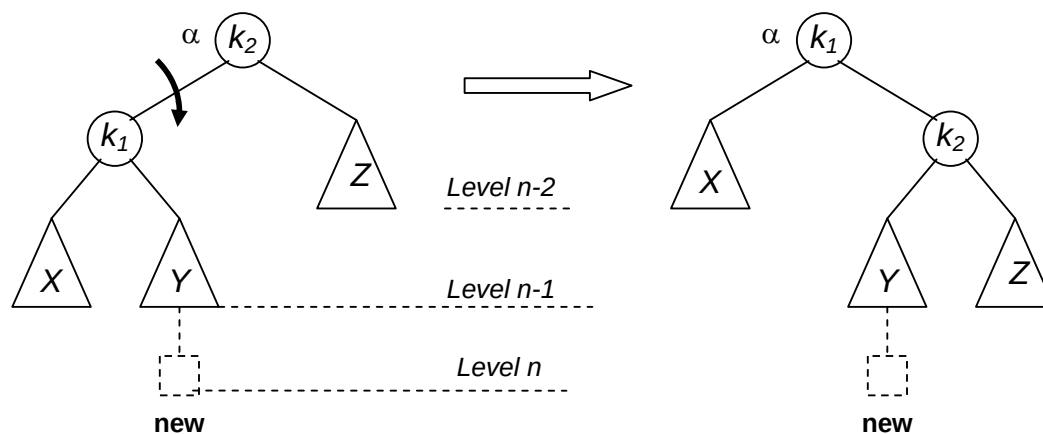
- Cases of rotations
- Left-right double rotation to fix case 2
- Right-left double rotation to fix case 3
- C++ Code for avlInsert method

Cases of rotations

In the previous lecture, we discussed how to make insertions in the AVL tree. It was seen that due to the insertion of a node, the tree has become unbalanced. Resultantly, it was difficult to fix it with the single rotation. We have analyzed the insertion method again and talked about the α node. The new node will be inserted at the left or right subtree of the α 's left child or at the left or right subtree of the α 's right child. Now the question arises whether the single rotation help us in balancing the tree or not. If the new node is inserted in the left subtree of the α 's left child or in the right subtree of α 's right child, the balance will be restored through single rotation. However, if the new node goes inside the tree, the single rotation is not going to be successful in balancing the tree.

We face four scenarios in this case. We said that in the case-1 and case-4, single rotation is successful while in the case-2 and case-3 single rotation does not work. Let's see the tree in the diagram given below.

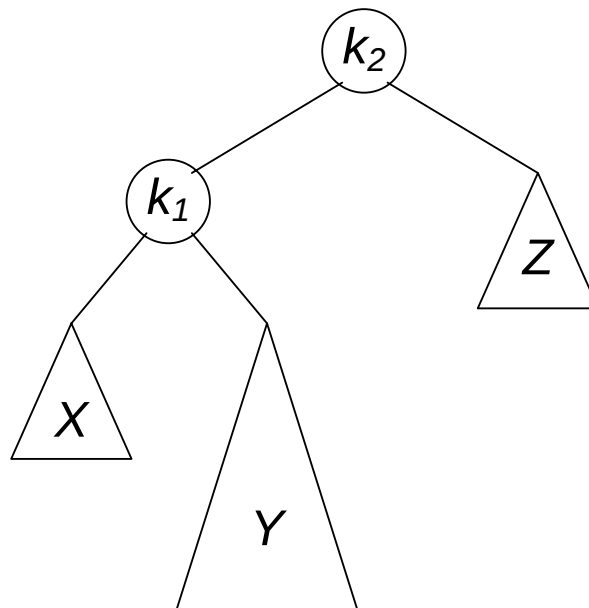
Single right rotation fails to fix case 2.



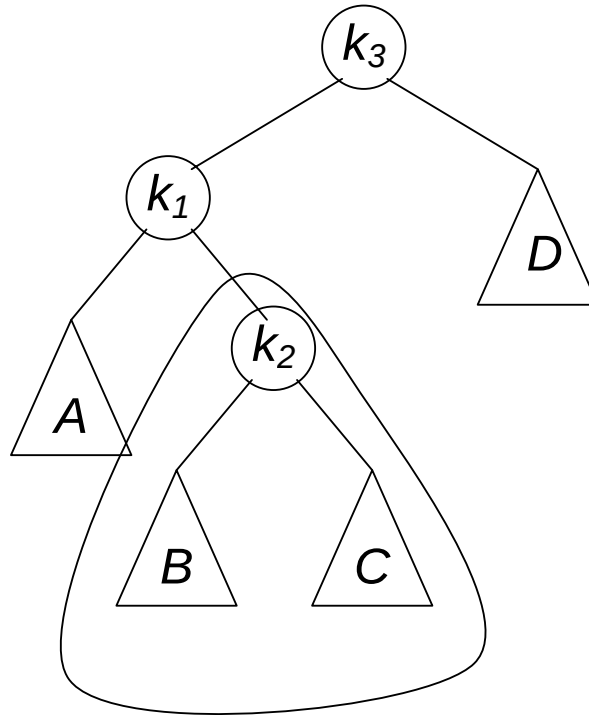
In the above tree, we have α node as k_2 , which has a left child as k_1 . Whereas X and Y are its left and right children. The node k_2 has a right child Z . Here the newly inserted

node works as the left or right child of node Y . Due to this insertion, one level is increased in the tree. We have applied single rotation on the link of k_1 and k_2 . The right side tree in the figure is the post-rotation tree. The node k_1 is now at the top while k_2 comes down and node Y changes its position. Now if you see the levels of the node, these are seen same. Have a look on the level of the α node i.e. k_1 which reflects that the difference between the left and right side levels is still 2. So the single rotation does not work here.

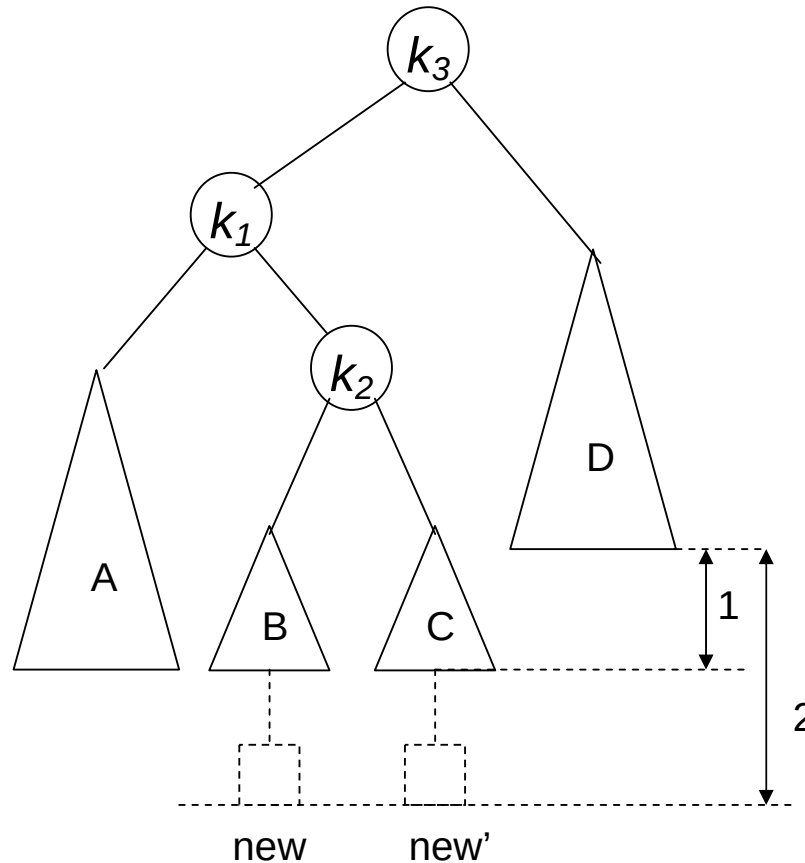
Let's see how we can fix that problem. A fresh look on the following diagram will help us understand the problem.



Here k_2 is the root node while k_1 and Z are the right and left children respectively. The new node is inserted under Y so we have shown Y in a big triangle. The new node is inserted in the right subtree of k_1 , increasing its level by 1. Y is not empty as the new node was inserted in it. If Y is empty, the new node will be inserted under k_1 . It means that Y has a shape of a tree having a root and possibly left and right subtrees. Now view the entire tree with four subtrees connected with 3 nodes. See the diagram below.



We have expanded the Y and shown the root of Y as $K2$, B and C are its left and right subtrees. We have also changed the notations of other nodes. Here, we have A , B , C and D as subtrees and $k1$, $k2$ and $k3$ as the nodes. Let's see where the new node is inserted in this expanded tree and how can we restore its balance. Either tree B or C is two levels deeper than D . But we are not sure which one is deeper. The value of new node will be compared with the data in $k2$ that will decide that this new node should be inserted in the right subtree or left subtree of the $k2$. If the value in the new node is greater than $k2$, it will be inserted in the right subtree i.e. C . If the value in the new node is smaller than $k2$, it will be inserted in the left subtree i.e. B . See the diagram given below:



New node inserted at either of the two spots

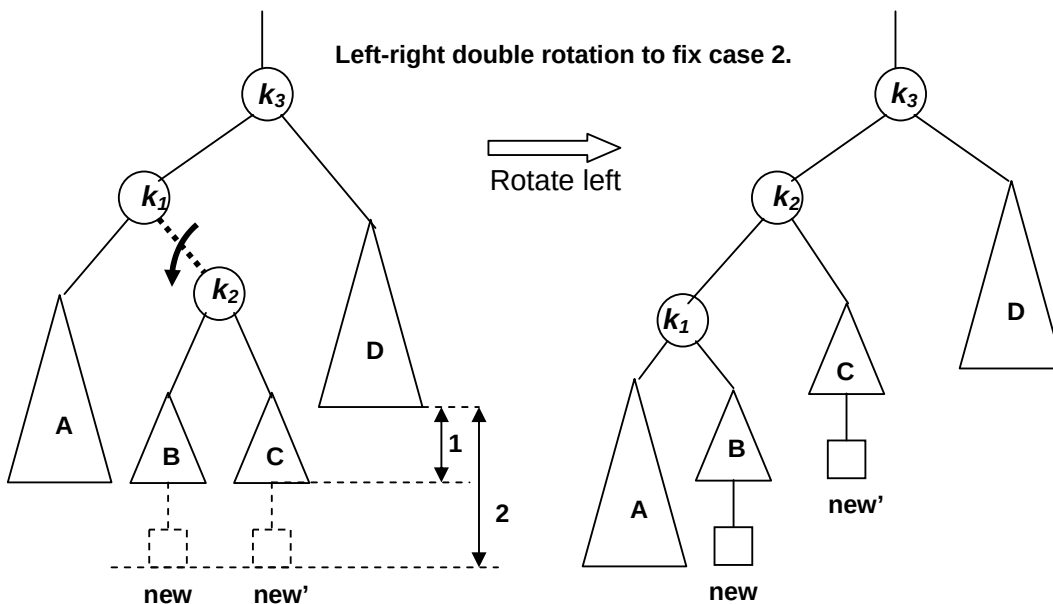
We have seen the both possible locations of the new node in the above diagram. Let's see the difference of levels of the right and left subtrees of the k_3 . The difference of B or C from D is 2. Therefore the expansion of either of B or C , due to the insertion of the new node, will lead to a difference of 2. Therefore, it does not matter whether the new node is inserted in B or C . In both of the cases, the difference becomes 2. Then we try to balance the tree with the help of single rotation. Here the single rotation does not work and the tree remains unbalanced. To re-balance it, k_3 cannot be left as the root. Now the question arises if k_3 cannot become root, then which node will become root? In the single rotation, k_1 and k_3 were involved. So either k_3 or k_1 will come down. We have two options i.e. left rotation or right rotation. If we turn k_1 into a root, the tree will be still unbalanced. The only alternative is to place k_2 as the new root. So we have to make k_2 as root to balance the tree. How can we do that?

If we make k_2 the root, it forces k_1 to be k_2 's left child and k_3 to be its right child. When we carry out these changes, the condition is followed by the inorder traversal. Let's see the above tree in the diagram. In that diagram, the k_3 is the root and k_1 is its left child while k_2 is the right child of k_1 . Here, we have A , B , C and D as subtrees. You should know the inorder traversal of this tree. It will be A , k_1 , B , k_2 , C , k_3 and D where A , B , C and D means the complete inorder traversal of these subtrees. You should memorize this tree traversal.

Now we have to take k_2 as the root of this tree and rearrange the subtrees A , B , C and D . k_1 will be the left child of k_2 while k_3 is going to be its right child. Finally if we traverse this new tree, it will be the same as seen above. If it is not same, it will mean that there is something wrong in the rotation. We have to find some other solution. Now let's see how we can rotate our tree so that we get the desired results.

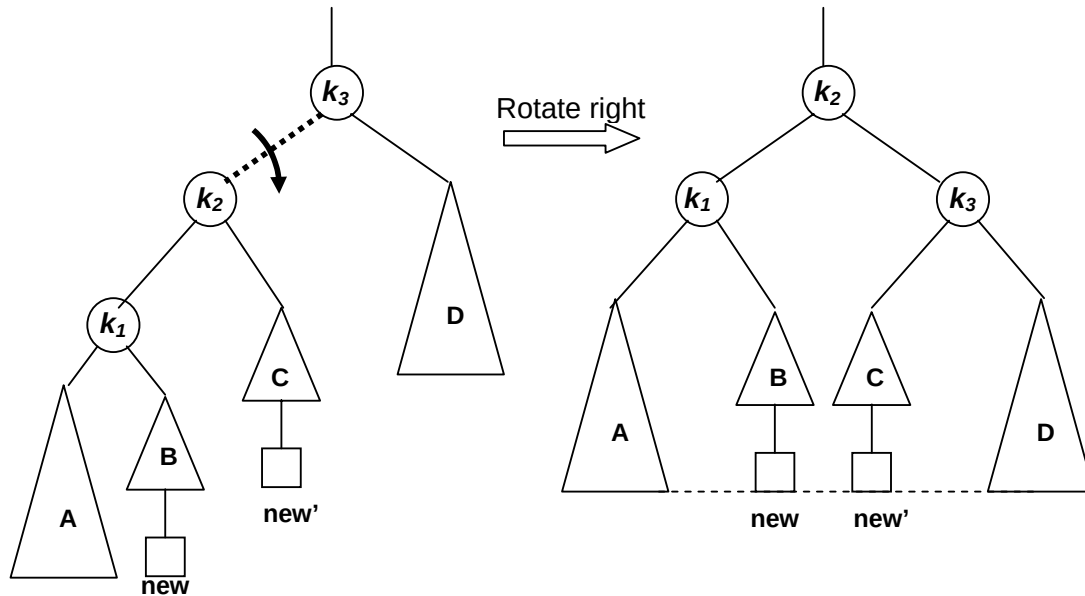
Left-right double rotation to fix case 2

We have to perform a double rotation to achieve our desired results. Let's see the diagram below:



On the left side, we have the same tree with k_3 as its root. We have also shown the new nodes as new and new' i.e. the new node will be attached to B or C . At first, we will carry out the left rotation between k_1 and k_2 . During the process of left rotation, the root k_1 comes down and k_2 goes up. Afterwards, k_1 will become the left child of k_2 and the left subtree of k_2 i.e. B , will become the right subtree of k_1 . This is the single rotation. You can see the new rotated tree in the above figure. It also shows that the B has become the right child of the k_1 . Moreover, the new node is seen with the B . Now perform the inorder traversal of this new rotated tree. It is A , k_1 , B , k_2 , C , k_3 and D . It is same as witnessed in case of the inorder traversal of original tree. With this single rotation, the k_2 has gone one step up while k_1 has come down. Now k_2 has become the left child of k_3 . We are trying to make the k_2 the root of this tree. Now what rotation should we perform to achieve this?

Now we will perform right rotation to make the k_2 the root of the tree. As a result, k_1 and k_2 have become its left and right children respectively. The new node can be inserted with B or C . The new tree is shown in the figure below:

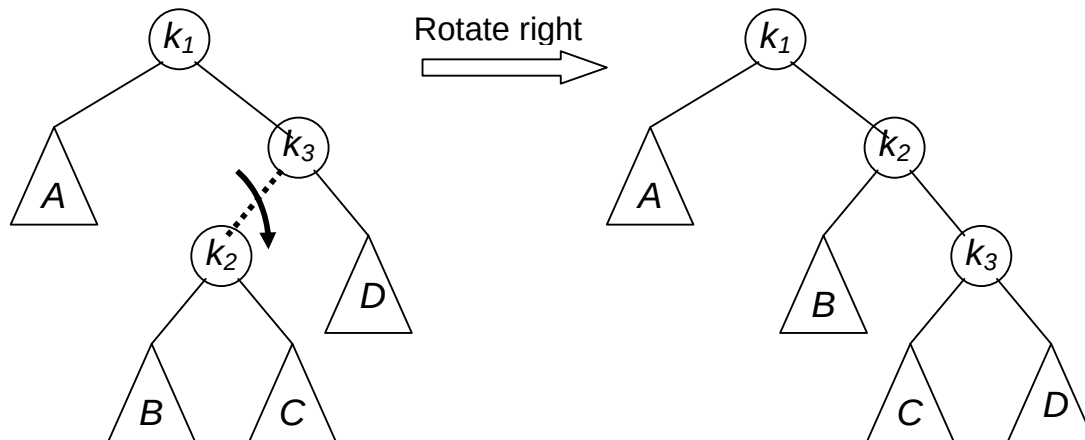


Now let's see the levels of *new* and *new'*. Of these, one is the new node. Here you can see that the levels of *new*, *new'* i.e. A and D are the same. The new tree is now a balanced one. Let's check the inorder traversal of this tree. It should be the same as that of the original tree. The inorder traversal of new tree is A, k_1, B, k_2, C, k_3 and D , which is same as that of the original tree.

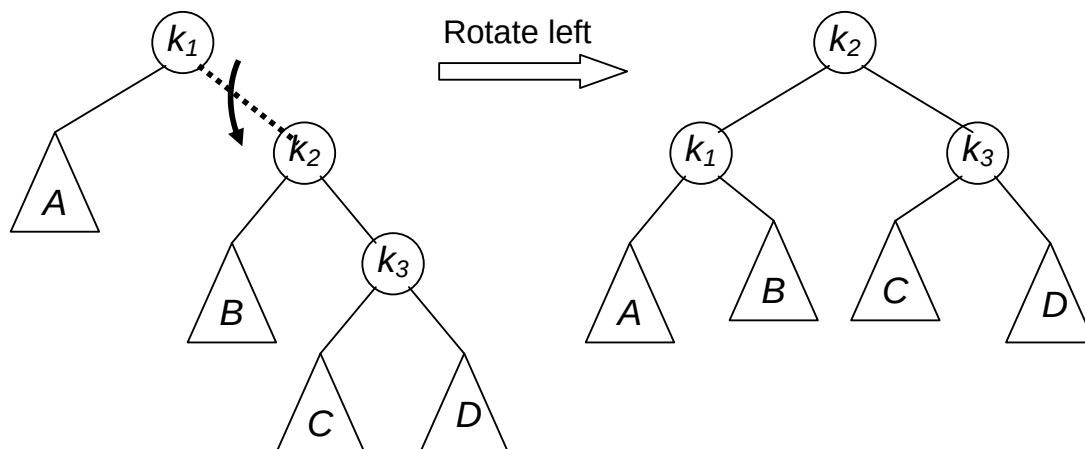
This is known as double rotation. In double rotation, we perform two single rotations. As a result, the balance is restored and the AVL condition is again fulfilled. Now we will see in which order, the double rotation is performed? We performed a left rotation between k_1 and k_2 link, followed by a right rotation.

Right-left double rotation to fix case 3

In case, the node is inserted in left subtree of the right child, we encounter the same situation as discussed above. But here, we will perform right rotation at first before going for a left rotation. Let's discuss this symmetric case and see how we can apply double rotation here. First we perform the right rotation.

Right-left double rotation to fix case 3.

Here k_1 is the root of the tree while k_3 is the right child of the k_1 . k_2 is the inner child. It is the Y tree expanded again here and the new node will be inserted in the k_2 's right subtree C or left subtree B . As we have to transform the k_2 into the root of the tree, so the right rotation between the link k_2 and k_3 will be carried out. As a result of this rotation, k_2 will come up and k_3 will go down. The subtree B has gone up with the k_2 while subtree C is now attached with the k_3 . To make the k_2 root of the tree, we will perform the left rotation between then k_1 and k_2 . Let's see this rotation in the figure below:

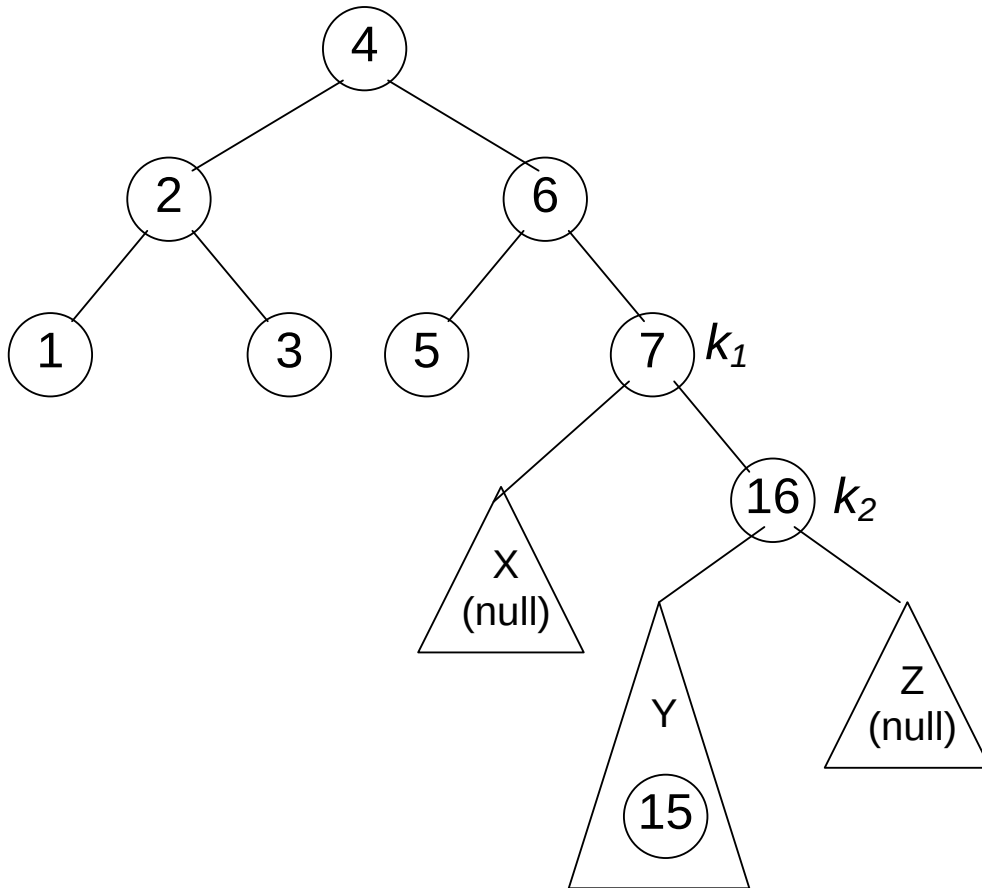


In the above figure at the right side, we have the final shape of the tree. You can see that k_2 has become the root of the tree. k_1 and k_3 are its left and right children respectively. While performing the inorder traversal, you will see that we have preserved our inorder traversal.

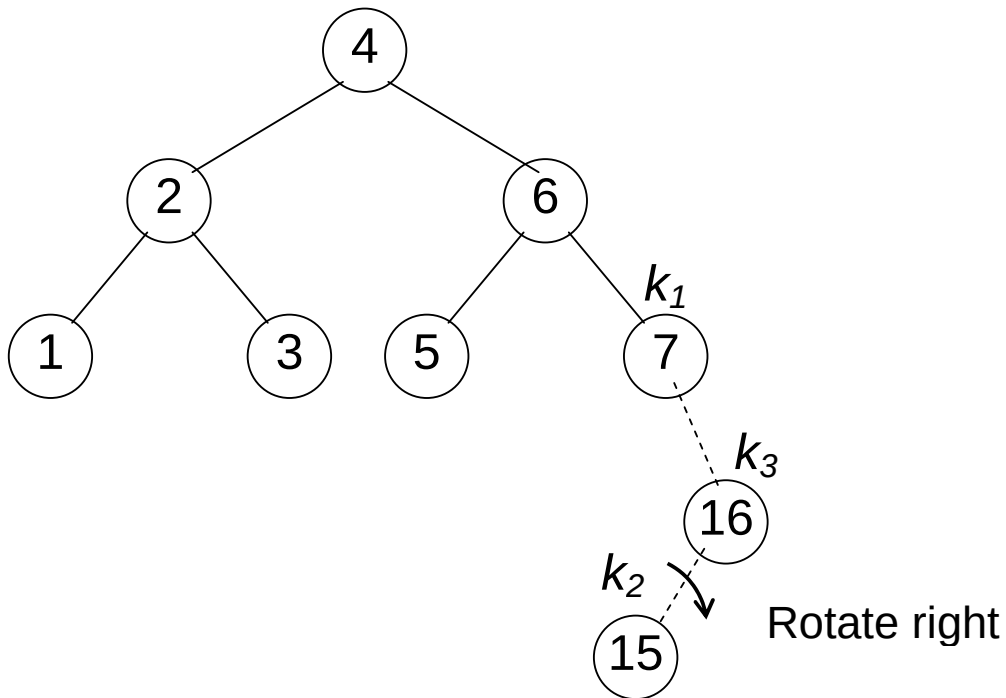
We have started this activity while building an example tree. We inserted numbers in it. When the balance factor becomes more than one, rotation is performed. During this process, we came at a point when single rotation failed to balance the tree. Now there is need to perform double rotation to balance the tree that is actually two single rotations. Do not take double rotation as some complex function, it is simply two single rotations in a special order. This order depends on the final position of the new

node. Either the new node is inserted at the right subtree of the left child of α node or at the left subtree of the right child of α node. In first case, we have to perform left-right rotation while in the second case, the right-left rotation will be carried out.

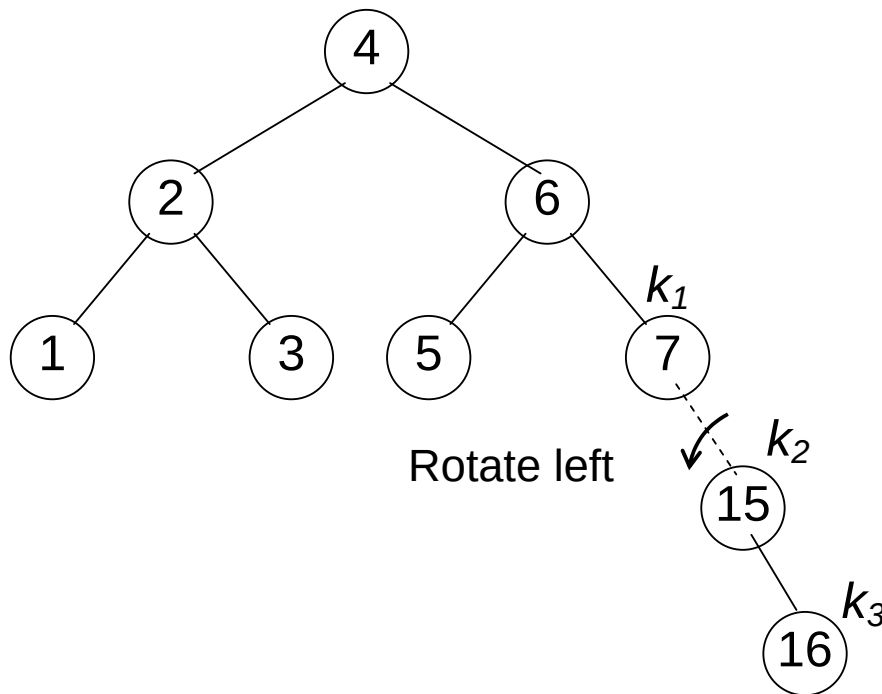
Let's go back to our example and try to complete it. So far, we have 1, 2, 3, 4, 5, 6, 7 and 16 in the tree and inserted 15 which becomes the left child of the node 16. See the figure below:



Here we have shown X , Y and Z in case of the double rotation. We have shown Y expanded and 15 is inside it. Here we will perform the double rotation, beginning with the right rotation first.

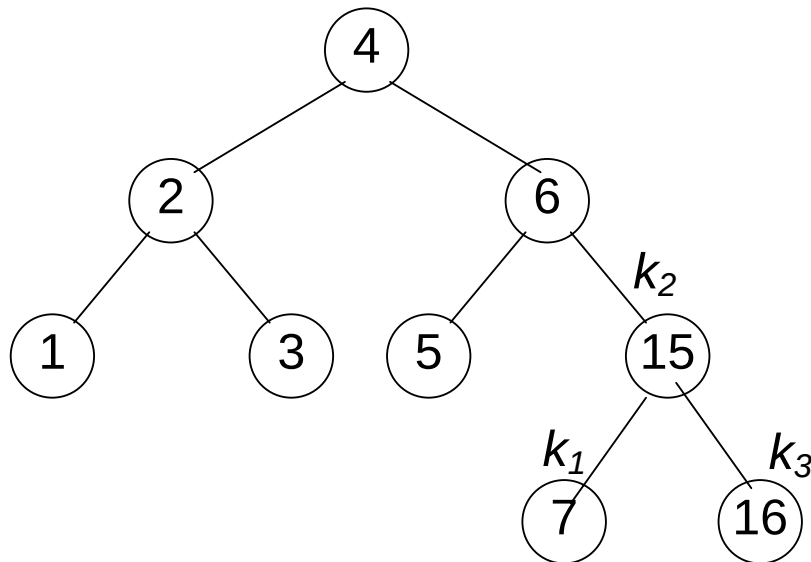


We have identified the k_1 , k_2 and k_3 nodes. This is the case where we have to perform right-left double rotation. Here we want to promote k_2 upwards. For this purpose, the right rotation on the link of k_2 and k_3 i.e. 15 and 16 will be carried out.



The node 15 now comes up while node 16 has gone down. We have to promote k_2 to the top and k_3 and k_1 will become its right and left children respectively. Now we will

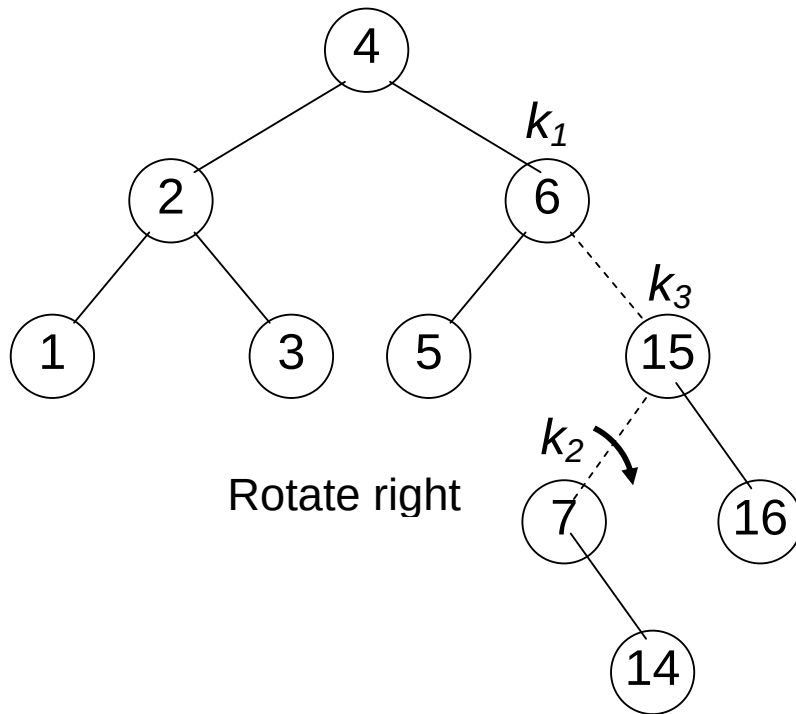
perform left rotation on the link of k_1 and k_2 i.e. 7 and 15. With this left rotation, 15 goes up and 7 and 16 become its left and right children respectively.



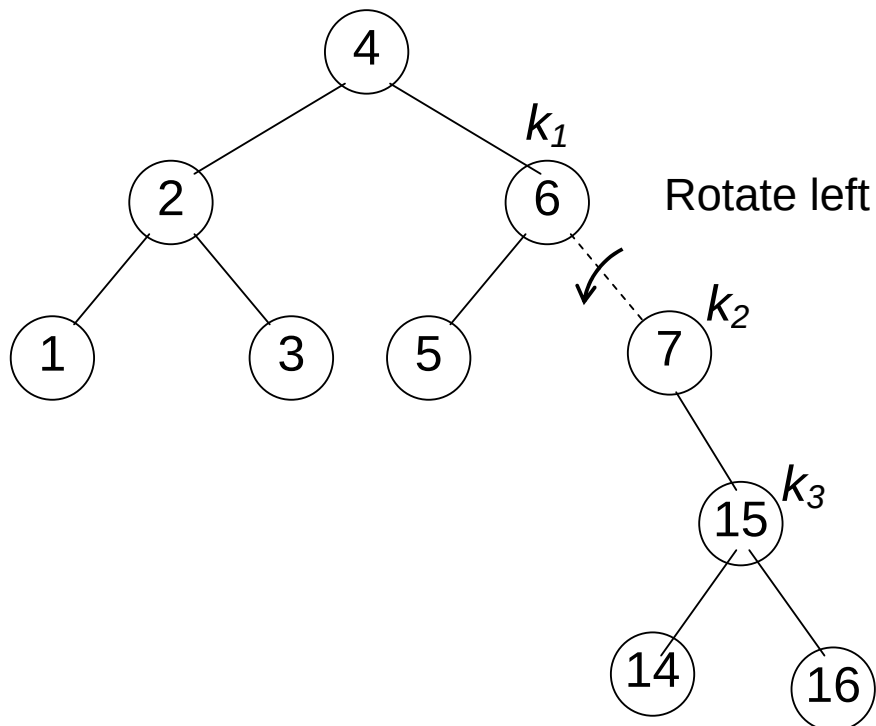
Here we have to check two things. At first, the tree is balanced or not i.e. the AVL condition is fulfilled or not. Secondly we will confirm that the inorder traversal is preserved or not. The inorder traversal should be the same as that of the inorder traversal of original tree. Let's check these two conditions. The depth of the left subtree of node 4 is 2 while the depth of the right subtree of node 4 is three. Therefore, the difference of the levels at node 4 is one. So the AVL condition is fulfilled at node 4. At node 6, we have one level on its left side while at the right side of node 6, there are two levels. As the difference of levels is one, therefore node 6 is also balanced according to the AVL condition. Similarly other nodes are also fulfilling the AVL condition. If you see the figure above, it is clear that the tree is balanced.

We are doing all this to avoid the link list structure. Whenever we perform rotation on the tree, it becomes clear from the figure that it is balanced. If the tree is balanced, in case of searching, we will not have to go very deep in the tree. After going through the mathematical analysis, you will see that in the worst case scenario, the height of the tree is $1.44 \log_2 n$. This means that the searching in AVL is logarithmic. Therefore if there are ten million nodes in an AVL tree, its levels will be roughly as $\log_2(10 \text{ million})$ which is very few. So the traversal in an AVL tree is very simple.

Let's insert some more nodes in our example tree. We will perform single and double rotations, needed to make the tree balanced. The next number to be inserted is 14. The position of node 14, according to the inorder traversal, is the right child of 7. Let's see this in the diagram as:

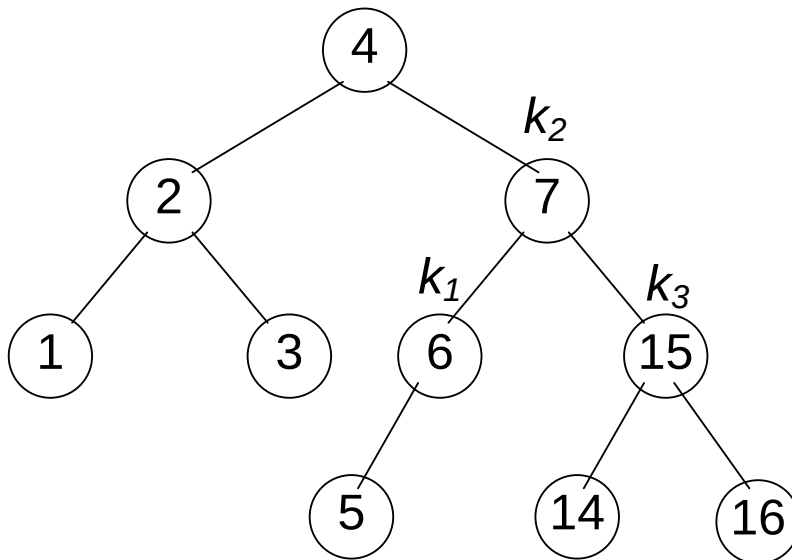


The new node 14 is inserted as the right child of 7 that is the inner subtree of 15. Here we have to perform double rotation again. We have identified the k_1 , k_2 and k_3 . k_2 has to become the root of this subtree. The nodes k_1 and k_3 will come down with their subtrees while k_2 is going to become the root of this subtree. After the right rotation the tree will be as:



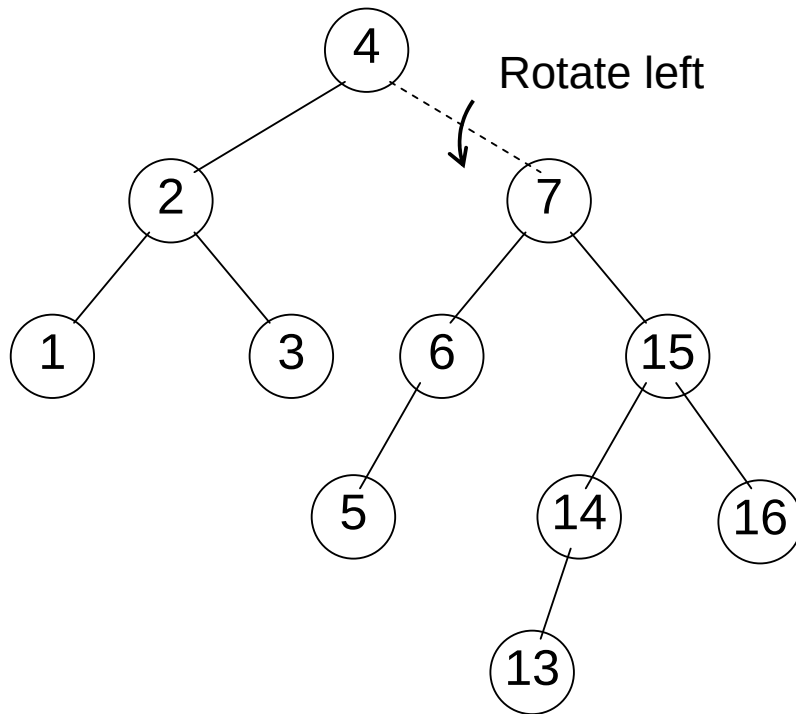
With the right rotation, k_2 has come one step up while k_3 has been turned into the right child of k_2 but k_1 is still up. Now we will perform a left rotation on the link of k_1 and k_2 to make the k_2 root of this subtree. Now think that after this rotation and rearrangement of node what will be the shape of the tree.

After the double rotation, the final shape of the tree will be as:

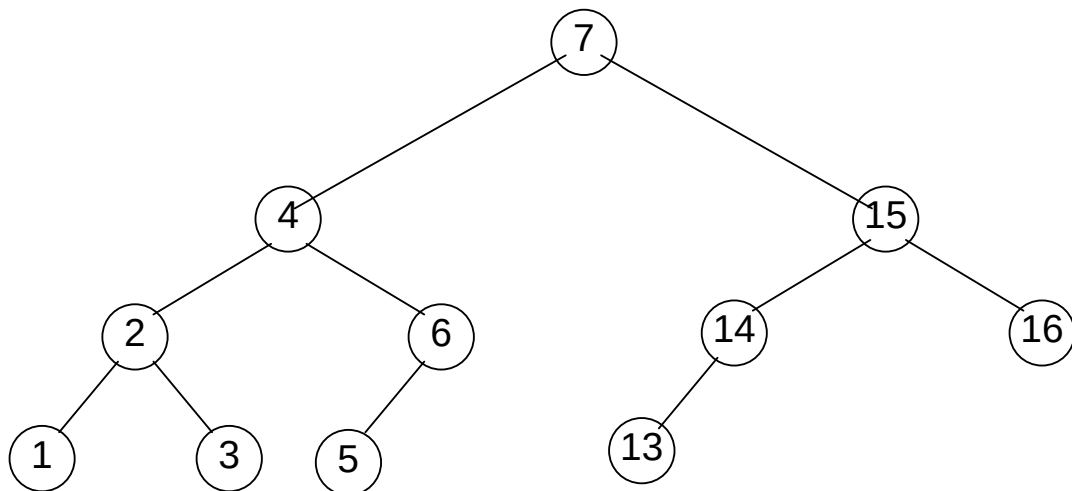


k_2 has become the root of the subtree. k_1 has attained the role of the left child of k_2 and k_3 has become the right child of the k_2 . The other nodes 5, 14 and 16 have been rearranged according to the inorder traversal. The node 7 has come up where as node 6 and 15 have become its left and right child. Now just by viewing the above figure, it is clear that the tree is balanced according to the AVL condition. Also if we find out its inorder traversal, it should be the same as the inorder traversal of original tree. The inorder traversal of the above tree is 1, 2, 3, 4, 5, 6, 7, 14, 15, and 16. This is in sorted order so with the rotations the inorder traversal is preserved.

Let's insert some more numbers in our tree. The next number to be inserted is 13.



We have to perform single rotation here and rearrange the tree. It will look like as:



The node 7 has become the root of the tree. The nodes 4, 2, 1, 3, 6, 5 have gone to its left side while the nodes 15, 14, 13, 16 are on its right side. Now try to memorize the tree which we build with these sorted numbers. If you remember that it looks like a link list. The root of that tree was 1. After that we have its right child as 2, the right child of 2 as 3, then its right child 4 and so on up to 16. The shape of that tree looks exactly like a linked list. Compare that with this tree. This tree is a balanced one. Now if we have to traverse this tree for search purposes, we have to go at the most three levels.

Now you must be clear why we need to balance the trees especially if we have to use the balanced search trees. While dealing with this AVL condition, it does not matter

whether the data, provided to a programmer is sorted or unsorted. The data may be sorted in ascending order or descending order. It may contain alphabets or names in any order. We will keep our tree balanced during the insertion and tree will be balanced at each point. Our tree will not be balanced in the end, it will be balanced with each insertion. It will not be completely balanced. At the maximum, if we pick any node, the difference in the levels of its right subtree and left subtree will not be more than 1.

Now if we have 9 or 10 nodes in the tree and take $\log$ of this, it will be near 3. Therefore our tree has three levels after that there are its leaf nodes. Please keep this in mind that originally we have thought BST as an abstract data type. We have defined operations on it like *insert*, *remove* and the major method was *find*. The *find* method takes a data item and searches the tree. It will also show that this data item exists or not in the tree. We also right *findMin* and *findMax* methods. In case of a BST, we can find the minimum and maximum value in it. The minimum will be the left most node of the tree while the right most node of BST will give the maximum value. You can confirm it by applying this on the above tree.

C++ Code for avlInsert method

Now let's see the C++ code of *avlinsert* method. Now we have to include this balancing procedure in the *insert* method. We have already written this *insert* method which takes some value and adds a node in the tree. That procedure does not perform balancing. Now we will include this balancing feature in our *insert* method so that the newly created tree fulfills the AVL condition.

Here is the code of the function.

```
/* This is the function used to insert nodes satisfying the AVL condition.*/

TreeNode<int>* avlInsert(TreeNode<int>* root, int info)
{
    if( info < root->getInfo() ){
        root->setLeft(avlInsert(root->getLeft(), info));
        int htdiff = height(root->getLeft()) - height(root->getRight());
        if( htdiff == 2 )
            if( info < root->getLeft()->getInfo() ) // outside insertion case
                root = singleRightRotation( root );
            else // inside insertion case
                root = doubleLeftRightRotation( root );
    }
    else if( info > root->getInfo() ) {
        root->setRight(avlInsert(root->getRight(), info));
        int htdiff = height(root->getRight()) - height(root->getLeft());
        if( htdiff == 2 )
            if( info > root->getRight()->getInfo() )
                root = singleLeftRotation( root );
            else
                root = doubleRightLeftRotation( root );
    }
}
```

```

// else a node with info is already in the tree. In
// case, reset the height of this root node.
int ht = Max(height(root->getLeft()), height(root->getRight()));
root->setHeight( ht + 1 ); // new height for root.
return root;
}

```

We have named the function as *avlInsert*. The input arguments are *root* node and the *info* is of the type *int*. In our example, we are having a tree of *int* data type. But of course, we can have any other data type. The return type of *avlInsert* method is *TreeNode<int>**. This *info* will be inserted either in the right subtree or left subtree of *root*. The first *if* statement is making this decision by comparing the value of *info* and the value of *root*. If the new *info* value is less than the *info* value of *root*, the new node will be inserted in the left subtree of *root*. In the next statement, we have a recursive call as seen in the following statement.

```
avlInsert(root->getLeft(), info)
```

Here we are calling *avlInsert* method again. We are passing it the first node of the left subtree and the *info* value. We are trying to insert this *info* in the left subtree of *root* if it does not already exist in it. In the same statement, we are setting the left of *root* as the return of the *avlInsert* method. Why we are doing *setLeft*? As we know that this is an AVL tree and the new data is going to be inserted in the left subtree of *root*. We have to balance this tree after the insertion of the node. After insertion, left subtree may be rearranged to balance the tree and its root can be changed. You have seen the previous example in which node 1 was the root node in the start and in the end node 7 was its root node. During the process of creation of that tree, the root nodes have been changing. After the return from the recursive call in our method, the left node may be different than the one that was before insertion. The complete call is as:

```
root->setLeft(avlInsert(root->getLeft(), info));
```

Here we are inserting the new node and also getting the root of the left subtree after the insertion of the new node. If the root of the left subtree has been changed after the insertion, we will update our tree by assigning this new left-node to the left of the *root*.

Due to the insertion of the new node if we have rearranged the tree, then the balance factor of the root node can be changed. So after this, we check the balance factor of the node. We call a function *height* to get the height of left and right subtree of the *root*. The *height* function takes a node as a parameter and calculates the height from that node. It returns an integer. After getting the heights of left and right subtrees, we take its difference as:

```
int htdiff = height(root->getLeft()) - height(root->getRight());
```

Now if the difference of the heights of left and right subtrees is greater than 1, we have to rebalance the tree. We are trying to balance the tree during the insertion of new node in the tree. If this difference is 2, we will perform single rotation or double

rotation to balance the tree. We have to deal with one of the four cases as discussed earlier. This rotation will be carried out with only one node where the balance factor is 2. As a result, the tree will be balanced. We do not need to rotate each node to balance the tree. This is characteristic of an AVL tree.

We have seen these rotations in the diagrams. Now we have to perform these ones in the code. We have four cases. The new node is inserted in the left or right subtree of the left child of α node or the new node is inserted in the left or right subtree of the right child of α node. How can we identify this in the code? We will identify this with the help of the *info* value of the new node.

We can check whether the difference (*htdiff*) is equal to 2 or not. We have *if* condition that checks that the *info* is less than the value of left node of the current node (α node). If this condition is true, it shows that it is the case of outside insertion. The node will be inserted as the left-most child. We have to perform single rotation. If the else part is executed, it means that this is the case of inside insertion. So we have to perform double rotation. We are passing α node to the rotation function. When a programmer performs these rotations, he gets another node that is assigned to root. This is due to the fact that the root may change during the rotations.

We encounter this case when the new node is inserted in the left subtree of the root. The new node can be inserted in the right subtree of the root. With respect to symmetry, that code is also similar. You can easily understand that. In the code, we have *if-else* condition that checks whether the *info* is greater than the value of root. Therefore, it will be inserted in the right subtree. Here again, we made a recursive call to *avlInsert* by passing it the first node of the right subtree of the root. When this recursive call is completed, we may have to perform rotations. We set the right node of the root as the node returned by the *avlInsert*. After this, we check the balance factor. We calculate the difference as:

```
int htdiff = height(root->getRight()) - height(root->getLeft());
```

In the previous case, we have subtracted the height of right tree from left tree. Here we have subtraction the height of left tree from the right tree. This is due to the fact that we want to avoid the -ve value. But it does not matter whether we get the -ve value. We can take its absolute value and test it. If the value of *htdiff* is 2, we have to identify the case. We check that the new node is to be inserted in the right subtree of the right child of α node or in the left subtree of the right child of α node. If the *info* is greater than the value of the right child of root, it is the case 4. Here we will restore the balance of the tree by performing single rotation. If the *else* part is executed, we have to perform the double rotation. These rotation routines will return the root of the rearranged tree that will be set as a root.

In the end, we have the third case. This is the case when the *info* is equal to the root. It means that the data is already in the tree. Here we will readjust the height if it is needed. We will take the maximum height between the left and right subtree and set this height plus one as the height of the root. We have added one to the height after adding one node to the tree. In the end, we return the root. We have included the balancing functionality in our tree and it will remain balanced.

We have not seen the code of rotation routines so far. The code of these methods is also very simple. We have to move some of pointers. You can easily understand that code. We will discuss this in the next lecture. Please see the code of single and double rotation in the book and try to understand it.



CS301- Data Structures

NOV 25,2011

LATEST SOLVED MCQS FROM MIDTERM PAPERS

MC100401285	<i>Moaaz.pk@gmail.com MC100401285@gmail.com</i>	PSMD01
mc110200885	<i><u>virtualcheater@gmail.com</u></i>	VDGK01
mc110200030	<i><u>awais4you@gmail.com</u></i>	PLHR25

CS301-MIDTERM SOLVED MCQS WITH REFERENCES



Virtual University

Muhammad Moazz Siddiq MCS 1285
moaazpk@gmail.com
 Campus : Islamabad Office : Learning &
 Modern Studies (IBMS) Saminagar
(PSMD01)

Muhammad Tahir Farid MCS 2007
virtualcheater@gmail.com
 Virtual University Campus : ILC, Ilam
(VDGK01)

Muhammad Waqar MCS 2008
awais4you@gmail.com
 Campus : Abbas Institute Of
 Modern Studies (AIMS) Lahore
(PLHR25)

Muhammad Moazz Siddiq MCS 1285
moaazpk@gmail.com
 Campus : Islamabad Office : Learning &
 Modern Studies (IBMS) Saminagar
(PSMD01)

Muhammad Tahir Farid MCS 2007
virtualcheater@gmail.com
 Virtual University Campus : ILC, Ilam
(VDGK01)

Muhammad Waqar MCS 2008
awais4you@gmail.com
 Campus : Abbas Institute Of
 Modern Studies (AIMS) Lahore
(PLHR25)

MIDTERM EXAMINATION

Spring 2010

CS301- Data Structures

Question No: 1 (Marks: 1) - Please choose one

A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True (Page 101)**

Question No: 2 (Marks: 1) - Please choose one

The data of the problem is of 2GB and the hard disk is of 1GB capacity, to solve this problem we should

- ▶ Use better data structures
- ▶ **Increase the hard disk space (Page 5)**
- ▶ Use the better algorithm
- ▶ Use as much data as we can store on the hard disk

Question No: 3 (Marks: 1) - Please choose one

Consider the function X as under

```
int X (int& Value)
{
    return Value;
}
```

Now a and b are integers in a calling function. Which one of the following is a valid call to the above function X.

- ▶ a = X (b) ;
- ▶ **a = X (&b) ;**
- ▶ a = X (*b) ;
- ▶ None of the given options

Here function argument passing by reference method is used, so when we call a function we will give the variable reference as parameter.

Question No: 4 (Marks: 1) - Please choose one

In the call by value methodology, a copy of the object is passed to the called function.

- ▶ False
- ▶ **True (Page 202)**

Question No: 5 (Marks: 1) - Please choose one

The tree data structure is a

- ▶ Linear data structure
- ▶ **Non-linear data structure (Page 112)**
- ▶ Graphical data structure
- ▶ Data structure like queue

Question No: 6 (Marks: 1) - Please choose one

When should you use a **const reference** parameter?

- ▶ Whenever the parameter has huge size.
- ▶ **Whenever the parameter has huge size, the function changes the parameter within its body, and you do NOT want these changes to alter the actual argument.**
- ▶ Whenever the parameter has huge size, the function changes the parameter within its body, and you DO want these changes to alter the actual argument.
- ▶ Whenever the parameter has huge size, and the function does not change the parameter within its body.

Declaring a parameter as a const simply means that the function can't change the value of its parameters.

Question No: 7 (Marks: 1) - Please choose one

Here is the start of a C++ class declaration:

```
class foo
{
public:
    void x(foo f);
    void y(const foo f);
    void z(foo f) const;
...
}
```

Which of the three member functions can alter the PRIVATE member variables of the foo object that activates the function?

- ▶ Only x can alter the private member variables of the object that activates the function.
- ▶ Only y can alter the private member variables of the object that activates the function.
- ▶ Only z can alter the private member variables of the object that activates the function.
- ▶ **Two of the functions can alter the private member variables of the object that activates the function.**

Only the x and y can alter the private member variable of the foo class object. Last Option is more correct but not exact. In the last option the two function name are not mentioned

Question No: 8 (Marks: 1) - Please choose one

What is the maximum depth of recursive calls a function may make?

- ▶ 1
- ▶ 2
- ▶ n (where n is the argument)
- ▶ **There is no fixed maximum**

Question No: 9 (Marks: 1) - Please choose one

Suppose n is the number of nodes in a complete Binary Tree then maximum steps required for a search operation are,

- ▶ **$\log_2(n+1) - 1$ (Page 139)**
- ▶ $\log_2(n+1)$
- ▶ $\log_2(n) - 1$
- ▶ $\log_2(n)$

Question No: 10 (Marks: 1) - Please choose one

In the linked list implementation of the stack class, where does the push member function places the new entry on the linked list?

- ▶ **At the head** (Page 53)
- ▶ At the tail
- ▶ After all other entries that are greater than the new entry.
- ▶ After all other entries that are smaller than the new entry.

Question No: 11 (Marks: 1) - Please choose one

Suppose we have a *circular* array implementation of the queue class, with ten items in the queue stored at data[2] through data[11]. The CAPACITY is 42, i.e., the array has been declared to be of size 42. Where does the push member function place the new entry in the array?

- ▶ data[1]
- ▶ data[2]
- ▶ data[11]
- ▶ **data[12]**

Question No: 12 (Marks: 1) - Please choose one

The expression $AB+C^*$ is called?

- ▶ Prefix expression
- ▶ **Postfix expression** (Page 70)
- ▶ Infix expression
- ▶ None of these

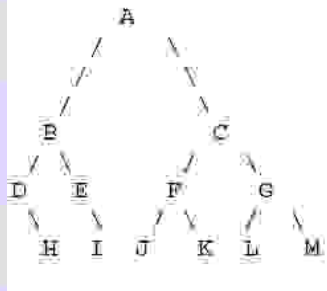
Question No: 13 (Marks: 1) - Please choose one

_____ is a binary tree where every node has a value, every node's left subtree contains only values less than or equal to the node's value, and every node's right subtree contains only values that are greater then or equal?

- ▶ Strictly Binary Tree
- ▶ **Binary Search tree** [Click here for detail](#)
- ▶ AVL tree
- ▶ All of these

Question No: 14 (Marks: 1) - Please choose one

Consider the following binary search tree (BST):



If node A in the BST is deleted, which two nodes are the candidates to take its place?

- ▶ J and I
- ▶ H and E
- ▶ D and E
- ▶ **L and M**

Question No: 15 (Marks: 1) - Please choose one

Let's call the node as that requires re-balancing. Consider the two cases given below:

- 1) An insertion into left sub tree of the left child of a
- 2) An insertion into right sub tree of the right child of a.

Which of the following statement is correct about these two cases?

- 1) The insertion occurs outside (i.e., left-left or right-right) in cases 1 and 2 single rotation can fix the balance in these two cases.
- 2) The insertion occurs inside ((i.e., left-left or right-right) in cases 1 and 2. single rotation cannot fix the balance in these two cases

Question No: 16 (Marks: 1) - Please choose one

We access elements in AVL Tree in,

- ▶ Linear way only
- ▶ **Non Linear way only**
- ▶ Both linear and non linear ways
- ▶ None of the given options.

Question No: 17 (Marks: 2)

AVL Tree is,

- ▶ **Non Linear data structure** [Click here for detail](#)
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.

MIDTERM EXAMINATION

Spring 2010

Question No: 1 (Marks: 1) - Please choose one

Which one of the following statement is NOT correct .

- ▶ **In linked list the elements are necessarily to be contiguous** [Click here for detail](#)
- ▶ In linked list the elements may locate at far positions in the memory
- ▶ In linked list each element also has the next to it
- ▶ In an array the elements are contiguous

Question No: 2 (Marks: 1) - Please choose one

Each operator in a postfix expression refers to the previous _____ operand(s).

- ▶ One
- ▶ **Two (Page 67)**
- ▶ Three
- ▶ Four

Question No: 3 (Marks: 1) - Please choose one

Which one of the following calling methods does not change the original value of the argument in the calling function?

- ▶ None of the given options
- ▶ **Call by passing the value of the argument** [Click here for detail](#)
- ▶ Call by passing reference of the argument
- ▶ Call by passing the address of the argument

Question No: 4 (Marks: 1) - Please choose one

A tree is an AVL tree if

- ▶ Any one node fulfills the AVL condition
- ▶ At least half of the nodes fulfill the AVL condition
- ▶ **All the nodes fulfill the AVL condition (Page 213)**
- ▶ None of the given options

Question No: 5 (Marks: 1) - Please choose one

Suppose currentNode refers to a node in a linked list (using the Node class with member variables called data and nextNode). What statement changes currentNode so that it refers to the next node?

- ▶ currentNode ++;
- ▶ currentNode = nextNode;
- ▶ currentNode += nextNode;
- ▶ **currentNode = currentNode->nextNode;**

Question No: 6 (Marks: 1) - Please choose one

A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True** (Page 101)

Question No: 7 (Marks: 1) - Please choose one

Which one is a self- referential data type?

- ▶ Stack
- ▶ Queue
- ▶ Link list
- ▶ **All of these** [Click here for detail](#)

Question No: 8 (Marks: 1) - Please choose one

Each node in doubly link list has,

- ▶ 1 pointer
- ▶ **2 pointers** (Page 39)
- ▶ 3 pointers
- ▶ 4 pointers

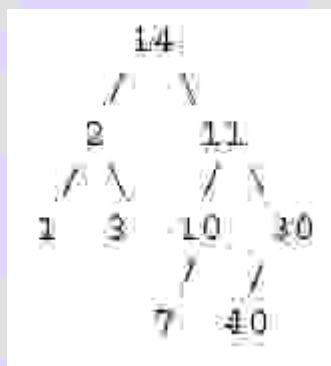
Question No: 9 (Marks: 1) - Please choose one

I have implemented the queue with a linked list, keeping track of a front pointer and a rear pointer. Which of these pointers will change during an insertion into an *EMPTY* queue?

- ▶ Neither changes
- ▶ Only front pointer changes.
- ▶ **Only rear pointer changes.**
- ▶ Both change.

Question No: 10 (Marks: 1) - Please choose one

Consider the following tree.



How many of the nodes have at least one sibling?

- ▶ 8
- ▶ 7
- ▶ 5
- ▶ 6

A sibling is an element that shares the same parent with another element

Question No: 11 (Marks: 1) - Please choose one

The nodes with no successor are called _____

- ▶ Root Nodes
- ▶ **Leaf Nodes**
- ▶ Both of these
- ▶ None of these

Question No: 12 (Marks: 1) - Please choose one

AVL Tree is,

- ▶ **Non Linear data structure** [Click here for detail](#)
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.

Question No: 13 (Marks: 1) - Please choose one

We access elements in AVL Tree in,

- ▶ Linear way only
- ▶ **Non Linear way only**
- ▶ Both linear and non linear ways
- ▶ None of the given options.

Question No: 14 (Marks: 1) - Please choose one

A binary search tree should have minimum of one _____ node/s at each level,

- ▶ One
- ▶ **Two** [Click here for detail](#)
- ▶ Three
- ▶ Four

Question No: 15 (Marks: 1) - Please choose one

Consider the following statements.

- (i) A binary tree can contain at least 2^L Nodes at level L.
- (ii) A complete binary tree of depth d is a binary tree that contains 2^L Nodes at each level L between 0 and d, both inclusive.
- (iii) The total number of nodes (T_n) in a complete binary tree of depth d is $2^{d+1} - 1$.
- (iv) The height of the complete binary tree can be written as $h = \log_2 (T_n + 1) - 1$ where T_n is Total number of Nodes.

Which one of the following is correct in respect of the above statements regarding the Binary trees?

- ▶ (i) and (iii) only
- ▶ (i), (ii) and (iii) only
- ▶ (ii) and (iii) only
- ▶ **(ii), (iii) and (iv) only** [Click here for detail](#)

Question No: 16 (Marks: 1) - Please choose one
“+” is a _____ operator.

- ▶ Unary
- ▶ **Binary** **(Page 64)**
- ▶ Ternary
- ▶ None of the above

MIDTERM EXAMINATION

Spring 2010

Question No: 1 (M a r k s: 1)

A subscript of an array may be an integer or an integer expression.

- ▶ **True** [Click here for detail](#)
- ▶ False

Question No: 2 (M a r k s: 1)

Doubly Linked List always has one NULL pointer.

- ▶ True
- ▶ **False** **(Page 43)**

Question No: 3 (M a r k s: 1)

In which of the traversal method, the recursive calls can be used to traverse a binary tree ?

- ▶ **In preorder traversal only** **(Page 143)**
- ▶ In inorder traversal only
- ▶ In postorder traversal only
- ▶ All of the given options

Question No: 4 (M a r k s: 1)

A tree is an AVL tree if

- ▶ Any one node fulfills the AVL condition
- ▶ At least half of the nodes fulfill the AVL condition
- ▶ **All the nodes fulfill the AVL condition** **(Page 213)**
- ▶ None of the given options

Question No: 5 (Marks: 1)

Suppose currentNode refers to a node in a linked list (using the Node class with member variables called data and nextNode). What boolean expression will be true when cursor refers to the tail node of the list?

- ▶ **(currentNode == null)**
- ▶ (currentNode->nextNode == null)
- ▶ (nextNode.data == null)
- ▶ (currentNode.data == 0.0)

Question No: 6 (Marks: 1) - Please choose one

Suppose that the class declaration of SomeClass includes the following function prototype.

bool LessThan(SomeClass anotherObject);

Which of the following tests in the client code correctly compares two class objects alpha and beta?

- ▶ if (alpha < beta)
- ▶ **if (alpha.LessThan(beta))** [Click here for detail](#)
- ▶ if (LessThan(alpha, beta))
- ▶ if (LessThan(alpha).beta)

Question No: 7 (Marks: 1)

In C what is the operation that you can not do with primitive types?

- ▶ Assign a value to primitive type using a literal
- ▶ Declare primitive types to be constant using the Const keyword
- ▶ **Create a new instance of primitive type with New keyword** [Click here for Detail](#)
- ▶ None of these

Question No: 8 (Marks: 1)

The operation for adding an entry to a stack is traditionally called :

- ▶ add
- ▶ append
- ▶ insert
- ▶ **push** (Page 53)

Question No: 9 (Marks: 1)

The operation for removing an entry from a stack is traditionally called:

- ▶ delete
- ▶ peek
- ▶ **pop** (Page 53)
- ▶ remove

Question No: 10 (Marks: 1)

Consider the following sequence of push operations in a stack:

```
stack.push('7');  
stack.push('8');  
stack.push('9');  
stack.push('10');  
stack.push('11');  
stack.push('12');
```

- ▶ 7 8 9 10 11 12
- ▶ 9 8 11 10 7 12
- ▶ 9 10 8 11 12 7
- ▶ 9 10 8 12 7 11

Question No: 11 (Marks: 1)

_____ is the maximum number of nodes that you can have on a stack-linked list ?

- ▶ Zero
- ▶ $2n$ (where n is the number of nodes in linked list)
- ▶ **Any Number** [Click here for detail](#)
- ▶ None of these

Question No: 12 (Marks: 1)

Which of the following can be used to reverse a string value,

- ▶ **Stack** [Click here for detail](#)
- ▶ Queue
- ▶ Both of these
- ▶ None of these

Question No: 14 (Marks: 1)

AVL Tree is,

- ▶ **Non Linear data structure** [Click here for detail](#)
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.

Question No: 15 (Marks: 1)

The following are statements related to queues.

- (i) The last item to be added to a queue is the first item to be removed
- (ii) A queue is a structure in which both ends are not used
- (iii) The last element hasn't to wait until all elements preceding it on the queue are removed
- (iv) A queue is said to be a last-in-first-out list or LIFO data structure.

Which of the above is/are related to normal queues?

- ▶ (iii) and (ii) only
- ▶ (i), (ii) and (iv) only
- ▶ (ii) and (iv) only
- ▶ **None of the given options** [Click here for detail](#)

Question No: 16 (Marks: 1)

An array is a group of consecutive related memory locations.

- ▶ **True** [Click here for detail](#)
- ▶ False

MIDTERM EXAMINATION

Spring 2010

Question No: 1 (Marks: 1) - Please choose one

In an array we can store data elements of different types.

- ▶ True
- ▶ **False** (Page 7)

Question No: 2 (Marks: 1) - Please choose one

In an array list the current element is

- ▶ **The first element** [Click here for detail](#)
- ▶ The middle element
- ▶ The last element
- ▶ The element where the current pointer points to

Question No: 3 (Marks: 1) - Please choose one

Which one of the following calling methods does not change the original value of the argument in the calling function?

- ▶ None of the given options
- ▶ **Call by passing the value of the argument** [Click here for detail](#)
- ▶ Call by passing reference of the argument
- ▶ Call by passing the address of the argument

Question No: 4 (Marks: 1) - Please choose one

Which one of the following statements is NOT correct?

- ▶ **Array size can be changed after its creation.** [Click here for detail](#)
- ▶ Link List size can be changed after its creation
- ▶ Binary Search Tree size can be changed after its creation
- ▶ AVL Tree size can be changed after its creation

Question No: 5 (Marks: 1) - Please choose one

Suppose that the class declaration of SomeClass includes the following function prototype. `bool LessThan( SomeClass anotherObject );`

Which of the following tests in the client code correctly compares two class objects alpha and beta?

- ▶ `if (alpha < beta)`
- ▶ **`if (alpha.LessThan(beta))`** [Click here for detail](#)
- ▶ `if (LessThan(alpha, beta))`
- ▶ `if (LessThan(alpha).beta)`

Question No: 6 (Marks: 1) - Please choose one

A queue is a----- data structure, whereas a stack is a -----data structure.

- ▶ **FIFO, LIFO** (Page 161,54)
- ▶ LIFO,FIFO
- ▶ none of these
- ▶ both of these

Question No: 7 (Marks: 1) - Please choose one

Which one of the following operators has higher priority than all of others?

- ▶ **Multiplication operator** [Click here for detail](#)
- ▶ Minus operator
- ▶ Plus operator
- ▶ Exponentiation operator

Question No: 8 (Marks: 1) - Please choose one

Each node in Binary Search Tree has

- ▶ 1 pointer
- ▶ **2 pointers** [Click here for detail](#)
- ▶ 3 pointers
- ▶ 4 pointers

Question No: 9 (Marks: 1) - Please choose one

Four statements about trees are below. Three of them are correct. Which one is INCORRECT?

- ▶ Trees are recursively defined multi-dimensional data structures tree
- ▶ The order of a tree indicates a maximum number of children allowed at each node of the
- ▶ A search tree is a special type of tree where all values (i.e. keys) are ordered
- ▶ **If Tree1's size is greater than Tree2's size, then the height of Tree1 must also be greater than Tree2's height.** [Click here for detail](#)

Question No: 10 (Marks: 1) - Please choose one

Which of the following is "TRUE" about arrays,

- ▶ We can increase the size of arrays after their creation.
- ▶ We can decrease the size of arrays after their creation.
- ▶ We can increase but can't decrease the size of arrays after their creation.
- ▶ **We can neither increase nor decrease the array size after their creation.** [Click here for detail](#)

Question No: 11 (Marks: 1) - Please choose one

Searching an element in an AVL tree take maximum in AVL tree,

- ▶ $\log_2(n+1)$
time (where n is no. of nodes)
- ▶ $\log_2(n+1) - 1$
- ▶ **$1.44 \log_2 n$** (Page 227)
- ▶ $1.66 \log_2 n$

Question No: 12 (Marks: 1) - Please choose one

There is/are case/s for rotation in an AVL tree,

- ▶ 1
- ▶ 3
- ▶ 2
- ▶ **4** (Page 229)

Question No: 13 (Marks: 1) - Please choose one

Consider the following statements.

- (i) A binary tree can contain at least $2L$ Nodes at level L .
- (ii) A complete binary tree of depth d is a binary tree that contains $2L$ Nodes at each level L between 0 and d , both inclusive.
- (iii) The total number of nodes (T_n) in a complete binary tree of depth d is $2^{d+1} - 1$.
- (iv) The height of the complete binary tree can be written as $h = \log_2(T_n + 1) - 1$ where T_n is Total number of Nodes.

Which one of the following is correct in respect of the above statements regarding the Binary trees?

- ▶ (i) and (iii) only
- ▶ (i), (ii) and (iii) only
- ▶ (ii) and (iii) only
- ▶ (ii), (iii) and (iv) only [Click here for detail](#)

Question No: 14 (Marks: 1) - Please choose one

Consider the following infix expression.

$$5 + 6/2$$

If one converts the above expression into postfix, what would be the resultant expression?

- ▶ $56/ + 2$
- ▶ **$5\ 6\ 2\ /\ +$** (Page 66)
- ▶ $5\ 6\ /\ 2\ +$
- ▶ $/62 + 5$

Question No: 15 (Marks: 1) - Please choose one

Which of the following is a non linear data structure?

- ▶ Linked List
- ▶ Stack
- ▶ Queue
- ▶ **Tree** (Page 112)

Question No: 16 (Marks: 1) - Please choose one

“+” is a operator.

- ▶ Unary
- ▶ **Binary** (Page 64)
- ▶ Ternary
- ▶ None of the above

MIDTERM EXAMINATION

Spring 2010

1. Addition of new items in stack make the pointer ----- by 2

- a. Increment, bits
- b. Increment, bytes
- c. Decrement, bits
- d. Decrement, bytes** [Click here for detail](#)

2. Next item in a linked list is known as

- a. Index
- b. Item
- c. Node** [Click here for detail](#)
- d. Child

3. What will be the postfix notation of $5+6/2$.

- a. $56+/2$
- b. $562+/$
- c. $562/+$** (Page 66)
- d. $5+62/$

4. In an AVL tree to delete a parent with two childs in a straight line following rotations will be required:-

- a. Single
- b. Double**
- c. Triple
- d. None.

5. To check the depth of an AVL tree following time will be taken:-

- a. $1.66 \log_2 n$
- b. $1.44 \log_2 n$** (Page 227)
- c. $\log_2 (n+1)-1$
- d. $1.66 \log_2 (n+1)$

6. BST is a Structure:-

- a. Linear
- b. Non Linear** [Click here for detail](#)
- c. Circular
- d. None of Above

7. After creation of an array:-

- a. Size can be increase but can not be decreased.
- b. Size can be decreased but can not be increased.
- c. Size can neither be increased nor be decreased.** [Click here for detail](#)
- d. Size can be increased and can also be decreased.

8. Each node in a BST has Pointers:-

- a. 1
- b. 2** [Click here for detail](#)
- c. 3
- d. 4

9. Highest Operators Precedence is of the following operator:-

- a. Plus
- b. Minus
- c. Multiply** [Click here for detail](#)
- d. Exponentiation

10. Following are the linear data structures:-

- a. Stacks
- b. Queues
- c. Both a & b (Page 52, 87)**
- d. None of the above

11. Each entry which points to a null value in a Singly Linked List is known as:-

- a. Node
- b. First Node
- c. Last Node**
- d. Head Node

12. Non recursive calls are faster than the Recursive calls.

- a. True (Page 323)**
- b. False

13. Tree data structure is a

- a. Linear
- b. Non Linear (Page 112) rep**
- c. Circular
- d. None of Above

14. What will be the valid postfix notation of $A+B*C-D$

- a. $ABC+*D-$
- b. $ABC*+D-$ (According to rule)**
- c. $ABCD+-*$
- d. $AB+D*C$

15. When an operator is used in between two operands this is which type of notation

- a. Prefix
- b. Postfix
- c. Infix (Page 64)**
- d. None of the Above

MIDTERM EXAMINATION

Spring 2009

Question No: 1 (Marks: 1) - Please choose one

Which one of the following is a valid postfix expression?

- ▶ $ab+c*d-$
- ▶ **$abc**d-$ (According to rule)**
- ▶ $abc+*d-$
- ▶ $(abc*)+d-$

Question No: 2 (Marks: 1) - Please choose one

The tree data structure is a

- ▶ Linear data structure
- ▶ **Non-linear data structure (Page 112)**
- ▶ Graphical data structure
- ▶ Data structure like queue

Question No: 3 (Marks: 1) - Please choose one

A **Compound Data Structure** is the data structure which can have multiple data items of same type or of different types. Which of the following can be considered compound data structure?

- ▶ **Arrays** [Click here for detail](#)
- ▶ LinkLists
- ▶ Binary Search Trees
- ▶ All of the given options

Question No: 4 (Marks: 1) - Please choose one

Suppose a pointer has been declared in main but has not assigned any variable address then

- ▶ That pointer points to First byte in main function
- ▶ **That pointer contains a NULL value**
- ▶ None of these
- ▶ **That pointer points to any memory address**

Question No: 5 (Marks: 1) - Please choose one

Here is the start of a C++ class declaration:

```
class foo
{
public:
void x(foo f);
void y(const foo f);
void z(foo f) const;
...
```

Which of the three member functions can alter the PRIVATE member variables of the foo object that activates the function?

- ▶ **Only x can alter the private member variables of the object that activates the function.**
- ▶ Only y can alter the private member variables of the object that activates the function.
- ▶ Only z can alter the private member variables of the object that activates the function.

► **Two of the functions can alter the private member variables of the object that activates the function.**

Only the x and y can alter the private member variable of the foo class object. Last Option is more correct but not exact. In the last option the two function name are not mentioned

Question No: 6 (Marks: 1) - Please choose one

The operation for removing an entry from a stack is traditionally called:

- delete
- peek
- **pop (Page 53)**
- remove

Question No: 7 (Marks: 1) - Please choose one

Which statement of the following statements is incorrect?

- Lists can be implemented by using arrays or linked lists
- A list is a sequence of one or more data items
- Stack is a special kind of list in which all insertions and deletions take place at one end
- **Stacks are easier to implement than lists**

Question No: 8 (Marks: 1) - Please choose one

Parameters in function call are passed using,

- **Stack (Page 80)**
- Queue
- Binary Search Tree
- AVL Tree

Question No: 9 (Marks: 1) - Please choose one

Consider the following sequence of push operations in a stack:

```
stack.push('7');  
stack.push('8');  
stack.push('9');  
stack.push('10');  
stack.push('11');  
stack.push('12');
```

- **7 8 9 10 11 12**
- 9 8 11 10 7 12
- 9 10 8 11 12 7
- 9 10 8 12 7 11

Question No: 10 (Marks: 1) - Please choose one

What is the maximum depth of recursive calls a function may make?

- 1
- 2
- n (where n is the argument)
- **There is no fixed maximum**

Question No: 11 (Marks: 1) - Please choose one

Consider the following function:

```
void test_a(int n)
{
    cout << n << " ";
    if (n>0)
        test_a(n-2);
}
```

What is printed by the call test_a(4)?

- ▶ 4 2 0
- ▶ 0 2 4
- ▶ 0 2
- ▶ 2 4

Question No: 12 (Marks: 1) - Please choose one

Queue follows,

- ▶ Last in First out
- ▶ First in Last out
- ▶ **First in First out** (Page 87)
- ▶ None of these

Question No: 13 (Marks: 1) - Please choose one

is a binary tree where every node has a value, every node's left subtree contains only values less than or equal to the node's value, and every node's right subtree contains only values that are greater then or equal ?

- ▶ **Strictly Binary Tree**
- ▶ **Binary Search tree** [Click here for detail](#)
- ▶ AVL tree
- ▶ All of these

Question No: 14 (Marks: 1) - Please choose one

Four statements about trees are below. Three of them are correct. Which one is INCORRECT?

- ▶ **Trees are recursively defined multi-dimensional data structures** [Click here for detail](#)
- ▶ The order of a tree indicates a maximum number of children allowed at each node of the tree
- ▶ A search tree is a special type of tree where all values (i.e. keys) are ordered
- ▶ If Tree1's size is greater than Tree2's size, then the height of Tree1 must also be greater than Tree2's height.

Question No: 15 (Marks: 1) - Please choose one

Below is a binary search tree. If we delete the value 50 using the algorithm we discussed, what value will be in the root of the remaining tree?

- ▶ 50
- ▶ 60
- ▶ 70
- ▶ 80

Question No: 16 (Marks: 1) - Please choose one

Is a data structure that can grow easily dynamically at run time without having to copy existing elements?

- ▶ Array
- ▶ **List**
- ▶ Both of these
- ▶ None of these

MIDTERM EXAMINATION

Spring 2009

Question No: 1 (Marks: 1) - Please choose one

Which one of the following statement is NOT correct .

- ▶ **In linked list the elements are necessarily to be contiguous** [Click here for detail](#)
- ▶ In linked list the elements may locate at far positions in the memory
- ▶ In linked list each element also has the address of the element next to it
- ▶ In an array the elements are contiguous

Question No: 2 (Marks: 1) - Please choose one

In a program a reference variable, say x, can be declared as

- ▶ **int &x ;** [Click here for detail](#)
- ▶ int *x ;
- ▶ int x ;
- ▶ None of the given options

Question No: 3 (Marks: 1) - Please choose one

Linked lists are collections of data items "lined up in a row" , insertions and deletions can be made only at the front and the back of a linked list.

- ▶ True
- ▶ **False** [Click here for detail](#)

Question No: 4 (Marks: 1) - Please choose one

A Linear Data Structure is the data structure in which data elements are arranged in a sequence or a linear list. Which of the following is Non Linear Data Structure?

- ▶ Arrays
- ▶ LinkLists
- ▶ **Binary Search Trees** [Click here for detail](#)
- ▶ None of these

Question No: 5 (Marks: 1) - Please choose one

A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True** (Page 101)

Question No: 6 (Marks: 1) - Please choose one

Which one of the following statements is correct?

- ▶ **Array size is fixed once it is created** [Click here for detail](#)
- ▶ Link List size is fixed once it is created.
- ▶ Binary Search Tree size is fixed once it is created
- ▶ AVL Tree size is fixed once it is created

Question No: 7 (Marks: 1) - Please choose one

Which one of the following is correct about pointers?

- ▶ They always point to different memory locations
- ▶ They may point to a single memory location
- ▶ The address of two pointer variables is same
- ▶ None of these

Question No: 8 (Marks: 1) - Please choose one

Which of the following abstract data types are NOT used by Integer Abstract Data type group?

- ▶ short
- ▶ Int
- ▶ **float** [Click here for detail](#)
- ▶ long

Question No: 9 (Marks: 1) - Please choose one

The operation for adding an entry to a stack is traditionally called :

- ▶ add
- ▶ append
- ▶ insert
- ▶ **push** (Page 53)

Question No: 10 (Marks: 1) - Please choose one

The operation for removing an entry from a stack is traditionally called:

- ▶ delete
- ▶ peek
- ▶ **pop** (Page 53)
- ▶ remove

Question No: 11 (Marks: 1) - Please choose one

We can add elements in QUEUE From _____

- ▶ Front
- ▶ **Rear** (Page 91)
- ▶ From Both Rare and Front
- ▶ None of these

Question No: 12 (Marks: 1) - Please choose one

The difference between a binary tree and a binary search tree is that ,a binary search tree has

▶ **two children per node whereas a binary tree can have none, one, or two children per node** [Click here for detail](#)

- ▶ in binary search tree nodes are inserted based on the values they contain
- ▶ in binary tree nodes are inserted based on the values they contain
- ▶ none of these

Question No: 13 (Marks: 1) - Please choose one

Suppose n is the number of nodes in a complete Binary Tree then maximum steps required for a search operation are,

▶ **$\log_2(n+1) - 1$** (Page 139)

- ▶ $\log_2(n+1)$
- ▶ $\log_2(n) - 1$
- ▶ $\log_2(n)$

Question No: 14 (Marks: 1) - Please choose one

The following is a segment of a C program.

```
int pqr(BinaryNode t)
{ if (t == null )
return -1;
else
return 1+max(pqr(t.left),pqr(t.right)) }
```

Identify, what the above program intend(s) to do?

- ▶ Compute the height of a binary tree using an in-order traversal
- ▶ Compute the height of a binary tree using a pre-order traversal
- ▶ Compute the depth of a binary tree using a pre-order traversal
- ▶ Compute the depth of a binary tree using a post-order traversal

Question No: 15 (Marks: 1) - Please choose one

Consider the following infix expression:

$3 + 5 * 6 - 7 * (8 + 5)$

Which of the following is a correct equivalent expression(s) for the above?

- ▶ $3 \ 6 \ 5 + * \ 7 \ 5 \ 8 + - *$
- ▶ $3 \ 6 \ 5 \ 7 \ 5 \ 8 + * + - *$
- ▶ $3 \ 5 \ 6 + * \ 7 \ 8 \ 5 + - *$
- ▶ **$3 \ 5 \ 6 * + \ 7 \ 8 \ 5 + * -$**

Question No: 16 (Marks: 1) - Please choose one

An array is a group of consecutive related memory locations.

- ▶ **True** [Click here for detail](#)
- ▶ False

Question No: 17 (Marks: 1)

Is this a correct statement? Give answer in Yes or No.

A node cannot be deleted, when the node to be deleted has both left and right subtrees.

False ---- No, it can be deleted.

Question No: 18 (Marks: 1)

Deleting a leaf node in binary search tree involves setting _____ pointer/s of that node's parent as null.

- 1
- 2
- 3
- 4



Virtual University

CS301- DTA STRUCTURE
(SOLVED MCQs)
FROM MIDTERM PAPERS
LECTURE (1-22)



BC190202640@vu.edu.pk
Junaidfaza108@gmail.com

For More Visit: vulmshelp.com

JUNAID MALIK
(0304-1659294)



AL-JUNAID TECH
INSTITUTE

PAID SERVICE
CS619 PROJECTS

Available training courses

- HTML
- JQUERY
- PHPMYSQL
- JAVASCRIPT
- BOOTSTRAP5
- NODEJS
- REACTJS
- CSS

LMS HANDLING

PAID
ASSIGNMENTS, QUIZ & QCM

95% RESULTS
ALL LMS ACTIVITIES

Contact Us :
+92 304 1659294
www.vulmshelp.com
junaidfaza108@gmail.com

AL-JUNAID TECH INSTITUTE

1. Consider the following infix expression. $7/8+9$

If one converts the above expression into postfix, what would be the resultant expression?

- a. $789/+$
 - b. $78/+9$
 - c. $/78+9$
 - d. $78/9+$
2. Suppose there are 100 elements in an equivalence classes so initially there will be 100 tree. The collection of tree is called _
- a. Cluster
 - b. Class
 - c. Forest
 - d. Bunch
3. For a perfect binary tree of height h , having N nodes the sum of height of nodes is
- a. $N-h-1$
 - b. $N-1$
 - c. $N-1+h$
 - d. $-(h-1)$
4. Sorting procedure normally takes _____ time
- a. $N \log N$
 - b. $2N$
 - c. $N*N*N$
 - d. N
5. there are 100 elements in an equivalence classes then will have _____ state initially
- a. 50
 - b. 100
 - c. 1000
 - d. 80
6. _____ objects (objects accessed by pointers) are called anonymous objects.
- a. Private
 - b. Nameless
 - c. Friend

AL-JUNAID TECH INSTITUTE

- d. Public
7. _____ is a self-balancing tree.
- a. AVL
 - b. Binary Tree
 - c. Binary Search Tree
 - d. ALV
8. Which of the following operation returns but do not removes top value of the stack?
- a. Push
 - b. Pop
 - c. Top
 - d. First
9. `Int htdiff = height(root->getLeft()) _____ height(root->getRight());` The above line of code is taken from AVL insert method. Complete it by selecting an appropriate symbol.
- a. -
 - b. +
 - c. /
 - d. *
10. Which operation of the queue data structure is used to insert an element into the Queue?
- a. Enqueue()
 - b. Dequeue()
 - c. Fornt()
 - d. Remove()
11. For making binary search tree for strings we need, _____ data type.
- a. Char
 - b. Int
 - c. Float
 - d. Double
12. Local variables defined inside function body are _____ automatically at the end of function execution.
- a. Created
 - b. Destroyed
 - c. Incremented

AL-JUNAID TECH INSTITUTE

- d. Decremental
- 9. If a node has three fields, then this node will be related to
 - a. Linked list
 - b. Doubly linked list
 - c. Circular linked list
 - d. All of the given
- 10. _____ is an area in computer memory that is allocated dynamically.
 - a. Heap
 - b. Stack
 - c. Queue
 - d. Linked list
- 11. Linked list use _____ to store data.
 - a. Array
 - b. 2-D Array
 - c. Variables
 - d. Linked Memory
- 12. The lifetime of a transient object cannot exceed that of the application.
 - a. True
 - b. False
 - c. In some cases
 - d. None of the given
- 13. To represent hierarchical relationship between elements, which data structure is suitable?
 - a. Dequeue
 - b. Priority
 - c. Stack
 - d. Tree
- 14. Local variables of a function are stored in,
 - a. Binary search tree
 - b. Stack
 - c. Queue
 - d. AVL tree
- 15. In-order traversal method traverses the data in
 - a. Non sorted order
 - b. Random order

AL-JUNAID TECH INSTITUTE

- c. Sorted order
- d. None of the given
16. Dequeue() operation of queue data structure is used to
- a. Get an element from the front of the queue
 - b. Remove an element from the front and return it
 - c. Insert an element at the front
 - d. Insert an element at the back
17. In a tree, we link the nodes in such a way that it _____ a linear structure.
- a. Does not remain
 - b. Forms
 - c. Reverses
 - d. Remains
18. Factorial is an example of _____ function.
- a. Recursive
 - b. Non-recursive
 - c. Cube
 - d. Log
19. All the objects created using _____ operator have to be explicitly destroyed using delete operator.
- a. New
 - b. Delete
 - c. Del
 - d. Create
20. In the calling function, after the execution of the function called, the program continues its execution from the _____ after the function call.
- a. Previous line
 - b. Next line
 - c. Beginning
 - d. None of the above
21. Last node in circular linked list contains
- a. At least one null pointer
 - b. No null pointer
 - c. Maximum two null pointers
 - d. None of the given

AL-JUNAID TECH INSTITUTE

22. One should be careful about transient _____ that are stored by reference in data structures.

a. Objects

b. Stack

c. Function

d. Tree

23. Binary search tree violates the condition of AVL tree when any node has balance equal to

a. 2 or -2

b. 1 or -1

c. 0

d. None of the options

24. Security of data is the main usage of AVL tree.

a. True

b. False

c. In some cases

d. None of the given

25. A binary tree whose every node has either zero or two children is called _____.

a. Complete binary tree

b. Binary search tree

c. Strictly binary tree

d. None of above

26. In internal memory organization of a process, there is some area of memory for static data that holds _____ variables.

a. Static

b. Global

c. Not static neither global

d. Both static and global

27. Following is true in case of using recursive method calls.

a. The code becomes very long

b. There is no effect on length of code

c. The code becomes very short

d. Code becomes very easy to understand

28. In which traversal method, the recursive calls can be used to traverse a binary tree?

a. In preorder traversal only

AL-JUNAID TECH INSTITUTE

- b. In inorder traversal only
 - c. In postorder traversal only
 - d. All of the given options
29. If we use doubly linked list to implement list then there is an issue of
- a. Next pointer of first node and pre pointer of last node are NULL
 - b. Next pointer of first node and next pointer of last node are NULL
 - c. Pre pointer of first node and next pointer of last node are NULL
 - d. Pre pointer of first node and pre pointer of last node are NULL
30. In case of insertion of left outer node in BST,
- a. We apply single right rotation to make it AVL tree.
 - b. We apply single left rotation to make it AVL tree.
 - c. We first apply right rotation and then left rotation to make it AVL tree.
 - d. We first apply left rotation and then right rotation to make it AVL tree.
31. Whenever we call a function, the compiler makes a stack, the top element of the stack is _____ of the function.
- a. First argument
 - b. Return address
 - c. Last argument
 - d. None of the above
32. Suppose a stack class has been defined using template. Now, we want to declare a Stack object of an int type. What will be the correct syntax?
- a. <int>Stack stack;
 - b. Stack<int> stack;
 - c. Stack int stack;
 - d. Int Stack stack;
33. In node class one field is an integer data and other field will be _____
- a. Pointer to class
 - b. Pointer to node
 - c. Pointer to integer
 - d. None of given options

AL-JUNAID TECH INSTITUTE

34. If there are 100,000 unique members (nodes) stored in a complete binary tree, the tree will have _____ levels.
- a. 10
 - b. 20
 - c. 30
 - d. 40
35. ~BinarySearchTree() is a _____.
- a. Constructor
 - b. Destructor
 - c. Switch case
 - d. Template method call
36. _____ is utilized at the time of memory allocation in dynamic manner.
- a. Stack
 - b. Queue
 - c. Heap
 - d. All of the given
37. How many cases of rotation are there in AVL tree?
- a. 2
 - b. 4
 - c. 6
 - d. 8
38. A list is the collection of items of the _____
- a. May be of same or may be of different type
 - b. Different type
 - c. Same type
 - d. None of the above
39. `Int * i =`
`new int[10];`
Above given code will:
- a. Create an integer having value 10
 - b. Allocate memory for 9 integers
 - c. Allocate memory for 10 integers
 - d. Create 10 pointers of integer type
40. The depth of a binary tree is
- a. Total number of nodes in the tree

AL-JUNAID TECH INSTITUTE

- b. Number of leaf nodes in the tree
 - c. Number of non-leaf nodes in the tree
 - d. Maximum level of a leaf
41. A _____ model attempts to model a real-world phenomenon
- a. Physical
 - b. Logical
 - c. Simulation
 - d. Conceptual
42. In level-order traversal for binary search tree, we visit the nodes at each level before proceeding to the next level, in a _____ order.
- a. Right-to-left
 - b. Left-to-right
 - c. Top-to-bottom
 - d. Bottom-to-top
43. The balance of a node in a binary tree is defined as the height of its _____ sub tree minus height of its right sub tree.
- a. Right
 - b. Left
 - c. Upper
 - d. Lower
44. Allocating and de-allocating memory for linked list nodes does take _____ time than pre-allocated array.
- a. Less
 - b. More
 - c. Equal
 - d. No
45. Consider we have performed the following operations on a stack of size 5. Push(10);
Push(20);
Push(30);
Pop();
Pop();
Pop();
Push(40);
Push(50);

AL-JUNAID TECH INSTITUTE

Pop();

After the completion of all operation, the top element in stack is

_____.

- a. 10
 - b. 20
 - c. 40
 - d. 50
46. Each operator in a postfix expression refers to the previous _____ operand(s).
- a. One
 - b. Two
 - c. Three
 - d. Four
47. In doubly linked list a node consists of three parts:
- a. 1 pointer and 2 objects
 - b. 2 pointers and 1 object
 - c. 3 objects
 - d. 3 pointers
48. For reference variables, _____ sign is used.
- a. Ampersand
 - b. Asterisk
 - c. Sigma
 - d. Dollar
49. Which of the following data structure is linear type?
- a. Stack
 - b. List
 - c. Queue
 - d. All of the above
50. The - - is a decrement operator in C++ that decreases the value of the operand by _____.
- a. One
 - b. Two
 - c. Three
 - d. Four
51. Which of the following statement is correct for the variable "current- -"?

AL-JUNAID TECH INSTITUTE

- a. $\text{Current} = \text{current} + 1$
 - b. $\text{Current} = \text{current} - 1$
 - c. $\text{Current} = \text{current} - 2$
 - d. $\text{Current} - 1 = \text{current}$
52. When an executable program runs, it is loaded in the computer memory and becomes a _____.
- a. Thread
 - b. .h file
 - c. Process
 - d. None of the above
53. start() method of list class is used to:
- a. Moves the "current" pointer to very first element
 - b. Moves the "current" pointer to very last element
 - c. Moves the "current" pointer to one step after the first element of the array
 - d. Moves the "current" pointer to one step before the first element of the array
54. $a * (b + c) - d$ is an example of _____ expression.
- a. Infix
 - b. Prefix
 - c. Postfix
 - d. Alfix
55. In a program a reference variable, say x, can be declared as
- a. `int &x;`
 - b. `int *x;`
 - c. `int x;`
 - d. none of the given options
56. We allocate memory dynamically by using _____ operator.
- a. This
 - b. New
 - c. Increment
 - d. Decrement
57. Get(?) method of list class is used to:
- a. Get element from the last position
 - b. Get element from the first position

AL-JUNAID TECH INSTITUTE

- c. Get element from the middle position
d. Get element at the given position
58. STL is a _____ that is a part of the official standard of C++.
- a. C program file
 - b. .h file
 - c. .cpp file
 - d. Library
59. There are _____ cases for deleting a node from binary search tree.
- a. 1
 - b. 2
 - c. 3
 - d. 4
60. In a list, tail() method of current pointer _____
- a. Returns the last element of the “current” pointer
 - b. Moves the “current” pointer to the very first element
 - c. Moves the “current” pointer to the very last element
 - d. Returns the first element of the “current” pointer
61. Suppose we have been given the following data set for a queue:
37524
What will be the resultant queue if we call a front() method?
- a. 7524
 - b. 37524
 - c. 75243
 - d. 5 2 4
62. Suppose we have been given the following data set for a queue:
7524
What will be the resultant queue if we call enqueue(3) method? Note that 7 is the front element whereas 4 is rear element of queue.
- a. 7524
 - b. 37524
 - c. 75243
 - d. 5243
63. Maximum time that an insertion operation can take in AVL tree is _____. Here log stands for log to the base of 2.
- a. Log (n)

AL-JUNAID TECH INSTITUTE

- b. $1.44 \log(n)$
 - c. $1.66 \log(n)$
 - d. $\log(n+1)$
64. _____ is when function is calling to itself.
- a. Loop
 - b. Recursion
 - c. Iteration
 - d. Nested loop
65. Left, right, info, and parent are the operations of _____ data structure.
- a. Stack
 - b. Tree
 - c. Queue
 - d. Linked list
66. Following is a keyword of C++
- a. Del
 - b. Delete
 - c. Remove
 - d. Eliminate
67. The expression $DE+H^*$ is called _____
- a. Prefix expression
 - b. Infix expression
 - c. Postfix expression
 - d. Hybrid expression
68. A software solution is said to be efficient if it solves the problem _____.
- a. By using some extra resources
 - b. Within no time
 - c. By consuming more hardware resources
 - d. Within its resources constraints
69. Each node in a singly linked list contains two fields, one field called data field while other field contains:
- a. Pointer to an integer
 - b. Pointer to character

AL-JUNAID TECH INSTITUTE

- c. Pointer to next node
d. Pointer to class
70. Each node in singly linked list contains _____.
a. One pointer
b. Two pointers
c. No pointer
d. NULL pointer
71. Every AVL tree is a binary search tree.
a. True
b. False
c. Not in some cases
d. None of the given
72. In singly linked list "next" field of node contains:
a. Address of next node
b. Object of next node
c. Object of current node
d. Address of head node
73. Two common models of simulation are _____ and _____.
a. Circuit-based simulation and Event-based simulation
b. Circuit-based simulation and Time-based simulation
c. Time-based simulation and Event-based simulation
d. None of the above
74. Stack and Queue can be implements using _____.
a. Singly Link List
b. Binary Tree
c. Binary search Tree
d. AVL Tree
75. What are the basic things associated with data structures?
a. Space for each data item it stores
b. Time to perform each basic operation
c. Programming effort
d. All of the above
76. In AVL tree insertion occurs on the inside in case _____ and 3 which a single rotation cannot fix.
a. 1
b. 2
c. 4

AL-JUNAID TECH INSTITUTE

- d. 5
77. add(12) method of linked list class will
- a. Add 12 nodes in linked list
 - b. Add 12 pointers in linked list
 - c. Add 12 as value in linked list
 - d. Add 12 values in linked list
78. Which of the following is a non- linear data structure?
- a. Stack
 - b. Queue
 - c. Tree
 - d. Linked list
79. Which data structure allows deleting data elements from front and inserting at rear?
- a. Stacks
 - b. Queues
 - c. Deques
 - d. Binary search tree
80. Array cells are _____ in computer memory.
- a. Contiguous
 - b. Random
 - c. Store in multiple Variables
 - d. Store in multiple functions
81. A kind of expression where the operator is present between two operands called _____ expressions.
- a. Infix
 - b. Postfix
 - c. Prefix
 - d. None of the above
82. back() method of list class is used to
- a. Moves the "current" pointer to backward one element.
 - b. Moves the "current" pointer to backward two element.
 - c. Moves the "current" pointer to backward three element.
 - d. Moves the "current" pointer to backward four element.
83. next() method of List class is used to:
- a. Moves the Current position backward one element

AL-JUNAID TECH INSTITUTE

- b. Moves the "Current" pointer to two steps after the last element of the array
 - c. Moves the Current position forward one element
 - d. Moves the "Current" pointer to two steps before the last element of the array
84. _____ tree has been named after two persons Adelson-Velskii and Landis.
- a. Binary
 - b. Black
 - c. AVL
 - d. VLA
85. There are four cases of rotation in an _____ tree.
- a. ELV
 - b. EVL
 - c. AVL
 - d. ALV
86. _____ is used for reference variable in C++.
- a. !
 - b. @
 - c. #
 - d. &
87. A _____ is a tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- a. Strict binary tree
 - b. Full binary tree
 - c. Perfect binary tree
 - d. Complete binary tree
88. A queue is a data structure where elements are
- a. Inserted at the front and removed from the back
 - b. Inserted and removed from the top
 - c. Inserted at the back and removed from the front
 - d. Inserted and removed from both ends
89. Longest path from root node to farthest leaf node is called _____ of tree.
- a. Level
 - b. Length

AL-JUNAID TECH INSTITUTE

c. Depth

d. Node level

90. New items are added at the _____ of the stack.

a. Bottom

b. Middle

c. Center

d. Top

91. Length() method of list class is used to:

a. Return the length of the array

b. Return the length of the list

c. Return the length of empty part of the array

d. Return the length of empty part of the list

92. The function calls are made with the help of _____.

a. Stack

b. Heap

c. Dynamic memory

d. External memory

93. An efficient program execute faster and helps in _____ the usage of resources like memory and disk

a. Maximizing

b. Minimizing

c. Equalizing

d. None of the given

94. Generalized code written for a class is called:

a. Function

b. Template

c. Structure

d. Stack

95. "set()" method of list class is used to:

a. Set the value of Pointer

b. Set the value of Null Nodes

c. Set the value of objects

d. Set the value of Value

96. In C++, we place the class interface in ____ file.

a. .cpp

b. .cppp

AL-JUNAID TECH INSTITUTE

- c. .h
d. .hh
97. Every _____ tree is a binary search tree.
a. AVL
b. binary
c. big
d. small
98. If both left and right nodes of a node are NULL then this type of node is called _____ node.
a. Non leaf
b. internal
c. inner
d. leaf
99. Which of the following function don't belongs to the stack class?
a. push()
b. pop()
c. crash()
d. top()
100. For a complete binary tree, the depth is calculated as _____.
a. $\log_2(\text{number of nodes}+1)-1$
b. $\log_2(\text{number of nodes} \times 1)+1$
c. $\log_2(\text{number of nodes}-1)-1$
d. $\log_2(\text{number of nodes}-1)+1$
101. _____ only removes items in reverse order as they were entered
a. Queue
b. Stack
c. Both of these
d. None of these
102. What will be the postfix expression of following infix expression? $D+E \times F/G$
a. $DE \times F/G$
b. $DE+F \times G/$
c. $DEF \times /+$
d. $DE+FG \times /$
103. The _____ of a node in a binary tree is defined as the height of its _____ sub tree minus height of its right sub tree.
a. Height

AL-JUNAID TECH INSTITUTE

- b. Balance
 - c. Width
 - d. None
104. The back() method decreases the value of Variable current by ____
- a. Four
 - b. Three
 - c. Two
 - d. One
105. In doubly linked list there is/are ____ pointer/s in each node.
- a. One
 - b. Two
 - c. Three
 - d. Four
106. Suppose we have the following values to be inserted in constructing AVL tree,
10, 13, 15, 5, 7, 8
Tell when first rotation will take place.
- a. After inserting node 13
 - b. After inserting node 15
 - c. After inserting node 5
 - d. After inserting node 7
107. Binary search algorithm cannot be applied to ____
- a. Sorted linked list
 - b. sorted binary trees
 - c. sorted linear array
 - d. None of given option
108. During the execution of a process operating system constructs focus things for that process which of the following is not part of that process
- a. A section for static data including global variable
 - b. Stack
 - c. Heap
 - d. Linked list
109. Which one of the following method does not change the original value of the argument in the calling function?
- a. Call by passing reference of the argument

AL-JUNAID TECH INSTITUTE

- b. Call by passing the address of the argument
 - c. Call by passing the value of the argument
 - d. None of the given options
110. _____ is the major factor to see the efficiency of a program.
- a. Quality
 - b. Time
 - c. Correctness
 - d. None of the given
111. While implementing stack with an array and to achieve LIFO behavior, we used push and pop elements at _____.
- a. The start of the array
 - b. The end of the array
 - c. The mid of the array
 - d. At least one position before array starting index.
112. A template is a function or class that is written with a _____
- a. Specific
 - b. Definite
 - c. Generic
 - d. None of the above
113. In the linked list implementation of the stack class, where does the push member function places the new entry on the linked list?
- a. After all other entries that are greater than the new entry
 - b. At the head
 - c. After all other entries that are smaller than the new entry
 - d. At the tail
114. The next field in the last node in a singly-linked list is set to _____.
- a. 0
 - b. 1
 - c. NULL
 - d. False
115. Consider the linked list having data [6, 72, 35, 65, 25] stored in it. While current pointer is pointing memory location having 72 stored in it. After calling remove() function on the following linked list current point will point to memory location having value?

AL-JUNAID TECH INSTITUTE

- a. 6
 - b. 35
 - c. 65
 - d. 25
116. If we write functions for recursive and non recursive inorder traversal method of BST, what will be the difference between its functions prototypes?
- a. Different return types
 - b. Different function names
 - c. Different arguments list
 - d. Nothing will be different
117. In singly linked list a node comprises of _____ field/s.
- a. One
 - b. Two
 - c. Three
 - d. Four
118. Doubly linked list always has _____ NULL pointers in a node.
- a. One
 - b. Two
 - c. Three
 - d. Four
119. A BST generated from the data in ascending order is _____.
- a. Linear
 - b. Nonlinear
 - c. Balanced
 - d. Un sorted
120. Linked list is generally considered an example of _____ type of memory location.
- a. Static
 - b. Compile time
 - c. Dynamic
 - d. None of given options
121. What's wrong with following loop?
- ```
while((i < 10) && (i > 24)){
 }
}
```
- a. The logical operator && cannot be used in a test condition
  - b. The while loop is an exit-condition loop

# AL-JUNAID TECH INSTITUTE

- c. The test condition is always false  
d. The test condition is always true
122. Making the tree unbalanced, it violates the \_\_\_\_\_ rule.  
a. Linked list  
b. Stack  
c. AVL  
d. Queue
123. copy() method of list data structure \_\_\_\_\_  
a. copy first item of list  
b. set one list to be a copy of another  
c. copy last item of list  
d. copy any item of list
124. Deleting a \_\_\_\_\_ node in BST is a \_\_\_\_\_ case.  
a. Root, simplest  
b. Left child, simplest  
c. Right child, simplest  
d. Leaf, simplest
125. We can not remove items randomly from \_\_\_\_\_.  
a. Stack  
b. Queue  
c. Both of these  
d. None of these
126. NULL is an invalid address and \_\_\_\_\_.  
a. Accessible  
b. Inaccessible  
c. Points to the start point of the list  
d. Points to the last point of the list
127. A stack carries \_\_\_\_\_ behavior.  
a. FIFO  
b. LIFO  
c. AVCO  
d. FEFO
128. The tree data structure is a  
a. Linear data structure  
b. Non-linear data structure  
c. Graphical data structure

# AL-JUNAID TECH INSTITUTE

- d. Data structure like queue
129. Can we store elements with different data types in a single array?
- a. Yes
  - b. No
  - c. In some cases
  - d. None of given
130. The lifetime of a transient object can exceed that of application which is accessing it.
- a. True
  - b. False
  - c. In some cases
  - d. None of the given
131. In a complete binary tree, for 25000 nodes the depth will be \_\_\_\_.
- a. 13
  - b. 14
  - c. 15
  - d. 16
132. The smallest value element in a binary search tree(Each node with left and right pointer) lies at
- a. Root Node
  - b. Left Child of Root
  - c. Right Most Node
  - d. Left Most Node
133. if we use array to implement list, then there is an issue that is gives difficulty when
- a. We will access values randomly
  - b. We will remove data from it
  - c. We will increase its size
  - d. We will decrease its size
134. if there are \_\_\_\_ nodes in an avl tree its levels will be roughly as  $\log_2(10 \text{ million})$
- a. 100 million
  - b. 10 million
  - c. 5 million

# AL-JUNAID TECH INSTITUTE

d. 2 million

135. Which one is the correct function call for the following function of calculatincube?

```
int cube(int&num)
{
.
.
.
}
```

- a. Cube(&num)
- b. Cube(&&num)
- c. Cube(\*num)
- d. Cube(num)

136. Which of the following is the correct option for priority Queue?

- a. The type of Queue that is not FIFO i.e the person who comes first may not leave first
- b. The type of Queue that is not FIFO i.e the person who comes first should leave first
- c. The type of Queue that is not FIFO i.e the person who comes first should leave first
- d. The type of Queue that is not FIFO i.e the person who comes first may not leave first

137. For String-based Binary Search Tree, We use ASCII values of characters for comparing among letters. This method is known as

- a. Lexicographic order
- b. Alphabet coding procedure
- c. Asymmetric technique
- d. heap-based approach

138. Elements in a queue data structure are added from\_\_\_ and remove from\_\_\_ .

- a. Rear end. front end
- b. Front end. Rear end
- c. Front end
- d. rear end

139. If numbers 5,222,4,48 are inserted in a queue. Which one will be remove first?

# AL-JUNAID TECH INSTITUTE

- a. 48
  - b. 4
  - c. 222
  - d. 5
140. A zigzag rotation is performed. In Left-Left case of rotation in AVL tree.
- a. True
  - b. False
  - c. In some cases
  - d. None of the above
141. During deletion of node from BST. if we found this node don't have in- order successor and predecessor. it means this node is \_\_\_\_
- a. Left Most node in the binary search tree
  - b. Right most node in the binary search tree
  - c. Root node
  - d. None of the given option
142. \_\_\_\_ rule applies for evaluating operators of same precedence in an expression
- a. right to left
  - b. Cascading
  - c. Associative
  - d. None of the above
143. What will be result of following postfix expression?  
123\*+2-
- a. 3
  - b. 4
  - c. 5
  - d. 10
144. The main use of AVL tree is:
- a. Searching of data
  - b. Storing of data
  - c. Insertion of data
  - d. Security of data
145.  $y = \&x[0];$   
In the above statement, we get the address of the first location of the array x and store it in y.  
Here "y" is:

# AL-JUNAID TECH INSTITUTE

- a. rvalue
  - b. xvalue
  - c. lvalue
  - d. zvalue
146. what will be postfix expression of the following infix expression? Infix expression  $a+b*c-d$
- a.  $ab+c*d-$
  - b.  $abc*+d-$
  - c.  $abcd+*-$
  - d.  $abc+*d-$
147. Which of the following is the correct conversion of infix to postfix expression?  $Z+B-(D-H)/K$
- a.  $ZB+D-H-K/$
  - b.  $ZB+DH-K-/$
  - c.  $ZB+DH-K/-$
  - d.  $ZB+DHK--/$
148. Leaf node of binary search tree contains \_\_\_\_\_.
- a. One null pointer
  - b. Three null pointers
  - c. Two null pointers
  - d. All of the given
149. A tree is an AVL tree if
- a. Any one node fulfills the AVL condition
  - b. At least half of the nodes fulfill the AVL condition
  - c. All the nodes fulfill the AVL condition
  - d. None of the given options
150. The \_\_\_\_\_ of a binary tree is the maximum level of its leaves (also called the depth).
- a. Level
  - b. Width
  - c. Height
  - d. None of the above
151. Memory address is stored in
- a. Address operator
  - b. Reference
  - c. Pointer
  - d. All of the given

# AL-JUNAID TECH INSTITUTE

152. A binary tree is said to be a \_\_\_\_\_ binary tree if every non-leaf node in a binary tree has non-empty left and right sub trees.
- a. Complete
  - b. Strictly
  - c. AVL
  - d. Perfect
153. “+” is a \_\_\_\_\_ operator.
- a. Unary
  - b. Binary
  - c. Ternary
  - d. None of the above
154. Which of the following a nonlinear tree
- a. Stack
  - b. Que
  - c. Tree
  - d. Linked List
155. For every Process execute the last part of the Memory is for \_\_\_\_\_ of the Program
- a. Data
  - b. Code
  - c. Stack
  - d. Heap
156. While implement non recursive traversal for binary search tree we need to implement \_\_\_\_\_
- a. Que
  - b. Stack
  - c. Min heap
  - d. Max Heap
157. Suppose we have the following values to be inserted in constructing AVL tree,  
20,23,25,10, 12,13  
Tell when first rotation will take place.
- a. After increasing Node 25
  - b. After increasing Node 23
  - c. After increasing Node 10
  - d. After increasing Node 12

# AL-JUNAID TECH INSTITUTE

158. If a tree has a 50 Node then the total linked in the tree will be

- a. 55
- b. 51
- c. 50
- d. 49

159. The percolate down procedure will move the smaller value \_\_\_\_ and Bigger value \_\_\_\_

- a. Left, right
- b. Right, left
- c. Down, up
- d. Up, down

160. Circular Linked List Solve the problem of \_\_\_\_ pointer/method of the doubly link list.

- a) Remove
- b) Null
- c) Add
- d) Find

161. isEmpty() method of stack class will return true when.

- a) Stack is full
- b) Stack is Partially
- c) Stack is not empty or Null
- d) Stack is empty

162. "New int[11]" will allocate memory for \_\_\_\_ integers.

- a) 13
- b) 12
- c) 11
- d) 10

163. For searching a particular number in Binary Search Tree (if it is not present), the maximum number of comparisons will be \_\_\_\_\_ comparison at each level.

- a) 1
- b) 2
- c) 3
- d) 4

164. In singly linked list which node will keep track of starting position of the list.



# AL-JUNAID TECH INSTITUTE

a) Next Node

b) Previous Node

c) Head Node

d) Last Node

165. If we singly linked list to implement list, then there is an issue that it gives difficulty when we:

a) Move forward in the list

b) Move backward in the list

c) We will increase its size

d) We will decrease its size

166. Can we store elements with different data types in a single array?

a) Yes

b) No

c) In some cases

d) None of the given

167. During union by size method, which data structure is used to improve the balancing of tree?

a) Array

b) Stack

c) Linked List

d) Tree

168. The difference between a "Binary Tree (BT)" and a "Binary Search Tree (BST)" is that,

a) A BST has two children per node whereas a BT can have none, one or two children per node

b) In BST nodes are inserted based on the values they contain

c) In BT nodes are inserted based on the values they contain

d) There is no difference

169. The main use of AVL tree is:

a) Searching of data

b) Storing of data

c) Insertion of data

d) Security of data

e) The expression

170. if (! heap ->isEmpty() )

checks

a) Heap is empty

b) Heap is full

# AL-JUNAID TECH INSTITUTE

c) Heap is not empty

d) Not a valid expression

171. A solution is said to be efficient if it solves the problem within its resource constraints i.e. hardware and time.

a) True

b) False

172. A queue where the dequeue operation does not depend upon FIFO, is called:

a) enqueue

b) simple queue

c) stack

d) priority queue

173. In simple or singly linked list there is/are \_\_\_\_\_ pointer/s in each node.

a) One

b) Two

c) Three

d) Four

174. Which one of the following statement is correct?

a) Array size is fixed once it is created

b) Link List size is fixed once it is created

c) Binary Search Tree size is fixed once it is created

d) AVL Tree size is fixed once it is created

175. Which of the is NOT true regarding the maze generation?

a) Randomly remove walls until the entrance and exit cells are in same set.

b) Removing a wall is the same as doing a union operation

c) Remove a randomly chosen wall if the cells it separates are already in same set

d) Do not remove a randomly chosen wall if the cells it separates are already in same set.

176. If Ahmad is cousin of Ali and Ali is cousin of Asad then Ahmad is also cousin of Asad. This statement has the following property

a) Reflexivity

b) Symmetry

c) Transitivity

d) All of the given

177. Stack and Queue can be implemented using \_\_\_\_\_.

# AL-JUNAID TECH INSTITUTE

a) Singly Link List

b) Binary Tree

c) Binary Search Tree

d) AVL Tree

178. A Linear Data Structure is the data structure in which data elements are arranged in a sequence or a linear list. Which of the following is Non Linear Data Structure?

a) Arrays

b) Linked Lists

c) Binary Search Trees

d) Stack

179. Recursive call of a function use \_\_\_\_\_ data structure.

a) Linked List

b) Queue

c) Stack

d) Table

180. The \_\_\_\_\_ of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

a) Height

b) Balance

c) Width

d) None of the above

181. Which operation of queue data structure is used to get front element from the queue and then remove it from the queue?

a) enqueue()

b) dequeue()

c) front()

d) remove()

182. Suppose we have been given the following data set for a Queue.

7 5 2 4

What will be the resultant Queue if we call enqueue(3) method?

Note that 7 is the front element whereas 4 is rear element of queue.

a) 7 5 2 4

b) 7 5 2 4

c) 7 5 2 4 3

d) 5 2 4 3

183. In C++, we place the class interface in \_\_\_\_\_ file.

a) .cpp

b) .cppp

# AL-JUNAID TECH INSTITUTE

c) .h

d) .hh

184. The array in binary search is sub divided \_\_\_\_\_.

a) Once

b) Twice

c) N time

d) **Untill a sublist is no more divisible**

185. If unions are done by weight (size), the depth of any element is never greater than

a)  $\log 3n$

b)  **$\log_2 n$**

c)  $n \log_2 n$

d)  $\log n * n$

186. \_\_\_\_\_ in AVL is logarithmic.

a) Updating

b) **Searching**

c) Deletion

d) Insertion

187. What is the depth of any tree if the union operation is performed by height?

a)  $O(N)$

b)  $O(N \log N)$

c)  **$O(\log N)$**

d)  $O(M \log N)$

188. Avl tree takes maximum \_\_\_\_\_ time to search an element.

a)  **$1.44 \log_2 n$**

b)  $\log_2(n+n)$

c)  $\log_2(n+1)+1$

d)  $1.88 \log_2 n$

189. Which of the following is correct about AVL Tree?

a) It is identical to BST except height of the left and right subtrees can differ by at least 1.

b) It is identical to BST except height of the left and right subtrees must differ by at least 1.

c) It is not identical to BST, it is totally different kind of tree.

d) **It is identical to BST except height of the left and right subtrees can differ by at most 1.**

190. "new int[11]" will allocate memory for \_\_\_\_\_ integers.

# AL-JUNAID TECH INSTITUTE

a) 10

b) 11

c) 12

d) 13

191. The \_\_\_\_\_ method of list data structure removes the element residing at the current position

a) Add

b) Next

c) Remove

d) Find

192. While implementing non-recursive traversal for Binary SearchTree, we need to implement \_\_\_\_\_.

a) Queue

b) Stack

c) Min heap

d) Max heap

193. Doubly Linked List always has \_\_\_\_\_ NULL pointer/s in a node.

a) One

b) Two

c) Three

d) Four

194. For a perfect binary tree of height  $h$ , having  $N$  nodes, the sum of heights of nodes is \_\_\_\_\_.

a)  $N - h - 1$

b)  $N - 1$

c)  $N - 1 + h$

d)  $N - (h - 1)$

195. If there are  $N$  elements in an array then the number of maximum steps needed to find an element using Binary Search is \_\_\_\_\_.

a)  $N$

b)  $N^2$

c)  $N \log_2 N$

d)  $\log_2 N$

196. Here is a small function definition:

```
void f(int i, int &k)
{
 i = 1;
```

# AL-JUNAID TECH INSTITUTE

k = 2;

}

Suppose that a main program has two integer variables x and y, which are given the value 0. Then the main program calls f(x,y); What are the values of x and y after the function f finishes?

- a) Both x and y are still 0.
- b) x is now 1, but y is still 0.
- c) x is still 0, but y is now 2.
- d) x is now 1, and y is now 2.

197. The worst case of searching in binary search tree (BST) is:

- a) When the data inserted in BST is sorted
- b) When the height of left sub-tree is greater than right sub-tree
- c) When the height of right sub-tree is greater than left sub-tree
- d) When the tree is balanced

198. What will be postfix expression of the following infix expression?

Infix Expression:  $a+b*c-d$

- a)  $ab+c*d-$
- b)  $abc*+d-$
- c)  $abcd+*-$
- d)  $abc+*d-$

199. What is the hash function used in linear probing?

- a)  $h_i(x) = \text{hash}(x) \bmod \text{table size}$
- b)  $h_i(x) = (\text{hash}(x) + f(i^2)) \bmod \text{table size}$
- c)  $h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{table size}$
- d)  $h_i(x) = X \bmod 17$

200. Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s)

- a) One
- b) Two
- c) Three
- d) Four

201. In the statement  $\text{int} \& a = b;$

- a) a and b pointing to two different memory location
- b) a and b are two different names of the same memory location
- c) a and b are two different variable names
- d) b hold the address of variable a

202. In 1990, Bill pugh proposed an enhancement on linked lists and the new data structure was termed as

# AL-JUNAID TECH INSTITUTE

- a) Linked list
- b) B-Tree
- c) Skip list
- d) Spelling checker

203. Which of the following statement is NOT correct regarding Table ADT?

- a) In a table, the type of information in columns may be different.
- b) A table consists of several columns known as entities
- c) The row of a table is called a record
- d) A major use of table is in databases where we build and use tables for keeping information

204. A binary tree of N nodes has \_\_\_\_\_ .

- a)  $\log_{10} N$  levels
- b)  $\log_2 N$  levels
- c)  $N / 2$  levels
- d)  $N \times 2$  levels

205. If the height of a perfect binary tree is 4. What will be the total number of nodes in it?

- a) 15
- b) 16
- c) 31
- d) 32

206. Which property of equivalence relation is satisfied if we say:  
Ahmad R(is related to) Ahmad

- a) Reflexivity
- b) Symmetry
- c) Transitivity
- d) All of the given

207. If ahmad is boss of ehsan and ehsan is boss of umer then ahmad is also boss of umer. The above mentioned relation is \_\_\_\_\_ .

- a) Reflexive
- b) Symmetry
- c) Transitive
- d) None of the given

208. Which of the following statement is false?

- a) Arrays are dense lists and static data structure
- b) data elements in linked list need not be stored in adjacent space in memory

# AL-JUNAID TECH INSTITUTE

- c) pointers store the next data element of a list
- d) linked lists are collection of the nodes that contain information part and next pointer

209. Binary search algorithm cannot be applied to \_\_\_\_\_.

- a) Sorted linked list
- b) sorted binary trees
- c) sorted linear array
- d) None of the given

210. Suppose there is an image segmented into pixels. Each pixel has \_\_\_\_\_ neighbour(s).

- a) 0
- b) 4
- c) 8
- d) 16

211. In singly linked list a node consists of two parts:

- a) Object and structure
- b) Two pointers
- c) Two objects
- d) Object and pointer

212. A hash function returns a \_\_\_\_\_ value.

- a) Integer
- b) Double
- c) Float
- d) Char

213. In a tree, we link the nodes in such a way that it \_\_\_\_\_ a linear structure.

- a) does not remain
- b) forms
- c) reverses
- d) remains

214. The principal benefit of a link list over a conventional array is that the order of the linked items may be \_\_\_\_\_ from the order that the data seems are stored in memory.

- a) Different
- b) Identical
- c) Same
- d) Equivalent

215. The Computer memory can be thought of as a/an.

- a) List



# AL-JUNAID TECH INSTITUTE

- b) Queue
  - c) Stack
  - d) Array
216. Before using the Pop method of a stack, the user must call the \_\_\_\_ method.
- a) isFull()
  - b) push()
  - c) pop()
  - d) isEmpty()
217. A stack carries \_\_\_\_ behavior.
- a) FIFO
  - b) FEFO
  - c) LIFO
  - d) AVCO
218. \_\_\_\_ Method returns the top element of the stack without removing it.
- a) Pop()
  - b) Front()
  - c) Push()
  - d) Top()
219. In Queue data structure element are removed from \_\_\_\_.
- a) Pop
  - b) Push
  - c) Rear
  - d) Front
220. In Queue data structure element are full from \_\_\_\_.
- a) Pop
  - b) Push
  - c) Rear
  - d) Front
221. \_\_\_\_\_
222. “end()” method of list performs its tasks in \_\_\_\_\_
- a) Many Steps
  - b) Three Steps
  - c) One Steps
  - d) Two Steps
223. Linked List use \_\_\_\_ to store data.
- a) Array
  - b) Variables

# AL-JUNAID TECH INSTITUTE

- c) Linked Memory
  - d) 2-D Array
224. Complete the push method code of stack void push (int x) {  
A{\_\_\_\_}=x}
- a) Count++
  - b) Count—
  - c) ++count
  - d) –count
225. A template is a Function or Class that is Written a \_\_\_\_ data type.
- a) Generic
  - b) None of These
  - c) Define
  - d) Specific
226. Trying to remove an element from an empty Stack is called \_\_\_\_
- a) Garbage Collection
  - b) Overflow stack
  - c) Empty collection
  - d) Underflow of Stack
227. Stack and Queue can be implemented using \_\_\_\_
- a) Binary Tree
  - b) Singly Linked List
  - c) AVL Tree
  - d) Binary Search Tree
228. A software solution is said to be efficient if it solves the problem.
- a) With its resource constraints
  - b) With No Time
  - c) By Consuming more hardware resources.
  - d) By Using some extra resources.
229. we cannot remove items randomly from \_\_\_\_
- a) Stack
  - b) Queue
  - c) Stack and Queue
  - d) List
230. Stack push(15) will push 15 on \_\_\_\_
- a) Top of the Stack
  - b) Anywhere of the Stack
  - c) Middle of the Stack
  - d) Bottom of the Stack

# AL-JUNAID TECH INSTITUTE

231. Which of the following is the correct conversion from infix to postfix expression  $A*B+C/(E-F)$ .

- a)  $ABC*+EF- /$
- b)  $AB*CFE- / *$
- c)  $AB*C+EF / *$
- d)  $AB+C*E-F /$

232. Convert the Given infix from  $12+60-23$  of expression in postfix form.

- a)  $12+60\ 23-$
- b)  $12\ 60+ -23$
- c)  $-12\ 60 -23$
- d) None

233.  $Y=\&x[0];$

In the above statement, we get the address of the Location of the array  $x$  and store it in  $y$ . Here “ $Y$ ” is;

- a) Xvalue
- b) Zvalue
- c) Ivalue
- d) Nvalue

234. Last Node in Circular Linked list Contains.

- a) Maximum Two points
- b) No Null Pointer
- c) Atleast one Null pointer
- d) None

235. Consider the linked list having data  $[6, 72, 35, 65, 25]$  stored in it. While current pointer is pointing memory location having 72 stored in it. After calling  $\text{add}(4)$  function on the following linked list current point will point to memory location having value?

- a) 36
- b) 4
- c) 72
- d) 25

236. there is no such Node whose next field is NULL which one of the given option support the statement

- a) Linked List
- b) Circular Link list
- c) Array
- d) Queue



CS301- Data Structures  
LATEST SOLVED SUBJECTIVES  
FROM MIDTERM PAPERS

DEC 09,2011

MC100401285

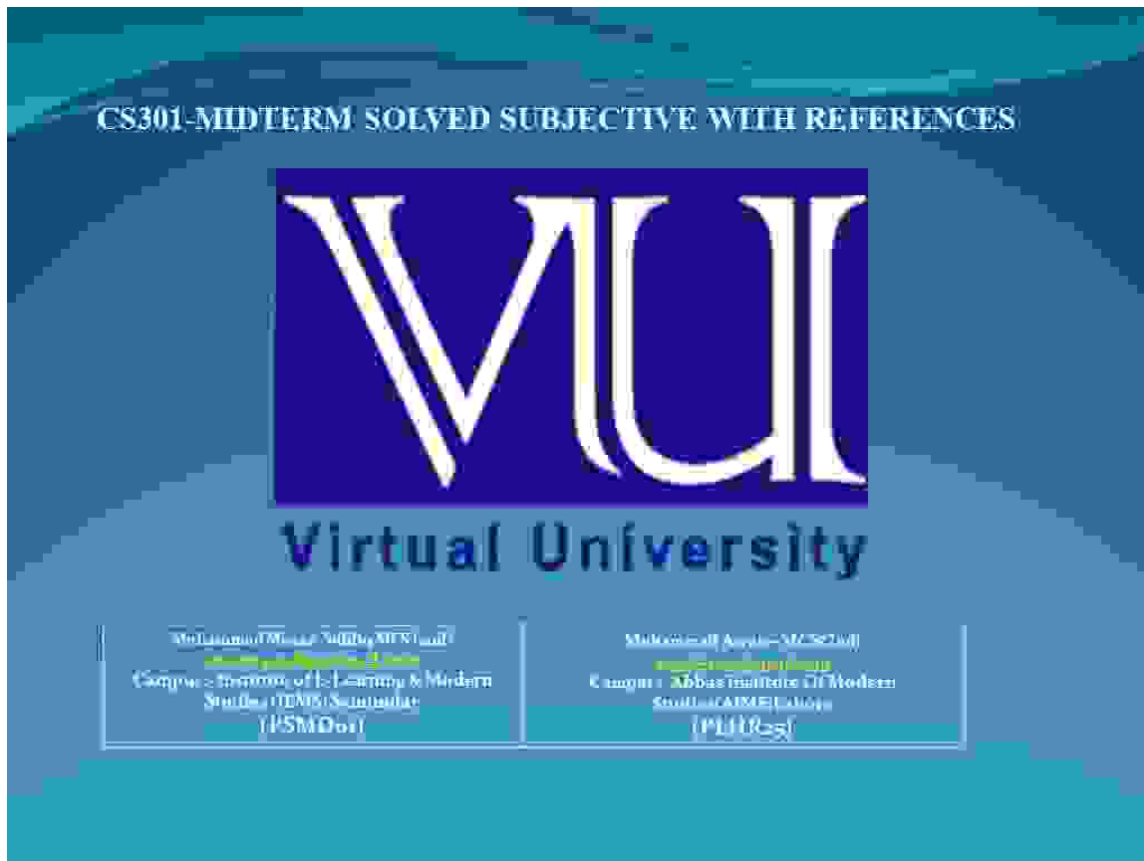
[Moaaaz.pk@gmail.com](mailto:Moaaaz.pk@gmail.com) [Mc100401285@gmail.com](mailto:Mc100401285@gmail.com)

PSMD01

MC110200030

[awais4you@gmail.com](mailto:awais4you@gmail.com)

PLHR25



## Cs301- current midterm paper-1(NOV 2011)

**Q:1 What is complete binary tree?**

**Answer:**

"A complete binary tree is a binary tree with the additional property that every node must have exactly two children if an internal node, and zero children if a leaf node."

**Q:2 How single left rotation is performed in AVL tree?**

**Answer:**

This rotation is almost similar to the single right rotation. We start from tree node k1. k1 is the root while node k2 is its right child. Due to the left rotation, the right child k2 of the root k1 will become the root. The k1 will go down to the left child of k2.

**Q:3 describe the following**

- (i) Height of tree
- (ii) Balance of Node

**(i) Height of tree**

**Answer: (Page 209)**

The height of a binary tree is the maximum level of its leaves (also called the depth).

**(ii) Balance of Node**

**Answer: (page 215)**

The balance of a node in a binary search tree is defined as the height of its left subtree minus height of its right subtree. In other words, at a particular node, the difference in heights of its left and right subtree gives the balance of the node.

## Cs301- current midterm paper-2(NOV 2011)

**Q1 what is binary tree brief it?**

**Answer:-**

"A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right sub-trees".

**Q2.what is different between binary tree and AVL tree**

**Answer:- (Page 212)**

An AVL tree is identical to a BST, barring the following possible differences:

- ❖ Height of the left and right subtrees may differ by at most 1.
- ❖ Height of an empty tree is defined to be (-1).

**Why queue is linear data structure.**

**Answer:-(Page 87)**

A queue is a linear data structure into which items can only be inserted at one end and removed from the other.

In contrast to the stack, which is a LIFO (Last In First Out) structure, a queue is a FIFO (First In First Out) structure.

**How we can delete a node with two Childs in a binary search tree using its right sub tree.**

**Answer:- Page 263**

The node to be deleted has either left child (subtree) or right child (subtree). In this case we bypass the node in such a way that we find the inorder successor of this node and then link the parent of the node to be deleted to this successor node. Thus the node was deleted.

**define const keyword, reference variable, Dangling reference**

**Answer: (Page 198)**

**Const keyword**

The const keyword is used for something to be constant. The actual meanings depend on where it occurs but it generally means something is to held constant. There can be constant functions, constant variables or parameters etc.

**Reference variable**

The references are pointers internally, actually they are constant pointers. You cannot perform any kind of arithmetic manipulation with references that you normally do with pointers.

**Dangling reference**

The pointer of the object (when object has already been deallocated or released) is called dangling pointer.

### **Cs301- current midterm paper-3(NOV 2011)**

**Q: 1 what is complete binary tree?**

**Answer: (Page 123)**

The definition of the complete binary tree is

A complete binary tree of depth d is the strictly binary tree all of whose leaves are level d.

**Q: 2 how single left rotation is performed in AVL tree?**

**Answer: rep**

**Q:3 describe the following**

**(i) Height of tree**

**(ii) Balance of Node**

**Answer: rep**

### **Cs301- current midterm paper-4(NOV 2011)**

**Q1 what is binary tree breif it?**

**Answer:- rep**

**Q2.what is different between binary tree and AVL tree**

**Answer: rep**

**Why queue is linear data structure.**

**Answer: rep**

## MIDTERM EXAMINATION 2011 (MAY)

### 1. What is function of length () method in the Queue

**Answer: (page 111)**

In Queue length() method, has a single statement i.e.

return size ;

This method returns the size of the queue, reflecting the number of elements in the queue. It is not the size of the array used internally to store the elements of the queue.

### 2. Explain the two cases in which we apply double rotation in An Avl tree

**Answer: (Page 229)**

Sometimes a single rotation is not sufficient to balance an unbalanced tree.

The two cases are following:

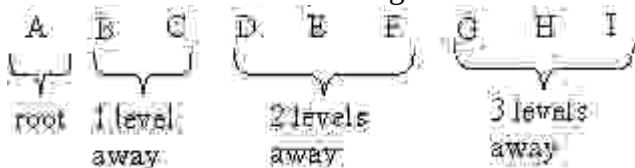
1. Insertion into the right subtree of the left child of X node (RL)
2. Insertion into the left subtree of the right child of X node (LR)

### 3. How can we perform level-order traversal on a tree?

**Answer:- (For More detail and code see Page 161 )**

We can print a binary tree level by level by employing recursive or non-recursive method.

The idea of a level-order traversal is to visit the root, then visit all nodes "1 level away" (depth 2) from the root (left to right), then all nodes "2 levels away" (depth 3) from the root, etc. For the example tree, the goal is to visit the nodes in the following order:



A level-order traversal requires using a queue (rather than a recursive algorithm, which implicitly uses a stack). A queue is a FIFO structure, which can make the level-order traversal easier.

### 4. How can the dangling reference problem be avoided?

**Answer:- ( Page 200 )**

To avoid dangling reference, don't return the reference of a local variable (transient) from a function.

## MIDTERM EXAMINATION 2011 (MAY)

**Q:1 What is complete binary tree?**

**Answer: rep**

**Q:2 How single left rotation is performed in AVL tree?**

**Answer: rep**

**What is binary tree breif it?**

**Answer:- rep**

**Why queue is linear data structure.**

**Answer:-** rep

**Q: Define const keyword, reference variable , Dangling reference**

**Answer: rep**

**Q:3 Describe the following**

**(i) Height of tree**

**Answer: rep**

**Q.21A: - Similarity between AVL and Binary Search Tree?**

**Answer:-** rep

**Q.21B: - How AVL is different from Binary Search Tree?**

**Answer:-** (<http://answers.yahoo.com/question/index?qid=20071213000314AAU8p08>)

A binary tree is a tree data structure in which each node has at most two children. Typically the child nodes are called left and right. One common use of binary trees is binary search trees; another is binary heaps.

While an AVL tree is a self-balancing binary search tree.

In an AVL tree the heights of the two child sub-trees of any node differ by at most one, therefore it is also called height-balanced. Lookup, insertion, and deletion all take  $O(\log n)$  time in both the average and worst cases. Additions and deletions may require the tree to be rebalanced by one or more tree rotations.

**Q.22: - What the maximum no of comparisons we have to perform during insertion in BST?**

**Answer:-** ( Page 139 )

If we have a complete binary tree with  $n$  numbers of nodes, the depth  $d$  of the tree can be found by the following equation:

$$d = \log_2(n + 1) - 1$$

If the tree is complete binary or near-to-complete, we need  $\log_2(n)$  number of comparison. Whereas in a linked list, the comparisons required could be a maximum of  $n$ .

## MIDTERM EXAMINATION 2010

**Question No: 19 ( Marks: 2 )**

**What is Function Call Stack Give short answer?**

**Answer:-**

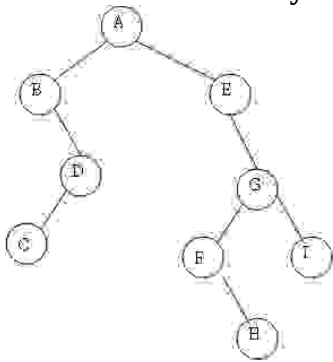
Stacks play a key role in implementation of function calls in programming languages.

In C++, for example, the “call stack” is used to pass function arguments and receive return values.

The call stack is also used for “local variables”



**Question No: 21 ( Marks: 3 )**  
**Here is a small binary tree.**



**Write the order of the nodes visited in**

- A Post-order traversal : (L, R, N)
- A level-order traversal

**Answer:-**

Post-order traversal:: CDBHFIGEA

level-order traversal:: ABEDGCFIH

## MIDTERM EXAMINATION 2010

**Question No: 18 ( Marks: 2 )**

**What are the applications of Binary Tree?**

**Answer:-**

The binary tree is a useful data structure for rapidly storing sorted data and rapidly retrieving stored data.

It can hold objects that are sorted by their keys.

It can represent a structured object, An operating system maintains a disk's file system as a tree, where file folders act as tree nodes:

## MIDTERM EXAMINATION 2010

**Question No: 17 ( Marks: 1 )**

**Give the names of basic Queue Operations**

**Answer:- (page 88)**

**Queue Operations**

The queue data structure supports the following operations:

**Operation Description**

- |            |                                                  |
|------------|--------------------------------------------------|
| enqueue(X) | Place X at the <i>rear</i> of the queue.         |
| dequeue()  | Remove the <i>front</i> element and return it.   |
| front()    | Return <i>front</i> element without removing it. |
| isEmpty()  | Return TRUE if queue is empty, FALSE otherwise   |

**Question No: 18 ( Marks: 1 )**

**Give one benefit of using Stack.**

**Answer:- (Page 53)**

In computer science, a stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. the data structure itself enforces the proper order of calls.

**Question No: 21 ( Marks: 5 )**

**Give short answers of the following questions:**

**1. Why List wastes less memory as compared to Arrays.**

**Answer:-**

1. Linked lists do not need contiguous blocks of memory; extremely large data sets stored in an array might not be able to fit in memory.
2. Linked list storage does not need to be preallocated (again, due to arrays needing contiguous memory blocks).
3. Inserting or removing an element into a linked list requires one data update, inserting or removing an element into an array requires n (all elements after the modified index need to be shifted).

Array is a collection of same data type. In linked list there are two field one is address and other is pointer. In array elements are arranged in a specific order

**2. Why we can change the size of list after its creation when we can not do that in simple arrays?**

**Answer:-**

Some how the answer will be same as part 1 because Inserting or removing an element into a linked list requires one data update, inserting or removing an element into an array requires n (all elements after the modified index need to be shifted).

Array is a collection of same data type. The size of array is mentioned with its declaration. in arrays the elements are in contiguous position. One must after another. While in linked list we gave the address of next element in the next part of node.

## **MIDTERM EXAMINATION 2010**

**Question No: 19 ( Marks: 2 )**

**Describe any two uses of priority queues?**

**Answer:- (page 101)**

Priority queue being used at many places especially in the operating systems. In operating systems, we have queue of different processes. If some process comes with higher priority, it will be processed first. Here we have seen a variation of queue. We will use the priority queue in the simulation. The events will be inserted in the queue and the event going to occur first in future, will be popped.

**Question No: 21****Describe the Use of Queues?****Answer:- (Page 96)****Use of Queues:**

Out of the numerous uses of the queues, one of the most useful is *simulation*. A simulation program attempts to model a real-world phenomenon. Many popular video games are simulations, e.g., SimCity, Flight Simulator etc. Each object and action in the simulation has a counterpart in the real world. Computer simulation is very powerful tool and it is used in different high tech industries, especially in engineering projects. For example, it is used in aero plane manufacturing. Actually Computer Simulation is full-fledged subject of Computer Science and contains very complex Mathematics, sometimes. For example, simulation of computer networks, traffic networks etc.

**Question No: 22 ( Marks: 3 )****How we evaluate postfix expressions?****Answer: (Page 67)****Evaluating Postfix Expression:**

- Each operator in a postfix expression refers to the previous two operands.
- Each time we read an operand, we push it on a stack.
- When we reach an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack.

**MIDTERM EXAMINATION 2010****Question No: 18 ( Marks: 1 )****What is meant by an empty stack?****Answer: (Page 54)**

Empty stack means that there is no element on the stack. IsEmpty() boolean function returns true if stack is empty, false otherwise.

**Question No: 19 ( Marks: 2 )****Is the following statement correct? If your answer is No, then correct it.****“A tree is an AVL tree if at least half of the nodes fulfill the AVL condition”****Answer: (page 214)**

Statement is wrong, Correct statement is:

A tree is an AVL tree if all the nodes fulfill the AVL condition.

**MIDTERM EXAMINATION 2010****Question No: 17 ( Marks: 2 )****Which process places data at the back of the queue?****Answer: (page 87)**

“ enqueue(X) “ Place X at the rear/back of the queue.

**Question No: 18 ( Marks: 2 )**

**How we can delete a node with two Childs in a binary search tree using its right sub tree.**

**Answer:- (Page 263)**

The node to be deleted has either left child (sub-tree) or right child (sub-tree). In this case we bypass the node in such a way that we find the in-order successor of this node and then link the parent of the node to be deleted to this successor node. Thus the node was deleted.

**Question No: 19 ( Marks: 2 )**

**Why we use Reference Variables. Give one example.**

**Answer:- [Click here for detail](#)**

C++ references allow you to create a second name for the a variable that you can use to read or modify the original data stored in that variable.

```
#include
```

```
main()
```

```
{
```

```
int x;
```

```
int& foo = x;
```

```
// foo is now a reference to x so this sets x to 56
```

```
foo = 56;
```

```
std::cout << x <<std::endl;
```

```
}
```

**17. Define Reference Variable, Dangling Reference & Const**

**Answer:-**

In the C++ programming language, a **reference** is a simple reference datatype that is less powerful but safer than the pointer type inherited from C. The name C++ reference may cause confusion, as in computer science a reference is a general concept datatype, with pointers and C++ references being specific reference datatype implementations Dangling Reference & Const

**Dangling pointers** and **wild pointers** in computer encoding are pointers that do not point to a valid object of the suitable type. These are special cases of violations of memory safety

**18. What is the use of Reference Variable, Give example? 3 Marks**

**Answer:-**

A reference variable is used to refer a particular object Location which resides on a garbage collectable heap. c++ supports one more type of variable called reference variable. In addition to pointer variable and value variable of reference variable behaves similar to both vaule variable and pointer variable. Reference variable has all access that variable has.

**Question No: 18 ( Marks: 2 )**

**How we can avoid the problem of Dangling reference**

**Answer:-**

To avoid dangling reference, don't return the reference of a local variable (Transient) from a function.

**CS301 Updated 250+ MCqs**

**Lec 1 - 20 Spring 2022**

**Quiz 1 and Quiz 2**

**#Midterm Preparation**

**REMEMBER OWN PRAYER**

**Orange Monkey Team**

**Important NOTE:-**

**Like share and comment too ..ager koi**  
**mistake hogaye ho to sorry and comment**  
**main lazmi mention krdijiye ga takey main**  
**correction krlon file main .... And like**  
**zaroor krein is hamein or bhe apsab**  
**keyliye files makes krny ke himat milti hey**  
**...JAZAKALLAH**

---

**Orange Monkey Team**



1. In which traversal method root node is visited at left first subtree?

Inorder.....confirm

### In-order Traversal

In this traversal method the left subtree is visited first

2. In which traversal method root node is visited at first step?

Preorder.....confirm

### Pre-order Traversal

In this traversal method, the root node is visited first

3. In which traversal method root node is visited at last step?

Postorder.....confirm

### Post-order Traversal

In this traversal method the root node is visited last

4. In level-order traversal for binary Search Tree \_\_\_\_\_ data structure is used.

Queue....confirm

In level order traversal of Binary Search Tree, \_\_\_\_\_ data structure is used.

Queue

5. The statement the insertion operation in avl tree generally takes more time than insertion operation in simple binary search tree

Always correct....confirm

The insertions into AVL trees make take more time than in BST when random elements are inserted, but that is quite a minute difference. The AVL trees has more practical applications than in BST because of its balancing property and thus it can be used in real-world applications.

6. \_\_\_\_\_ is used for reference variables in C++

&....confirm

The symbol & is used for reference variable.

7. The first statement in the given code is called as:

```
Class BinarySearchTree;
Class BinaryNode{
Friend class BinarySearchTree;
};
```

Forward declaration

8. Which of the following traversal method traverses the binary tree in sorted order?

Inorder....confirm

3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20 This sorted order is only in the inorder traversal.

9. The \_\_\_\_\_ symbol is used when we want to get the value of a variable using pointer.

\*.....confirm

By using \* operator we can access the value of a variable through a pointer .

10. The \_\_\_\_\_ of every node should be 1,0 or -1 otherwise it will not be an AVL tree.

Balance....confirm

means that the balance of every node should be 1,0 or -1. Otherwise it will not be an AVL tree.

11. The \_\_\_\_ tree has been named after two person Adelson-Velskii and Landis

AVL ....confirm

### AVL Tree

AVL tree is named after two persons Adelson-Velskii and Landis.

12. What will be output of the given C++ code?

```
Char lhs[] = "Army";
Char rhs[] = "Army";
Cout << strcmp(lhs, rhs);
}
```

0.....confirm

C:\Users\Daniyal\Desktop\Untitled1.exe

```
Process exited after 0:00:00.005 with return value 0
Press any key to continue.
```

13. \_\_\_\_\_ of an empty AVL tree is defined to be -1

Height....confirm

14. Linked list use \_\_\_\_\_ to store data.'

Variables.....confirm



11. Linked list is a data structure that stores data in a linear fashion. It is a sequence of nodes, each containing a data element and a pointer to the next node. It is a type of linear data structure. It is a type of linear data structure. It is a type of linear data structure.

15. AVL stands for

Adelson-Velskii and Landis....confirm

**AVL Tree**

AVL tree is named after two persons Adelson-Velskii and Landis.

16. The balance of a node in a binary tree is defined as the height of its \_\_\_\_ sub tree minus height of its right sub tree.

Left....confirm

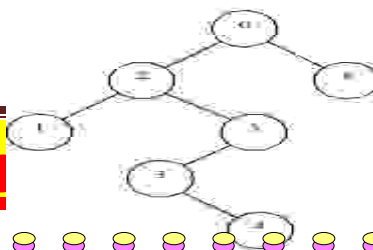
The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

17. The \_\_\_\_ sign before the name of the variable means that the address of the variable is being passed.

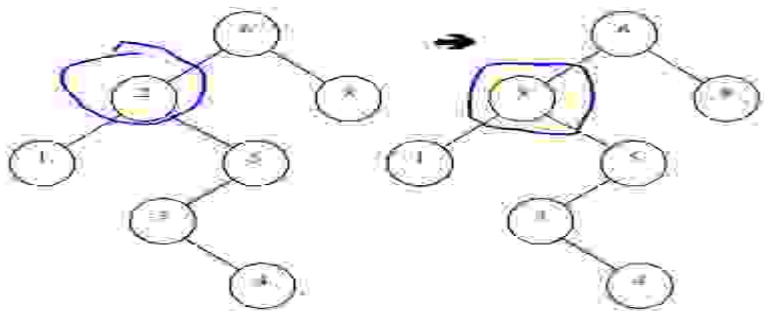
&.....confirm

The '&' sign before the name of the variable means that the address of the variable is being passed. The called function will take it as a pointer. Remember that

18. If we delete node 2 from the given BST then which node will replace it?



3....confirm



19. During in order traversal using recursive calls if we found a node is NULL it means this node will satisfy following condition.

It will not have left child....confirm

20. A tree is an AVL tree if

All the nodes fulfill the AVL condition....confirm



21. Which of the following calling method does not change the original value of variable passed to a function on argument?

Call by passing as value....confirm

Which one of the following calling methods does not change the original value of the argument in the calling function?

- ▶ None of the given options
- ▶ Call by passing the value of the argument

22. Which of the following data structure is/are linear type?

Heap and stack

23. Which of the following is correct syntax to pass a constant variable by reference to a function?

Int display (const int& val)

The const keyword appears before a function parameter

int movePiece(const Piece & entirePiece)

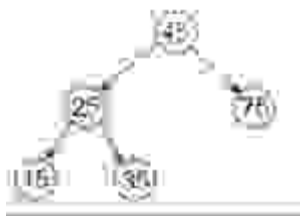
24. In C++ we place the class interface in \_\_\_\_\_ file,

.h.....confirm

25. If we write functions for recursive and non recursive inorder traversal method of BST, what will be the difference between its functions prototypes?

Nothing will be different.....confirm

26. What will be the preorder traversal of the given tree?



45,25,15,35,75....confirm

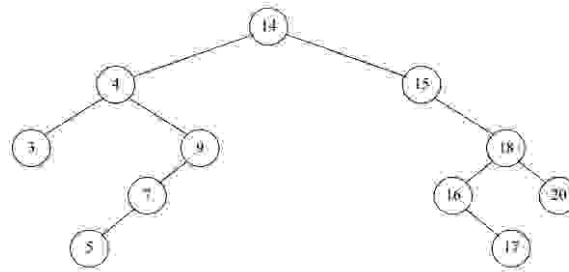


Fig 13.3: Preorder: 14 4 3 9 7 6 15 18 16 17 20

27. Binary Search Tree violates the condition of AVL tree when any node has balance factor equal to

0.....confirm

necessarily. Rather, the tree will be called an AVL tree if the balance factor of every node in a tree is 0, 1 or -1. By the way, if the balance factor of each node inside the tree is 0, it will be perfectly balanced tree.

28. Binary Search Tree violates the condition of AVL tree when any node has balance factor equal to

1 or -1....confirm

29. After deletion of a node from a binary search tree \_\_\_\_\_ traversal method should be maintained.

Inorder

In this delete case, we need to find the **inorder** successor of the node to be deleted which is the minimum value in the right subtree. Then we will put the value of this node to the node to be deleted. This node is saved in the variable **minNode**. The node to be deleted is now pointed by **tree**. Then we get the value of the **inorder** successor, stored in the **minNode** and assign it to the node to be deleted. Now we have to delete the **minNode**. For this purpose, we call the **remove** method recursively providing it the right subtree and the value of the **minNode**. We have also changed the value of the current node and can also send this value to the **remove** method. For clarity purposes, we provide it the **minNode** value. The node returned by the **remove** method is stored

Page 173 of 505

in the temporary variable **t**. Then we assign **t** to the right subtree of **tree**. We do it due to the fact that the tree has to re-adjusted after the deletion of **minNode** in the right subtree.

30. Link list can be special case of

Binary search tree

31. To create a reference variable we need to use \_\_\_\_\_ sign.

Ampersand.....confirm

The symbol **&** used for reference variable has a different purpose and is used in its occurrence in the code. In C++ programming, we have seen that when the **ampersand** sign i.e. **&** appears in front of a variable name, it is the address operator. It

32. At a particular node the difference in heights of its left and right subtree gives the \_\_\_\_\_ of the node.

Balance.....confirm

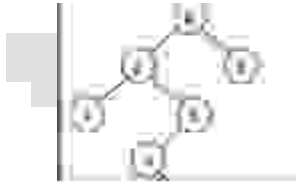
minus height of its right subtree. In other words, at a particular node, the difference in heights of its **left and right subtree** gives the balance of the node.

33. For string-based binary search tree, we use ASCII values of characters for comparing among letters. This method is known as \_\_\_\_\_.

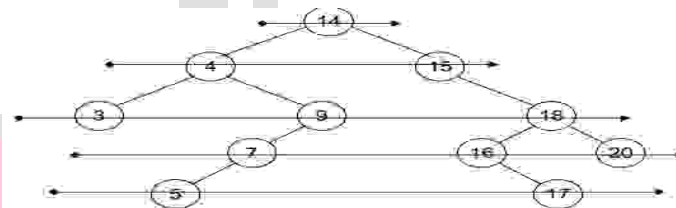
Lexicographic order....confirm

will happen in case of strings. You must remember, every character of a string has an associated ASCII value. We can make a lexicographic order of characters based on their ASCII values. For example, the ASCII value of B (66) is greater than A (65), therefore, character B is greater than character A. Similarly if a word is starting with the letter A, it is smaller than the words starting from B or any other character up to Z. This is called lexicographic order. C++ provides us overloading facility to define the same operators (<, >, <=, >=, ==).

34. What will be level order traversal of the given BST?



6,2,8,1,5,3,4.....confirm



level order: 14 4 15 3 9 18 7 16 20 5 17

Fig 14.4: Level order traversal

35. \_\_\_\_\_ parameter passing ( by value or by reference) is similar to PASCAL.

C++.....confirm

In PASCAL you can pass a parameter by value or by reference like C++

36. We can make a lexicographic order of characters based on their \_\_\_\_\_.

ASCII values.....confirm

associated ASCII value. We can make a lexicographic order of characters based on their ASCII values. For example, the ASCII value of B (66) is greater than A (65), therefore, character B is greater than character A. Similarly if a word is starting with the letter A, it is smaller than the words starting from B or any other character up to Z.



37. If we return the reference of a local variable from a function it will cause.

Dangling reference....confirm

To avoid **dangling reference**, don't return the reference of a local variable (transient) from a function.

38. Which one is the correct function call for the following function of calculating cube?

```
Int cube(int& num)
```

```
{
.
.
.
}
```

Cube(num)

39. Whenever a node is deleted from a binary search tree \_\_\_\_\_ has to be maintained.

Inorder traversal....confirm

Fig 15.10: Deleting a Node From BST

If we want to delete the node containing number 4 then we have to adjust the right subtree pointer in the node containing value 2 to the *inorder* successor of 4. The important point is that the **inorder traversal order has to be maintained** after the delete

40. For a complete binary tree with n numbers of nodes the depth is calculated as \_\_\_\_\_.

$\log_2(n+1)-1$ ....confirm

binary tree. It was witnessed that if we have a complete binary tree with  $n$  numbers of nodes, the depth  $d$  of the tree can be found by the following equation:

$$d = \log_2(n+1) - 1$$

41. What will be the return type of the findMin method in the statement given below?

```
Int& findMin()const;
```

Integer reference

42. Which of the following is TRUE about recursive calls?

recursive calls always contains a terminating condition....confirm

There is always a terminating condition in recursive call of recursive procedure.

43. Which of the following is correct syntax to declare a const function?

Const int& findMin();

44. A node in AVL tree can become imbalanced due to.

Both insertion and deletion operation....confirm  
computation should be understood as it is used for the insertion and deletion of nodes in an AVL tree. We may come across a situation, when the tree does not remain

45. We can use binary search tree with \_\_\_\_\_.

Both integer and string data....confirm

46. Following is a keyword of C++ \_\_\_\_\_.

Delete....confirm

delete is a keyword of C++

47. In a strictly complete binary tree the number of \_ at any level k will be  $2^k$ .

Nodes....confirm

the number of nodes at any level k will be  $2^k$

48. A hash function returns a \_\_\_ value

Integer....confirm

A hash function returns a \_\_\_\_\_ value.

Integer

49. Following is true in case of using recursive method calls.



The code becomes very short....confirm

50. Suppose we have the following values to be inserted in constructing AVL tree, 10,13,15,5,7,8 tell when first rotation will take place

After inserting node 15....confirm

SUPPOSE WE HAVE THE FOLLOWING VALUES TO BE INSERTED IN CONSTRUCTING AVL TREE  
10,13,15,5,7,8  
Tell when first rotation will take place  
After inserting the node 15

51. Suppose we have the following values to be inserted in constructing AVL tree, 20,23,25,10,12,13 tell when first rotation will take place

After inserting node 25.....confirm

CONSTRUCTING AVL TREE  
20,23,25,10,12,13  
Tell when first rotation will take place  
After inserting node 25

52. If the root of a tree is at level three in case of a complete binary tree i.e nodes will be at \_\_\_\_\_.

8....confirm

there will be 4 nodes in case of a complete binary tree. Similarly at level 3, the number of nodes will be 8 and so on. As discussed earlier, in a complete binary tree,

53. If the root of a tree is at level two in case of a complete binary tree i.e nodes will be at \_\_\_\_\_.

Level 4....confirm

tree is at level zero, its two children (subtrees) i.e. nodes will be at level 1. At level 2, there will be 4 nodes in case of a complete binary tree. Similarly at level 3, the

54. If the root of a tree is at level zero its two children subtree i.e nodes will be at \_\_\_\_\_.

Level 1....confirm

tree is at level zero, its two children (subtrees) i.e. nodes will be at level 1.

55. In Preorder traversal of a binary tree the second step is \_\_\_\_\_.

Traverse the left subtree....confirm



56. In a complete binary tree for 25000 nodes the depth will be \_\_\_\_\_.

14.....confirm



57. When a node to be deleted from a BST has both left and right child after deletion its \_\_\_\_\_ node will replace this node.

inorder Successor....confirm

Case III: The node to be deleted has both the left and right children (subtree). This is the most difficult case. In this case we find the inorder successor of the node. The left most node of the right subtree will be the inorder successor of it. We put the value of that inorder successor node into the node to be deleted. After it we delete the inorder successor node recursively.

58. In tree, the search operation is \_\_\_\_\_ as compared to the linked list.

Very fast....confirm



59. Factorial is an example of \_\_\_\_\_ function

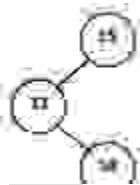
Recursive.....confirm

60. Function signatures are also called:

Function prototype....confirm

The function signature is also called the function prototype

61. Why following tree is not AVL tree?



Because \_\_\_\_\_.

The balance factor of root node is two

62. A BST generated from the data in ascending order is \_\_\_\_\_.

Linear....confirm

A BST generated from the data in ascending order is Linear

63. From operating system point of view the recursive function calls are made with the help of \_\_\_\_\_.

Stack....confirm

64. In which traversal method the recursive calls can be used to traverse a binary tree?

In preorder, postorder and inorder traversal only....confirm

at each level before proceeding to the next level. At each level, we visit the nodes in a left-to-right order. The traversal methods discussed earlier are inorder, preorder and post order. We print the node or to the left subtree or to the right subtree. In the

65. The functions calls are made with the help of \_\_\_\_\_.

Stack.....confirm

function calls are made with the help of stacks

66. What will be the output of following code?

```
Void explode(int& num){
 Num++;}
Main(){
 Int power = 15;
 explode(power);
 cout<<power;
}
```

16

67. What will be the output of following code?

```
Main(){
 Int x=15;
 Int &y;
 Y=x;
 Y=15;
 Cout<<x;
}
```

Compile time error.....confirm

```
1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5 int x = 15;
6 int &y;
7 y = x;
8 y = 15;
9 cout<<x;
10 }
```

68. We allocate memory dynamically by using \_\_\_\_\_ operator.

New....confirm

When we allocate memory dynamically by using new operator

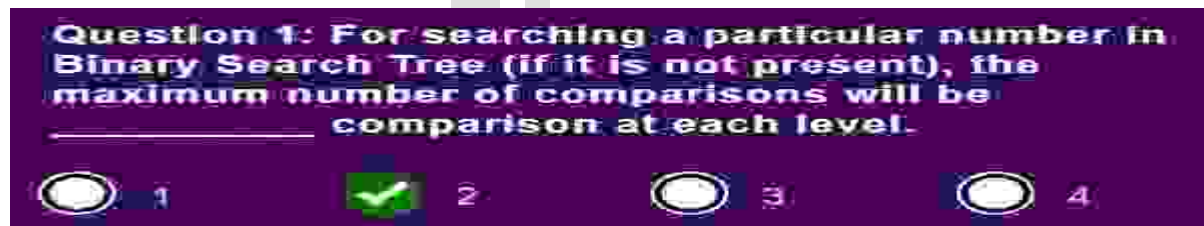
69. When a function calls another the parameters and return address of the calling function are put in \_\_\_\_\_.

Stack....confirm

We know that function calls are made with the help of stacks. When a function calls some other function, the parameters and return address of the function is put in a stack. The local variables of the function are also located at the stack and the control

70. For searching a particular number in binary search tree (if it is not present) the maximum number of comparisons will be \_\_\_\_\_ comparisons(s) at each level.

2....confirm



71. In which of the following function signature the value of variable num cannot be changed in function body?

Int cube(const int& num).....confirm

71. In which of the following function signatures, the value of variable 'num' cannot be changed in function body?

int cube(const int& num)

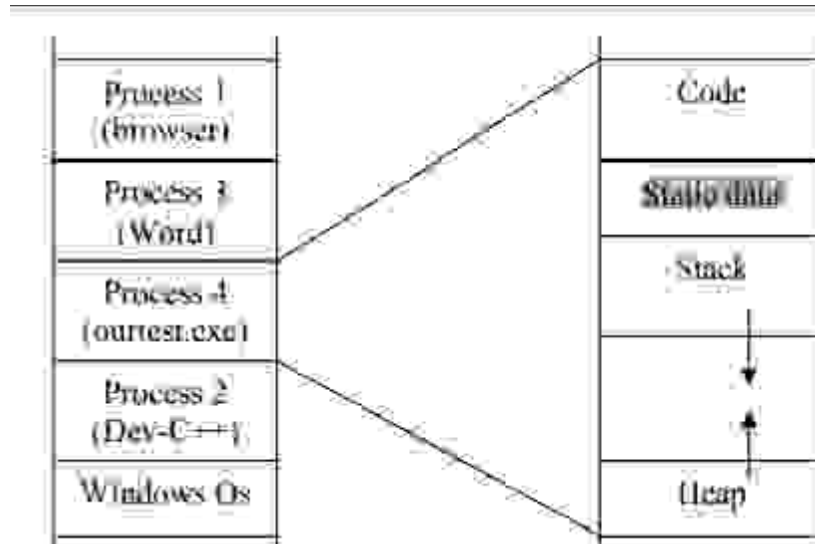
72. \_\_\_\_\_ is a self balancing tree.

AVL.....confirm

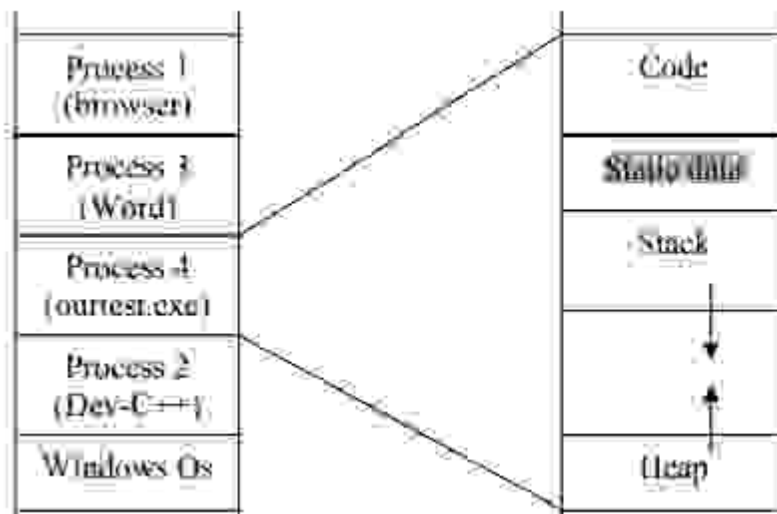
AVL - An AVL tree is a self-balancing binary search tree.

73. In the perspective of memory organization every process executing the first part of the memory is for \_\_\_\_\_ of the program.

Code....confirm

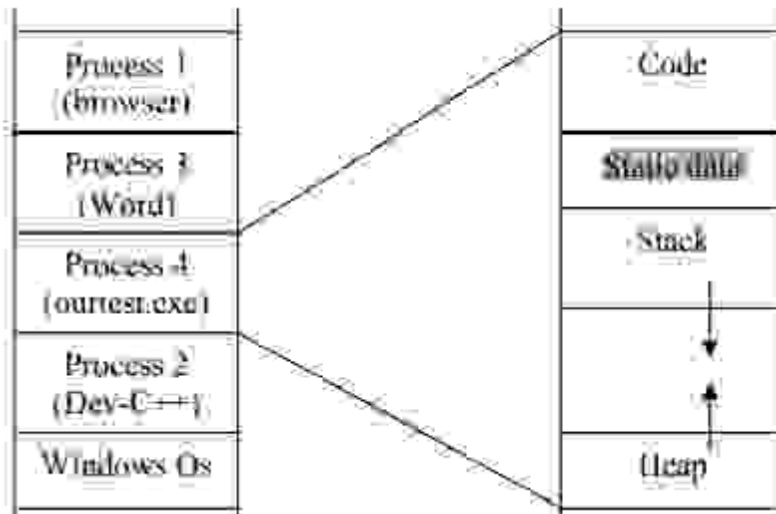


74. In the perspective of memory organization every process executing the third part of the memory is for \_\_\_\_\_ of the program.  
Stack.....confirm



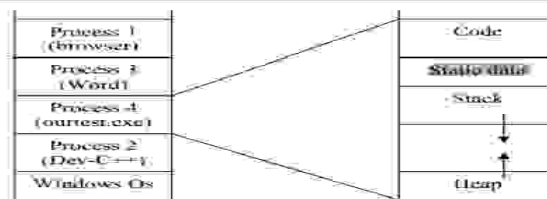
75. In the perspective of memory organization every process executing the last part of the memory is for the \_\_\_\_\_ of the program.  
Heap....confirm





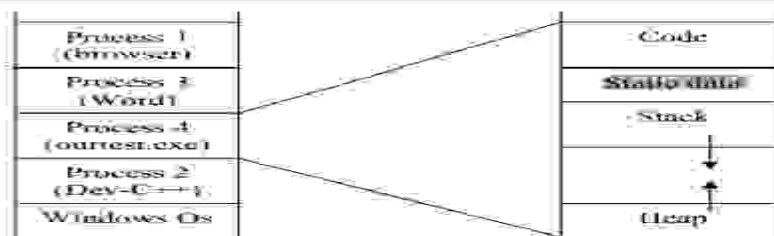
76. In the perspective of memory organization each process is divided into \_\_\_\_\_ sections.

4



77. In the perspective of memory organization every process executing the second part of the memory is for \_\_\_\_\_ of the program.

Static code



78. The structure of tree with \_\_\_\_\_ data is like a linked list

Sorted.....confirm

will be better to recap things talked about in the previous lecture. We built a balanced search tree (BST) with sorted data. The numbers put in that tree were in increasing sorted order. The tree built in this way was like a linked list. It was witnessed that the

**79. Which one of the following case is the very simple to delete a node from BST?**

The node to be deleted is the leaf node i.e it has no right or left child....confirm

The node to be deleted is the leaf node i.e it has no right or left child. It is very simple to delete such node. We make the pointer in the parent node pointing to this node as

**80. Which one of the following case is the most complicated case to delete a node from BST?**

Node to be deleted has both the left and right children.....confirm

Case III: The node to be deleted has both the left and right children (subtree). This is the most difficult case. Let's recap this case. The following figure shows the tree and

**81. After performing which task on the list, implementing array has to shift the entire remaining elements one place to the left.**

By removing element at start.....confirm

elements on the right of this empty space one place to the left. If we remove an element from the beginning of the list, then we have to shift the entire remaining elements one place to the left. Suppose there is a large number of elements, say 10000 or 20000 in the list. We remove the first element from the list. Now to fill this space:

**82. In array list the worst case of removing an element is.**

To remove an element from the end of the list.....confirm



In array list the (Raise For Success) worst case of removing an element is:

- To remove an element from the end of the list Correct

83. End() method of list performs it's tasks in

One step....confirm

The other methods back(), start() and end() also perform their tasks only in one step

84. Start() method of list class is used to :

Moves the "current" pointer to very first element

85. In singly linked list "next" field of node contains

Address of the next node....confirm

IN A SINGLY LINKED LIST, EACH NODE CONTAINS A FIELD THAT POINTS TO THE ADDRESS OF NEXT NODE

86. If we do not have to move too much in the list then use of which one type of list is not necessary/essential

Circular linked list.....confirm

advantages depend on its use. If we do not have to move too much in the list and have no problem checking the NULL, there is little need a circularly-linked list. But this facility is available to us.

87. Null is an invalid address and \_\_\_\_.

Inaccessible....confirm

memory address: NULL in the last node: NULL is an invalid address and is inaccessible

88. Suppose a stack class has been defined using template. Now, we want to declare a stack object of an int type. What will be the correct syntax.

Stack<int> stack....confirm

In the above code, consider the line

Stack<int> mystack

This line means that while creating an object *mystack* of Stack, the generic data type is

89. Which off the following is not a form of expression?

Pastfix....confirm

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions. It is easiest to demonstrate the differences by looking at examples of

90. The back() method decreases the value of variable current by

One....confirm

The back() method decreases the value of variable current by 1

91. In the linked list implementation of the stack class, where does the push member function places the new entry on the linked list?

At the head.....confirm

In the linked list implementation of the stack class, where does the push member function places the new entry on the linked list?

At the head

92. Whenever we call a function, the compiler makes a \_\_\_\_ that it uses to fulfill this function call.

Stack.....confirm

Whenever we call a function, the compiler makes a stack that it uses to fulfill this function call. The compiler puts the entries on the stack in the way

93. Which of the following is not a Data structure?

Memory cell .....confirm

94. STL is a \_\_\_\_ that is a part of the official standard of C++.

Library.....confirm

STL is an important tool, pre-developed for us. It is available as a library. Different data structures like stack, queue etc is also there in STL. We can

95. Compiler uses which one of the following in functions call.

Stack....confirm

Question 1: Compiler uses which one of the following in Function calls \_\_\_\_\_.



Stack



Queue



Binary  
Search Tree



AVL Tree

96. Which one option is associated with the data structures?

Time to perform each basic operation.....confirm

Time to perform each basic operation.....confirm

- Time to perform each basic operation.....confirm
- Time to perform each basic operation.....confirm
- Time to perform each basic operation.....confirm

97. Which of the following line of code is incorrect?

Float \*I = float new

98. Which kind of data structure is used to evaluate postfix expression?

Stack.....confirm

We are going to evaluate the postfix expression with the help of stack.

99. We cannot remove items randomly from \_\_\_\_\_.

Stack and queue.....confirm

125 We can not remove items randomly from

A) Stack

B) Queue

C) Both of these

100. Length() method of list class is used to \_\_\_\_.

Return the length of the list.....confirm

\$length() that simply returns the size of the list

101. A template is a function or class that is written with a \_\_\_\_\_ data type

Generic.....confirm

In C++ language, a template is a function or class that is written with a generic data type. When a programmer uses this function or class, the generic data type is replaced.

102. Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

One...confirm

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

1. One

103. The principal benefit of a linked list over a conventional array is that the order of the linked items may be  
**Same...confirm**

The principal benefit of a linked list over a conventional array is that the order of the linked items may be the same. In an array, the order of the items is fixed. In a linked list, the order of the items is determined by the pointers that connect them.

104. isEmpty() method of stack class will return true when  
**Stack is empty....confirm**

The isEmpty() method of the Stack class returns true if the stack is empty and false otherwise. This is a boolean function that returns true if there is no element in the stack.

105. End() method of list performs its task in  
**One step.....confirm**

The End() method of the list class performs its task in one step. It removes the last element from the list.

106. In queue data structure elements are removed from \_\_\_\_\_.  
**Front....confirm**

Queues are data structures that follow the First in First Out (FIFO) principle. The first element that is added to the queue is the first one to be removed. Elements are always added to the back and removed from the front. Think of it as a line of people waiting for a bus.

107. A software solution is said to be efficient if it solves the problem  
**Within its resource constraints....confirm**

A solution is said to be efficient if it solves the problem within the required resource constraints. Examples of resource constraints include the total space available to store it.

108. Generalized code written for a class is called.

### Template....confirm

nodes, a binary tree is created. This class is a template class.

109. New int[11] will allocate memory for \_\_\_\_\_ integers.

### 11...confirm

"new int[11]" will allocate memory for \_\_\_\_\_ integers.

1. 10
2. 11

110. At the time while evaluating an expression, if we want to give higher precedence to lower precedence operator, we used.

### Parenthesis.....confirm

parenthesis in the math, especially at the times when we want to give a higher precedence to an operator of lower precedence. For example, if there is a  $+$  operator

111. "+" is a \_\_\_\_\_ operator.

### Binary.....confirm

Binary operator who hoty hein jo 2 operand key sath use hoty hon ..like  $A + B$ ..idher A or B operand hein or  $+$  operator hey.. or Uniary symbol just one operand key sath use hoty hein like  $(- -a)$  idher  $- -$  ye operator hey or a operand hein

112. The stack implementing an array causes worst case behavior when the insertion and deletion of an element done from

### Beginning of an array.....confirm

that the stack is internally implemented with the help of array. The worst case for insertion and deletion from an array may happen when we insert and delete from the beginning of the array. We have to shift elements to the right for insertion and left for

113. Each node in singly linked list contains \_\_\_\_\_.

### One pointer....confirm



Each node in singly link list has,

1, 1 pointer

114.  $2*(7+8)-6$  is an example of \_\_\_ expression

Infix.....confirm

how can we convert the infix expression  $A = (B * C)$

115. If we use doubly linked list to implement list then there is an issue of;

Pre pointer of first node and next pointer of last node are

NULL....confirm

pointers. we can move forward and backward very easily. Now we face another problem that the prev pointer of first node and the next pointer of the last node are NULL. Therefore, we have to be careful in case of NULL pointers. To remove the

116. Allocating and de-allocating memory for linked list nodes does take \_ time than pre-allocated array.

More.....confirm

Allocating and de-allocating memory for list nodes does take more time than pre-allocated array. Memory allocation and de-allocation has cost in terms of time

117. Circular linked list always has \_ NULL pointer/s in a node.

0.....confirm

A circular list is a list that does not contain any pointer pointing to NULL

118. The type of expression in which operator succeeds its operands is \_\_\_\_\_ expression.

Postfix.....confirm

## postfix expression

Explanation: The expression in which operator succeeds its operands is called postfix expression.

119. Before using the push method of stack the user must call the \_\_\_\_\_ method.

isFull(); ....confirm

Before using the push method of stack, the user must call the \_\_\_\_\_ method.

- isEmpty();
- isFull(); Correct

120. What will be postfix expression of following infix expression?

$4 * 1 \uparrow 7 + 2$

41  $\uparrow$  7\*2+.....confirm

(1)  $4 * 1 \uparrow 7 + 2$

Infix to Postfix

Postfix: 41 7\*2+=

121. If the input symbol is not ')' and the stack is not empty then which of the following option will be pushed on the stack.

Operator.....confirm

If the input symbol is not ')' and the stack is not empty, we will push the operator on the stack. Otherwise, it is advisable to pop the stack and discard the '('. The following

122. There is no such node whose next field is NULL, which one of the given option supports the statement .

Circular link list.....confirm



123. Trying to remove an element from an empty stuck is called \_\_\_\_\_.

Underflow of stack....confirm

Trying to remove an element from an empty stack is called \_\_\_\_\_.

1. Garbage collector
2. Overflow of stack
3. Empty collection
4. Underflow of stack

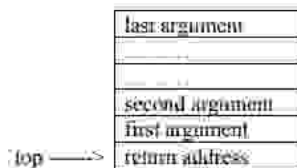
124. In doubly linked list a node consists of three parts:

2 pointers and 1 object...confirm

The doubly linked list consists of three parts: data, previous pointer, and next pointer.

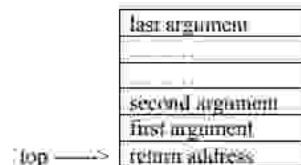
125. Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function

return address...confirm



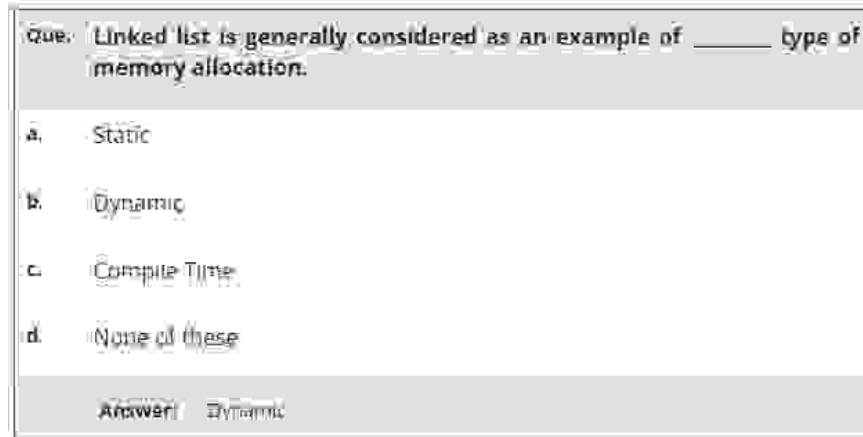
126. Whenever we call a function, the compiler makes a stack, the last element of the stack is \_\_\_\_\_ of the function

last argument.....confirm



127. Linked list is generally considered an example of \_\_\_\_\_ type of memory location.

Dynamic....confirm

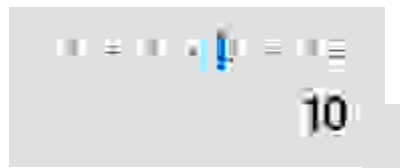


128. In singly linked list “next” field of node contains  
Address of the next node.....confirm



129. After evaluating this expression what will be the result?  
 $7+3*3/(7-4)$

10....confirm



130. In the calling function after the execution of the function called the program continues its execution from the \_\_ after the function call.

Next line....confirm



131. Consider the linked list implementation of a stack. Which of the following node is considered as top of the stack?  
First node.....confirm

Consider infix to postfix conversion which is the following node is considered as top of the stack?

- A Middle Node
- B Last Node
- C First Node
- D Any Node

132. “+” is a \_\_\_\_\_ operator

Binary.....confirm

#### Precedence of Operators

The order in which the operations are performed in an expression is called the precedence of operators. The operators are given a priority in the order of execution. The operators with higher priority are executed first. The operators with lower priority are executed last.

133. Convert the given infix form  $12 + 60 - 23$  if expression in postfix form.

$12\ 60 + 23 -$  .....confirm

Infix:  $12 + 60 - 23$

Infix to Postfix

Postfix:  $12\ 60 + 23 -$

134. Which of the following is the correct conversation from infix to postfix expression?

$A * B + C / (E - F)$

$AB * CEF - / +$ .....confirm

Infix :  $A * B - C / E - F$

Infix to Postfix

Postfix :  $AB * C / EF - =$

Step by

135. Stack.push(15) will push 15 on \_\_\_\_\_.

Top of stack....confirm

Stack: 15

Stack: 15

136. In simple or singly linked list there is/are \_\_\_ pointer/s in each node.

One.....confirm

In simple or singly linked list there is/are  
\_\_\_\_\_ pointer/s in each node.

1. One

137. Which of the following statement is correct for the variable “current—“?

Current = current - - ....confirm

current = current - 1

138. Circular linked list solve the problem of \_\_\_\_\_ pointers/method of the doubly link list.

Add....confirm

When you solve the problem by using the add, next, length, next, last, and get methods of the class List.

139. Which one of the following is not a one-step method in list implementation using array?

### Find()....confirm

The other methods: `get()`, `length()` etc are **one-step** methods. They carry out their operation in one instruction. There is no need of any loop or other programming structures to perform the task. The `get()` method gets the value from the specified position just in one step. Similarly the `update()` method sets a value at the specific position just in one step. The `length()` method returns the value of the size of the list. The other methods `back()`, `start()` and `end()` also perform their tasks only in one step.

140. Local variables of a function are stored in,

Stack.....confirm

One should be careful about transient objects that are stored by reference in data structures. We know that the local variables of a function are created in call stack. Those variables

141. A stack carries \_\_\_\_\_ behavior.

LIFO....confirm

Primarily, there are two operations of a stack: `push()` and `pop()`. A **stack** follows **LIFO** (Last In First Out) principle.

142. The `--` is a decrement operator in C++ that decreases the value of the operand by \_\_\_\_\_.

One....confirm

The `--` is a decrement operator in C++ that **decreases the value** of the operand by one. (The above statement is also the definition)

143. In singly linked list which node will keep track of starting position of the list.

Next node.....confirm

In singly linked list which node **will keep** track of starting position of the list.

1. **Next Node**

144. Consider the linked list having data [6, 72, 35, 65, 25] stored in it. While current pointer is pointing to memory location having 72 stored in it. After calling add (4) function on the following linked list current point will point to memory location having value?

4...confirm

Consider the linked list having data [6, 72, 35, 65, 25] stored in it. While current pointer is pointing to memory location having 72 stored in it. After calling add(4) function on the following linked list current point will point to memory location having value?

1. 72
2. 35
3. 4

145. Consider the linked list having data [6, 72, 35, 65, 25] stored in it. While current pointer is pointing to memory location having 72 stored in it. After calling remove () function on the following linked list current point will point to memory location having value?

6....confirm



146. Consider the following expression:  
 $P = A * B + C - D$   
In the above expression “-“ is a \_\_\_\_\_ operator

Binary

147.  $Y = \&x[0];$

In the above statement, we get the address of the first location of the array x and store it in y. Here “y” is

Lvalue...confirm

```
x = &x[0];
```

In the above statement, we get the address of the first location of the array x and store it in y. &x is *lvalue* so it can be used on left hand side. This means that the above statement is correct.

148. The first step to add a node in a linked list between the existing nodes is

`newNode->setnext(currentNode->getNext());`

The first step (Raise For Success) to add a node in a linked list between the existing nodes is:

- `newNode->setNext();`
- `newNode->setNext(currentNode->getNext());` Correct

149. Josephus problem is resolved by the implementation of \_\_\_\_\_.

Circular linked list....confirm

Now we will see an example where Josephus problem is implemented. This is a problem in which n people are standing in a circle and they would like to choose a leader. The way

150. In node class one field is an integer data and other field will be \_\_\_\_.

Pointer to node....confirm

151. Clear() of a list is used to \_\_\_\_\_.

Clear a list(remove all elements)...confirm

```
clear()
```

152. createList() of a list is used to \_\_\_\_\_.

Create a new list (presumably empty)....confirm

```
createList()
```



153. `copy()` of a list is used to \_\_\_\_\_.

Set one list to be a copy of another....confirm

|                     |                                      |
|---------------------|--------------------------------------|
| <code>copy()</code> | Set one list to be a copy of another |
|---------------------|--------------------------------------|

154. `insert(X,?)` of a list is used to \_\_\_\_\_.

Insert element X at a particular position in the list....confirm

|                          |                                                       |
|--------------------------|-------------------------------------------------------|
| <code>insert(X,?)</code> | Insert element X at a particular position in the list |
|--------------------------|-------------------------------------------------------|

155. `remove(?)` of a list is used to \_\_\_\_\_.

Remove element at some position in the list...confirm

|                        |                                             |
|------------------------|---------------------------------------------|
| <code>remove(?)</code> | Remove element at some position in the list |
|------------------------|---------------------------------------------|

156. `get(?)` of a list is used to \_\_\_\_\_.

Get element at a given position...confirm

|                     |                                 |
|---------------------|---------------------------------|
| <code>get(?)</code> | Get element at a given position |
|---------------------|---------------------------------|

157. `update(X,?)` of a list is used to \_\_\_\_\_.

Replace the element at a given position with X...confirm

|                          |                                              |
|--------------------------|----------------------------------------------|
| <code>update(X,?)</code> | Replace the element at given position with X |
|--------------------------|----------------------------------------------|

158. `find(X)` of a list is used to \_\_\_\_\_.

Determine if the element X is in the list...confirm

|                      |                                       |
|----------------------|---------------------------------------|
| <code>find(X)</code> | Determine if element X is in the list |
|----------------------|---------------------------------------|

159. `length()` method of a list class is used to \_\_\_\_\_.



Returns the length of the list...confirm

`length()` Returns the length of the list.

160. First element of an int array can be manipulated by using the index zero as:

`Int[0];`

To manipulate the first element, we will use the index 0. `arr[0]` will give the first element in the array.

161. Which of the following is known as Last-In, First-Out or Lifo data structure?

Stack...confirm

Which one of the following is known as "Last-In, First-Out" or LIFO Data Structure?

1. Linked List
2. Stack

162. The computer memory can be thought of as a/an

Array ....confirm

The computer memory is thought of as an array.

163. In doubly linked list there is/are \_\_\_\_\_ pointer/s in each node.

Two.....confirm

In doubly linked list there are two pointers in each node.

One

Two

164. Which one option is correct to know the first element of the linked list

Head pointer....confirm

165. While implementing stack with an array and to achieve LIFO behavior, we used push and pop elements at

The end of the array.....confirm

You know that while implementing stack with an array and to achieve *lifo* behavior we used **push** and **pop** elements at the end of the array. Instead of pushing and

166. Each node in a singly linked list contains two fields, one field called data field while other field contains.

Pointer to node.....confirm

Que: Each Node contain minimum two fields one field called data field to store data. Another field is of type \_\_\_\_\_.

a. Pointer to Class  
b. Pointer to an Integer  
c. Pointer to Character  
d. Pointer to Node

ANSWER: **Pointer to Node**

167. What will be postfix expression of following infix expression?  $4 * 1 / 7 + 2$

$41/7*2+.....$ confirm

$(4 * 1) / 7 + 2$

**Infix to Postfix**

Postfix:  $41/7*2+$

168. When add() operation of a linked list is called the following action is done.

A new node is made....confrim

We can create a new node in the following manner in the **add()** operation of the linked list.

169. What will be the postfix expression of following infix expression?

$D + F * F / G$

DEF\*G/+.....confirm

What will be the postfix expression of following infix expression?  
 $D + E * F / G$

1. DE\*F/G+

2. DE+F\*G/

3. DEF\*G/+

170. To create a \_\_\_\_\_ we link the last node with the first node in the list.

Circularly-linked list....confirm

NULL pointer is given to the first node and also to the last node. If we connect the pointer of last node to the first node, then it will create a circularly-linked list.

171. Array cells are \_\_\_\_\_ in computer memory.

Contiguous

array cells are  
contiguous in  
computer memory

172. Stack can be implemented by which of the following data structure?

Array or Linked list....confirm

Stack can be implemented using array and linked list

173. Add(12) method of linked list class will;

Add 12 as value in linked list.....confirm

add(12) method of linked list class will  
add 12 as value in linked list

174. Current pointer always points to the \_\_\_\_\_ of the linked list.

First node....confirm

Current pointer always points to the first node of the linked list. This is because the current pointer is initialized to the address of the first node at the start of the program. If we don't initialize it, it will not be possible to know the starting point of the linked list.

175. \_\_\_\_\_ is the major factor to see the efficiency of a program.

Time.....confirm

Time is the major factor to see the efficiency of a program. A program is considered efficient if it runs quickly and uses less memory.

176. An efficient program executes faster and helps in \_\_\_\_\_ the usage of resources like memory and disk.

Minimize....confirm

An efficient program executes faster and helps in minimizing the usage of resources like memory and disk. This is because an efficient program is designed to use resources wisely and avoid unnecessary operations.

177. A list is the collection of items of the \_\_\_\_\_.

Same type.....confirm

A list is a collection of items of the same type. It is a data structure that stores a sequence of elements.

178. Consider the following push operations of a Stack

Stack push (4);

Stack push (6);

Stack push (5);

Stack push (8);

If a user calls a pop() operation then which value will be returned?

8...confirm

179. When an executable program runs, it is loaded in the computer memory and becomes a \_\_\_\_\_.

Process.....confirm

memory and **current** process

180. Head node pointer always points to the \_\_\_\_\_ of the linked list?

**Current node**

anti-clockwise direction. **Head node pointer** will remain at its position. You don't need to change it. If there is a need to remove the node pointed by head node then you

181. The abstract data type refers to the basic mathematical concept that defines the

**Data type....confirm**

explained with the use of a particular language or software, data structure. The **data type** is a concept that is used to define the data type. We have additional two different implementations of the data type.

182. Set() method of list class is used to

**Set the value of object....confirm**

In line 12, the value of the passed in parameter is set to the newly created node.

Page 86 of 505

CS301 - Data Structures

Lecture No. 09

object using the **set()** method

183. The next field in the last node in a singly-linked list is set to \_\_\_\_-.

**NULL....confirm**

**Circularly-linked lists**

Let's talk about circularly linked list. The **next** pointer of the last node in a singly-linked list is set to the first node. The time to traverse the circularly-linked list is  $O(n)$ . Moving along a singly-linked list has to be done in a sequential manner. Doubly-linked lists have two

184. The postfix form of the expression  $A+B*C$  and  $(A+B)*C$  will be \_\_

Different ...confirm

employ the infix form of expressions. Consider the infix expression  $A=B+C$  and  $(A+B)*C$ . The postfix versions are  $ABC++$  and  $ABC+*$  respectively. The order

185. Each node in a singly linked list contains two fields, one field called data field while other field contains

Pointer to next node.....confirm

Each node in a singly linked list contains two fields: one field called data field while other field contains

(a) Pointer to next node

(b) Pointer to next node

(c) Pointer to next node

186. On performing which task on the list implementing array has to visit the entire from start to end

Finding an element.....confirm

\* find

The word 'find' in this context means to search for a particular element in the list. If found, the currentNode pointer is moved to that node. As there is no order in the list, we have to start search from the beginning of the list. We have to check the value of each node and compare it with a value to be searched. If found, a return true and pointer to the currentNode pointer in that node otherwise return false. Suppose that 'x' is not in the list, in this case, we have to search the list from start to end and return false. This is the worst case scenario. Therefore, the time complexity of this operation is  $O(n)$ .

187. Consider the linked list having data [6, 72, 35, 65, 25] stored in it. While current pointer is pointing to memory location having 35 stored in it. What will be the resultant linked list after calling add(18) function on the given linked list

6, 72, 35, 18, 65, 25.....confirm

188. In \_\_ various cells of memory are not located continuously

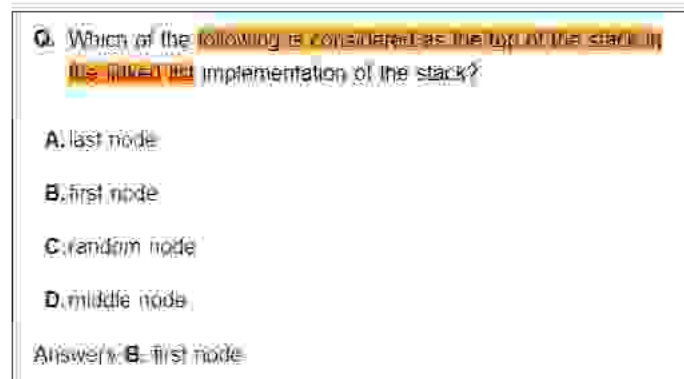
Linked list....confirm

There is no need of more linked memory as linked list is a linear data structure. In this process, each cell of the memory will only contain the reference of the next node. Therefore, the time complexity of this operation is  $O(1)$ .

189. Consider the linked list implementation of a stack. Which of the following node is considered as top of the stack?



First node....confirm



190. Memory address is stored in Pointer.....confirm

Memory address (Raise For Success) is stored in:

- Address operator
- Reference
- **Pointer Correct**

191. “\_” is a \_\_\_\_\_ operator  
Unary.....confirm



192. Which of the following is the limitation of the array?  
Fix size.....confirm

Which of the following (Raise For Success) is the limitation of the array?

- Cannot store user define objects
- Cannot store more than 1000 elements
- Have to random access
- **Fix size Correct**

193. In singly linked list a node consists of two parts:  
Object and pointer....confirm

In singly linked list a node consists of two parts:

1. Object and structure
2. Two pointers
3. Two objects
4. **Object and pointer**

194. Stack and Queue can be implemented using \_\_\_\_\_.

Singly Link List or doubly link list....confirm

195. \_\_\_\_\_ is a collection of elements arranged in a linear order.

Stack.....confirm

196. Consider the following infix expression

$$7/8 + 9$$

If one converts the above expression into postfix, what would be the resultant expression?

78/9+ .....confirm

197. Which of the following operation is used to get an element from the top of stack?

Push(x).....confirm

198. Which one of the following operations returns and removes top value of the stack.

Pop( )....confirm

199. Which one of the following operations returns top value of the stack without removing it.



## Top( ).....confirm

|       |                                                           |
|-------|-----------------------------------------------------------|
| Top 1 | Return the top element without removing it from the stack |
|-------|-----------------------------------------------------------|

200. \_\_\_\_\_ method returns the top element of the stack without removing it.

## Top()...confirm

|       |                                                            |
|-------|------------------------------------------------------------|
| top() | Return the top element without removing it from the stack. |
|-------|------------------------------------------------------------|

201. What will be the postfix expression of following infix expression?

$$\mathbf{A} * \mathbf{B} / \mathbf{C} + \mathbf{D} - \mathbf{E}$$

**AB + CD/ + E - ..confirm**

What **will be the postfix** expression of following infix expression?  
A\*B/C+D-E

1.  $AB / C * D + E -$
2.  $AB * C / D + E -$
3.  $AB * C + DE / -$
4.  $AB + CD / + E -$

202. In the statement template <T>. T represents \_\_\_\_\_

## Generic data type....confirm

Intrepid = Collins, US  
Stock = 1 || Stock() = 1

This means that we are writing a **single-use** structure that stores a **value** of a data type. It is common for a variable declared with a non-static data type (e.g. `int`) will be **shared** with the class name before the access procedure (e.g. `int i`) while storing a **number** of the class. Then there is the **implementation** of the **structure** or **object** we define the value of `i` as `4096` and allocate memory to **define the value** of `i` to be `4096` as `MAXSTACKED` (or `size` of `i`) and **get the starting address** in the **memory** model. If it pertains to **note** that we create an array of `int` (`int i[10]`) **same memory** by dividing the `sizeof` variable, we define the structure `struct` (`struct`) **same memory** by dividing the

203. The `_` method of `list` will position the `currentNode` and `lastcurrentNode` at the start of the list

## Start ()...confirm

have noted the elements with data values 5, 17, 4, 8, 24, 48 and 12. Using the *next* pointer, we can traverse the *list* (as much as the *currentNode* and *lastCurrentNode* in the case of the *list*). Now there is no need to worry about the implementation of the *list*.

204. Which of the following is the correct conversion of infix to postfix expression?

ZB+DH-K/-....confirm

Infix Z-B\*(D-H)/K

Infix to Postfix

Postfix ZB-DH-K/

205. Which of the following function don't belongs to the stack class?

Crash ()....confirm

which of the following function don't belongs to the stack class?

1. push()

2. pop()

3. crash()

206. Consider we have performed the following operations on a stack of size 5  
Pust(10); Push(20); Push(30); Pop(); Pop(); Push(40);

1 or 4 or 20...both...confirm

Consider we have performed the following operations:  
on stack of size 5.

Push(10);

Push(20);

Push(30);

Pop();

Pop();

Push(40);

1. 10

2. 20

3. 40

4. 20

207. Consider we have performed the following operations on a stack of size 5  
Pust(10); Push(20); Push(30); Pop(); Push(40); Push(50); Pop();

4....confirm

208. In singly linked list a node comprises of \_\_\_\_\_ field/s

Two.....confirm

A singly link list's node is divided into two field

209. Each operator in a postfix expression refers to the previous \_\_ operands.

One.....confirm

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

1. One

210. Which of the following signature of the constructor function for a stack is correct as a generic data type?

Template<stack T>;

211. Last node in circular linked list contains

No null pointer....confirm

211. Last node in circular linked list contains  
a) At least one null pointer  
b) No null pointer

212. Back() method of list class is used to  
Move the current position backward one element....confirm

Back() | Move the current position backward one element

213. Next() method of list class is used to  
Move the current position forward one element....confirm

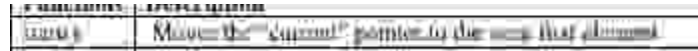
Next() | Move the current position forward one element

214. Tail() method of list class is used to  
Move the "current" pointer to the very last element....confirm

Tail() | Move the current pointer to the very last element

215. Start () method of list class is used to

Moves the “current” pointer to the very first element....confirm



216. To search an element which method of the list searches the entire list from beginning to end?

Find().....confirm

To search an element (Raise For Success) which method of the list searches the entire list from beginning to end?

- end()::
- find():: Correct

217. In \_ various cells of memory

218. If we use array to implement list, then there is an issue that gives difficulty when:

We will increase its size.....confirm

If we use array to implement list, then there is an issue that it gives difficulty when:

1. We will access values randomly
2. We will remove data from it
3. We will increase its size

All Updated MCqs the yahn tak file bhe upload krdon ga tommarrow ya phir 7 jun ko...JAZAKALLAH...Yad rahy ye promise nhe ker rhaa hon ...zindagi or mout ka hmy nhe pata hota ....try kron ga apni...copy that ☺))

219. For a perfect binary tree of height 4, what will be the sum of highest of node.

$$S = (2^{h+1} - 1) - (h + 1) = (2^{4+1} - 1) - (4 + 1) = (2^5 - 1) - (5) = 31 - 5 = 26$$

220. If Ahmed is cousin of Ali and Ali is cousin of Asad then Ahmed is also cousin of Asad.

Transitivity....confirm

If Ahmad is **cousin of Ali** and Ali is cousin of Asad then Ahmad is also cousin of Asad. This statement has the following property

1. Reflexivity
2. Symmetry
3. **Transitivity**

221. Suppose there are 100 elements in an equivalence class, so initially there will be 100 trees the collection of these trees is called.

Forest....confirm

Suppose there are 100 elements **in an** equivalence class, so initially there will be 100 trees. The collection of these trees is called

1. Cluster
2. Class
3. **Forest**

222. The percolate Down procedure will move the smaller value and bigger value.

Up,down....confirm

will move downwards. In this regard, percolate procedures may be of great help. The **percolate down** procedure will move the smaller value up and bigger value down.

223. For a perfect binary tree of height h, having N nodes, the sum of height of nodes is

$N - h - 1$ ....confirm

the **sum of heights** of nodes will be equal to  $N - h - 1$

224. If we want to find median of 50 elements, then after applying builtHeap method, how manytime deleteMin method will be called

25....confirm

If we want to find median of 50 elements, then after applying buildHeap method, how many times deleteMin method will be called?

1. 5
2. 25

225. The Expression if( !heap-> isEmpty() ) check Heap is not empty....confirm

The expression if ( ! heap -> isFull() ) check

1. Heap is not empty
2. Heap is not full

226. Which of the following heap method increase the value if key at position p by the amount delta?

decreaseKey(p, delta).....confirm

Which of the following heap method lowers the value of key at position 'p' by the amount 'delta'?

1. IncreaseKey(p,delta)
2. decreaseKey(p,delta)

227. Given the values are the array representation of heap; 12 23 26 31 34 44 56 64 78 100. What is the 5<sup>th</sup> smallest element in the given heap?

34.....confirm

given the values are the array representation of heap: 12 23 26 31 34 44 56 64 78 100 What is the 5th smallest element in the given heap?

1. 31
2. 34

228. Given the values are the array representation of heap; 12 23 26 31 34 44 56 64 78 100. If we perform 4 deleteMin operations the last element deleted is?

31....confirm

given the values are the array representation of heap:  
12 23 26 31 34 44 56 64 78 100  
If we perform 4 deleteMin operations, the last element deleted is \_\_\_\_\_

1. 31

229. If the elements are 3, 5, 1, 6 in which order they will be removed from the Queue one by one?

3 5 1 6

230. The \_\_\_\_\_ of a binary tree is the maximum level of its leaves (also called the depth).

Height...confirm

The height of a binary tree is the maximum level of its leaves (also called the depth)

231. The \_\_\_\_\_ of a binary tree is the maximum level of any leaf in the tree.

Depth...confirm

The depth of a binary tree is the maximum level of any leaf in the tree

232. Following is a keyword of C++ \_\_\_\_\_.

Delete....confirm

Following is a keyword of C++ \_\_\_\_\_.

1. del

2. delete



233. Analyze the given code carefully and identify which type of binary tree traversal this is;

```
Void traversal (treeNode<int>*treeNode)
{
if(treeNode !=Null)
{
cout << *(treeNode->getinfo())<< “ ”;
Traversal (treeNode->getleft());
Traversal (treeNode->getRight());
}
}
```

Preorder....confirm

Let's write the C++ code for it. Following is the code of the *preorder* method

```
void preorder(TreeNode<int>* treeNode)
{
if (treeNode != NULL)
{
cout << *(treeNode->getinfo()) << " ";
preorder(treeNode->getleft());
preorder(treeNode->getRight());
}
}
```

234. The \_\_\_\_ of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

Balance....confirm

The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

235. If p is pointing to a node in an existing tree then info(p) means;

Returns the contents of node p...confirm



If  $p$  is pointing to a node in an existing tree, then

- $\text{left}(p)$  returns pointer to the left subtree
- $\text{right}(p)$  returns pointer to right subtree
- $\text{parent}(p)$  returns the father of  $p$
- $\text{brother}(p)$  returns brother of  $p$
- $\text{infix}(p)$  returns content of the node

236. If  $p$  is pointing to a node in an existing tree then  $\text{left}(p)$  means;  
Returns pointer to the left subtree  $p$ .....confirm

If  $p$  is pointing to a node in an existing tree, then

- $\text{left}(p)$  returns pointer to the left subtree

237. If  $p$  is pointing to a node in an existing tree then  $\text{right}(p)$  means;  
Returns pointer to right subtree....confirm

If  $p$  is pointing to a node in an existing tree, then

- $\text{left}(p)$  returns pointer to the left subtree
- $\text{right}(p)$  returns pointer to right subtree

238. If  $p$  is pointing to a node in an existing tree then  $\text{parent}(p)$  means;  
Returns the father of  $p$ .....confirm

If  $p$  is pointing to a node in an existing tree, then

- $\text{left}(p)$  returns pointer to the left subtree
- $\text{right}(p)$  returns pointer to right subtree
- $\text{parent}(p)$  returns the father of  $p$

239. If  $p$  is pointing to a node in an existing tree then  $\text{brother}(p)$  means;  
Returns brother of  $p$ .....confirm

If  $p$  is pointing to a node in an existing tree, then

- $\text{left}(p)$  returns pointer to the left subtree
- $\text{right}(p)$  returns pointer to right subtree
- $\text{parent}(p)$  returns the father of  $p$
- $\text{brother}(p)$  returns brother of  $p$

240. Every \_\_\_\_\_ tree is a binary search tree.

AVL.....confirm

Every AVL tree is a binary search tree.

241. If a function has recursive call as the last statement it is known as \_\_\_\_.

Tail recursion....confirm

has recursive call as the last statement, it is known as tail recursion.

242. During in-order traversal using recursive calls, if we found a node is NULL it means this node will satisfy following condition

It will not have left child.....confirm

During in-order traversal using recursive calls, if we found a node is NULL, it means this node will satisfy following condition:  
It will not have left child.

243. In binary search tree deleting a node is easy if it is a \_\_\_\_\_ node

Root....confirm

In binary search tree deleting a node is easy if it is a \_\_\_\_\_ node

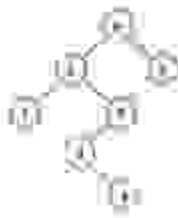
Root

244. Which of the following operation returns the contents of a binary tree node?

Info(p)....confirm

If  $p$  is pointing to a node in an existing tree, then:  
• left( $p$ ) returns pointer to the left subtree.  
• right( $p$ ) returns pointer to right subtree.  
• parent( $p$ ) returns the father of  $p$ .  
• brother( $p$ ) returns brother of  $p$ .  
• info( $p$ ) returns content of the node.

245. What will be level order of the given BST?



6,2,8,1,5,3,4...confirm

246. In a complete binary tree the number of nodes at level 5 are

31....confirm

no. of level in binary tree  $2(5+1)-1 = 31$

247. One should be careful about transient \_\_\_\_\_ that are stored by reference in data structures.

Objects....confirm

One should be careful about transient objects that are stored by reference in data structures.

248. In a tree we link the nodes in such a way that it \_\_\_\_ a linear structure.

Does not remain....confirm

We link the nodes in such a way that it does not remain a linear structure.

249. The function calls are made with the help of \_\_\_\_\_.

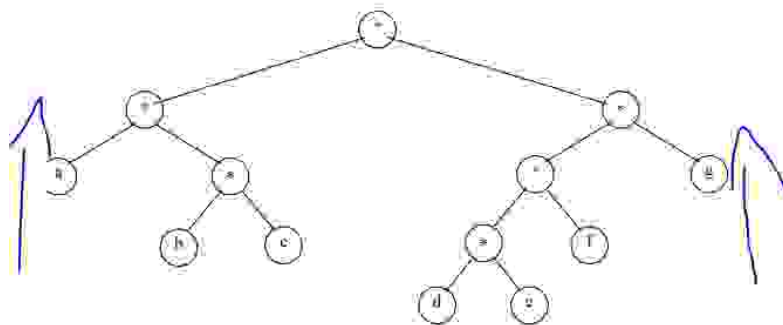
Stack....confirm

that function calls are made with the help of stacks

250. What will be the postorder traversal of the given tree?



15,35,25,75,45.....confirm



Postorder traversal: a b c d e f g

251. A binary tree is said to be a \_\_\_\_\_ binary tree if every non-leaf node in a binary tree has non-empty left and right subtrees.

Strictly....confirm

There is a version of the binary tree, called strictly binary tree. A binary tree is said to be a strictly binary tree if every non-leaf node in a binary tree has non-empty left and right subtrees.

252. In which of the following tree for each node the value of root node is greater than left child and smaller than right child?

Complete binary tree...confirm

253. In which of the following tree for each node the value of root node is greater than left child and smaller than right child?

(a) Binary Search Tree

(b) Full Binary Tree

(c) Complete Binary Tree

253. Two common models of simulation are \_\_\_\_\_ and \_\_\_\_\_.

Time-based, event-based....confirm

Let's discuss little bit about the simulation models. Two common models of simulation are time-based simulation and event-based simulation. In time-based

254. Generally there is/are \_ case(s) to delete a node from BST.

3.....confirm

There are \_\_\_\_\_ cases for **balancing a binary tree** tree.

a. 1  
b. 2  
c. 3  
d. 4

255. When a function calls another function the parameters and return address of the calling function are put in \_\_\_\_\_.

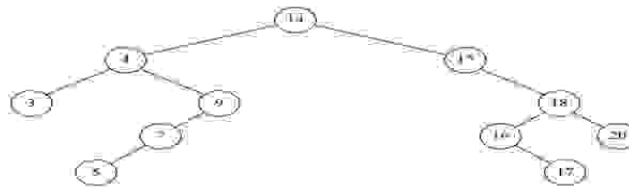
Stack....confirm

We know that function calls are made with the help of stacks. When a function calls some other function, the **return address and value** address of the function is put in a stack. The local variables of the function are also located in the stack and the control

256. What will be the inorder traversal of the given tree?



15,25,35,45,75....confirm



(Fig. 15.4) Inorder: 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20

257. ~BinarySearchTree() is a \_\_\_\_\_  
Constructor.....confirm

~BinarySearchTree() is a \_\_\_\_\_.

1. **Constructor**

258. We can calculate the \_\_\_\_\_ of a subtree by counting its levels from the bottom.

Height....confirm

We can calculate the height **of a subtree** by counting its levels from the bottom

259. If there is a strictly complete binary tree of depth 3 the total number of nodes in it will be?

7....confirm

If there is a strictly complete binary tree of depth 3 the total number of nodes it will be:

7

260. In which of the following function signatures the value of variable “num” cannot be changed in function body?

Int cube (const int& num).....confirm

In which of the following functions, the value of variable cannot be changed in function body?

Select correct option

int cube (int num)

int cube (int\* num)

int cube (const int& num) ok

261. A \_\_\_\_\_ model attempts to model a real-world phenomenon.

Simulation.....confirm

One of the numerous uses of the queues, one of the most useful is *simulation*. A simulation program attempts to *model* a real-world phenomenon. Many popular video

262. The process of getting the value of a variable using pointers is called  
Deferencing.....confirm

The process of getting the value of a variable using a pointer is called

Dereferencing

263. When a function calling itself is called as \_\_\_\_\_.

(Recursive or recursion) both.....confirm

A function that calls itself is said to be recursive



264. A \_\_\_\_\_ is a tree in which every level except possibly the last is completely filled

Complete binary tree....confirm

A \_\_\_\_\_ is a tree in which every level except possibly the last is completely filled

Complete binary tree

265. \_\_\_\_\_ objects (objects accessed by pointers) are called anonymous objects.

Nameless objects....confirm

starting addresses of those objects to e1 and e2 pointers. Nameless objects (objects accessed by pointers) are called anonymous objects.

266. In the perspective of memory organization each process is divided into \_\_\_\_\_ sections.

4.....confirm

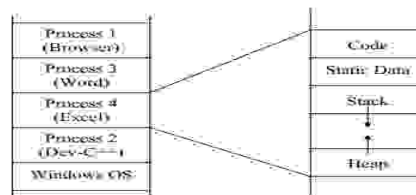


Fig 18.1: Memory Organization

267. Analyze the given code carefully and choose the statement which will be used if we want to set 7 as the value of root?

Int x [] = {14,15,4,9,7,18,4,5,16,4,20,17};

TreeNode \* root = new TreeNode( );

Root->setinfo( &x[4]);

268. In tree, the search operation is \_\_\_\_\_ as compared to the linked list.

Very fast....confirm

Thus in a tree the search is very fast as compared to the linked list

269. Level-order traversal for binary search tree can be implemented \_\_\_\_\_.

Only through recursive method .....confirm

Level order traversal (Raise For Success) for Binary Search Tree can be implemented \_\_\_\_\_.

- Only through recursive method. Correct

270. \_\_\_\_\_ is an area in computer memory that is used to allocate memory dynamically.

Heap....confirm

Heap is an area in computer memory that is allocated dynamically

271. Suppose we want to create the right child of a node 'p' and set the value x in it. Which of the following statement is correct for this operation?

setRight (x,p)....confirm

Suppose we want to create the right child of 'p' and set the value x in it, which of the following statement is correct in this operation:

Set Right (x,p)

272. In level-order traversal for binary Search Tree at each level we visit the nodes in \_\_\_\_\_ order

Left-to-right.....confirm

traversal procedures discussed previously. In level-order traversal we visit the nodes at each level before proceeding to the next level. At each level we visit the nodes in a left-to-right order. The traversal methods discussed earlier are inorder, preorder and

273. Which of the following function(s) is/are used for function call by using pointers?



```
void func1(int& num) {
 num++;
}
void func2(int* num){
 num++;
}
```

**Func2 only....confirm**

```
// Function 2: call by using pointers
int intMinus2(int* oldVal)
{
 *oldVal = *oldVal - 2;
 return *oldVal;
}
```

274. Which of the following function(s) is/are used for function call by value?

```
void func1(int& num) {
 num++;
}
void func2(int num){
 num++;
}
```

**func2 only....confirm**

```
//Function 1: call by value
int intMinus1(int oldVal)
{
 oldVal = oldVal - 1;
 return oldVal;
}
```

275. Which of the following function(s) is/are used for function call by reference?

```
void func1(int& num) {
 num++;
}
void func2(int num){
 num++;
}
```

**func1 only....confirm**

```
// Function 3. call by reference
int minMax3(int& oldVal)
{
 oldVal = oldVal * 3;
 return oldVal;
}
```

276. In a program a reference variable with name 'x' can be declared as

**Int &x;....confirm**

276. In a program a reference variable with name 'x' can be declared as **int &x;....confirm**

277. Suppose the elements are 3,5,1,6, if we want to enqueue '8' then dequeue all the elements what will be the order of elements?

**8,3,5,1,6**

After dequeue() is called once

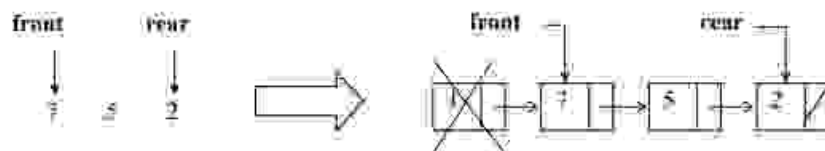


Fig 4. Removal of one element from queue using dequeue()

278. Which of the following data structure is linear type?

**(List, Queue, Stack)....All options**

278. Which of the following data structure is linear type?  
☐ a) Stack  
☐ b) List  
☐ c) Queue  
☒ d) All of the above

279. Analyze the given code carefully and identify which type of binary tree traversal this is;

```
void traversal (treeNode<int>*treeNode)
{
if(treeNode != Null)
{
traversal (treeNode->getleft());
cout << *(treeNode->getinfo())<< " ";
Traversal (treeNode->getRight());
}
}
```

**Inorder....confirm**

*Here is the code of the inorder function.*

```
void inorder(TreeNode<int>* treeNode)
{
if (treeNode != NULL)
{
inorder(treeNode->getLeft());
cout << *(treeNode->getInfo()) << " ";
inorder(treeNode->getRight());
}
}
```

280. Analyze the given code carefully and identify which type of binary tree traversal this is;

```
void traversal (treeNode<int>*treeNode)
{
if(treeNode != Null)
{
traversal (treeNode->getleft());
Traversal (treeNode->getRight());
cout << *(treeNode->getinfo())<< " ";
}
}
```

**Postorder.....confirm**

```

void postorder(TreeNode<int>*& treeNode)
{
 if (treeNode != NULL)
 {
 postorder(treeNode->getLeft());
 postorder(treeNode->getRight());
 cout << "(treeNode->data)" << endl;
 }
}

```

281. Doubly Linked list always has one Null pointer.

**False....confirm**

which is implemented using 'single linked list'. Objective: To doubly linked list. The overhead is the addition of two pointers (variables) are added within each node. In that case,

*CS301 quiz(1) file*

*By*

*<https://vuonlinehelp.blogspot.com/>*

*Compose by (ALI HASSAN)*

If the elements are 3, 5, 1, 6 in which order they will be removed from the queue one by one

3516

The.....of a binary tree is a maximum level of its leaves (also the depth).

height

.....of the an empty AVL tree is define to be

Height

Following is the keyword of c++

Delete

In the perspective of memory organization every process expecting the first part of memory is for.....of the program.

code

The.....of a node in a binary tree is define as the height of its left

subtree minus height of its right subtree.

**balance**

If p is pointing to a node in an existing tree then info (p) means

**Return the content of node p.**

Following is a keyword of c++.....

**Delete**

Every.....tree is a binary search tree

**AVL**

If a function has recursive call as last statement it is known as.....

**Tail recursive**

During in a order traversal call if we found a node is null it means this node will satisfy following condition

**it will not have left child**

In binary search tree deleting a node is easy if it is a..... node

**Root**

Which of the following operation returns the content of binary tree node?

**Info (p)**

1 2 6

5 8

3

4

1 2 3 4 5 6 8

in a complete binary tree the number of node of level 5 are

30

One should be careful about transient.....that are stored by reference I data structure

Objects

In a tree we link the node in such a way that it.....a linear structure

does not remain

the function call are mode with the help of.....

stack

18. 15 25 45

35 75

15 35 25 75 45

A binary is said to be a.....binary tree if every non leaf node in binary tree has non empty left and right structure

Strictly

In which of the following function signature the value of variable num cannot be changed in function body

Binary search tree

Two common method of simulations are.....are.....

**Time based simulation and event based simulation**

Generally there are case (s)to delete a node from BST

**1**

When a function calls another function the parameters and address of the calling function are put in...

**Stock**

Binary search tree ()is a.....

**Destructor**

We can calculate the.....of subtree by counting its level from the bottom

**Nodes**

If there is a strictly complete binary tree of depth 3 the total number of nodes it will be

**7**

A .....model attempt to model real word phenomenon

**Simulation**

The process of getting the value of a variable using a pointer is called

**Dereferencing**

When a function calling itself is called as.....



## Recursion

A.....is a tree in which every except possibly the last, is completely filled

## Strict binary tree

In AVL tree during insertion, a single rotation can fix the balance in cases \_\_\_\_\_ and 4.

1

If a node is inserted in outer side of a node in binary search tree then to make it AVL tree,

## We may have to apply single rotation

A zigzag rotation is performed. In Left-Left case of rotation in AVL tree.

## False

The principal benefit of a linked list over a conventional array is that the order of the linked items may be\_\_\_\_\_ from the order that the data items are stored in memory.

## Same

Stack.push(15) will push 15 on \_\_\_\_\_.

## Top of the stack

Suppose we have the following values to be inserted in constructing AVL tree,

10,13,15,5,7,8

Tell when first rotation will take place,

### After inserting the node 15

All the objects created using \_\_\_\_\_ operator have to be explicitly destroyed using the delete operator.

### New

During the execution of a process, operating system constructs four things for that process. Which of the following is not part of that process?

### Linked list

There are \_\_\_\_\_ cases of Rotation in AVL tree.

### 4

Which of the following line of code is incorrect?

`Float *l = float new` (not confirmed)

In simple or singly linked list there is/are \_\_\_\_\_ pointer/s in each node.

### One

Suppose there are three nodes tree with three numbers 14, 4, 15. Following is not a permutation, or combination for output them.

(4, 4, 15)

The type of expression in which operator succeeds its operands is \_\_\_\_\_ expression.

### Postfix

The \_\_\_\_\_ of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

### Balance

If both left and right nodes of a node are NULL then this type of node is called a \_\_\_\_\_ node.

### Non leaf (not confirmed)

The function calls are made with the help of\_\_\_\_\_.

### Stack

Which data structure is needed to convert infix expression to postfix expression?

### Stack

In case of insertion of right inner node in BST,

### Left rotation then right to make AVL

A BST generated from the data in ascending order is \_\_\_\_\_.

### Linear

Which operation of queue data structure is used to get front element from the queue and then remove it from the queue?

### Remove ()

In \_\_\_\_\_ various cells of memory are not located continuously.

### Linked list

The \_\_\_\_\_ method of list will position the currentNode and lastCurrentNode at the start of the list.

### Start

Which operation of queue data structure is used to insert an element into the Queue?

Enqueueer ()

From Operating System point of view, the recursive function calls are made with the help of \_\_\_\_\_.

Queue

We can make a lexicographic order of characters based on their \_\_\_\_\_.

ASCII values

What are the basic things associated with data structures?

All of the above

Consider the following push operations of a Stack:

Stack.push(4);

Stack.push(6);

Stack.push(5);

Stack.push(8);

If a user calls a pop() operation, then which value will be returned?

8 (stack is last in first out)

~BinarySearchTree( ) is a \_\_\_\_\_.

Destructor

Array cells are \_\_\_\_\_ in computer memory.

Contiguous

Leaf node of binary search tree contains \_\_\_\_\_

One Null Pointer

dequeue() operation of queue data structure is used to \_\_\_\_\_

remove an element from the front and return it

The depth of a binary tree is

Maximum level of a leaf

Which of the following operation returns but do not removes top value of the stack?

Top

Which of the following data structure is linear type?

All of above

In doubly linked list a node consists of three parts:

2 pointers and 1 object

Last node in circular linked list contains

No null pointer

A \_\_\_\_\_ model attempts to model a real-world phenomenon

Simulation

Factorial is an example of \_\_\_\_\_ function.

Recursive

The balance of a node in a binary tree is defined as the height of its \_\_\_\_\_ sub tree minus height of its right sub tree.

**Left**

In case of insertion of left outer node in BST,

We first apply left rotation and then rotation to make it an AVL Tree.

`int htdiff = height(root->getLeft()) - height(root->getRight());` The above line of code is taken from AVL insert method. Complete it by selecting an appropriate symbol.

**Minus (-)**

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

**Two**

For making Binary Search Tree for Strings we need, \_\_\_\_\_ data type.

**Int**

A \_\_\_\_\_ model attempts to model a real-world phenomenon

**Simulation**

The lifetime of a transient object cannot exceed that of the application.

**True**

Which one of the following calling method does not change the original value of the argument in the calling function?

Call by passing the value of the argument

In level-order traversal for Binary Search Tree, we visit the nodes at each level before proceeding to the next level, in a \_\_\_\_\_ order.

Left-to-right

In doubly linked list a node consists of three parts:

1 pointer and 2 objects

Binary Search Tree violates the condition of AVL tree when any node has balance equal to

1 or -1

To search an element in AVL tree, it takes maximum  $1.88 \log_2 n$  time

True

How many cases of rotation are there in AVL tree?

4

\_\_\_\_\_ tree has been named after two persons Adelson-Velskii and Landis.

AVL

Which of the following is a nonlinear data structure?

Tree

AVL tree is nonlinear data structure.

True

\_\_\_\_\_ is when function is calling to itself.

### Recursion

Which of the following function don't belongs to the stack class?

### Crash ()

Which one is the cored function call for the following function of calculating cube?

```
int cube(int& num)
```

`cube (num)`

Binary search algorithm cannot be applied to \_\_\_\_

### None of the given

copy() method of list data structure \_\_\_\_\_

### Set one list to be a copy of another

While implementing non-recursive traversal for Binary Search Tree, we need to implement

### Stack

Following is true in case of using Recursive method calls

### The code becomes very short

\_\_\_\_\_is a self-balancing tree.

### Binary Search Tree

While implementing stack with an array and to achieve LIFO behavior, we used push and pop elements at



The start of the array

Generalized code written for a class is called

Structure

Array cells are \_\_\_\_\_ in computer memory

Contiguous

In singly linked list a node comprises of \_\_\_\_\_ field/s.

Two

Which of the following statement is false?

Pointers store the next data element of a list

Leaf node of binary search tree contains \_\_\_\_\_

One null Pointer

What will be the result of evaluating following expression?

$5 + 3 * 2 / (6 - 3)$

7

Want and de-allocating memory for linked list nodes does take less time than pre-allocated array

MORE

Suppose you are writing a class for Node class and forgot to write the constructor of the class, then

Compiler will automatically generate a default constructor

Factorial is an example of \_\_\_\_\_ function.

## Recursive

A binary tree is said to be a \_\_\_\_\_ binary tree if every non-leaf node in a binary tree has non-empty left and right subtrees.

## Strictly

Memory address is stored in

## Pointer

In level-order traversal for Binary Search Tree, \_\_\_\_\_ data structure is used.

## Queue

Which of the following is the correct option for priority Queue?

The type of queues that is FIFO i.e. the person who comes first should leave first.

AVL tree is linear data structure.

## False

AVL tree is a binary search tree

## True

The post order traversal of a binary tree is DEBFCA. Find out the pre order traversal

A

Λ

B C

$\wedge /$

D E F

ABDCEF

In a tree, we link the nodes in such a way that it \_\_\_\_\_ a linear structure.

Does not remain

add(12) method of linked list class will:

Add 12 as value in linked list

Which of the following is the correct conversion of infix to postfix expression?

$Z+B-(D-H)/K$

ZB+DH-K/-

In singly linked list "next" field of node contains:

Address of next node

Which of the following line of code is incorrect?

The lifetime of a transient object can exceed that of the application which is accessing it.

A list is the collection of items of the \_\_\_\_\_

same type

back() method of List class is used to:

Moves the "current" pointer to backward one element.

Suppose you are writing a class for Node class and forgot to write the constructor of the class, then

\_\_\_\_\_ is the maximum height of the AVL tree.

$1.44\log_2 n$

length() method of List class is used to:

Return the length of the list

There are four cases of rotation in an \_\_\_\_\_ tree.

AVL

Allocating and de-allocating memory for linked list nodes does take \_\_\_\_\_ time than pre-allocated array.

More

\_\_\_\_\_ is utilized at the time of memory allocation in dynamic manner.

Heap

In which case of insertion we require double rotation to make the AVL tree balance.

None

The back() method decreases the value of variable current by \_\_\_\_\_.

One

In array list the worst case of removing an element is

To remove the item from start

Left, right, info, and parent are the operations of \_\_\_\_\_ data structure.

Tree

Which of the following is TRUE for search operations in a binary tree as compared to linked list or an array?

It increase the number of comparison

Which of the following is a correct statement?

An AVL tree is identical to BST except the height of left and right subtree can differ by at least 1

length() method of List class is used to:

Return the length of the list

Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function.

Return Address

Whenever we call a function, the compiler makes a \_\_\_\_\_ that it uses to fulfill this function call.

Stack

\_\_\_\_\_ is the major factor to see the efficiency of a program.

Time

\_\_\_\_\_ rule applies for evaluating operators of same precedence in an expression

None

The postfix form of the expression  $A + B * C$  and  $(A + B) * C$  will be \_\_\_\_\_.

Same

If we use array to implement list, then there is an issue that it gives difficulty when:

We will access value randomly

During in-order traversal using recursive calls, if we found a node is NULL. It means this node will satisfy following condition.

It will not have left child

Which one the following is more closer to AVL tree, (chose the best option)

Binary Search Tree

start() method of List class is used to:

Move the current pointer to first element

Consider the following infix expression.  $7/8 + 9$

If one converts the above expression into postfix, what would be the resultant expression?

78/9+

In internal memory organization of a process, there is some area of memory for static data that holds \_\_\_\_\_ variables.

Both Static and Global

In Left-Right case of rotation in \_\_\_\_\_ tree. A double rotation is performed.

AVL

If numbers 5, 222, 4, 48 are inserted in a queue, which one will be removed first?

5

Which one is not the property of binary tree?

Sibling node should be same parent (Not Sure)

Each node in singly linked list contains\_\_\_\_\_

One Pointers

What's wrong with following loop? while( (i < 10) && (i > 24)) { }

The Condition is always false

Which one of the following is TRUE about recursion?

1. Recursive function calls consume a lot of memory.
2. Recursion extensively uses stack memory.
3. Recursion extensively uses stack memory.
4. Iteration is more efficient than iteration.

Which of the following is not a data structure ?

1. Linked list

2. Queue
3. stack
4. Memory cell

“+” is a \_\_\_\_\_operator.

1. Unary
2. Binary
3. Ternary
4. None of the above

When an executable program run, it is loaded in the memory and becomes a\_\_\_\_\_.

1. Thread
2. .h file
3. Process
4. None of the above

[www.vustudents.ning.com](http://www.vustudents.ning.com)

In array list the worst case of removing an element is

1. To remove an element from the end of the list
2. To remove an element from the middle of the list
3. To remove an element from the start of the list
4. We cannot remove element from an array list

The tree data structure is a

1. Linear data structure



2. Graphical data structure
3. Non-linear data structure (Not Confirm)
4. Data structure like queue

In\_\_\_\_\_, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node.

1. Linked list
2. doubly-link list
3. array
4. Structure

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s)

1. one
2. two
3. three
4. four

Which of the following operations returns top value of the stack?

1. push
2. pop
3. top
4. First

In the calling function, after the execution of the function called, the program continues its execution from the \_\_\_\_\_ after the function call.

1. Previous line
2. Next line
3. Beginning
4. None of the above

\_\_\_\_\_ only removes items in reverse order as they were entered.

1. Queue
2. Stack
3. Both of these
4. None of these

In the linked list implementation of the stack class, where does the push member function place the new entry on the linked list?

1. After all other entries that are greater than the new entry.
2. At the head
3. After all other entries that are smaller than the new entry.
4. At the tail

Which of the following is known as "Last-In, First-Out" or LIFO Data Structure?

1. Linked List
2. Stack
3. Queue
4. Tree

A queue is a \_\_\_\_\_data structure, whereas a stack is a \_\_\_\_\_data structure.

1. FIFO, LIFO

Each node in doubly link list has

1. 2
2. 3
3. 4

Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function

1. First argument
2. Return address
3. Last argument
4. none of the above

\_\_\_\_\_ is the stack characteristic but \_\_\_\_\_ was implemented because of the size limitation of the array.

1. isFull(),isEmpty()

2. pop(), push()
3. isEmpty() , isFull()

Stack and Queue can be implemented using \_\_\_\_\_,

1. Singly Link List
2. Binary Tree
3. Binary Search Tree
4. AVL Tree

The \_\_\_\_\_ method of list will position the currentNode and lastCurrentNode at the start of the list.

1. Remove
2. next
3. start
4. back

In array list the worst case of removing an element is

1. To remove an element from the end of the list

2. To remove an element from the middle of the list
3. To remove an element from the start of the list
4. We cannot remove element from an array list

STL is a \_\_\_\_\_ that is a part of the official standard of C++.

1. C program file

2. .h file

When an executable program run, it is loaded in the memory and becomes a\_\_\_\_\_.

1. Thread

2. .h file

3. Process

4. None of the above

Each node in doubly link list has,

1. pointer

2. pointers

3. pointers

4. pointer

A queue is a \_\_\_\_\_data structure, whereas a stack is a \_\_\_\_\_data structure.

1. FIFO, LIFO

2. LIFO,FIFO

3. both of these

4. none of these

Which one of the following is TRUE about recursion?

1. Recursive function calls consume a lot of memory.

2. Recursion extensively uses stack memory.

3. Recursion extensively uses stack memory.
4. Iteration is more efficient than iteration.

To create a \_\_\_\_\_ we link the last node with the first node in the list.

1. Double linked list
2. Circularly-linked list
3. Linked list
4. None of the above

\_\_\_\_\_ is the stack characteristic but \_\_\_\_\_ was implemented because of the size limitation of the array.

1. isFull(),isEmpty()
2. pop(), push()



3. isEmpty() , isFull()

4. push(),pop()

The principal benefit of a linked list over a conventional array is that the order of the linked items may be\_\_\_\_\_ from the order that the data items are stored in memory.

1. Same

2. Identical

3. Different

4. Equivalent

In\_\_\_\_\_ the 'next' returns false when it reaches to the last node due to the fact that the next field of the last node is set to NULL.

1. Circular linked list

2. Triple linked list

### 3. Singly linked list

4. None of the above

If numbers 5, 222, 4, 48 are inserted in a queue, which one will be removed first?

1. 48

2. 4

3. 333

4. 5

Which boolean expression indicates whether the numbers in two nodes (p and q) are the same. Assume that neither p nor q is null.

1. `p == q`

2. `p.data == q.data`

3. `p.nextNode == q.nextNode`

4. `p.data == q`

Local variables of a function are stored in,

1. Binary Search Tree
2. Stack
3. Queue
4. AVL Tree

A queue is a data structure where elements are

1. inserted at the front and removed from the back.
2. inserted and removed from the top.
3. inserted at the back and removed from the front.
4. inserted and removed from both ends.

A kind of expressions where the operator is present between two

operands called \_\_\_\_\_expressions.

1. Infix

2. Postfix

3. Prefix

4. None of the above

A template is a function or class that is written with a \_\_\_\_\_data type.

1. Specific

2. Definite

3. Generic

4. None of the above.

Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function.

1. First argument

2. Return address

3. Last argument

4. None of the above

Which of the following can be used to reverse a string value,

1. Stack

2. Queue

3. Both of these

4. None of these

A queue is a \_\_\_\_\_data structure, whereas a stack is a \_\_\_\_\_data structure.

1. FIFO, LIFO

2. LIFO,FIFO
3. both of these
4. none of these

Stack and Queue can be implemented using \_\_\_\_\_,

1. Singly Link List
2. Binary Tree
3. Binary Search Tree
4. AVL Tree

Which of the following is not a data structure ?

1. Linked list
2. Stack
3. Queue
4. Memory cell

Compiler uses which one of the following in Function calls,

1. Stack
2. Queue
3. Binary Search Tree
4. AVL Tree

The principal benefit of a linked list over a conventional array is that the order of the linked items may be\_\_\_\_\_ from the order that the data items are stored in memory.

1. Same
2. Identical
3. Different
4. Equivalent

Doubly Linked List always has one NULL pointer.

1. True
2. False

We cannot remove items randomly from \_\_\_\_\_

1. Stack
2. Queue
3. Both of these
4. None of these

\_\_\_\_\_ is the maximum number of nodes that you can have on a stack-linked list ?

1. Zero
2.  $2n$  (where  $n$  is the number of nodes in linked list)
3. Any Number
4. None of these



“+” is a \_\_\_\_\_operator.

1. Unary
2. Binary
3. Ternary
4. None of the above

In\_\_\_\_\_, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node.

1. Linked list
2. doubly-link list
3. array
4. structure

A queue is a data structure where elements are

1. inserted at the front and removed from the back.
2. inserted and removed from the top.

3. inserted at the back and removed from the front.

4. inserted and removed from both ends.

\_\_\_\_\_ is the stack characteristic but \_\_\_\_\_ was implemented because of the size limitation of the array.

1. isFull(),isEmpty()

2. pop(), push()

3. isEmpty() , isFull()

4. push(),pop()

The \_\_\_\_\_ method of list will position the currentNode and lastCurrentNode at the start of the list.

1. Remove

2. Next

3. Start

4. Back

Parameters in function call are passed using,

- 
1. Stack
  2. Queue
  3. Binary Search Tree
  4. AVL Tree

If numbers 5, 222, 4, 48 are inserted in a queue, which one will be removed first?

1. 48
2. 4
3. 333
4. 5

For a perfect binary tree of height 4, what will be the sum of highest of node

1. 31
2. 30
3. 27

4. 26

If Ahmed is cousin of Ali and Ali is cousin of Asad then Ahmed is also cousin of Asad. This statement has the following property

1. Reflexivity
2. Symmetry
3. Transitivity
4. All of the above

Which one of the following is NOT the property of equivalence relation?

1. Reflexive
2. Symmetric
3. Transitive
4. Associative

The main reason of using heap in priority queue is

1. Improve performance

2. Code readable
3. Less code
4. Heap can't be used in priority queues

The total numbers of nodes on 10th level of perfect binary tree are

1. 256
2. 512
3. 1024
4. Can't be determined

Suppose there are 100 elements in an equivalence class, so initially there will be 100 trees, the collection of these trees is called \_\_\_\_\_.

1. Cluster
2. Class
3. Forest
4. Bunch

The percolate Down procedure will move the smaller value and

bigger value.

1. Left, right
2. Right, left
3. Down, up
4. Up, down

For a perfect binary tree of height  $h$ , having  $N$  nodes, the Sum of height of nodes is

1.  $N - h - 1$
2.  $N - 1$
3.  $N - 1 + h$
4.  $N - (h - 1)$

Which of the following method is helpful in creating the heap at once?

1. Insert
2. Add
3. Update

4. percolateDown

If ahmad is boss of ehsan and ehsan is boss of umer then ahmad is also boss of umer, the above-mentioned relation is\_\_\_\_\_ .

1. Reflexive
2. Symmetry
3. Transitive
4. None of given

If we want to find median of 50 elements, then after applying builtHeap method, how many times deleteMin method will be called?

1. 5
2. 25
3. 35
4. 50

Which of the following properties are satisfied by equivalence relationship?

1. Reflexive, symmetric

2. Reflexive, transitive

3. Symmetric, transitive

4. Reflexive, symmetric and transitive

Sorting procedure normally takes\_\_\_ time.

1.  $N \log N$

2.  $2N$

3.  $N * N * N$

4.  $N$

15. The Expression

if ( ! heap->isFull() )

Check

1. Heap is empty

2. Heap is full

3. Heap is not empty



#### 4. Heap is not full

The Expression

if ( ! heap->isEmpty() )

Check

1. Heap is empty

2. Heap is full

3. Heap is not empty

4. Not a valid expression

Given the values are the array representation of heap: 12 23 26 31  
34 44 56 64 78

100 If we perform 4 deleteMin operation, the last element deleted is\_\_\_\_\_ .

1. 31

2. 34

3. 44

4. 56

Which of the following heap method increase the value if key at

position 'p' by the amount 'delta'?

1. increaseKey(p, delta)

2. decreaseKey(p, delta)

3. percolateDown(p, delta)

4. remove(p, delta)

Which of the following heap method lowers the value if key at position 'p' by the amount 'delta'?

1. increaseKey(p, delta)

2. decreaseKey(p, delta)

3. percolateDown(p, delta)

4. remove(p, delta)

Which property of equivalence relation is satisfied if we say:

Ahmad R(is related to)Ahmad

1. Reflexivity

2. Symmetry

3. Transitivity

4. All of Above

Given the values are the array representation of heap; 12 23 26 31 34 44 56 64 78 100 What is the 5th smallest element in the given heap?

1. 31

2. 34

3. 44

4. 56

**CS301 Quiz No.01 Solution Fall2021**

For a perfect binary tree of height 4, what will be the sum of highest of node

26

If Ahmed is cousin of Ali and Ali is cousin of Asad then Ahmed is also cousin of Asad. This statement has the following property

Transitivity

Suppose there are 100 elements in an equivalence class, so initially there will be 100 trees, the collection of these trees is called.

Forest

The percolate Down procedure will move the smaller value and bigger value.

Up, down

For a perfect binary tree of height  $h$ , having  $N$  nodes, the sum of height of nodes is

$N-h-1$

If we want to find median of 50 elements, then after applying builtHeap method, how

Manytime deleteMin method will be called?

25

The Expression

If(!heap-> isEmpty () )

Check

Heap is not empty

Which of the following heap method increase the value if Key at position 'p' by the Amount 'delta'?

Increase Key(p,delta)

Given the values are the array representation of heap; 12 23 26 31 34 44 56 64 78 100

What is the 5<sup>th</sup> smallest element in the given heap?

34

-----

Which of the following data structure linear type

All of above

in a program a reference variable with name 'x' can be declared as

int &'x';

Object (object oriented by pointers) are called anonymous object

Friend

in the perspective each process divided into.....section

4

In tree the search operation is.....as compared to the linked list

Very fast

Level order transversal for binary tree search can be implemented.....

Only through non recursive method

.....is an area in computer memory that is used to allocate memory dynamically

Stock

Suppose we want to create the right child of 'p' and set the value x in it. which of the following statement is correct in this operation

Set Right (x,p)

In level order traversal for binary search tree of each level we visit node in.....order

Right to left

We can calculate the.....of a subtree counting the level from the bottom

Data items

A .....is a tree in which every level except possibly the last is completely filled

Complete binary tree

Which of the following calling method does not change the original value of variable passed to a function as an argument?

### Call by passing as value

The \_\_\_\_\_ of a binary tree is the maximum level of its leaves (also called the depth).

Select correct option:

Level

Width

Height

None of the above

Question # 2 of 10 ( Start time: 06:40:41 PM ) Total Marks: 1

If we write functions for recursive and non recursive inorder traversal method of BST, what will be the difference between its functions prototypes?

Select correct option:

Different return types

Different function names

Different arguments list

Nothing will be different

Question # 3 of 10 ( Start time: 06:40:41 PM ) Total Marks: 1

binary tree whose every node has either zero or two children is called \_\_\_\_\_

Select correct option:

Complete binary tree

Binary search tree

Strictly binary tree

None of above

Question # 4 of 10 ( Start time: 06:40:41 PM ) Total Marks: 1

Which one of the following calling method does not change the original value of the argument in the calling function?

Select correct option:

Call by passing reference of the argument

Call by passing the address of the argument

Call by passing the value of the argument

None of the given options

Question # 5 of 10 ( Start time: 06:40:41 PM ) Total Marks: 1

Longest path from root node to farthest leaf node is called \_\_\_\_\_ of tree

Select correct option:

Level

Length

Depth

Node level

Question # 6 of 10 ( Start time: 06:40:41 PM ) Total Marks: 1

Which of the following statement is false?

Select correct option:

Arrays are dense lists and static data structure

data elements in linked list need not be stored in adjacent space in memory

pointers store the next data element of a list

linked lists are collection of the nodes that contain information part and next pointer

The \_\_\_\_\_ of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

Select correct option:

Height

Balance

Width

None of the above



A tree is an AVL tree if

Select correct option:

Any one node fulfills the AVL condition

At least half of the nodes fulfill the AVL condition

**All the nodes fulfill the AVL condition**

None of the given options

In a program a reference variable, say x, can be declared as

Select correct option:

**int &x ;**

int \*x ;

int x ;

None of the given options

In which of the following functions, the value of variable cannot be changed in function body?

Select correct option:

int cube( int num)

int cube(int& num)

**int cube(const int& num)**

int cube(int\* num)

What is the maximum depth of recursive calls a function may make?

**n (where n is the argument)**

Here is a small function definition:

```
void f(int i, int &k)
```

```
{
```

```

i = 1;
k = 2;
}

```

Suppose that a main program has two integer variables x and y, which are given the value 0. Then the main program calls f(x,y); What are the values of x and y after the function f finishes?

x is still 0, but y is now 2.

Write in postfix form  $5 * (9 - 7)$

5 9 7 - \*

Which of the following is NOT true regarding the maze generation?

Remove a randomly chosen wall if the cells it separates are already in the same set

An array x of 100 integers is declared as

int x[100];

Local variables defined inside function body are \_\_\_\_\_ automatically at the end of function execution.

Created

Which of the following statement is NOT correct regarding Table ADT?

A table consists of several columns known as entities

In doubly linked list there is/are \_\_\_\_\_ pointer/s in each node.

Two

"new int[11]" will allocate memory for \_\_\_\_\_ integers.

11

If a tree has 20 edges/links, then the total number of nodes in the tree will be:

21

In the statement `int& a=b;`

`b` hold the address of variable `a`

Which of the following is NOT an open addressing technique to resolve collisions?

Double hashing

Consider the function X as under

```
int X (int& Value)
```

```
{
```

```
return Value;
```

```
}
```

Now `a` and `b` are integers in a calling function. Which one of the following is a valid call to the above function `X`?

`a = X(*b);`

For a perfect binary tree of height 4, What will be the sum of heights of nodes?

26

If unions are done by weight (size), the depth of any element is never greater than

$\log_2 n$

Which of the following statement is NOT true for reference variable?

References can be uninitialized.

The tree data structure is a

Non-linear data structure

In singly linked list a node consists of two parts:

Object and pointer

The simplest case in a BST to delete a node is:

When the node, that is to be deleted is a leaf node

A binary tree with 24 internal nodes has \_\_\_\_\_ external nodes.

25

$a * (b+c)-d$  is an example of \_\_\_\_\_ expression.

infix

given the values are the array representation of heap:

12 23 26 31 34 44 56 64 78 100

If we perform 4 deleteMin operations, the last element deleted is \_\_\_\_\_.

31

In \_\_\_\_\_, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node.

Linked list

Trying to remove an element from an empty stack is called \_\_\_\_\_.

Underflow of stack

For reference variables, \_\_\_\_\_ sign is used.

ampersand

Which property of equivalence relation is satisfied if we say:  
Ahmad R(is related to) Ahmad

Reflexivity

Suppose there is an image segmented into pixels. Each pixel has \_\_\_\_\_ neighbour(s).

8

The most difficult case for deleting a node from Binary Search Tree is when the node to be deleted \_\_\_\_\_.

is Leaf node

When an executable program runs, it is loaded in the computer memory and becomes a \_\_\_\_\_ .

Process

If class A defines class B as its friend, then:

Class B can access private members of Class A

In the post-order traversal of a binary search tree, nodes process as:

Left-subtree, Right-subtree, Root

The expression  $AB+C^*$  is called?

Postfix expression

Binary Search can be categorized into which of following?

## Divide and conquer

The main use of AVL tree is:

## Storing of data

Select the one FALSE statement about binary trees:

Every binary tree has at least one node.

A table consists of several columns, known as

## Fields

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

## One

If there are 56 internal nodes in a binary tree then how many external nodes this binary tree will have?

## 57

Which of the following function don't belong to the stack class?

## crash()

Which of the following statement is false?

Arrays are dense lists and static data structure

During union by size method, which data structure is used to improve the balancing of tree?

## Array

Non-recursive calls are faster than which of the following calls?

**Recursive**

Binary Search Tree violates the condition of AVL tree when any node has balance equal to

**0**

Each node in Binary Search Tree has

**2 pointers**

What will be postfix expression of the following infix expression?

*Infix Expression:  $a + b * c - d$*

**$abc * + d -$**

The balance of a node is the result of "height of left subtree" \_\_\_\_\_ "height of right subtree".

**Minus**

A complete binary tree having "N" nodes consists of \_\_\_\_\_ Levels.

**$\log_2 (N+1) - 1$**

Consider a hash table of size seven, with starting index zero, and a hash function  $(3x + 4) \bmod 7$ . Assuming the hash table is initially empty, which of the following is the contents of table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that '\_' denotes an empty location in the table.

**1, 8, 10, \_, \_, \_, 3**

Which one of the following is known as "Last-In, First-Out" or LIFO Data Structure?

**Stack**

Parameters in function call are passed using,

**Stack**

If we singly linked list to implement list, then there is an issue that it gives difficulty when we:

**Move backward in the list**

Which of the following is correct about AVL Tree?

**It is identical to BST except height of the left and right subtrees can differ by at most 1.**

Insertion in a linked list can be done at

**Front, Back and somewhere in the middle**

If the height of a perfect binary tree is 4. What will be the total number of nodes in it?

**31**

AVL tree is linear data structure.

**TRUE**

During the union by size method, all the array element initialized to -1 shows:

**Initial condition of tree**

Consider the following pseudo code

*declare a stack of characters*

*while ( there are more characters in the word to read )*

*{*

*read a character*

*push the character on the stack*



```

}
while (the stack is not empty)
{
 pop a character off the stack
 write the character to the screen
}

```

What is written to the screen for the input "apples"?

selppa

Consider the linked list having data [6, 72, 35, 65, 25] stored in it. While current pointer is pointing to memory location having 72 stored in it. After calling add(4) function on the following linked list current point will point to memory location having value?

4

One difference between a queue and a stack is

Queues use two ends of the structure, stacks use only one.

next() method of List class is used to:

Moves the current position forward one element

get(?) method of list class is used to:

Get element at the given position

If we want to find median of 50 elements, then after applying buildHeap methd, how many times deleteMin method will be called?

25

In simple or singly linked list there is/are \_\_\_\_\_ pointer/s in each node.

One

If we have 1000 sets each containing a single different person.  
Which of the following relation will be true on each set:

Reflexive

The total number of nodes on 10th level of a perfect binary tree are  
:

1024

If there are 100 elements in a heap, and 100 deleteMin operation  
are performed, will get \_\_\_\_\_ list.

Sorted

In simple implementation of stack, isFull() method is used due to  
\_\_\_\_\_.

Limitation of array

If a tree has 50 nodes, then the total edges/links in the tree will be:

49

Following is a keyword of C++ \_\_\_\_\_ .

Delete

In level-order traversal for Binary Search Tree, at each level, we  
visit the nodes in \_\_\_\_\_ order.

Left-to-right

To search an element in AVL tree, it takes maximum  $1.88 \log_2 n$   
time.

FALSE

If we want to find 3rd minimum element from an array of elements, then after applying buildHeap method, how many times deleteMin method will be called?

3

If there are N elements in an array then the number of maximum steps needed to find an element using Binary Search is

\_\_\_\_\_.

$\log_2 N$

The \_\_\_\_\_ method of list data structure removes the element residing at the current position

Remove

Allocating and de-allocating memory for linked list nodes take \_\_\_\_\_ time than pre-allocated array.

More

In a tree, we link the nodes in such a way that it \_\_\_\_\_ a linear structure.

Forms

Longest path from root node to farthest leaf node is called \_\_\_\_\_ of tree.

Depth

In which of the following function signatures, the value of variable "num" cannot be changed in function body?

int cube(int num)

Binary search algorithm cannot be applied to \_\_\_\_\_ .

sorted linked list

The expression  
if (! heap ->isFull() )  
check

Heap is not full

Which of the following is correct statement?

An AVL tree is identical to BST except height of the left and right subtree can differ by at most 1.

The easiest case of deleting a node from BST is the case in which the node to be deleted \_\_\_\_\_.

Is a leaf node

Every AVL is \_\_\_\_\_.

Binary Search Tree

\_\_\_\_\_ is when function is calling to itself.

Recursion

For a perfect binary tree of height  $h$ , having  $N$  nodes, the sum of heights of nodes is \_\_\_\_\_.

$N - h - 1$

The binary tree is the maximum level of its leaves (also called the depth).

Height

Which one of the following is NOT true regarding the skip list?

List Sh contains only the  $n$  special keys

Suppose we have been given the following data set for a Queue.

7 5 2 4

What will be the resultant Queue if we call enqueue(3) method?

Note that 7 is the front element whereas 4 is rear element of queue.

7 5 2 4 3

If there are N external nodes in a binary tree then what will be the no. of internal nodes in this binary tree?

N - 1

```
int htdiff = height(root->getLeft()) _____ height(root->getRight());
```

The above line of code is taken from AVL insert method.

Complete it by selecting an appropriate symbol.

-

Which of the following can be used to reverse a string value?

Stack

The next field in the last node of a singly-linked list is set to

\_\_\_\_\_.

'NULL

If there are N internal nodes in a binary tree then what will be the no. of external nodes in this binary tree?

N + 1

A hash function returns a \_\_\_\_\_ value.

Integer

The union operation is based on size or weight but the reducing the in-between links or path compression from \_\_\_\_\_ to the \_\_\_\_\_ is done by the find method.

nodes, root

If there are 23 external nodes in a binary tree then what will be the no. of internal nodes in this binary tree?

22

\_\_\_\_\_ in AVL is logarithmic.

Searching

In case of insertion of right inner node in BST,

we need to apply single left rotation to make it AVL tree.

Consider the following infix expression.

$7/8 + 9$

If one converts the above expression into postfix. What would be the resultant expression?

7 8 9 / +

What are the basic things associated with data structures?

All of the given

Which of the following properties are satisfied by Equivalence relationship?

Reflexive, Symmetric and Transitive

For compiler, a postfix expression is easier to evaluate than infix expression?

True

The expression  $DE + H^*$  is called \_\_\_\_\_.

Hybrid expression

Consider the following expression:

$$x - y * a + b / c$$

Which of the following is a correct equivalent expression for the above?

$$x - y * a + b / c$$

In the call by \_\_\_\_\_ methodology, a copy of the object is passed to the called function.

Value

There are four cases of rotation in an \_\_\_\_\_ tree.

AVL

In singly linked list which node will keep track of starting position of the list.

Next Node

The depth of a binary tree is

Maximum level of a leaf

What will be the value of root of an AVL and BST if built from the same data?

Root of AVL is square of root of BST

In 1990, Bill Pugh proposed an enhancement on linked lists and the new data structure was termed as

Skip list

Avl tree takes maximum \_\_\_\_\_ time to search an element.

1.44  $\log_2 n$

Can we store elements with different data types in a single array?

Yes

The \_\_\_\_\_ of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

Balance

What is the depth of any tree if the union operation is performed by height?

$O(\log N)$

If a hash function returns 4 for more than one value, this condition is called \_\_\_\_\_.

Collision

Which one of the following calling methods does not change the original value of the argument in the calling function?

Call by passing the value of the argument

Every AVL tree is a binary search tree.

TRUE

The difference between a "Binary Tree (BT)" and a "Binary Search Tree (BST)" is that,

In BST nodes are inserted based on the values they contain



The worst case of searching in binary search tree (BST) is:

When the data inserted in BST is sorted

Suppose there are 100 elements in an equivalence class, so initially there will be 100 trees. The collection of these trees is called

Forest

Factorial is an example of \_\_\_\_\_ function.

Recursive

Suppose a hash function returns 6 for the given value. At which index of array the value will be saved?

6

A binary tree of N nodes has \_\_\_\_\_ .

$\log_2 N$  levels

Every AVL is \_\_\_\_\_.

Binary Search Tree

If Ahmad is cousin of Ali and Ali is cousin of Asad then Ahmad is also cousin of Asad. This statement has the following property

Transitivity

What is the time complexity of binary search with iteration?

$O(\log n)$

A solution is said to be efficient if it solves the problem within its resource constraints i.e. hardware and time.

True

Searching an element in an AVL tree takes maximum \_\_\_\_\_ time (where n is number of nodes in AVL tree).

1.44  $\log_2 n$

Which of the following is not a form of expression?

Pastfix

Heap can be used to implement

Priority Queue

Sorting procedure normally takes \_\_\_\_\_ time.

$N \log N$

Consider we have performed the following operations on stack of size 5.

Push(10);

Push(20);

Push(30);

Pop();

Pop();

Push(40);

20

Which one of the following is NOT the property of equivalence relation?

Associative

What is the hash function used in linear probing?

$h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{table size}$

Recursive call of a function use \_\_\_\_\_ data structure.

**Stack**

Left, right, info, and parent are the operation of \_\_\_\_\_ data structure.

**Tree**

For Binary Search Tree, we call the findMax() method as \_\_\_\_\_.

**findMin(tree->getLeft())**

From Operating System point of view, the recursive function calls are made with the help of \_\_\_\_\_.

**Stack**

Which of the following is true regarding the maze generation?

**All of the given**

If there are 100 elements in an equivalence class then we will have \_\_\_\_\_ sets initially.

**100**

given the values are the array representation of heap: 12 23 26 31 34 44 56 64 78 100

What is the 5th smallest element in the given heap?

**34**

Which operation of queue data structure is used to get front element from the queue and then remove it from the queue?

**enqueue()**

\_\_\_\_\_ is an area in computer memory that is allocated dynamically.

**Heap**

The tree structure is a

**Non-linear data structure**

Binary Tree traversal can be performed with the help of \_\_\_\_\_ .

**Both Recursive and Non-recursive calls**

New items are added at the \_\_\_\_\_ of the stack.

**Top**

Leaf node of binary search tree contains \_\_\_\_\_.

**All of the given**

Suppose there are a set of fruits and a set of vegetables. Both sets are \_\_\_\_\_ sets.

**Disjoint**

Which of the following operations returns "most recently entered value" from the stack?

**Top**

back() method of list class is used to:

**Moves the "current" pointer to backward one element.**

Stack and Queue can be implemented using \_\_\_\_\_.

**Singly Link List**

The array in binary search is sub divided \_\_\_\_\_.

Untill a sublist is no more divisible

Which of the following heap method lowers the value of key at position 'p' by the amount 'delta'?

decreaseKey(p,delta)

What will be the postfix expression of following infix expression?  
 $A * B / C + D - E$

$A B + C D / + E -$

An efficient program executes faster and helps in \_\_\_\_\_ the usage of resources like memory and disk.

Maximizing

Array cells are \_\_\_\_\_ in computer memory.

Contiguous

Suppose there is an image of  $7 * 7$ , now we will have matrix of \_\_\_\_\_ rows and \_\_\_\_\_ columns.

7, 7

An array is a group of \_\_\_\_\_ memory locations.

Consecutive

for every process executing, the last part of the memory is for \_\_\_\_\_ of the program.

Heap

What will be the postfix expression of following infix expression?  
 $D + E * F / G$

DEF\*G/+

The expression  
 if (! heap ->isEmpty() )  
 checks

Heap is not empty

\_\_\_\_\_ is used for Reference variables in C++.

&

For searching a particular number in Binary Search Tree (if it is not present), the maximum number of comparisons will be \_\_\_\_\_ comparison at each level.

2

Suppose n is the number of nodes in a complete Binary Tree, then maximum steps required for a search operation are

$\log_2 (n+1) - 1$

If we use array to implement list, then there is an issue that it gives difficulty when:

We will increase its size

Consider the following function:

```
void test_a(int n)
{
 cout << n << " ";
 if (n>0)
 test_a(n-2);
}
```

}

What is printed by the call test\_a(4)?

4 2 0

Doubly Linked List always has \_\_\_\_\_ NULL pointer/s in a node.

Two

The percolateDown procedure will move the smaller value \_\_\_\_\_ and bigger value \_\_\_\_\_.

up, down

What will be the result of evaluation following expression?  
 $5 + 3 * 2 / (6 - 3)$

7

\_\_\_\_\_ only removes items in reverse order as they were entered.

Stack

Linked list always contains elements that can be described as,

Self-referential

What will be postfix expression of the following infix expression?  
 Infix Expression:  $a + b * c - d$

$abc * + d -$

In tree, the search operation is \_\_\_\_\_ as compared to the linked list.

Very fast

Each node in doubly link list has

1 pointer

Which of the is NOT true regarding the maze generation?

Remove a randomly chosen wall if the cells it separates are already in same set

The new operation in C++ for dynamically allocating memory returns,

Pointer to the memory it has allocated

Which of the following applications may use a stack?

Parentheses balancing program

Which one of the following statement is correct?

Array size is fixed once it is created

What will be result of following postfix expression?

1 2 3 \* + 2 -

4

When we compare recursive method calls and non-recursive method calls, following statement is true.

Recursion is implemented in the same way as other functioncalls are implemented

A queue where the dequeue operation does not depend upon FIFO, is called:

Priority queue



A Linear Data Structure is the data structure in which data elements are arranged in a sequence or a linear list. Which of the following is Non Linear Data Structure?

Binary Search Trees

\_\_\_\_\_ is the stack characteristic but \_\_\_\_\_ was implemented because of the size limitation of the array.

isEmpty(), isFull()

$Y = \&x[0];$

In the above statement, we get address of the first location of the array x and store it in y. Here "y" is:

lvalue

Each node in singly link list has,

1 pointer

If ahmad is boss of ehsan and ehsan is boss of umer then ahmad is also boss of umer.

The above mentioned relation is \_\_\_\_\_.

Transitive

Which of the following possible operation are performed on Table ADT?

Insert, Find, Remove

Which of the following operation returns but do not removes top value of the stack?

top

While implementing non-recursive traversal for Binary SearchTree, we need to implement \_\_\_\_\_ .

Stack

~BinarySearchTree() is a \_\_\_\_\_ .

Constructor

Which property of equivalence relation is satisfied if we say:  
Ahmad is cousin of Ali and Ali is also cousin of Ahmad

Symmetry

The total number of nodes on 5th level of a perfect binary tree are :

32

In C++, we place the class interface in \_\_\_\_\_ file.

.cpp

The variables which are destroyed automatically when a function's execution ends are:

Local variables defined inside function body

If the elements are 3, 5, 1, 6 in which order they will be removed from the queue one by one

3516

The.....of a binary tree is a maximum level of its leaves (also the depth).

height

.....of the an empty AVL tree is define to be

**Height**

Following is the keyword of c++

**Delete**

In the perspective of memory organization every process expecting the first part of memory is for.....of the program.

**code**

The.....of a node in a binary tree is define as the height of its left subtree minus height of its right subtree.

**balance**

If p is pointing to a node in an existing tree then info (p)means

**Return the content of node p.**

Following is a keyword of c++.....

**Delete**

Every.....tree is a binary search tree

**AVL**

If a function has recursive call as last statement it is known as.....

**Tail recursive**

During in a order traversal call if we found a node is null it means this node will satisfy following condition

it will not have left child

In binary search tree deleting a node is easy if it is a..... node

Root

Which of the following operation returns the content of binary tree node?

Info (p)

14. 1 2 6  
       5 8  
       3  
        4

1 2 3 4 5 6 8

in a complete binary tree the number of node of level 5 are

30

One should be careful about transient.....that are stored by reference I data structure

Objects

In a tree we link the node in such a way that it.....a linear structure

does not remain

the function call are made with the help of.....

stack

18.    15   25   45  
         35   75

15 35 25 75 45

A binary is said to be a.....binary tree if every non leaf node in binary tree has non empty left and right structure

Strictly

In which of the following function signature the value of variable num cannot be changed in function body

Binary search tree

Two common method of simulations are.....are.....

Time based simulation and event based simulation

Generally there are case (s) to delete a node from BST

1

When a function calls another function the parameters and address of the calling function are put in...

Stock

Binary search tree () is a.....

Destructor

We can calculate the.....of subtree by counting its level from the bottom

## Nodes

If there is a strictly complete binary tree of depth 3 the total number of nodes it will be

7

A .....model attempt to model real word phenomenon

Simulation

The process of getting the value of a variable using a pointer is called

## Dereferencing

When a function calling itself is called as.....

## Recursion

A.....is a tree in which every except possibly the last, is completely filled

## Strict binary tree

In AVL tree during insertion, a single rotation can fix the balance in cases \_\_\_\_\_ and 4.

1

If a node is inserted in outer side of a node in binary search tree then to make it AVL tree,

**We may have to apply single rotation**

A zigzag rotation is performed. In Left-Left case of rotation in AVL tree.

**False**

The principal benefit of a linked list over a conventional array is that the order of the linked items may be\_\_\_\_\_ from the order that the data items are stored in memory.

**Same**

Stack.push(15) will push 15 on \_\_\_\_\_.

**Top of the stack**

Suppose we have the following values to be inserted in constructing AVL tree,

10,13,15,5,7,8

Tell when first rotation will take place,

**After inserting the node 15**

All the objects created using \_\_\_\_\_ operator have to be explicitly destroyed using the delete operator.

**New**

During the execution of a process, operating system constructs four things for that process. Which of the following is not part of that process?

## Linked list

There are \_\_\_\_\_ cases of Rotation in AVL tree.

4

Which of the following line of code is incorrect?

Float \*I = float new (not confirmed)

In simple or singly linked list there is/are \_\_\_\_\_ pointer/s in each node.

One

Suppose there are three nodes tree with three numbers 14, 4, 15. Following is not a permutation, or combination for output them.

(4, 4, 15)

The type of expression in which operator succeeds its operands is \_\_\_\_\_ expression.

Postfix

The \_\_\_\_\_ of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

Balance

If both left and right nodes of a node are NULL then this type of node is called a \_\_\_\_\_ node.

Non leaf (not confirmed)

The function calls are made with the help of \_\_\_\_\_.

Stack



Which data structure is needed to convert infix expression to postfix expression?

**Stack**

In case of insertion of right inner node in BST,

**Left rotation then right to make AVL**

A BST generated from the data in ascending order is \_\_\_\_\_.

**Linear**

Which operation of queue data structure is used to get front element from the queue and then remove it from the queue?

**Remove ()**

In \_\_\_\_\_ various cells of memory are not located continuously.

**Linked list**

The \_\_\_\_\_ method of list will position the currentNode and lastCurrentNode at the start of the list.

**Start**

Which operation of queue data structure is used to insert an element into the Queue?

**Enqueuer ()**

From Operating System point of view, the recursive function calls are made with the help of \_\_\_\_\_.

**Queue**

We can make a lexicographic order of characters based on their \_\_\_\_\_.

**ASCII values**

What are the basic things associated with data structures?

**All of the above**

Consider the following push operations of a Stack:

Stack.push(4);

Stack.push(6);

Stack.push(5);

Stack.push(8);

If a user calls a pop() operation, then which value will be returned?

**8** (stack is last in first out)

~BinarySearchTree( ) is a \_\_\_\_\_.

**Destructor**

Array cells are \_\_\_\_\_ in computer memory.

**Contiguous**

Leaf node of binary search tree contains \_\_\_\_\_

**One Null Pointer**

dequeue() operation of queue data structure is used to \_\_\_\_\_

**remove an element from the front and return it**

The depth of a binary tree is

**Maximum level of a leaf**

Which of the following operation returns but do not removes top value of the stack?

**Top**

Which of the following data structure is linear type?

**All of above**

In doubly linked list a node consists of three parts:

**2 pointers and 1 object**

Last node in circular linked list contains

**No null pointer**

A\_\_\_\_\_ model attempts to model a real-world phenomenon

**Simulation**

Factorial is an example of \_\_\_\_\_function.

**Recursive**

The balance of a node in a binary tree is defined as the height of its \_\_\_\_\_ sub tree minus height of its right sub tree.

**Left**

In case of insertion of left outer node in BST,

**We first apply left rotation and then rotation to make it an AVL Tree.**

`int htdiff = height(root->getLeft()) height(root->getRight());` The above line of code is taken from AVL insert method. Complete it by selecting an appropriate symbol.

**Minus (-)**

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

**Two**

For making Binary Search Tree for Strings we need, \_\_\_\_\_ data type.

**Int**

A \_\_\_\_\_ model attempts to model a real-world phenomenon

**Simulation**

The lifetime of a transient object cannot exceed that of the application.

**True**

Which one of the following calling method does not change the original value of the argument in the calling function?

**Call by passing the value of the argument**

In level-order traversal for Binary Search Tree, we visit the nodes at each level before proceeding to the next level, in a \_\_\_\_\_ order.

**Left-to-right**

In doubly linked list a node consists of three parts:

1 pointer and 2 objects

Binary Search Tree violates the condition of AVL tree when any node has balance equal to

1 or -1

To search an element in AVL tree, it takes maximum  $1.88 \log_2 n$  time

True

HOW many cases of rotation are there in AVL tree?

4

\_\_\_\_\_ tree has been named after two persons Adelson-Velskii and Landis.

AVL

Which of the following is a nonlinear data structure?

Tree

AVL tree is nonlinear data structure.

True

\_\_\_\_\_ is when function is calling to itself.

Recursion

Which of the following function don't belongs to the stack class?

Crash ()

Which one is the core function call for the following function of calculating cube?

int cube(int& num)

**cube (num)**

Binary search algorithm cannot be applied to \_\_\_\_\_

**None of the given**

copy() method of list data structure \_\_\_\_\_

**Set one list to be a copy of another**

While implementing non-recursive traversal for Binary Search Tree, we need to implement

**Stack**

Following is true in case of using Recursive method calls

**The code becomes very short**

\_\_\_\_\_is a self-balancing tree.

**Binary Search Tree**

While implementing stack with an array and to achieve LIFO behavior, we used push and pop elements at

**The start of the array**

Generalized code written for a class is called

**Structure**

Army cells are \_\_\_\_\_ in computer memory

**Contiguous**

In singly linked list a node comprises of \_\_\_\_\_ field/s.

Two

Which of the following statement is false?

Pointers store the next data element of a list

Leaf node of binary search tree contains \_\_\_\_\_

One null Pointer

What will be the result of evaluating following expression?  
 $5+3*2/(6-3)$

7

Want and de-allocating memory for linked list nodes does take  
less time than pre-allocated array

MORE

Suppose you are writing a class for Node class and forgot to write  
the constructor of the class, then

Compiler will automatically generate a default constructor

Factorial is an example of \_\_\_\_\_function.

Recursive

A binary tree is said to be a \_\_\_\_\_ binary tree if every non-leaf  
node in a binary tree has non-empty left and right subtrees.

Strictly

Memory address is stored in

Pointer

In level-order traversal for Binary Search Tree, \_\_\_\_\_ data structure is used.

Queue

Which of the following is the correct option for priority Queue?

The type of queues that is FIFO i.e. the person who comes first should leave first.

AVL tree is linear data structure.

False

AVL tree is a binary search tree

True

The post order traversal of a binary tree is DEBFCA. Find out the pre order traversal

A

Λ

B C

Λ /

D E F

ABDCEF

In a tree, we link the nodes in such a way that it \_\_\_\_\_ a linear structure.

Does not remain



add(12) method of linked list class will:

Add 12 as value in linked list

Which of the following is the correct conversion of infix to postfix expression?

$Z+B-(D-H)/K$

ZB+DH-K/-

In singly linked list “next” field of node contains:

Address of next node

Which of the following line of code is incorrect?

The lifetime of a transient object can exceed that of the application which is accessing it.

A list is the collection of items of the \_\_\_\_\_

same type

back() method of List class is used to:

Moves the “current” pointer to backward one element.

Suppose you are writing a class for Node class and forgot to write the constructor of the class, then

\_\_\_\_\_ is the maximum height of the AVL tree.

$1.44\log_2 n$

length() method of List class is used to:

Return the length of the list

There are four cases of rotation in an \_\_\_\_\_ tree.

**AVL**

Allocating and de-allocating memory for linked list nodes does take \_\_\_\_\_ time than pre-allocated array.

**More**

\_\_\_\_\_ is utilized at the time of memory allocation in dynamic manner.

**Heap**

In which case of insertion we require double rotation to make the AVL tree balance.

**None**

The back() method decreases the value of variable current by \_\_\_\_\_.

**One**

In array list the worst case of removing an element is

**To remove the item from start**

Left, right, info, and parent are the operations of \_\_\_\_\_ data structure.

**Tree**

Which of the following is TRUE for search operations in a binary tree as compared to linked list or an array?

**It increase the number of comparison**

Which of the following is a correct statement?

An AVL tree is identical to BST except the height of left and right subtree can differ by at least 1

length() method of List class is used to:

Return the length of the list

Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function.

Return Address

Whenever we call a function, the compiler makes a \_\_\_\_\_ that it uses to fulfill this function call.

Stack

\_\_\_\_\_ is the major factor to see the efficiency of a program.

Time

\_\_\_\_\_ rule applies for evaluating operators of same precedence in an expression

None

The postfix form of the expression  $A + B * C$  and  $(A + B) * C$  will be \_\_\_\_\_.

Same

If we use array to implement list, then there is an issue that it gives difficulty when:

We will access value randomly

During in-order traversal using recursive calls, if we found a node is NULL. It means this node will satisfy following condition.

It will not have left child

Which one the following is more closer to AVL tree, (chose the best option)

Binary Search Tree

start() method of List class is used to:

Move the current pointer to first element

Consider the following infix expression.  $7/8 + 9$

If one converts the above expression into postfix, what would be the resultant expression?

$78/9+$

In internal memory organization of a process, there is some area of memory for static data that holds \_\_\_\_\_ variables.

Both Static and Global

In Left-Right case of rotation in \_\_\_\_\_ tree. A double rotation is performed.

AVL

If numbers 5, 222, 4, 48 are inserted in a queue, which one will be removed first?

5

Which one is not the property of binary tree?

Sibling node should be same parent (Not Sure)

Each node in singly linked list contains\_\_\_\_\_

**One Pointers**

What's wrong with following loop? while( (i < 10) && (i > 24)) {  
}

**The Condition is always false**

Which one of the following is TRUE about recursion?

1. Recursive function calls consume a lot of memory.
2. **Recursion extensively uses stack memory.**
3. Recursion extensively uses stack memory.
4. Iteration is more efficient than iteration.

Which of the following is not a data structure ?

1. Linked list
2. Queue
3. stack
4. **Memory cell**

“+” is a \_\_\_\_\_ operator.

1. Unary
2. **Binary**
3. Ternary
4. None of the above

When an executable program run, it is loaded in the memory and becomes a\_\_\_\_\_.

1. Thread
2. .h file
3. **Process**
4. None of the above

[www.vustudents.ning.com](http://www.vustudents.ning.com)

In array list the worst case of removing an element is

1. To remove an element from the end of the list
2. To remove an element from the middle of the list
3. To remove an element from the start of the list
4. We cannot remove element from an array list

The tree data structure is a

1. Linear data structure
2. Graphical data structure
3. Non-linear data structure (Not Confirm)
4. Data structure like queue

In\_\_\_\_\_, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node.

1. Linked list
2. doubly-link list
3. array
4. Structure

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s)

1. one
2. two
3. three
4. four

Which of the following operations returns top value of the stack?

1. push
2. **pop**
3. top
4. First

In the calling function, after the execution of the function called, the program continues its execution from the \_\_\_\_\_ after the function call.

1. Previous line
2. **Next line**
3. Beginning
4. None of the above

\_\_\_\_\_ only removes items in reverse order as they were entered.

1. Queue
2. Stack
3. Both of these
4. **None of these**

In the linked list implementation of the stack class, where does the push member function place the new entry on the linked list?

1. After all other entries that are greater than the new entry.
2. **At the head**
3. After all other entries that are smaller than the new entry.
4. At the tail

Which of the following is known as "Last-In, First-Out" or LIFO Data Structure?

1. Linked List
2. Stack
3. Queue
4. Tree

A queue is a \_\_\_\_\_ data structure, whereas a stack is a \_\_\_\_\_ data structure.

1. FIFO, LIFO

Each node in doubly link list has

1. 2
2. 3
3. 4

Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function

1. First argument
2. Return address
3. Last argument
4. none of the above

\_\_\_\_\_ is the stack characteristic but \_\_\_\_\_ was implemented because of the size limitation of the array.



1. isFull(),isEmpty()
2. pop(), push()
3. isEmpty() , isFull()

Stack and Queue can be implemented using \_\_\_\_\_,

1. Singly Link List
2. Binary Tree
3. Binary Search Tree
4. AVL Tree

The \_\_\_\_\_ method of list will position the currentNode and lastCurrentNode at the start of the list.

1. Remove
2. next
3. start
4. back

In array list the worst case of removing an element is

1. To remove an element from the end of the list
2. To remove an element from the middle of the list
3. To remove an element from the start of the list
4. We cannot remove element from an array list

STL is a \_\_\_\_\_ that is a part of the official standard of C++.

1. C program file
2. .h file

When an executable program run, it is loaded in the memory and becomes a\_\_\_\_\_.

1. Thread
2. .h file
3. Process
4. None of the above

Each node in doubly link list has,

1. pointer
2. pointers
3. pointers
4. pointer

A queue is a \_\_\_\_\_data structure, whereas a stack is a \_\_\_\_\_data structure.

1. FIFO, LIFO
2. LIFO,FIFO
3. both of these
4. none of these

Which one of the following is TRUE about recursion?

1. Recursive function calls consume a lot of memory.
2. Recursion extensively uses stack memory.
3. Recursion extensively uses stack memory.
4. Iteration is more efficient than iteration.

To create a \_\_\_\_\_ we link the last node with the first node in the list.

1. Double linked list
2. Circularly-linked list
3. Linked list
4. None of the above

\_\_\_\_\_ is the stack characteristic but \_\_\_\_\_ was implemented because of the size limitation of the array.

1. isFull(),isEmpty()
2. pop(), push()
3. isEmpty() , isFull()
4. push(),pop()

The principal benefit of a linked list over a conventional array is that the order of the linked items may be \_\_\_\_\_ from the order that the data items are stored in memory.

1. Same
2. Identical
3. Different
4. Equivalent

In \_\_\_\_\_ the 'next' returns false when it reaches to the last node due to the fact that the next field of the last node is set to NULL.

1. Circular linked list
2. Triple linked list
3. Singly linked list

4. None of the above

If numbers 5, 222, 4, 48 are inserted in a queue, which one will be removed first?

1. 48
2. 4
3. 333
4. 5

Which boolean expression indicates whether the numbers in two nodes (p and q) are the same. Assume that neither p nor q is null.

1.  $p == q$
2.  $p.data == q.data$
3.  $p.nextNode == q.nextNode$
4.  $p.data == q$

Local variables of a function are stored in,

1. Binary Search Tree
2. Stack
3. Queue
4. AVL Tree

A queue is a data structure where elements are

1. inserted at the front and removed from the back.
2. inserted and removed from the top.
3. inserted at the back and removed from the front.

4. inserted and removed from both ends.

A kind of expressions where the operator is present between two operands called \_\_\_\_\_ expressions.

1. Infix
2. Postfix
3. Prefix
4. None of the above

A template is a function or class that is written with a \_\_\_\_\_ data type.

1. Specific
2. Definite
3. Generic
4. None of the above.

Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function.

1. First argument
2. Return address
3. Last argument
4. None of the above

Which of the following can be used to reverse a string value,

1. Stack
2. Queue
3. Both of these
4. None of these

A queue is a \_\_\_\_\_ data structure, whereas a stack is a \_\_\_\_\_ data structure.

1. FIFO, LIFO
2. LIFO, FIFO
3. both of these
4. none of these

Stack and Queue can be implemented using \_\_\_\_\_,

1. Singly Link List
2. Binary Tree
3. Binary Search Tree
4. AVL Tree

Which of the following is not a data structure ?

1. Linked list
2. Stack
3. Queue
4. Memory cell

Compiler uses which one of the following in Function calls,

1. Stack

2. Queue
3. Binary Search Tree
4. AVL Tree

The principal benefit of a linked list over a conventional array is that the order of the linked items may be \_\_\_\_\_ from the order that the data items are stored in memory.

1. **Same**
2. Identical
3. Different
4. Equivalent

Doubly Linked List always has one NULL pointer.

1. True
2. **False**

We cannot remove items randomly from \_\_\_\_\_

1. Stack
2. Queue
3. Both of these
4. **None of these**

\_\_\_\_\_ is the maximum number of nodes that you can have on a stack-linked list ?

1. Zero
2.  $2n$  (where  $n$  is the number of nodes in linked list)
3. **Any Number**
4. None of these

“+” is a \_\_\_\_\_ operator.

1. Unary
2. Binary
3. Ternary
4. None of the above

In \_\_\_\_\_, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node.

1. Linked list
2. doubly-link list
3. array
4. structure

A queue is a data structure where elements are

1. inserted at the front and removed from the back.
2. inserted and removed from the top.
3. inserted at the back and removed from the front.
4. inserted and removed from both ends.

\_\_\_\_\_ is the stack characteristic but \_\_\_\_\_ was implemented because of the size limitation of the array.

1. isFull(),isEmpty()
2. pop(), push()
3. isEmpty() , isFull()
4. push(),pop()

The \_\_\_\_\_ method of list will position the currentNode and lastCurrentNode at the start of the list.



1. Remove
2. Next
3. Start
4. Back

Parameters in function call are passed using,

1. Stack
2. Queue
3. Binary Search Tree
4. AVL Tree

If numbers 5, 222, 4, 48 are inserted in a queue, which one will be removed first?

1. 48
2. 4
3. 333
4. 5

For a perfect binary tree of height 4, what will be the sum of highest of node

1. 31
2. 30
3. 27
4. 26

If Ahmed is cousin of Ali and Ali is cousin of Asad then Ahmed is also cousin of Asad. This statement has the following property

1. Reflexivity

2. Symmetry
3. Transitivity
4. All of the above

Which one of the following is NOT the property of equivalence relation?

1. Reflexive
2. Symmetric
3. Transitive
4. Associative

The main reason of using heap in priority queue is

1. Improve performance
2. Code readable
3. Less code
4. Heap can't be used in priority queues

The total numbers of nodes on 10th level of perfect binary tree are

1. 256
2. 512
3. 1024
4. Can't be determined

Suppose there are 100 elements in an equivalence class, so initially there will be 100 trees, the collection of these trees is called \_\_\_\_\_.

1. Cluster
2. Class
3. Forest
4. Bunch

The percolate Down procedure will move the smaller value and bigger value.

1. Left, right
2. Right, left
3. Down, up
4. Up, down

For a perfect binary tree of height  $h$ , having  $N$  nodes, the Sum of height of nodes is

1.  $N - h - 1$
2.  $N - 1$
3.  $N - 1 + h$
4.  $N - (h - 1)$

Which of the following method is helpful in creating the heap at once?

1. Insert
2. Add
3. Update
4. percolateDown

If ahmad is boss of ehsan and ehsan is boss of umer then ahmad is also boss of umer, the above-mentioned relation is\_\_\_\_\_ .

1. Reflexive
2. Symmetry
3. Transitive
4. None of given

If we want to find median of 50 elements, then after applying builtHeap method, how many times deleteMin method will be called?

1. 5
2. 25
3. 35
4. 50

Which of the following properties are satisfied by equivalence relationship?

1. Reflexive, symmetric
2. Reflexive, transitive
3. Symmetric, transitive
4. Reflexive, symmetric and transitive

Sorting procedure normally takes\_\_\_\_ time.

1.  $N \log N$
2.  $2N$
3.  $N * N * N$
4.  $N$

15. The Expression  
if ( ! heap->isFull() )  
Check

1. Heap is empty
2. Heap is full
3. Heap is not empty
4. Heap is not full

The Expression  
if ( ! heap->isEmpty() )  
Check

1. Heap is empty
2. Heap is full

- 3. Heap is not empty
- 4. Not a valid expression

Given the values are the array representation of heap: 12 23 26 31 34 44 56 64 78

100 If we perform 4 deleteMin operation, the last element deleted is \_\_\_\_\_ .

- 1. 31
- 2. 34
- 3. 44
- 4. 56

Which of the following heap method increase the value if key at position 'p' by the amount 'delta'?

- 1. increaseKey(p, delta)
- 2. decreaseKey(p, delta)
- 3. percolateDown(p, delta)
- 4. remove(p, delta)

Which of the following heap method lowers the value if key at position 'p' by the amount 'delta'?

- 1. increaseKey(p, delta)
- 2. decreaseKey(p, delta)
- 3. percolateDown(p, delta)
- 4. remove(p, delta)

Which property of equivalence relation is satisfied if we say:  
Ahmad R(is related to)Ahmad

- 1. Reflexivity
- 2. Symmetry
- 3. Transitivity
- 4. All of Above

Given the values are the array representation of heap; 12 23 26 31 34 44 56 64 78 100 What is the 5th smallest element in the given heap?

1. 31
2. 34
3. 44
4. 56

# AL-JUNAID INSTITUTE OF GROUP

## CS301-Data Structure

Solved MCQS for Mid terms papers

Solved by JUNAID MALIK and Team



AL-Junaaid Institute

Contact no: 03041659294



## LMS Handling

### Subject Enrollment

- 1) Online classes available  
HTML, CSS, Javascript  
Python, etc
- 2) Solved Graded Activities
  - Assignment's
  - Quizzes
  - GDB's
- 3) Solved Quiz File's
- 4) Short Note's
- 5) Past paper's &  
Current paper's

Website Link  
[vulmshelp.com](http://vulmshelp.com)

### Final project CS619

- 1) SRS  
(Software Requirement's  
Specification)
- 2) DD  
(Design Document)
- 3) Test phase + viva
- 4) Viva preparation
- 5) Final Deliverable

Gmail  
[junaidfazal08@gmail.com](mailto:junaidfazal08@gmail.com)

# **AL-JUNAID TECH INSTITUTE**

1. Consider the following infix expression.  $7/8+9$

If one converts the above expression into postfix, what would be the resultant expression?

- a.  $789/+$
  - b.  $78/+9$
  - c.  $/78+9$
  - d.  $78/9+$
2. Suppose there are 100 elements in an equivalence classes so initially there will be 100 tree. The collection of tree is called \_
- a. Cluster
  - b. Class
  - c. Forest
  - d. Bunch
3. For a perfect binary tree of height  $h$ , having  $N$  nodes the sum of height of nodes is
- a.  $N-h-1$
  - b.  $N-1$
  - c.  $N-1+h$
  - d.  $-(h-1)$
4. Sorting procedure normally takes \_\_\_\_\_ time
- a.  $N \log N$
  - b.  $2N$
  - c.  $N*N*N$
  - d.  $N$
5. there are 100 elements in an equivalence classes then will have \_\_\_\_\_ state initially
- a. 50
  - b. 100
  - c. 1000
  - d. 80
6. \_\_\_\_\_ objects (objects accessed by pointers) are called anonymous objects.
- a. Private
  - b. Nameless
  - c. Friend
  - d. Public



# **AL-JUNAID TECH INSTITUTE**

7. \_\_\_\_\_ is a self-balancing tree.
- a. AVL
  - b. Binary Tree
  - c. Binary Search Tree
  - d. ALV
8. Which of the following operation returns but do not removes top value of the stack?
- a. Push
  - b. Pop
  - c. Top
  - d. First
9. `Int htdiff = height(root->getLeft()) _____ height(root->getRight());` The above line of code is taken from AVL insert method. Complete it by selecting an appropriate symbol.
- a. -
  - b. +
  - c. /
  - d. \*
10. Which operation of the queue data structure is used to insert an element into the Queue?
- a. Enqueue()
  - b. Dequeue()
  - c. Fornt()
  - d. Remove()
11. For making binary search tree for strings we need, \_\_\_\_\_ data type.
- a. Char
  - b. Int
  - c. Float
  - d. Double
12. Local variables defined inside function body are \_\_\_\_\_ automatically at the end of function execution.
- a. Created
  - b. Destroyed
  - c. Incremented
  - d. Decrementd

# **AL-JUNAID TECH INSTITUTE**

9. If a node has three fields, then this node will be related to
- Linked list
  - Doubly linked list**
  - Circular linked list
  - All of the given
10. \_\_\_\_\_ is an area in computer memory that is allocated dynamically.
- Heap**
  - Stack
  - Queue
  - Linked list
11. Linked list use \_\_\_\_\_ to store data.
- Array
  - 2-D Array
  - Variables**
  - Linked Memory
12. The lifetime of a transient object cannot exceed that of the application.
- True**
  - False
  - In some cases
  - None of the given
13. To represent hierarchical relationship between elements, which data structure is suitable?
- Dequeue
  - Priority
  - Stack
  - Tree**
14. Local variables of a function are stored in,
- Binary search tree
  - Stack**
  - Queue
  - AVL tree
15. In-order traversal method traverses the data in
- Non sorted order
  - Random order
  - Sorted order**

# **AL-JUNAID TECH INSTITUTE**

- d. None of the given
16. Dequeue() operation of queue data structure is used to \_\_\_\_\_
- a. Get an element from the front of the queue
  - b. Remove an element from the front and return it**
  - c. Insert an element at the front
  - d. Insert an element at the back
17. In a tree, we link the nodes in such a way that it \_\_\_\_\_ a linear structure.
- a. Does not remain**
  - b. Forms
  - c. Reverses
  - d. Remains
18. Factorial is an example of \_\_\_\_\_ function.
- a. Recursive**
  - b. Non-recursive
  - c. Cube
  - d. Log
19. All the objects created using \_\_\_\_\_ operator have to be explicitly destroyed using delete operator.
- a. New**
  - b. Delete
  - c. Del
  - d. Create
20. In the calling function, after the execution of the function called, the program continues its execution from the \_\_\_\_\_ after the function call.
- a. Previous line
  - b. Next line**
  - c. Beginning
  - d. None of the above
21. Last node in circular linked list contains
- a. At least one null pointer
  - b. No null pointer**
  - c. Maximum two null pointers
  - d. None of the given

# **AL-JUNAID TECH INSTITUTE**

22. One should be careful about transient \_\_\_\_\_ that are stored by reference in data structures.
- a. Objects
  - b. Stack
  - c. Function
  - d. Tree
23. Binary search tree violates the condition of AVL tree when any node has balance equal to
- a. 2 or -2
  - b. 1 or -1
  - c. 0
  - d. None of the options
24. Security of data is the main usage of AVL tree.
- a. True
  - b. False
  - c. In some cases
  - d. None of the given
25. A binary tree whose every node has either zero or two children is called \_\_\_\_\_.
- a. Complete binary tree
  - b. Binary search tree
  - c. Strictly binary tree
  - d. None of above
26. In internal memory organization of a process, there is some area of memory for static data that holds \_\_\_\_\_ variables.
- a. Static
  - b. Global
  - c. Not static neither global
  - d. Both static and global
27. Following is true in case of using recursive method calls.
- a. The code becomes very long
  - b. There is no effect on length of code
  - c. The code becomes very short
  - d. Code becomes very easy to understand
28. In which traversal method, the recursive calls can be used to traverse a binary tree?
- a. In preorder traversal only

# **AL-JUNAID TECH INSTITUTE**

- b. In inorder traversal only
  - c. In postorder traversal only
  - d. All of the given options
29. If we use doubly linked list to implement list then there is an issue of
- a. Next pointer of first node and pre pointer of last node are NULL
  - b. Next pointer of first node and next pointer of last node are NULL
  - c. Pre pointer of first node and next pointer of last node are NULL
  - d. Pre pointer of first node and pre pointer of last node are NULL
30. In case of insertion of left outer node in BST,
- a. We apply single right rotation to make it AVL tree.
  - b. We apply single left rotation to make it AVL tree.
  - c. We first apply right rotation and then left rotation to make it AVL tree.
  - d. We first apply left rotation and then right rotation to make it AVL tree.
31. Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function.
- a. First argument
  - b. Return address
  - c. Last argument
  - d. None of the above
32. Suppose a stack class has been defined using template. Now, we want to declare a Stack object of an int type. What will be the correct syntax?
- a. <int>Stack stack;
  - b. Stack<int> stack;
  - c. Stack int stack;
  - d. Int Stack stack;
33. In node class one field is an integer data and other field will be \_\_\_\_\_
- a. Pointer to class
  - b. Pointer to node
  - c. Pointer to integer
  - d. None of given options

# **AL-JUNAID TECH INSTITUTE**

34. If there are 100,000 unique members (nodes) stored in a complete binary tree, the tree will have \_\_\_\_\_ levels.
- a. 10
  - b. 20**
  - c. 30
  - d. 40
35. ~BinarySearchTree() is a \_\_\_\_\_.
- a. Constructor
  - b. Destructor**
  - c. Switch case
  - d. Template method call
36. \_\_\_\_\_ is utilized at the time of memory allocation in dynamic manner.
- a. Stack
  - b. Queue
  - c. Heap**
  - d. All of the given
37. How many cases of rotation are there in AVL tree?
- a. 2
  - b. 4**
  - c. 6
  - d. 8
38. A list is the collection of items of the \_\_\_\_\_.
- a. May be of same or may be of different type
  - b. Different type
  - c. Same type**
  - d. None of the above
39. `Int * i =`  
`new int[10];`  
Above given code will:
- a. Create an integer having value 10
  - b. Allocate memory for 9 integers
  - c. Allocate memory for 10 integers**
  - d. Create 10 pointers of integer type
40. The depth of a binary tree is
- a. Total number of nodes in the tree

# **AL-JUNAID TECH INSTITUTE**

- b. Number of leaf nodes in the tree
  - c. Number of non-leaf nodes in the tree
  - d. Maximum level of a leaf
41. A \_\_\_\_\_ model attempts to model a real-world phenomenon
- a. Physical
  - b. Logical
  - c. Simulation
  - d. Conceptual
42. In level-order traversal for binary search tree, we visit the nodes at each level before proceeding to the next level, in a \_\_\_\_\_ order.
- a. Right-to-left
  - b. Left-to-right
  - c. Top-to-bottom
  - d. Bottom-to-top
43. The balance of a node in a binary tree is defined as the height of its \_\_\_\_\_ sub tree minus height of its right sub tree.
- a. Right
  - b. Left
  - c. Upper
  - d. Lower
44. Allocating and de-allocating memory for linked list nodes does take \_\_\_\_\_ time than pre-allocated array.
- a. Less
  - b. More
  - c. Equal
  - d. No
45. Consider we have performed the following operations on a stack of size 5. Push(10);  
Push(20);  
Push(30);  
Pop();  
Pop();  
Pop();  
Push(40);  
Push(50);

# **AL-JUNAID TECH INSTITUTE**

Pop();

After the completion of all operation, the top element in stack is

\_\_\_\_\_.

- a. 10
  - b. 20
  - c. 40**
  - d. 50
46. Each operator in a postfix expression refers to the previous  
\_\_\_\_\_ operand(s).
- a. One
  - b. Two**
  - c. Three
  - d. Four
47. In doubly linked list a node consists of three parts:
- a. 1 pointer and 2 objects
  - b. 2 pointers and 1 object**
  - c. 3 objects
  - d. 3 pointers
48. For reference variables, \_\_\_\_\_ sign is used.
- a. Ampersand**
  - b. Asterisk
  - c. Sigma
  - d. Dollar
49. Which of the following data structure is linear type?
- a. Stack
  - b. List
  - c. Queue
  - d. All of the above**
50. The - - is a decrement operator in C++ that decreases the value of  
the operand by \_\_\_\_\_.
- a. One**
  - b. Two
  - c. Three
  - d. Four
51. Which of the following statement is correct for the variable  
“current- -“?



# **AL-JUNAID TECH INSTITUTE**

- a.  $\text{Current} = \text{current} + 1$
  - b.  **$\text{Current} = \text{current} - 1$**
  - c.  $\text{Current} = \text{current} - 2$
  - d.  $\text{Current} - 1 = \text{current}$
52. When an executable program runs, it is loaded in the computer memory and becomes a \_\_\_\_\_.
- a. Thread
  - b. .h file
  - c. **Process**
  - d. None of the above
53. start() method of list class is used to:
- a. **Moves the “current” pointer to very first element**
  - b. Moves the “current” pointer to very last element
  - c. Moves the “current” pointer to one step after the first element of the array
  - d. Moves the “current” pointer to one step before the first element of the array
54.  $a * (b + c) - d$  is an example of \_\_\_\_\_ expression.
- a. **Infix**
  - b. Prefix
  - c. Postfix
  - d. Alfix
55. In a program a reference variable, say x, can be declared as
- a. **`int &x;`**
  - b. `int *x;`
  - c. `int x;`
  - d. none of the given options
56. We allocate memory dynamically by using \_\_\_\_\_ operator.
- a. This
  - b. **New**
  - c. Increment
  - d. Decrement
57. Get(?) method of list class is used to:
- a. Get element from the last position
  - b. Get element from the first position

# **AL-JUNAID TECH INSTITUTE**

- c. Get element from the middle position  
d. Get element at the given position
58. STL is a \_\_\_\_\_ that is a part of the official standard of C++.
- a. C program file
  - b. .h file
  - c. .cpp file
  - d. Library
59. There are \_\_\_\_\_ cases for deleting a node from binary search tree.
- a. 1
  - b. 2
  - c. 3
  - d. 4
60. In a list, tail() method of current pointer \_\_\_\_\_
- a. Returns the last element of the “current” pointer
  - b. Moves the “current” pointer to the very first element
  - c. Moves the “current” pointer to the very last element
  - d. Returns the first element of the “current” pointer
61. Suppose we have been given the following data set for a queue:  
37524  
What will be the resultant queue if we call a front() method?
- a. 7524
  - b. 37524
  - c. 75243
  - d. 5 2 4
62. Suppose we have been given the following data set for a queue:  
7524  
What will be the resultant queue if we call enqueue(3) method? Note that 7 is the front element whereas 4 is rear element of queue.
- a. 7524
  - b. 37524
  - c. 75243
  - d. 5243
63. Maximum time that an insertion operation can take in AVL tree is \_\_\_\_\_. Here log stands for log to the base of 2.
- a. Log (n)

# **AL-JUNAID TECH INSTITUTE**

- b.  $1.44 \log(n)$
  - c.  $1.66 \log(n)$
  - d.  $\log(n+1)$
64. \_\_\_\_\_ is when function is calling to itself.
- a. Loop
  - b. Recursion
  - c. Iteration
  - d. Nested loop
65. Left, right, info, and parent are the operations of \_\_\_\_\_ data structure.
- a. Stack
  - b. Tree
  - c. Queue
  - d. Linked list
66. Following is a keyword of C++
- a. Del
  - b. Delete
  - c. Remove
  - d. Eliminate
67. The expression  $DE+H^*$  is called \_\_\_\_\_
- a. Prefix expression
  - b. Infix expression
  - c. Postfix expression
  - d. Hybrid expression
68. A software solution is said to be efficient if it solves the problem \_\_\_\_\_.
- a. By using some extra resources
  - b. Within no time
  - c. By consuming more hardware resources
  - d. Within its resources constraints
69. Each node in a singly linked list contains two fields, one field called data field while other field contains:
- a. Pointer to an integer
  - b. Pointer to character

# **AL-JUNAID TECH INSTITUTE**

- c. Pointer to next node
- d. Pointer to class
70. Each node in singly linked list contains \_\_\_\_\_
- a. One pointer
- b. Two pointers
- c. No pointer
- d. NULL pointer
71. Every AVL tree is a binary search tree.
- a. True
- b. False
- c. Not in some cases
- d. None of the given
72. In singly linked list “next” field of node contains:
- a. Address of next node
- b. Object of next node
- c. Object of current node
- d. Address of head node
73. Two common models of simulation are \_\_\_\_\_ and \_\_\_\_\_.
- a. Circuit-based simulation and Event- based simulation
- b. Circuit-based simulation and Time-based simulation
- c. Time-based simulation and Event- based simulation
- d. None of the above
74. Stack and Queue can be implements using \_\_\_\_\_.
- a. Singly Link List
- b. Binary Tree
- c. Binary search Tree
- d. AVL Tree
75. What are the basic things associated with data structures?
- a. Space for each data item it stores
- b. Time to perform each basic operation
- c. Programming effort
- d. All of the above
76. In AVL tree insertion occurs on the inside in case \_\_\_\_\_ and 3 which a single rotation cannot fix.
- a. 1
- b. 2
- c. 4

# **AL-JUNAID TECH INSTITUTE**

- d. 5
77. add(12) method of linked list class will
- a. Add 12 nodes in linked list
  - b. Add 12 pointers in linked list
  - c. Add 12 as value in linked list
  - d. Add 12 values in linked list
78. Which of the following is a non- linear data structure?
- a. Stack
  - b. Queue
  - c. Tree
  - d. Linked list
79. Which data structure allows deleting data elements from front and inserting at rear?
- a. Stacks
  - b. Queues
  - c. Deques
  - d. Binary search tree
80. Array cells are \_\_\_\_\_ in computer memory.
- a. Contiguous
  - b. Random
  - c. Store in multiple Variables
  - d. Store in multiple functions
81. A kind of expression where the operator is present between two operands called \_\_\_\_\_ expressions.
- a. Infix
  - b. Postfix
  - c. Prefix
  - d. None of the above
82. back() method of list class is used to
- a. Moves the "current" pointer to backward one element.
  - b. Moves the "current" pointer to backward two element.
  - c. Moves the "current" pointer to backward three element.
  - d. Moves the "current" pointer to backward four element.
83. next() method of List class is used to:
- a. Moves the Current position backward one element

# **AL-JUNAID TECH INSTITUTE**

- b. Moves the "Current" pointer to two steps after the last element of the array
  - c. Moves the Current position forward one element
  - d. Moves the "Current" pointer to two steps before the last element of the array
84. \_\_\_\_\_ tree has been named after two persons Adelson-Velskii and Landis.
- a. Binary
  - b. Black
  - c. AVL
  - d. VLA
85. There are four cases of rotation in an \_\_\_\_\_ tree.
- a. ELV
  - b. EVL
  - c. AVL
  - d. ALV
86. \_\_\_\_\_ is used for reference variable in C++.
- a. !
  - b. @
  - c. #
  - d. &
87. A \_\_\_\_\_ is a tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- a. Strict binary tree
  - b. Full binary tree
  - c. Perfect binary tree
  - d. Complete binary tree
88. A queue is a data structure where elements are
- a. Inserted at the front and removed from the back
  - b. Inserted and removed from the top
  - c. Inserted at the back and removed from the front
  - d. Inserted and removed from both ends
89. Longest path from root node to farthest leaf node is called \_\_\_\_\_ of tree.
- a. Level
  - b. Length
  - c. Depth

# **AL-JUNAID TECH INSTITUTE**

- d. Node level
90. New items are added at the \_\_\_\_\_ of the stack.
- a. Bottom
  - b. Middle
  - c. Center
  - d. Top
91. Length() method of list class is used to:
- a. Return the length of the array
  - b. Return the length of the list
  - c. Return the length of empty part of the array
  - d. Return the length of empty part of the list
92. The function calls are made with the help of \_\_\_\_\_.
- a. Stack
  - b. Heap
  - c. Dynamic memory
  - d. External memory
93. An efficient program execute faster and helps in \_\_\_\_\_ the usage of resources like memory and disk
- a. Maximizing
  - b. Minimizing
  - c. Equalizing
  - d. None of the given
94. Generalized code written for a class is called:
- a. Function
  - b. Template
  - c. Structure
  - d. Stack
95. "set()" method of list class is used to:
- a. Set the value of Pointer
  - b. Set the value of Null Nodes
  - c. Set the value of objects
  - d. Set the value of Value
96. In C++, we place the class interface in \_\_\_\_ file.
- a. .cpp
  - b. .cppp
  - c. .h

# **AL-JUNAID TECH INSTITUTE**

- d. .hh
97. Every \_\_\_\_\_ tree is a binary search tree.
- a. AVL
  - b. binary
  - c. big
  - d. small
98. If both left and right nodes of a node are NULL then this type of node is called \_\_\_\_\_ node.
- a. Non leaf
  - b. internal
  - c. inner
  - d. leaf
99. Which of the following function don't belongs to the stack class?
- a. push()
  - b. pop()
  - c. crash()
  - d. top()
100. For a complete binary tree, the depth is calculated as \_\_\_\_\_
- a.  $\log_2(\text{number of nodes}+1)-1$
  - b.  $\log_2(\text{number of nodes} \times 1)+1$
  - c.  $\log_2(\text{number of nodes}-1)-1$
  - d.  $\log_2(\text{number of nodes}-1)+1$
101. \_\_\_\_\_ only removes items in reverse order as they were entered
- a. Queue
  - b. Stack
  - c. Both of these
  - d. None of these
102. What will be the postfix expression of following infix expression?  $D+E \times F/G$
- a.  $DE \times F/G$
  - b.  $DE+F \times G/$
  - c.  $DEF \times /+$
  - d.  $DE+FG \times /$
103. The \_\_\_\_\_ of a node in a binary tree is defined as the height of its \_\_\_\_\_ sub tree minus height of its right sub tree.
- a. Height
  - b. Balance



# **AL-JUNAID TECH INSTITUTE**

- c. Width
  - d. None
104. The back() method decreases the value of Variable current by\_\_\_\_\_
- a. Four
  - b. Three
  - c. Two
  - d. One
105. In doubly linked list there is/are \_\_\_\_\_ pointer/s in each node.
- a. One
  - b. Two
  - c. Three
  - d. Four
106. Suppose we have the following values to be inserted in constructing AVL tree,  
10, 13, 15, 5, 7, 8  
Tell when first rotation will take place.
- a. After inserting node 13
  - b. After inserting node 15
  - c. After inserting node 5
  - d. After inserting node 7
107. Binary search algorithm cannot be applied to \_\_\_\_\_
- a. Sorted linked list
  - b. sorted binary trees
  - c. sorted linear array
  - d. None of given option
108. During the execution of a process operating system constructs focus things for that process which of the following is not part of that process
- a. A section for static data including global variable
  - b. Stack
  - c. Heap
  - d. Linked list
109. Which one of the following method does not change the original value of the argument in the calling function?
- a. Call by passing reference of the argument
  - b. Call by passing the address of the argument

# **AL-JUNAID TECH INSTITUTE**

- c. Call by passing the value of the argument
- d. None of the given options
110. \_\_\_\_\_ is the major factor to see the efficiency of a program.
- a. Quality
- b. Time
- c. Correctness
- d. None of the given
111. While implementing stack with an array and to achieve LIFO behavior, we used push and pop elements at \_\_\_\_\_.
- a. The start of the array
- b. The end of the array
- c. The mid of the array
- d. At least one position before array starting index.
112. A template is a function or class that is written with a \_\_\_\_\_.
- a. Specific
- b. Definite
- c. Generic
- d. None of the above
113. In the linked list implementation of the stack class, where does the push member function places the new entry on the linked list?
- a. After all other entries that are greater than the new entry
- b. At the head
- c. After all other entries that are smaller than the new entry
- d. At the tail
114. The next field in the last node in a singly-linked list is set to \_\_\_\_\_.
- a. 0
- b. 1
- c. NULL
- d. False
115. Consider the linked list having data [6, 72, 35, 65, 25] stored in it. While current pointer is pointing memory location having 72 stored in it. After calling remove() function on the following linked list current point will point to memory location having value?
- a. 6

# **AL-JUNAID TECH INSTITUTE**

- b. 35
  - c. 65
  - d. 25
116. If we write functions for recursive and non recursive inorder traversal method of BST, what will be the difference between its functions prototypes?
- a. Different return types
  - b. Different function names
  - c. Different arguments list
  - d. Nothing will be different
117. In singly linked list a node comprises of \_\_\_\_\_ field/s.
- a. One
  - b. Two
  - c. Three
  - d. Four
118. Doubly linked list always has \_\_\_\_\_ NULL pointers in a node.
- a. One
  - b. Two
  - c. Three
  - d. Four
119. A BST generated from the data in ascending order is \_\_\_\_\_.
- a. Linear
  - b. Nonlinear
  - c. Balanced
  - d. Un sorted
120. Linked list is generally considered an example of \_\_\_\_\_ type of memory location.
- a. Static
  - b. Compile time
  - c. Dynamic
  - d. None of given options
121. What's wrong with following loop?
- ```
while((i < 10) && (i > 24)){  
}
```
- a. The logical operator && cannot be used in a test condition
 - b. The while loop is an exit-condition loop
 - c. The test condition is always false

AL-JUNAID TECH INSTITUTE

- d. The test condition is always true
122. Making the tree unbalanced, it violates the _____ rule.
- a. Linked list
 - b. Stack
 - c. AVL**
 - d. Queue
123. copy() method of list data structure _____
- a. copy first item of list
 - b. set one list to be a copy of another**
 - c. copy last item of list
 - d. copy any item of list
124. Deleting a _____ node in BST is a _____ case.
- a. Root, simplest
 - b. Left child, simplest
 - c. Right child, simplest
 - d. Leaf, simplest**
125. We can not remove items randomly from _____
- a. Stack
 - b. Queue
 - c. Both of these**
 - d. None of these
126. NULL is an invalid address and _____.
- a. Accessible
 - b. Inaccessible**
 - c. Points to the start point of the list
 - d. Points to the last point of the list
127. A stack carries _____ behavior.
- a. FIFO
 - b. LIFO**
 - c. AVCO
 - d. FEFO
128. The tree data structure is a
- a. Linear data structure
 - b. Non-linear data structure**
 - c. Graphical data structure
 - d. Data structure like queue

AL-JUNAID TECH INSTITUTE

129. Can we store elements with different data types in a single array?
- a. Yes
 - b. No**
 - c. In some cases
 - d. None of given
130. The lifetime of a transient object can exceed that of application which is accessing it.
- a. True
 - b. False**
 - c. In some cases
 - d. None of the given
131. In a complete binary tree, for 25000 nodes the depth will be _____.
- a. 13
 - b. 14**
 - c. 15
 - d. 16
132. The smallest value element in a binary search tree(Each node with left and right pointer) lies at
- a. Root Node
 - b. Left Child of Root
 - c. Right Most Node
 - d. Left Most Node**
133. if we use array to implement list, then there is an issue that is gives difficulty when
- a. We will access values randomly
 - b. We will remove data from it
 - c. We will increase its size
 - d. We will decrease its size
134. if there are ____ nodes in an avl tree its levels will be roughly as $\log_2(10 \text{ million})$
- a. 100 million
 - b. 10 million**
 - c. 5 million

AL-JUNAID TECH INSTITUTE

- d. 2 million
135. Which one is the correct function call for the following function of calculatincube?
- ```
int cube(int&num)
{
.
.
.
}
```
- a. Cube(&num)  
b. Cube(&&num)  
c. Cube(\*num)  
d. Cube(num)
136. Which of the following is the correct option for priority Queue?
- a. The type of Queue that is not FIFO i.e the person who comes first may not leave first  
b. The type of Queue that is not FIFO i.e the person who comes first should leave first  
c. The type of Queue that is not FIFO i.e the person who comes first should leave first  
d. The type of Queue that is not FIFO i.e the person who comes first may not leave first
137. For String-based Binary Search Tree, We use ASCII values of characters for comparing among letters. This method is known as \_\_\_\_\_
- a. Lexicographic order  
b. Alphabet coding procedure  
c. Asymmetric technique  
d. heap-based approach
138. Elements in a queue data structure are added from \_\_\_\_\_ and remove from \_\_\_\_\_.
- a. Rear end. front end  
b. Front end. Rear end  
c. Front end  
d. rear end
139. If numbers 5,222,4,48 are inserted in a queue. Which one will be remove first?
- a. 48

# **AL-JUNAID TECH INSTITUTE**

- b. 4
  - c. 222
  - d. 5
140. A zigzag rotation is performed. In Left-Left case of rotation in AVL tree.
- a. True
  - b. False
  - c. In some cases
  - d. None of the above
141. During deletion of node from BST. if we found this node don't have in- order successor and predecessor. it means this node is \_\_\_\_\_
- a. Left Most node in the binary search tree
  - b. Right most node in the binary search tree
  - c. Root node
  - d. None of the given option
142. \_\_\_\_\_ rule applies for evaluating operators of same precedence in an expression
- a. right to left
  - b. Cascading
  - c. Associative
  - d. None of the above
143. What will be result of following postfix expression?  
123\*+2-
- a. 3
  - b. 4
  - c. 5
  - d. 10
144. The main use of AVL tree is:
- a. Searching of data
  - b. Storing of data
  - c. Insertion of data
  - d. Security of data
145.  $y = \&x[0];$   
In the above statement, we get the address of the first location of the array x and store it in y.  
Here "y" is:
- a. rvalue

# **AL-JUNAID TECH INSTITUTE**

- b. xvalue
  - c. lvalue**
  - d. zvalue
146. what will be postfix expression of the following infix expression? Infix expression  $a+b*c-d$
- a.  $ab+c*d-$
  - b.  $abc*+d-$**
  - c.  $abcd+*-$
  - d.  $abc+*d-$
147. Which of the following is the correct conversion of infix to postfix expression?  $Z+B-(D-H)/K$
- a.  $ZB+D-H-K/$
  - b.  $ZB+DH-K-/$
  - c.  $ZB+DH-K/-$**
  - d.  $ZB+DHK--/$
148. Leaf node of binary search tree contains \_\_\_\_\_.
- a. One null pointer
  - b. Three null pointers
  - c. Two null pointers**
  - d. All of the given
149. A tree is an AVL tree if
- a. Any one node fulfills the AVL condition**
  - b. At least half of the nodes fulfill the AVL condition
  - c. All the nodes fulfill the AVL condition
  - d. None of the given options
150. The \_\_\_\_\_ of a binary tree is the maximum level of its leaves (also called the depth).
- a. Level
  - b. Width
  - c. Height**
  - d. None of the above
151. Memory address is stored in
- a. Address operator
  - b. Reference
  - c. Pointer**
  - d. All of the given



# **AL-JUNAID TECH INSTITUTE**

152. A binary tree is said to be a \_\_\_\_\_ binary tree if every non-leaf node in a binary tree has non-empty left and right sub trees.
- a. Complete
  - b. Strictly**
  - c. AVL
  - d. Perfect
153. “+” is a \_\_\_\_\_ operator.
- a. Unary
  - b. Binary**
  - c. Ternary
  - d. None of the above
154. Which of the following a nonlinear tree
- a. Stack
  - b. Que
  - c. Tree**
  - d. Linked List
155. For every Process execute the last part of the Memory is for \_\_\_\_\_ of the Program
- a. Data
  - b. Code
  - c. Stack
  - d. Heap**
156. While implement non recursive traversal for binary search tree we need to implement \_\_\_\_\_
- a. Que
  - b. Stack**
  - c. Min heap
  - d. Max Heap
157. Suppose we have the following values to be inserted in constructing AVL tree,  
20,23,25,10, 12,13  
Tell when first rotation will take place.
- a. After increasing Node 25**
  - b. After increasing Node 23
  - c. After increasing Node 10
  - d. After increasing Node 12

# **AL-JUNAID TECH INSTITUTE**

158. If a tree has a 50 Node then the total linked in the tree will be
- a. 55
  - b. 51
  - c. 50
  - d. 49
159. The percolate down procedure will move the smaller value \_\_\_\_\_ and Bigger value \_\_\_\_\_
- a. Left, right
  - b. Right, left
  - c. Down, up
  - d. Up, down
160. Circular Linked List Solve the problem of \_\_\_\_\_ pointer/method of the doubly link list.
- a) Remove
  - b) Null
  - c) Add
  - d) Find
161. isEmpty() method of stack class will return true when.
- a) Stack is full
  - b) Stack is Partially
  - c) Stack is not empty or Null
  - d) Stack is empty
162. "New int[11]" will allocate memory for \_\_\_\_\_ integers.
- a) 13
  - b) 12
  - c) 11
  - d) 10
163. For searching a particular number in Binary Search Tree (if it is not present), the maximum number of comparisons will be \_\_\_\_\_ comparison at each level.
- a) 1
  - b) 2
  - c) 3
  - d) 4
164. In singly linked list which node will keep track of starting position of the list.

# **AL-JUNAID TECH INSTITUTE**

a) Next Node

b) Previous Node

c) Head Node

d) Last Node

165. If we singly linked list to implement list, then there is an issue that it gives difficulty when we:

a) Move forward in the list

b) Move backward in the list

c) We will increase its size

d) We will decrease its size

166. Can we store elements with different data types in a single array?

a) Yes

b) No

c) In some cases

d) None of the given

167. During union by size method, which data structure is used to improve the balancing of tree?

a) Array

b) Stack

c) Linked List

d) Tree

168. The difference between a "Binary Tree (BT)" and a "Binary Search Tree (BST)" is that,

a) A BST has two children per node whereas a BT can have none, one or two children per node

b) In BST nodes are inserted based on the values they contain

c) In BT nodes are inserted based on the values they contain

d) There is no difference

169. The main use of AVL tree is:

a) Searching of data

b) Storing of data

c) Insertion of data

d) Security of data

e) The expression

170. if (! heap ->isEmpty() )

checks

a) Heap is empty

b) Heap is full

# **AL-JUNAID TECH INSTITUTE**

c) Heap is not empty

d) Not a valid expression

171. A solution is said to be efficient if it solves the problem within its resource constraints i.e. hardware and time.

a) True

b) False

172. A queue where the dequeue operation does not depend upon FIFO, is called:

a) enqueue

b) simple queue

c) stack

d) priority queue

173. In simple or singly linked list there is/are \_\_\_\_\_ pointer/s in each node.

a) One

b) Two

c) Three

d) Four

174. Which one of the following statement is correct?

a) Array size is fixed once it is created

b) Link List size is fixed once it is created

c) Binary Search Tree size is fixed once it is created

d) AVL Tree size is fixed once it is created

175. Which of the is NOT true regarding the maze generation?

a) Randomly remove walls until the entrance and exit cells are in same set.

b) Removing a wall is the same as doing a union operation

c) Remove a randomly chosen wall if the cells it separates are already in same set

d) Do not remove a randomly chosen wall if the cells it separates are already in same set.

176. If Ahmad is cousin of Ali and Ali is cousin of Asad then Ahmad is also cousin of Asad. This statement has the following property

a) Reflexivity

b) Symmetry

c) Transitivity

d) All of the given

177. Stack and Queue can be implemented using \_\_\_\_\_.

# **AL-JUNAID TECH INSTITUTE**

a) Singly Link List

b) Binary Tree

c) Binary Search Tree

d) AVL Tree

178. A Linear Data Structure is the data structure in which data elements are arranged in a sequence or a linear list. Which of the following is Non Linear Data Structure?

a) Arrays

b) Linked Lists

c) Binary Search Trees

d) Stack

179. Recursive call of a function use \_\_\_\_\_ data structure.

a) Linked List

b) Queue

c) Stack

d) Table

180. The \_\_\_\_\_ of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

a) Height

b) Balance

c) Width

d) None of the above

181. Which operation of queue data structure is used to get front element from the queue and then remove it from the queue?

a) enqueue()

b) dequeue()

c) front()

d) remove()

182. Suppose we have been given the following data set for a Queue.

7 5 2 4

What will be the resultant Queue if we call enqueue(3) method?

Note that 7 is the front element whereas 4 is rear element of queue.

a) 7 5 2 4

b) 7 5 2 4

c) 7 5 2 4 3

d) 5 2 4 3

183. In C++, we place the class interface in \_\_\_\_\_ file.

a) .cpp

b) .cppp

# AL-JUNAID TECH INSTITUTE

c) .h

d) .hh

184. The array in binary search is sub divided \_\_\_\_\_.

a) Once

b) Twice

c) N time

d) **Untill a sublist is no more divisible**

185. If unions are done by weight (size), the depth of any element is never greater than

a)  $\log 3n$

b)  **$\log_2 n$**

c)  $n \log_2 n$

d)  $\log n * n$

186. \_\_\_\_\_ in AVL is logarithmic.

a) Updating

b) **Searching**

c) Deletion

d) Insertion

187. What is the depth of any tree if the union operation is performed by height?

a)  $O(N)$

b)  $O(N \log N)$

c)  **$O(\log N)$**

d)  $O(M \log N)$

188. Avl tree takes maximum \_\_\_\_\_ time to search an element.

a)  **$1.44 \log_2 n$**

b)  $\log_2(n+n)$

c)  $\log_2(n+1)+1$

d)  $1.88 \log_2 n$

189. Which of the following is correct about AVL Tree?

a) It is identical to BST except height of the left and right subtrees can differ by at least 1.

b) It is identical to BST except height of the left and right subtrees must differ by at least 1.

c) It is not identical to BST, it is totally different kind of tree.

d) **It is identical to BST except height of the left and right subtrees can differ by at most 1.**

190. "new int[11]" will allocate memory for \_\_\_\_\_ integers.

# AL-JUNAID TECH INSTITUTE

a) 10

**b) 11**

c) 12

d) 13

191. The \_\_\_\_\_ method of list data structure removes the element residing at the current position

a) Add

b) Next

**c) Remove**

d) Find

192. While implementing non-recursive traversal for Binary SearchTree, we need to implement \_\_\_\_\_ .

a) Queue

**b) Stack**

c) Min heap

d) Max heap

193. Doubly Linked List always has \_\_\_\_\_ NULL pointer/s in a node.

a) One

**b) Two**

c) Three

d) Four

194. For a perfect binary tree of height  $h$ , having  $N$  nodes, the sum of heights of nodes is \_\_\_\_\_ .

**a)  $N - h - 1$**

b)  $N - 1$

c)  $N - 1 + h$

d)  $N - (h - 1)$

195. If there are  $N$  elements in an array then the number of maximum steps needed to find an element using Binary Search is \_\_\_\_\_ .

a)  $N$

b)  $N^2$

c)  $N \log_2 N$

**d)  $\log_2 N$**

196. Here is a small function definition:

```
void f(int i, int &k)
{
 i = 1;
```

# AL-JUNAID TECH INSTITUTE

k = 2;

}

Suppose that a main program has two integer variables x and y, which are given the value 0. Then the main program calls f(x,y); What are the values of x and y after the function f finishes?

- a) Both x and y are still 0.
- b) x is now 1, but y is still 0.
- c) x is still 0, but y is now 2.
- d) x is now 1, and y is now 2.

197. The worst case of searching in binary search tree (BST) is:

- a) When the data inserted in BST is sorted
- b) When the height of left sub-tree is greater than right sub-tree
- c) When the height of right sub-tree is greater than left sub-tree
- d) When the tree is balanced

198. What will be postfix expression of the following infix expression?

Infix Expression:  $a+b*c-d$

- a)  $ab+c*d-$
- b)  $abc*+d-$
- c)  $abcd+*-$
- d)  $abc+*d-$

199. What is the hash function used in linear probing?

- a)  $hi(x) = \text{hash}(x) \bmod \text{table size}$
- b)  $hi(x) = (\text{hash}(x) + f(i^2)) \bmod \text{table size}$
- c)  $hi(x) = (\text{hash}(x) + f(i)) \bmod \text{table size}$
- d)  $hi(x) = X \bmod 17$

200. Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s)

- a) One
- b) Two
- c) Three
- d) Four

201. In the statement  $\text{int} \& a=b;$

- a) a and b pointing to two different memory location
- b) a and b are two different names of the same memory location
- c) a and b are two different variable names
- d) b hold the address of variable a

202. In 1990, Bill pugh proposed an enhancement on linked lists and the new data structure was termed as



# **AL-JUNAID TECH INSTITUTE**

a) Linked list

b) B-Tree

c) Skip list

d) Spelling checker

203. Which of the following statement is NOT correct regarding Table ADT?

a) In a table, the type of information in columns may be different.

b) A table consists of several columns known as entities

c) The row of a table is called a record

d) A major use of table is in databases where we build and use tables for keeping information

204. A binary tree of N nodes has \_\_\_\_\_.

a)  $\log_{10} N$  levels

b)  $\log_2 N$  levels

c)  $N/2$  levels

d)  $N \times 2$  levels

205. If the height of a perfect binary tree is 4. What will be the total number of nodes in it?

a) 15

b) 16

c) 31

d) 32

206. Which property of equivalence relation is satisfied if we say:

Ahmad R(is related to) Ahmad

a) Reflexivity

b) Symmetry

c) Transitivity

d) All of the given

207. If ahmad is boss of ehsan and ehsan is boss of umer then ahmad is also boss of umer. The above mentioned relation is \_\_\_\_\_.

a) Reflexive

b) Symmetry

c) Transitive

d) None of the given

208. Which of the following statement is false?

a) Arrays are dense lists and static data structure

b) data elements in linked list need not be stored in adjacent space in memory

# **AL-JUNAID TECH INSTITUTE**

- c) pointers store the next data element of a list
- d) linked lists are collection of the nodes that contain information part and next pointer

209. Binary search algorithm cannot be applied to \_\_\_\_\_ .

- a) **Sorted linked list**
- b) sorted binary trees
- c) sorted linear array
- d) None of the given

210. Suppose there is an image segmented into pixels. Each pixel has \_\_\_\_\_ neighbour(s).

- a) 0
- b) 4
- c) **8**
- d) 16

211. In singly linked list a node consists of two parts:

- a) Object and structure
- b) Two pointers
- c) Two objects
- d) **Object and pointer**

212. A hash function returns a \_\_\_\_\_ value.

- a) **Integer**
- b) Double
- c) Float
- d) Char

213. In a tree, we link the nodes in such a way that it \_\_\_\_\_ a linear structure.

- a) does not remain
- b) **forms**
- c) reverses
- d) remains

214. The principal benefit of a link list over a conventional array is that the order of the linked items may be \_\_\_\_\_ from the order that the data seems are stored in memory.

- a) Different
- b) Identical
- c) **Same**
- d) Equivalent

215. The Computer memory can be thought of as a/an.

- a) List

# **AL-JUNAID TECH INSTITUTE**

- b) Queue
  - c) Stack
  - d) Array
216. Before using the Pop method of a stack, the user must call the \_\_\_\_ method.
- a) isFull()
  - b) push()
  - c) pop()
  - d) isEmpty()
217. A stack carries \_\_\_\_ behavior.
- a) FIFO
  - b) FEFO
  - c) LIFO
  - d) AVCO
218. \_\_\_\_ Method returns the top element of the stack without removing it.
- a) Pop()
  - b) Front()
  - c) Push()
  - d) Top()
219. In Queue data structure element are removed from \_\_\_\_.
- a) Pop
  - b) Push
  - c) Rear
  - d) Front
220. In Queue data structure element are full from \_\_\_\_.
- a) Pop
  - b) Push
  - c) Rear
  - d) Front
- 221.
222. “end()” method of list performs its tasks in
- a) Many Steps
  - b) Three Steps
  - c) One Steps
  - d) Two Steps
223. Linked List use \_\_\_\_ to store data.
- a) Array
  - b) Variables

# **AL-JUNAID TECH INSTITUTE**

- c) Linked Memory
  - d) 2-D Array
224. Complete the push method code of stack void push (int x) {  
A{\_\_\_\_}=x}  
a) Count++  
b) Count—  
c) ++count  
d) –count
225. A template is a Function or Class that is Written a \_\_\_\_ data type.  
a) Generic  
b) None of These  
c) Define  
d) Specific
226. Trying to remove an element from an empty Stack is called \_\_\_\_  
a) Garbage Collection  
b) Overflow stack  
c) Empty collection  
d) Underflow of Stack
227. Stack and Queue can be implemented using \_\_\_\_  
a) Binary Tree  
b) Singly Linked List  
c) AVL Tree  
d) Binary Search Tree
228. A software solution is said to be efficient if it solves the problem.  
a) With its resource constraints  
b) With No Time  
c) By Consuming more hardware resources.  
d) By Using some extra resources.
229. we cannot remove items randomly from \_\_\_\_  
a) Stack  
b) Queue  
c) Stack and Queue  
d) List
230. Stack push(15) will push 15 on \_\_\_\_  
a) Top of the Stack  
b) Anywhere of the Stack  
c) Middle of the Stack  
d) Bottom of the Stack

# **AL-JUNAID TECH INSTITUTE**

231. Which of the following is the correct conservation from infix to postfix expression  $A*B+C/(E-F)$ .

- a)  $ABC*+EF-/$
- b)  $AB*CFE-/*$
- c)  $AB*C+EF/*$
- d)  $AB+C*E-F/$

232. Convert the Given infix from  $12+60-23$  of expression in postfix form.

- a)  $12+60\ 23-$
- b)  $12\ 60+ -23$
- c)  $-12\ 60 -23$
- d) None

233.  $Y=\&x[0];$

In the above statement, we get the address of the Location of the array x and store it in y. Here “Y” is;

- a) Xvalue
- b) Zvalue
- c) Ivalue
- d) Nvalue

234. Last Node in Circular Linked list Contains.

- a) Maximum Two points
- b) No Null Pointer
- c) Atleast one Null pointer
- d) None

235. Consider the linked list having data  $[6, 72, 35, 65, 25]$  stored in it. While current pointer is pointing memory location having 72 stored in it. After calling  $\text{add}(4)$  function on the following linked list current point will point to memory location having value?

- a) 36
- b) 4
- c) 72
- d) 25

236. there is no such Node whose next filed is NULL which one of the given option support the statement

- a) Linked List
- b) Circular Link list
- c) Array
- d) Queue

# DATA STRUCTURE (CS301)

## MID TERM MEGA FILE

SOLVED BY

## VU HELPER

❖ Which one of the following statement is NOT correct .

- ▶ In linked list the elements are necessarily to be contiguous
- ▶ **In linked list the elements may locate at far positions in the memory**
- ▶ In linked list each element also has the next to it
- ▶ In an array the elements are contiguous

❖ Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

- ▶ One
- ▶ **Two**
- ▶ Three
- ▶ Four

❖ Which one of the following calling methods does not change the original value of the argument in the calling function?

- ▶ None of the given options
- ▶ **Call by passing the value of the argument**
- ▶ Call by passing reference of the argument
- ▶ Call by passing the address of the argument

❖ A tree is an AVL tree if

- ▶ Any one node fulfills the AVL condition
- ▶ At least half of the nodes fulfill the AVL condition
- ▶ All the nodes fulfill the AVL condition
- ▶ **None of the given options**

❖ Suppose currentNode refers to a node in a linked list (using the Node class with member variables called data and nextNode). What statement changes currentNode so that it refers to the next node?

- ▶ currentNode ++;
- ▶ currentNode = nextNode;
- ▶ currentNode += nextNode;
- ▶ **currentNode = currentNode->nextNode;**

❖ A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True**

❖ Which one is a self-referential data type?

- ▶ Stack
- ▶ Queue
- ▶ **Link list**
- ▶ All of these

❖ Each node in doubly link list has,

- ▶ 1 pointer
- ▶ **2 pointers**
- ▶ 3 pointers
- ▶ 4 pointers

❖ I have implemented the queue with a linked list, keeping track of a front pointer and a rear pointer. Which of these pointers will change during an insertion into an *EMPTY* queue?

- ▶ Neither changes
- ▶ Only front pointer changes.
- ▶ **Only rear pointer changes.**
- ▶ Both change.

❖ Consider the following tree.



How many of the nodes have at least one sibling?

- ▶ 8
- ▶ 7
- ▶ **5**
- ▶ 6

❖ The nodes with no successor are called \_\_\_\_\_

- ▶ Root Nodes
- ▶ **Leaf Nodes**
- ▶ Both of these
- ▶ None of these

❖ AVL Tree is,

- ▶ **Non Linear data structure**
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.

❖ We access elements in AVL Tree in,

- ▶ Linear way only
- ▶ **Non Linear way only**
- ▶ Both linear and non linear ways
- ▶ None of the given options.

❖ A binary search tree should have minimum of \_\_\_\_\_ node/s at each level,

- ▶ **One**
- ▶ Two
- ▶ Three
- ▶ Four

❖ Consider the following statements.

- (i) A binary tree can contain at least  $2^L$  Nodes at level L.
- (ii) A complete binary tree of depth d is a binary tree that contains  $2^L$  Nodes at each level L between 0 and d, both inclusive.
- (iii) The total number of nodes ( $T_n$ ) in a complete binary tree of depth d is  $2^{d+1} - 1$ .
- (iv) The height of the complete binary tree can be written as  $h = \log_2 (T_n + 1) - 1$  where  $T_n$  is Total number of Nodes.

Which one of the following is correct in respect of the above statements regarding the Binary trees?

- ▶ (i) and (iii) only
- ▶ (i), (ii) and (iii) only
- ▶ (ii) and (iii) only
- ▶ **(ii), (iii) and (iv) only**

❖ “+” is a \_\_\_\_\_ operator.

- ▶ Unary
- ▶ **Binary**
- ▶ Ternary
- ▶ None of the above

❖ A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True**

❖ The data of the problem is of 2GB and the hard disk is of 1GB capacity, to solve this problem we should

- ▶ Use better data structures
- ▶ **Increase the hard disk space**
- ▶ Use the better algorithm
- ▶ Use as much data as we can store on the hard disk

❖ Consider the function X as under

```
int X (int& Value)
{
 return Value;
}
```



Now a and b are integers in a calling function. Which one of the following is a valid call to the above function X.

- ▶ a = X (b) ;
- ▶ a = X (&b) ;
- ▶ **a = X (\*b) ;**
- ▶ None of the given options

❖ In the call by value methodology, a copy of the object is passed to the called function.

- ▶ False
- ▶ **True**

❖ The tree data structure is a

- ▶ Linear data structure
- ▶ **Non-linear data structure**
- ▶ Graphical data structure
- ▶ Data structure like queue

❖ When should you use a const reference parameter?

- ▶ Whenever the parameter has huge size
- ▶ Whenever the parameter has huge size, the function changes the parameter within its body, and you do NOT want these changes to alter the actual argument.
- ▶ Whenever the parameter has huge size, the function changes the parameter within its body, and you DO want these changes to alter the actual argument.

▶ **Whenever the parameter has huge size, and the function does not change the parameter within its body.**

❖ Here is the start of a C++ class declaration:

```
class foo
{
 public:
 void x(foo f);
 void y(const foo f);
 void z(foo f) const;
...

```

Which of the three member functions can alter the PRIVATE member variables of the foo object that activates the function?

▶ **Only x can alter the private member variables of the object that activates the function. (R )**

▶ Only y can alter the private member variables of the object that activates the function.

▶ Only z can alter the private member variables of the object that activates the function.

▶ Two of the functions can alter the private member variables of the object that activates the function.

❖ What is the maximum depth of recursive calls a function may make?

- ▶ 1
- ▶ 2
- ▶ **n (where n is the argument)**
- ▶ There is no fixed maximum

- ❖ Suppose  $n$  is the number of nodes in a complete Binary Tree then maximum steps required for a search operation are,

- ▶  **$\log_2(n+1) - 1$**
- ▶  $\log_2(n+1)$
- ▶  $\log_2(n) - 1$
- ▶  $\log_2(n)$

- ❖ In the linked list implementation of the stack class, where does the push member function places the new entry on the linked list?

- ▶ **At the head**
- ▶ At the tail
- ▶ After all other entries that are greater than the new entry.
- ▶ After all other entries that are smaller than the new entry.

- ❖ Suppose we have a *circular* array implementation of the queue class, with ten items in the queue stored at `data[2]` through `data[11]`. The CAPACITY is 42, i.e., the array has been declared to be of size 42. Where does the push member function place the new entry in the array?

- ▶ `data[1]`
- ▶ `data[2]`
- ▶ **`data[11]`**
- ▶ `data[12]`

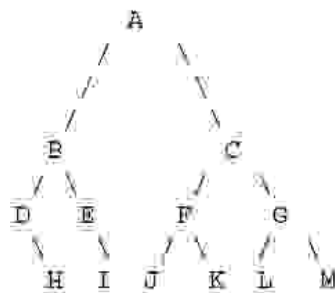
- ❖ The expression  $AB+C^*$  is called ?

- ▶ Prefix expression
- ▶ **Postfix expression**
- ▶ Infix expression
- ▶ None of these

- ❖ \_\_\_\_\_ is a binary tree where every node has a value, every node's left subtree contains only values less than or equal to the node's value, and every node's right subtree contains only values that are greater than or equal ?

- ▶ **Strictly Binary Tree**
- ▶ Binary Search tree
- ▶ AVL tree
- ▶ All of these

- ❖ Consider the following binary search tree (BST):



If node A in the BST is deleted, which two nodes are the candidates to take its place?

- ▶ **J and I (Pending)**

- ▶ H and E
- ▶ D and E
- ▶ L and M

❖ Let's call the node as a that requires re-balancing. Consider the two cases given below:

- 1) An insertion into left subtree of the left child of a
- 2) An insertion into right subtree of the right child of a.

Which of the following statement is correct about these two cases.

- ▶ The insertion occurs outside (i.e., left-left or right-right) in cases 1 and 2. single rotation can fix the balance in these two cases.
- ▶ The insertion occurs inside ((i.e., left-left or right-right) in cases 1 and 2. single rotation cannot fix the balance in these two cases

❖ We access elements in AVL Tree in,

- ▶ Linear way only
- ▶ **Non Linear way only**
- ▶ Both linear and non linear ways
- ▶ None of the given options.

❖ AVL Tree is,

- ▶ **Non Linear data structure**
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.

❖ Which one of the following is a valid postfix expression?

- ▶ **ab+c\*d-**
- ▶ abc\*+d-
- ▶ abc+\*d-
- ▶ (abc\*)+d-

❖ The tree data structure is a

- ▶ Linear data structure
- ▶ **Non-linear data structure**
- ▶ Graphical data structure
- ▶ Data structure like queue

❖ A Compound Data Structure is the data structure which can have multiple data items of same type or of different types. Which of the following can be considered compound data structure?

- ▶ Arrays
- ▶ LinkLists
- ▶ **Binary Search Trees**
- ▶ All of the given options

❖ Suppose a pointer has been declared in main but has not assigned any variable address then

- ▶ ☐ That pointer points to First byte in main function
- ▶ **That pointer contains a NULL value**
- ▶ ☐ None of these
- ▶ That pointer points to any memory address



- ❖ Here is the start of a C++ class declaration:

```
class foo
{
 public:
 void x(foo f);
 void y(const foo f);
 void z(foo f) const;
```

**Which of the three member functions can alter the PRIVATE member variables of the foo object that activates the function?**

▶ **Only x can alter the private member variables of the object that activates the function.**

▶ Only y can alter the private member variables of the object that activates the function.

▶ ☐ Only z can alter the private member variables of the object that activates the function.

▶ Two of the functions can alter the private member variables of the object that activates the function.

- ❖ **The operation for removing an entry from a stack is traditionally called:**

▶ delete

▶ peek

▶ **pop**

▶ remove

- ❖ **Which statement of the following statements is incorrect?**

▶ Lists can be implemented by using arrays or linked lists

▶ A list is a sequence of one or more data items

▶ **Stack is a special kind of list in which all insertions and deletions take place at one end**

▶ Stacks are easier to implement than lists

- ❖ **Parameters in function call are passed using,**

▶ Stack

▶ **Queue**

▶ Binary Search Tree

▶ AVL Tree

- ❖ **Consider the following sequence of push operations in a stack:**

```
stack.push('7');
stack.push('8');
stack.push('9');
stack.push('10');
stack.push('11');
stack.push('12');
```

▶ ☐ **7 8 9 10 11 12**

▶ 9 8 11 10 7 12

▶ 9 10 8 11 12 7

▶ 9 10 8 12 7 11

❖ What is the maximum depth of recursive calls a function may make?

- ▶ 1
- ▶ 2

▶ **n (where n is the argument)**

▶ There is no fixed maximum

❖ Consider the following function:

```
void test_a(int n)
{
 cout << n << " ";
 if (n>0)
 test_a(n-2);
}
```

What is printed by the call test\_a(4)?

- ▶ 4 2
- ▶ 0 2 4
- ▶ 0 2
- ▶ **2 4**

❖ Queue follows,

- ▶ Last in First out
- ▶ First in Last out
- ▶ **First in First out**
- ▶ None of these

❖ \_\_\_\_\_ is a binary tree where every node has a value, every node's left subtree contains only values less than or equal to the node's value, and every node's right subtree contains only values that are greater than or equal ?

▶ **Strictly Binary Tree**

- ▶ Binary Search tree
- ▶ AVL tree
- ▶ All of these

❖ Four statements about trees are below. Three of them are correct. Which one is INCORRECT?

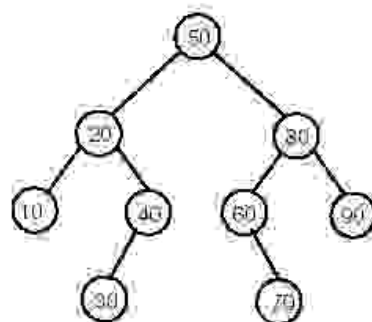
▶ ☐ **Trees are recursively defined multi-dimensional data structures**

▶ The order of a tree indicates a maximum number of children allowed at each node of the tree

▶ A search tree is a special type of tree where all values (i.e. keys) are ordered

▶ If Tree1's size is greater than Tree2's size, then the height of Tree1 must also be greater than Tree2's height.

❖ Below is a binary search tree. If we delete the value 50 using the algorithm we discussed, what value will be in the root of the remaining tree?



- ▶ 50
- ▶ 60
- ▶ 70
- ▶ **80**

❖ \_\_\_\_\_ is a data structure that can grow easily dynamically at run time without having to copy existing elements.

- ▶ Array
- ▶ **List**
- ▶ Both of these
- ▶ None of these

❖ \_\_\_\_\_ only removes items in reverse order as they were entered.

- ▶ **Stack**
- ▶ Queue
- ▶ Both of these
- ▶ None of these

❖ Here is a small function definition:

```
void f(int i, int &k)
{
 i = 1;
 k = 2;
}
```

Suppose that a main program has two integer variables x and y, which are given the value 0. Then the main program calls f(x,y); What are the values of x and y after the function f finishes?

- ▶ Both x and y are still 0.
- ▶ x is now 1, but y is still 0.
- ▶ **x is still 0, but y is now 2.**
- ▶ x is now 1, and y is now 2.

❖ Select the one *FALSE* statement about binary trees:

- ▶ **Every binary tree has at least one node.**
- ▶ Every non-empty tree has exactly one root node.
- ▶ Every node has at most two children.
- ▶ Every non-root node has exactly one parent.

❖ Every AVL is \_\_\_\_\_

- ▶ Binary Tree
- ▶ Complete Binary Tree
- ▶ None of these
- ▶ **Binary Search Tree**

❖ Searching an element in an AVL tree take maximum \_\_\_\_\_ time (where n is no. of nodes in AVL tree),

- ▶  $\log_2(n+1)$
- ▶  $\log_2(n+1) - 1$



▶ **1.44  $\log_2 n$**

▶  $1.66 \log_2 n$

- ❖ Suppose that we have implemented a *priority queue* by storing the items in a heap. We are now executing a reheapification downward and the out-of-place node has priority of 42. The node's parent has a priority of 72, the left child has priority 52 and the node's right child has priority 62. Which statement best describes the status of the reheapification.

- ▶ The reheapification is done.
- ▶ The next step will interchange the two children of the out-of-place node.
- ▶ The next step will swap the out-of-place node with its parent.
- ▶ The next step will swap the out-of-place node with its left child.

- ❖ Suppose you implement a heap (with the largest element on top) in an array. Consider the different arrays below, determine the one that *cannot* possibly be a heap:

- ▶ 7 6 5 4 3 2 1
- ▶ 7 3 6 2 1 4 5
- ▶ 7 6 4 3 5 2 1
- ▶ 7 3 6 4 2 5 1

- ❖ If there are 23 external nodes in a binary tree then what will be the no. of internal nodes in this binary tree?

- ▶ 23
- ▶ 24
- ▶ 21
- ▶ **22**

- ❖ If there are N external nodes in a binary tree then what will be the no. of internal nodes in this binary tree?

- ▶ **N - 1**
- ▶ N+1
- ▶ N+2
- ▶ N

- ❖ Which one of the following is NOT the property of equivalence relation:

- ▶ Reflexive
- ▶ Symmetric
- ▶ Transitive
- ▶ **Associative**

- ❖ The definition of Transitivity property is

- ▶ For all element x member of S,  $x R x$
- ▶ For all elements x and y,  $x R y$  if and only if  $y R x$
- ▶ **For all elements x, y and z, if  $x R y$  and  $y R z$  then  $x R z$**
- ▶ For all elements w, x, y and z, if  $x R y$  and  $w R z$  then  $x R z$

- ❖ Union is a \_\_\_\_\_ time operation.

- ▶ **Constant**
- ▶ Polynomial
- ▶ Exponential
- ▶ None of the given options



❖ Which of the following is NOT a correct statement about Table ADT.

- ▶ In a table, the type of information in columns may be different.
- ▶ A table consists of several columns, known as entities.
- ▶ The row of a table is called a record.
- ▶ A major use of table is in databases where we build and use tables for keeping information.

❖ In the worst case of deletion in AVL tree requires \_\_\_\_\_.

- ▶ Only one rotation
- ▶ Rotation at each non-leaf node
- ▶ Rotation at each leaf node
- ▶ Rotations equal to  $\log_2 N$

❖ Binary Search is an algorithm of searching, used with the \_\_\_\_\_ data.

- ▶ Sorted
- ▶ Unsorted
- ▶ Heterogeneous
- ▶ Random

❖ Which of the following statement is correct?

- ▶ A Threaded Binary Tree is a binary tree in which every node that does not have a left child has a THREAD (in actual sense, a link) to its INORDER successor.
- ▶ A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its PREORDER successor.
- ▶ A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor.
- ▶ A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its POSTORDER successor.

❖ By using \_\_\_\_\_ we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

- ▶ Binary tree only
- ▶ Threaded binary tree
- ▶ Heap data structure
- ▶ Huffman encoding

❖ Which of the following statement is NOT true about threaded binary tree?

- ▶ Right thread of the right-most node points to the *dummy* node.
- ▶ Left thread of the left-most node points to the *dummy* node.
- ▶ The left pointer of dummy node points to the root node of the tree.
- ▶ Left thread of the right-most node points to the *dummy* node.

❖ Consider a min heap, represented by the following array:

11,22,33,44,55 After inserting a node with value 66. Which of the following is the updated min heap?

- ▶ 11,22,33,44,55,66
- ▶ 11,22,33,44,66,55
- ▶ 11,22,33,66,44,55
- ▶ 11,22,66,33,44,55



- ❖ Consider a min heap, represented by the following array:  
3,4,6,7,5 After calling the function deleteMin(). Which of the following is the updated min heap?

- ▶ 4,6,7,5
- ▶ 6,7,5,4
- ▶ 4,5,6,7
- ▶ 4,6,5,7

- ❖ We can build a heap in \_\_\_\_\_ time.

▶ **Linear**

- ▶ Exponential
- ▶ Polynomial
- ▶ None of the given options

- ❖ Suppose we are sorting an array of eight integers using quick sort, and we have just finished the first partitioning with the array looking like this:

2 5 1 7 9 12 11 10

Which statement is correct?

▶ **The pivot could be either the 7 or the 9.**

- ▶ The pivot could be the 7, but it is not the 9.
- ▶ The pivot is not the 7, but it could be the 9.
- ▶ Neither the 7 nor the 9 is the pivot.

- ❖ Which formula is the best approximation for the depth of a heap with  $n$  nodes?

▶  **$\log$  (base 2) of  $n$**

- ▶ The number of digits in  $n$  (base 10), e.g., 145 has three digits
- ▶ The square root of  $n$
- ▶  $n$

- ❖ Suppose you implement a Min heap (with the smallest element on top) in an array. Consider the different arrays below; determine the one that *cannot* possibly be a heap:

- ▶ 16, 18, 20, 22, 24, 28, 30
- ▶ 16, 20, 18, 24, 22, 30, 28
- ▶ 16, 24, 18, 28, 30, 20, 22
- ▶ 16, 24, 20, 30, 28, 18, 22

- ❖ While joining nodes in the building of Huffman encoding tree if there are more nodes with same frequency, we choose the nodes \_\_\_\_\_.

▶ **Randomly**

- ▶ That occur first in the text message
- ▶ That are lexically smaller among others.
- ▶ That are lexically greater among others

- ❖ Consider the following paragraph with blanks.

A ..... is a linear list where ..... and ..... take place at the same end . This end is called the .....

What would be the correct filling the above blank positions?

- ▶ (i) queue (ii) insertion (iii) removals (iv) top

- ▶ (i) stack (ii) insertion (iii) removals (iv) bottom
- ▶ **(i) stack (ii) insertion (iii) removals (iv) top**
- ▶ (i) tree (ii) insertion (iii) removals (iv) top

❖ A binary tree with 33 internal nodes has \_\_\_\_\_ links to internal nodes.

- ▶ 31
- ▶ **32**
- ▶ 33
- ▶ 66

❖ Which traversal gives a decreasing order of elements in a heap where the max element is stored at the top?

- ▶ post-order
- ▶ level-order
- ▶ inorder
- ▶ None of the given options

❖ What requirement is placed on an array, so that *binary search* may be used to locate an entry?

- ▶ The array elements must form a heap.
- ▶ The array must have at least 2 entries.
- ▶ The array must be sorted.
- ▶ The array's size must be a power of two.

❖ Which of the following is a non linear data structure?

- ▶ Linked List
- ▶ Stack
- ▶ Queue
- ▶ **Tree**

Question No: 1 ( Marks: 1 ) - Please choose one

In the statement `int x[6];`, we cannot assign any value to x because x is not an lvalue.

- ▶ True
- ▶ **False**

Question No: 2 ( Marks: 1 ) - Please choose one

What will be postfix expression of the following infix expression?

Infix Expression :  $a+b*c-d$

- ▶  $ab+c*d-$
- ▶  **$abc*+d-$**
- ▶  $abc+*d-$
- ▶  $abcd+*-$

Question No: 3 ( Marks: 1 ) - Please choose one

Which one of the following calling methods does not change the original value of the argument in the calling function?

- ▶ None of the given options

- ▶ Call by passing the value of the argument
- ▶ Call by passing reference of the argument
- ▶ **Call by passing the address of the argument**

**Question No: 4 ( Marks: 1 ) - Please choose one**  
**In a program a reference variable, say x, can be declared as**

- ▶ **int &x ;**
- ▶ int \*x ;
- ▶ int x ;
- ▶ None of the given options

**Question No: 5 ( Marks: 1 ) - Please choose one**

**A tree is an AVL tree if**

- ▶ **None of the given options**

**Question No: 6 ( Marks: 1 ) - Please choose one**

Consider the following pseudo code  
declare a stack of characters

```
while (there are more characters in the word to read)
{
 read a character
 push the character on the stack
}
while (the stack is not empty)
{
 pop a character off the stack
 write the character to the screen
}
```

What is written to the screen for the input "apples"?

- ▶ selpa
- ▶ **selppa**
- ▶ apples
- ▶ aaappppplleess

**Question No: 7 ( Marks: 1 ) - Please choose one**

In the following C++ code, how many function calls are made?

```
int x, y, z;
x = 2;
y = 3 + x;
z = foobar(x,y);
```

- ▶ 1
- ▶ 4
- ▶ 7
- ▶ **8**

**Question No: 8 ( Marks: 1 ) - Please choose one**

**We can add elements in QUEUE From \_\_\_\_\_**

- ▶ **Front**
- ▶ Rear
- ▶ From Both Rare and Front

- ▶ None of these

**Question No: 9 ( Marks: 1 ) - Please choose one**  
**Consider the following tree.**

How many of the nodes have at least one sibling?

- ▶ 8
- ▶ 7
- ▶ 5
- ▶ 6

**Question No: 10 ( Marks: 1 ) - Please choose one**  
**Consider the following tree.**

How many descendants does the root have?

- ▶ 5
- ▶ 6
- ▶ 7
- ▶ 8

**Question No: 11 ( Marks: 1 ) - Please choose one**  
Below is a binary search tree. If we delete the value 50 using the algorithm we discussed, what value will be in the root of the remaining tree?

- ▶ 50
- ▶ 60
- ▶ 70
- ▶ 80

**Question No: 12 ( Marks: 1 ) - Please choose one**  
We access elements in AVL Tree in,

- ▶ Linear way only
- ▶ **Non Linear way only**
- ▶ Both linear and non linear ways
- ▶ None of the given options.

**Question No: 13 ( Marks: 1 ) - Please choose one**

Which of the following statement regarding binary tree is NOT correct.

- ▶ A binary tree can contain at least  $2^L$  Nodes at level L.
- ▶ A complete binary tree of depth d is a binary tree that contains  $2^L$  Nodes at each level L between 0 and d, both inclusive.
- ▶ The total number of nodes ( $T_n$ ) in a complete binary tree of depth d is  $2^{d+1} - 1$ .
- ▶ **The height of the complete binary tree can be written as  $h = \log_2 (T_n + 1) - 1$  where  $T_n$  is Total number of Nodes.**

**Question No: 14 ( Marks: 1 ) - Please choose one**  
The following are statements related to queues.

- The last item to be added to a queue is the first item to be removed
- A queue is a structure in which both ends are not used
- The last element hasn't to wait until all elements preceding it on the queue are removed
- A queue is said to be a last-in-first-out list or LIFO data structure.

Which of the above is/are related to normal queues?

- ▶ **(iii) and (ii) only**
- ▶ (i), (ii) and (iv) only

- ▶ (ii) and (iv) only
- ▶ None of the given options

**Question No: 15 ( Marks: 1 ) - Please choose one**

The \_\_\_\_\_ method of list data structure removes the element residing at the current position.

- ▶ Add
- ▶ next
- ▶ **remove**
- ▶ find

**Question No: 16 ( Marks: 1 ) - Please choose one**

it will be efficient to place stack elements at the start of the list because insertion and removal take \_\_\_\_\_ time.

- ▶ Variable
- ▶ **Constant**
- ▶ Inconsistent
- ▶ None of the above

**Question No: 1 ( Marks: 1 ) - Please choose one**

Which one of the following statement is NOT correct .

- ▶ In linked list the elements are necessarily to be contiguous
- ▶ **In linked list the elements may locate at far positions in the memory**
- ▶ In linked list each element also has the address of the element next to it
- ▶ In an array the elements are contiguous

**Question No: 2 ( Marks: 1 ) - Please choose one**

In a program a reference variable, say x, can be declared as

- ▶ **int &x ;**
- ▶ int \*x ;
- ▶ int x ;
- ▶ None of the given options

**Question No: 3 ( Marks: 1 ) - Please choose one**

Linked lists are collections of data items "lined up in a row" , insertions and deletions can be made only at the front and the back of a linked list.

- ▶ True
- ▶ **False**

**Question No: 4 ( Marks: 1 ) - Please choose one**

A Linear Data Structure is the data structure in which data elements are arranged in a sequence or a linear list. Which of the following is Non Linear Data Structure?

- ▶ Arrays
- ▶ LinkLists
- ▶ **Binary Search Trees**
- ▶ None of these

**Question No: 5 ( Marks: 1 ) - Please choose one**

A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True**

**Question No: 6 ( Marks: 1 ) - Please choose one**

Which one of the following statements is correct?

- ▶ ☐ Array size is fixed once it is created.

- ▶ Link List size is fixed ☐ once it is created.
- ▶ Binary Search Tree size is ☐ ☐ fixed once it is created
- ▶ AVL Tree size is fixed ☐ once it is created

**Question No: 7 ( Marks: 1 ) - Please choose one**

Which one of the following is correct about pointers?

- ▶ They always point to ☐ different memory locations
- ▶ They may point to a single ☐ ☐ memory location
- ☐ ▶ The address of two pointer variables is same
- ▶ None of these ☐

**Question No: 8 ( Marks: 1 ) - Please choose one**

Which of the following abstract data types are NOT used by Integer Abstract Data type group?

- ▶ ☐ short
- ▶ int ☐
- ▶ float ☐
- ▶ long ☐ ☐

**Question No: 9 ( Marks: 1 ) - Please choose one**

The operation for adding an entry to a stack is traditionally called :

- ▶ add ☐
- ▶ append ☐
- ▶ insert ☐ ☐
- ☐ ▶ push

**Question No: 10 ( Marks: 1 ) - Please choose one**

The operation for removing an entry from a stack is traditionally called:

- ▶ delete ☐
- ▶ peek ☐
- ▶ ☐ pop
- ▶ remove ☐

**Question No: 11 ( Marks: 1 ) - Please choose one**

We can add elements in QUEUE From \_\_\_\_\_

- ▶ Front ☐
- ▶ ☐ Rear
- ▶ From Both Rare and Front ☐
- ▶ None of these ☐
- ☐

**Question No: 12 ( Marks: 1 ) - Please choose one**

The difference between a binary tree and a binary search tree is that ,

- ▶ a binary search tree has ☐ ☐ two children per node whereas a binary tree can have none, one, or two children per node
- ▶ ☐ in binary search tree nodes are inserted based on the values they contain
- in binary tree nodes are inserted based on the values they contain
- ▶ none of these ☐ ☐

**Question No: 13 ( Marks: 1 ) - Please choose one**

Suppose n is the number of nodes in a complete Binary Tree then maximum steps required for a search operation are,

- ▶ ☐  $\log_2 (n+1) - 1$
- ▶  $\log_2 (n+1)$
- ▶  $\log_2 (n) - 1$
- ▶  $\log_2 (n)$

**Question No: 14 ( Marks: 1 ) - Please choose one**

The following is a segment of a C program.

```
int pqr(BinaryNode t)
{ if (t == null)
return -1;
```



else

return 1+max(pqr(t.left),pqr(t.right)) }

Identify, what the above program intend(s) to do?

- ▶ Compute the height of a ☐ ☐ binary tree using an in-order traversal
- ▶ Compute the height of a ☐ ☐ binary tree using a pre-order traversal
- ▶ ☐ Compute the depth of a binary tree using a pre-order traversal
- ▶ Compute the depth of a ☐ ☐ binary tree using a postorder traversal

**Question No: 15 ( Marks: 1 ) - Please choose one**

Consider the following infix expression:

$3 + 5 * 6 - 7 * (8 + 5)$

Which of the following is a correct equivalent expression(s) for the above?

- ▶  $3 \ 6 \ 5 + * \ 7 \ 5 \ 8 + - *$
- ▶  $3 \ 6 \ 5 \ 7 \ 5 \ 8 + * + - *$
- ▶  $3 \ 5 \ 6 + * \ 7 \ 8 \ 5 + - *$
- ▶  $3 \ 5 \ 6 * + \ 7 \ 8 \ 5 + * -$

**Question No: 16 ( Marks: 1 ) - Please choose one**

An array is a group of consecutive related memory locations.

- ▶ True
- ▶ False

**Question No: 17 ( Marks: 1 )**

Is this a correct statement?

Give answer in Yes or No.

- ▶ A node cannot be deleted, when the node to be deleted has both left and right subtrees.
- ▶ No, it can be deleted.

**Question No: 18 ( Marks: 1 )**

Deleting a leaf node in binary search tree involves setting \_\_\_\_\_ pointer/s of that nodes parent as null.

- ▶ 1
- ▶ 2
- ▶ 3
- ▶ 4

**Question No: 1 ( Marks: 1 ) - Please choose one**

A subscript of an array may be an integer or an integer expression.

- ▶ True
- ▶ False

**Question No: 2 ( Marks: 1 ) - Please choose one**

Doubly Linked List always has one NULL pointer.

- ▶ True
- ▶ False

**Question No: 3 ( Marks: 1 ) - Please choose one**

In which of the traversal method, the recursive calls can be used to traverse a binary tree ?

- ▶ In preorder traversal only
- ▶ In inorder traversal only
- ▶ In postorder traversal only
- ▶ All of the given options

**Question No: 4 ( Marks: 1 ) - Please choose one**

A tree is an AVL tree if

- ▶ Any one node fulfills the AVL condition
- ▶ At least half of the nodes fulfill the AVL condition
- ▶ All the nodes fulfill the AVL condition
- ▶ None of the given options

**Question No: 5 ( Marks: 1 ) - Please choose one**

Suppose currentNode refers to a node in a linked list (using the Node class with member variables called data and nextNode). What boolean expression will be true when cursor refers to the tail node of the list?

- ▶ (currentNode == null)
- ▶ (currentNode->nextNode == null)
- ▶ (nextNode.data == null)
- ▶ (currentNode.data == 0.0)

**Question No: 6 ( Marks: 1 ) - Please choose one**

Suppose that the class declaration of SomeClass includes the following function prototype.  
bool LessThan( SomeClass anotherObject );

**Which of the following tests in the client code correctly compares two class objects alpha and beta?**

- ▶ if (alpha < beta)
- ▶ if (alpha.LessThan(beta))
- ▶ if (LessThan(alpha, beta))
- ▶ if (LessThan(alpha).beta)

**Question No: 7 ( Marks: 1 ) - Please choose one**

In C what is the operation that you can not do with primitive types?

- ▶ Assign a value to primitive type using a literal
- ▶ Declare primitive types to be constant using the Const keyword
- ▶ Create a new instance of primitive type with New keyword
- ▶ None of these

**Question No: 8 ( Marks: 1 ) - Please choose one**

The operation for adding an entry to a stack is traditionally called :

- ▶ add
- ▶ append
- ▶ insert
- ▶ push

**Question No: 9 ( Marks: 1 ) - Please choose one**

The operation for removing an entry from a stack is traditionally called:

- ▶ delete
- ▶ peek
- ▶ pop
- ▶ remove

**Question No: 10 ( Marks: 1 ) - Please choose one**

Consider the following sequence of push operations in a stack:

```
stack.push('7');
stack.push('8');
stack.push('9');
stack.push('10');
stack.push('11');
stack.push('12');
```

- ▶ 7 8 9 10 11 12
- ▶ 9 8 11 10 7 12
- ▶ 9 10 8 11 12 7
- ▶ 9 10 8 12 7 11

**Question No: 11 ( Marks: 1 ) - Please choose one**

\_\_\_\_\_ is the maximum number of nodes that you can have on a stack-linked list ?

- ▶ Zero



- ▶  $2n$  (where  $n$  is the number of nodes in linked list)
- ▶ Any Number
- ▶ None of these

**Question No: 12 ( Marks: 1 ) - Please choose one**

Which of the following can be used to reverse a string value,

- ▶ Stack
- ▶ Queue
- ▶ Both of these
- ▶ None of these

**Question No: 13 ( Marks: 1 ) - Please choose one**

Consider the following tree,

How many leaves does it have?

- ▶ 2
- ▶ 4
- ▶ 6
- ▶ 9

**Question No: 14 ( Marks: 1 ) - Please choose one**

AVL Tree is,

- ▶ **Non Linear data structure**
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.

**Question No: 15 ( Marks: 1 ) - Please choose one**

The following are statements related to queues.

- (i) The last item to be added to a queue is the first item to be removed
- (ii) A queue is a structure in which both ends are not used
- (iii) The last element hasn't to wait until all elements preceding it on the queue are removed
- (iv) A queue is said to be a last-in-first-out list or LIFO data structure.

Which of the above is/are related to normal queues?

- ▶ (iii) and (ii) only
- ▶ (i), (ii) and (iv) only
- ▶ (ii) and (iv) only
- ▶ None of the given options

**Question No: 16 ( Marks: 1 ) - Please choose one**

An array is a group of consecutive related memory locations.

- ▶ True
- ▶ False

**Question No: 1 ( Marks: 1 ) - Please choose one**

In an array we can store data elements of different types.

- ▶ True
- ▶ **False**

**Question No: 2 ( Marks: 1 ) - Please choose one**

In an array list the current element is

- ▶ The first element
- ▶ The middle element
- ▶ The last element
- ▶ **The element where the current pointer points to**

**Question No: 3 ( Marks: 1 ) - Please choose one**

Which one of the following calling methods does not change the original value of the argument in the calling function?

- ▶ None of the given options
- ▶ **Call by passing the value of the argument**
- ▶ Call by passing reference of the argument
- ▶ Call by passing the address of the argument

**Question No: 4 ( Marks: 1 ) - Please choose one**

Which one of the following statements is NOT correct?

- ▶ Array size can be changed after its creation.
- ▶ Link List size can be changed after its creation
- ▶ Binary Search Tree size can be changed after its creation
- ▶ AVL Tree size can be changed after its creation

**Question No: 5 ( Marks: 1 ) - Please choose one**

Suppose that the class declaration of SomeClass includes the following function prototype.

`bool LessThan( SomeClass anotherObject );`

**Which of the following tests in the client code correctly compares two class objects alpha and beta?**

- ▶ `if (alpha < beta)`
- ▶ `if (alpha.LessThan(beta))`
- ▶ `if (LessThan(alpha, beta))`
- ▶ `if (LessThan(alpha).beta)`

**Question No: 6 ( Marks: 1 ) - Please choose one**

A queue is a \_\_\_\_\_ data structure, whereas a stack is a \_\_\_\_\_ data structure.

- ▶ **FIFO, LIFO**
- ▶ LIFO, FIFO
- ▶ none of these
- ▶ both of these

**Question No: 7 ( Marks: 1 ) - Please choose one**

Which one of the following operators has higher priority than all of others ?

- ▶ Multiplication operator
- ▶ Minus operator
- ▶ Plus operator
- ▶ Exponentiation operator

**Question No: 8 ( Marks: 1 ) - Please choose one**

Each node in Binary Search Tree has

- ▶ 1 pointer
- ▶ 2 pointers

- ▶ 3 pointers
- ▶ 4 pointers

**Question No: 9 ( Marks: 1 ) - Please choose one**

Four statements about trees are below. Three of them are correct. Which one is INCORRECT?

- ▶ Trees are recursively defined multi-dimensional data structures
- ▶ The order of a tree indicates a maximum number of children allowed at each node of the tree
- ▶ A search tree is a special type of tree where all values (i.e. keys) are ordered
- ▶ If Tree1's size is greater than Tree2's size, then the height of Tree1 must also be greater than Tree2's height.

**Question No: 10 ( Marks: 1 ) - Please choose one**

Which of the following is "TRUE" about arrays,

- ▶ We can increase the size of arrays after their creation.
- ▶ We can decrease the size of arrays after their creation.
- ▶ We can increase but can't decrease the size of arrays after their creation.
- ▶ We can neither increase nor decrease the array size after their creation.

**Question No: 11 ( Marks: 1 ) - Please choose one**

Searching an element in an AVL tree take maximum \_\_\_\_\_ time (where n is no. of nodes in AVL tree),

- ▶  $\log_2(n+1)$
- ▶  $\log_2(n+1) - 1$
- ▶  **$1.44 \log_2 n$**
- ▶  $1.66 \log_2 n$

**Question No: 12 ( Marks: 1 ) - Please choose one**

There is/are \_\_\_\_\_ case/s for rotation in an AVL tree,

- ▶ 1
- ▶ 3
- ▶ 2
- ▶ 4

**Question No: 13 ( Marks: 1 ) - Please choose one**

Consider the following statements.

- A binary tree can contain at least  $2^L$  Nodes at level L.
- A complete binary tree of depth d is a binary tree that contains  $2^L$  Nodes at each level L between 0 and d, both inclusive.
- The total number of nodes ( $T_n$ ) in a complete binary tree of depth d is  $2^{d+1} - 1$ .
- The height of the complete binary tree can be written as  $h = \log_2 (T_n + 1) - 1$  where  $T_n$  is Total number of Nodes.

Which one of the following is correct in respect of the above statements regarding the Binary trees?

- ▶ (i) and (iii) only
- ▶ (i), (ii) and (iii) only
- ▶ (ii) and (iii) only
- ▶ (ii), (iii) and (iv) only

**Question No: 14 ( Marks: 1 ) - Please choose one**

Consider the following infix expression.

$5 + 6/2$

If one converts the above expression into postfix, what would be the resultant expression?

- ▶  $56 / + 2$
- ▶  $5\ 6\ 2 / +$
- ▶  $5\ 6 / 2 +$
- ▶  $/62 + 5$

**Question No: 15 ( Marks: 1 ) - Please choose one**

Which of the following is a non linear data structure?

- ▶ Linked List
- ▶ Stack
- ▶ Queue
- ▶ **Tree**

**Question No: 16 ( Marks: 1 ) - Please choose one**

“+” is a \_\_\_\_\_ operator.

- ▶ Unary
- ▶ **Binary**
- ▶ Ternary
- ▶ None of the above

**Question: ( Marks: 1 ) - Please choose one**

In a complete binary tree of depth 5 the number of non-leaf nodes is

- 15
- 32
- 16
- 31

**Question: ( Marks: 1 ) - Please choose one**

Which of the following is NOT a linear data structure?

- Linked List
- Stack
- Queue
- **Tree**

**Question: ( Marks: 1 ) - Please choose one**

Recursive function calls are implemented internally using a data structure

- Stack
- Link-List
- **Tree**
- Queue

**Question: ( Marks: 1 ) - Please choose one**

We access elements in AVL Tree in,

- Linear way only
- Non Linear way only
- Both linear and non linear ways
- None of the given options.

**Question: ( Marks: 1 ) - Please choose one**

Consider the following tree,

How many leaves does it have?

- 2
- 4
- 6
- 9

**Question: ( Marks: 1 ) - Please choose one**

In the statement `int x[6];`, we cannot assign any value to x because x is not an lvalue.

- True
- **False**

**Question: ( Marks: 1 ) - Please choose one**

In the following C++ code, how many function calls are made?

```
int x, y, z;
x = 2;
y = 3 + x;
z = foobar(x,y);
```

- 1
- 4
- 7
- 8

**Question: ( Marks: 1 ) - Please choose one**

Consider the following infix expression:

$3 + 5 * 6 - 7 * (8 + 5)$

Which of the following is a correct equivalent expression(s) for the above?

- $65 + * 758 + - *$
- $65758 + * + - *$
- $56 + * 785 + - *$
- $356 * + 785 + * -$

**Question: ( Marks: 1 ) - Please choose one**

A subscript of an array may be an integer or an integer expression.

- True
- False
- 

**Question: ( Marks: 1 ) - Please choose one**

Which of the following is "TRUE" about arrays,

- We can increase the size of arrays after their creation.
- We can decrease the size of arrays after their creation.
- We can increase but can't decrease the size of arrays after their creation.
- We can neither increase nor decrease the array size after their creation.

**Question: ( Marks: 1 ) - Please choose one**

Searching an element in an AVL tree take maximum \_\_\_\_\_ time (where n is no. of nodes in AVL tree),

- $\text{Log}_2(n+1)$
- $\text{Log}_2(n+1) - 1$
- **$1.44 \text{Log}_2 n$**
- $1.66 \text{Log}_2 n$

**Question: ( Marks: 1 ) - Please choose one**

There is/are \_\_\_\_\_ case/s for rotation in an AVL tree,

- 1
- 3
- 2
- 4

**Question: ( Marks: 1 ) - Please choose one**

Consider the following infix expression.

$5 + 6/2$

If one converts the above expression into postfix, what would be the resultant expression?

- $56 / + 2$
- $562 / +$
- $56 / 2 +$
- $/62 + 5$

Question No: 16 ( Marks: 1 ) - Please choose one  
“+” is a \_\_\_\_\_ operator.

- Unary
- **Binary**
- Ternary
- None of the above



[www.vuplanet.com](http://www.vuplanet.com)

1. Addition of new items in stack make the pointer ----- by 2 :-
  - a. Increment, bits
  - b. Increment, bytes
  - c. Decrement, bits
  - d. Decrement, bytes**
2. Next item in a linked list is known as:-
  - a. Index
  - b. Item
  - c. Node**
  - d. Child
3. What will be the postfix notation of  $5+6/2$ .
  - a.  $56+/2$
  - b.  $562+ /$
  - c.  $562/+$**
  - d.  $5+62/$
4. In an AVL tree to delete a parent with two childs in a straight line following rotations will be required:-
  - a. Single
  - b. Double**
  - c. Triple
  - d. None.
5. To check the depth of an AVL tree following time will be taken:-
  - a.  $1.66 \log_2 n$
  - b.  $1.44 \log_2 n$**
  - c.  $\log_2(n+1)-1$
  - d.  $1.66 \log_2(n+1)$
6. BST is a \_\_\_\_\_ Structure:-
  - a. Linear
  - b. Non Linear**
  - c. Circular
  - d. None of Above
7. After creation of an array:-
  - a. Size can be increase but can not be decreased.
  - b. Size can be decreased but can not be increased.
  - c. Size can neither be increased nor be decreased.**
  - d. Size can be increased and can also be decreased.
8. Each node in a BST has \_\_\_\_\_ Pointers:-
  - a. 1
  - b. 2**
  - c. 3
  - d. 4
9. Highest Operators Precedence is of the following operator:-
  - a. Plus
  - b. Minus
  - c. Multiply
  - d. Exponentiation**
10. Following are the linear data structures:-
  - a. Stacks
  - b. Queues



**c. Both a & b**

d. None of the above

11. Each entry which points to a null value in a Singly Linked List is known as:-

a. Node

b. First Node

**c. Last Node**

d. Head Node

12. Non recursive calls are faster than the Recursive calls.

**a. True**

b. False

13. Tree data structure is a \_\_\_\_\_

a. Linear

**b. Non Linear**

c. Circular

d. None of Above

14. What will be the valid postfix notation of  $A+B*C-D$

a.  $ABC+*D-$

**b.  $ABC*+D-$**

c.  $ABCD+-*$

d.  $AB+D*C$

15. When an operator is used in between two operands this is which type of notation

a. Prefix

b. Postfix

**c. Infix**

d. None of the Above

Question No: 1 ( Marks: 1 ) - Please choose one

Which one of the following operations returns top value of the stack?

Push

Pop

**Top**

First

Question No: 2 ( Marks: 1 ) - Please choose one

In a complete binary tree of depth 4 the number of non-leaf nodes is

**7**

8

15

16

Question No: 3 ( Marks: 1 ) - Please choose one

Which of the following is NOT a linear data structure?

Linked List

Stack

Queue

**Tree**

Question No: 4 ( Marks: 1 ) - Please choose one

A Linear Data Structure is the data structure in which data elements are arranged in a sequence or a linear list. Which of the following is Non Linear Data Structure?

Arrays

LinkLists

**Binary Search Trees**

None of these



**Question No: 5 ( Marks: 1 ) - Please choose one**

In sequential access data structure, accessing any element in the data structure takes different amount of time. Tell which one of the following is sequential access data structure,

Arrays

Lists

**Both of these**

None of these

**Question No: 6 ( Marks: 1 ) - Please choose one**

If you know the size of the data structure in advance, i.e., at compile time, which one of the following is a good data structure to use.

**Array**

List

Both of these

None of these

**Question No: 7 ( Marks: 1 ) - Please choose one**

Recursive function calls are implemented internally using a data structure

Stack

Link-List

**Tree**

Queue

**Question No: 8 ( Marks: 1 ) - Please choose one**

Given a stack of n items, how many POP and PUSH operations need to be performed to remove the item at its bottom?

0 POP operation and 0 PUSH operation

1 POP operation and 1 PUSH operation

n POP operations and n PUSH operations

n POP operations and n-1 PUSH operations

**Question No: 9 ( Marks: 1 ) - Please choose one**

One difference between a queue and a stack is:

Queues require dynamic memory, but stacks do not.

Stacks require dynamic memory, but queues do not.

Queues use two ends of the structure; stacks use only one.

Stacks use two ends of the structure, queues use only one.

**Question No: 10 ( Marks: 1 ) - Please choose one**

In the following C++ code, how many function calls are made?

```
int x, y, z;
```

```
x = 2;
```

```
y = 3 + x;
```

```
z = foobar(x,y);
```

1

4

7

8

**Question No: 11 ( Marks: 1 ) - Please choose one**

Consider the following function:

```
void test_a(int n)
```

```
{
```

```
cout << n << " ";
```

```
if (n>0)
```

```
test_a(n-2);
```

}

What is printed by the call test\_a(4)?

- 4 2
- 0 2 4
- 0 2
- 2 4

**Question No: 12 ( Marks: 1 ) - Please choose one**

Consider the following tree,

How many leaves does it have?

- 2
- 4
- 6
- 9

**Question No: 13 ( Marks: 1 ) - Please choose one**

We access elements in AVL Tree in,

- Linear way only
- Non Linear way only**
- Both linear and non linear ways
- None of the given options.

**Question No: 14 ( Marks: 1 ) - Please choose one**

Consider the following statements.

- (i) A binary tree can contain at least 2 L Nodes at level L.
- (ii) A complete binary tree of depth d is a binary tree that contains 2 L Nodes at each level L between 0 and d, both inclusive.
- (iii) The total number of nodes ( $T_n$ ) in a complete binary tree of depth d is  $2^{d+1} - 1$ .
- (iv) The height of the complete binary tree can be written as  $h = \log_2(T_n + 1) - 1$  where  $T_n$  is Total number of Nodes.

Which one of the following is correct in respect of the above statements regarding the Binary trees?

- (i) and (iii) only
- (i), (ii) and (iii) only
- (ii) and (iii) only
- (ii), (iii) and (iv) only

**Question No: 15 ( Marks: 1 ) - Please choose one**

The following is a segment of a C program.

```
int pqr(BinaryNode t)
{ if (t == null)
 return -1;
 else
 return 1+max(pqr(t.left),pqr(t.right)) }
```

Identify, what the above program intend(s) to do?

- Compute the height of a binary tree using an in-order traversal
- Compute the height of a binary tree using a pre-order traversal
- Compute the depth of a binary tree using a pre-order traversal
- Compute the depth of a binary tree using a post-order traversal

**Question No: 16 ( Marks: 1 ) - Please choose one**

Consider the following infix expression.

$3 * 2 \ 5 - 1$

If one converts the above expression into postfix, what would be the resultant expression?

- $3 \ 2 \ * \ 5 \ 1 \ -$
- $3 \ 2 \ * \ 5 \ 1 \ -$
- $3 \ 2 \ * \ 5 \ - \ 1$



[www.vuplanet.com](http://www.vuplanet.com)



## CS301- Data Structures

**NOV 25,2011**

**LATEST SOLVED MCQS FROM MIDTERM PAPERS**

|                    |                                                      |               |
|--------------------|------------------------------------------------------|---------------|
| <b>MC100401285</b> | <i>Moaaz.pk@gmail.com      MC100401285@gmail.com</i> | <b>PSMD01</b> |
| <b>mc110200885</b> | <i><u>virtualcheater@gmail.com</u></i>               | <b>VDGK01</b> |
| <b>mc110200030</b> | <i><u>awais4you@gmail.com</u></i>                    | <b>PLHR25</b> |

## CS301-MIDTERM SOLVED MCQS WITH REFERENCES



**Virtual University**

Mohammad Moazz Siddiq MCS1285  
[moaazpk@gmail.com](mailto:moaazpk@gmail.com)  
 Campus : Islamabad Office Learning &  
 Modern Studies (IBMS) Samanabad  
**(PSMD01)**

Mohammad Tahir Farid MCS2001  
[virtualcheater@gmail.com](mailto:virtualcheater@gmail.com)  
 Virtual University Campus, IGC, Ilahi  
**(VDGK01)**

Mohammad Waqar MCS2003  
[awais4you@gmail.com](mailto:awais4you@gmail.com)  
 Campus : Abbas Institute Of  
 Modern Studies (AIMS) Lahore  
**(PLHR25)**

Mohammad Moazz Siddiq MCS1285  
[moaazpk@gmail.com](mailto:moaazpk@gmail.com)  
 Campus : Islamabad Office Learning &  
 Modern Studies (IBMS) Samanabad  
**(PSMD01)**

Mohammad Tahir Farid MCS2001  
[virtualcheater@gmail.com](mailto:virtualcheater@gmail.com)  
 Virtual University Campus, IGC, Ilahi  
**(VDGK01)**

Mohammad Waqar MCS2003  
[awais4you@gmail.com](mailto:awais4you@gmail.com)  
 Campus : Abbas Institute Of  
 Modern Studies (AIMS) Lahore  
**(PLHR25)**

## MIDTERM EXAMINATION

Spring 2010

CS301- Data Structures

### Question No: 1 ( Marks: 1 ) - Please choose one

A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True (Page 101)**

### Question No: 2 ( Marks: 1 ) - Please choose one

The data of the problem is of 2GB and the hard disk is of 1GB capacity, to solve this problem we should

- ▶ Use better data structures
- ▶ **Increase the hard disk space (Page 5)**
- ▶ Use the better algorithm
- ▶ Use as much data as we can store on the hard disk

### Question No: 3 ( Marks: 1 ) - Please choose one

Consider the function X as under

```
int X (int& Value)
{
 return Value;
}
```

Now a and b are integers in a calling function. Which one of the following is a valid call to the above function X.

- ▶ a = X (b) ;
- ▶ **a = X (&b) ;**
- ▶ a = X (\*b) ;
- ▶ None of the given options

Here function argument passing by reference method is used, so when we call a function we will give the variable reference as parameter.

### Question No: 4 ( Marks: 1 ) - Please choose one

In the call by value methodology, a copy of the object is passed to the called function.

- ▶ False
- ▶ **True (Page 202)**

### Question No: 5 ( Marks: 1 ) - Please choose one

The tree data structure is a

- ▶ Linear data structure
- ▶ **Non-linear data structure (Page 112)**
- ▶ Graphical data structure
- ▶ Data structure like queue

**Question No: 6 ( Marks: 1 ) - Please choose one**

When should you use a **const reference** parameter?

- ▶ Whenever the parameter has huge size.
- ▶ **Whenever the parameter has huge size, the function changes the parameter within its body, and you do NOT want these changes to alter the actual argument.**
- ▶ Whenever the parameter has huge size, the function changes the parameter within its body, and you DO want these changes to alter the actual argument.
- ▶ Whenever the parameter has huge size, and the function does not change the parameter within its body.

**Declaring a parameter as a const simply means that the function can't change the value of its parameters.**

**Question No: 7 ( Marks: 1 ) - Please choose one**

Here is the start of a C++ class declaration:

```
class foo
{
public:
 void x(foo f);
 void y(const foo f);
 void z(foo f) const;
...
}
```

Which of the three member functions can alter the PRIVATE member variables of the foo object that activates the function?

- ▶ Only x can alter the private member variables of the object that activates the function.
- ▶ Only y can alter the private member variables of the object that activates the function.
- ▶ Only z can alter the private member variables of the object that activates the function.
- ▶ **Two of the functions can alter the private member variables of the object that activates the function.**

**Only the x and y can alter the private member variable of the foo class object. Last Option is more correct but not exact. In the last option the two function name are not mentioned**

**Question No: 8 ( Marks: 1 ) - Please choose one**

What is the maximum depth of recursive calls a function may make?

- ▶ 1
- ▶ 2
- ▶ n (where n is the argument)
- ▶ **There is no fixed maximum**

**Question No: 9 ( Marks: 1 ) - Please choose one**

Suppose n is the number of nodes in a complete Binary Tree then maximum steps required for a search operation are,

- ▶  **$\log_2(n+1) - 1$  (Page 139)**
- ▶  $\log_2(n+1)$
- ▶  $\log_2(n) - 1$
- ▶  $\log_2(n)$

**Question No: 10 ( Marks: 1 ) - Please choose one**

In the linked list implementation of the stack class, where does the push member function places the new entry on the linked list?

- ▶ **At the head** (Page 53)
- ▶ At the tail
- ▶ After all other entries that are greater than the new entry.
- ▶ After all other entries that are smaller than the new entry.

**Question No: 11 ( Marks: 1 ) - Please choose one**

Suppose we have a *circular* array implementation of the queue class, with ten items in the queue stored at data[2] through data[11]. The CAPACITY is 42, i.e., the array has been declared to be of size 42. Where does the push member function place the new entry in the array?

- ▶ data[1]
- ▶ data[2]
- ▶ data[11]
- ▶ **data[12]**

**Question No: 12 ( Marks: 1 ) - Please choose one**

The expression  $AB+C^*$  is called?

- ▶ Prefix expression
- ▶ **Postfix expression** (Page 70)
- ▶ Infix expression
- ▶ None of these

**Question No: 13 ( Marks: 1 ) - Please choose one**

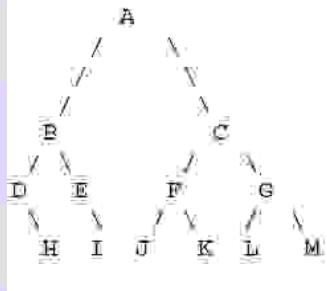
\_\_\_\_\_ is a binary tree where every node has a value, every node's left subtree contains only values less than or equal to the node's value, and every node's right subtree contains only values that are greater then or equal?

- ▶ Strictly Binary Tree
- ▶ **Binary Search tree** [Click here for detail](#)
- ▶ AVL tree
- ▶ All of these

**Question No: 14 ( Marks: 1 ) - Please choose one**

Consider the following binary search tree (BST):





If node A in the BST is deleted, which two nodes are the candidates to take its place?

- ▶ J and I
- ▶ H and E
- ▶ D and E
- ▶ **L and M**

**Question No: 15 ( Marks: 1 ) - Please choose one**

Let's call the node as that requires re-balancing. Consider the two cases given below:

- 1) An insertion into left sub tree of the left child of a
- 2) An insertion into right sub tree of the right child of a.

Which of the following statement is correct about these two cases?

- 1) The insertion occurs outside (i.e., left-left or right-right) in cases 1 and 2 single rotation can fix the balance in these two cases.
- 2) The insertion occurs inside ((i.e., left-left or right-right) in cases 1 and 2. single rotation cannot fix the balance in these two cases

**Question No: 16 ( Marks: 1 ) - Please choose one**

We access elements in AVL Tree in,

- ▶ Linear way only
- ▶ **Non Linear way only**
- ▶ Both linear and non linear ways
- ▶ None of the given options.

**Question No: 17 ( Marks: 2 )**

AVL Tree is,

- ▶ **Non Linear data structure** [Click here for detail](#)
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.



## MIDTERM EXAMINATION

### Spring 2010

#### Question No: 1 ( Marks: 1 ) - Please choose one

Which one of the following statement is NOT correct .

- ▶ **In linked list the elements are necessarily to be contiguous** [Click here for detail](#)
- ▶ In linked list the elements may locate at far positions in the memory
- ▶ In linked list each element also has the next to it
- ▶ In an array the elements are contiguous

#### Question No: 2 ( Marks: 1 ) - Please choose one

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

- ▶ One
- ▶ **Two (Page 67)**
- ▶ Three
- ▶ Four

#### Question No: 3 ( Marks: 1 ) - Please choose one

Which one of the following calling methods does not change the original value of the argument in the calling function?

- ▶ None of the given options
- ▶ **Call by passing the value of the argument** [Click here for detail](#)
- ▶ Call by passing reference of the argument
- ▶ Call by passing the address of the argument

#### Question No: 4 ( Marks: 1 ) - Please choose one

A tree is an AVL tree if

- ▶ Any one node fulfills the AVL condition
- ▶ At least half of the nodes fulfill the AVL condition
- ▶ **All the nodes fulfill the AVL condition (Page 213)**
- ▶ None of the given options

#### Question No: 5 ( Marks: 1 ) - Please choose one

Suppose currentNode refers to a node in a linked list (using the Node class with member variables called data and nextNode). What statement changes currentNode so that it refers to the next node?

- ▶ currentNode ++;
- ▶ currentNode = nextNode;
- ▶ currentNode += nextNode;
- ▶ **currentNode = currentNode->nextNode;**

**Question No: 6 ( Marks: 1 ) - Please choose one**

A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True (Page 101)**

**Question No: 7 ( Marks: 1 ) - Please choose one**

Which one is a self- referential data type?

- ▶ Stack
- ▶ Queue
- ▶ Link list
- ▶ **All of these** [Click here for detail](#)

**Question No: 8 ( Marks: 1 ) - Please choose one**

Each node in doubly link list has,

- ▶ 1 pointer
- ▶ **2 pointers (Page 39)**
- ▶ 3 pointers
- ▶ 4 pointers

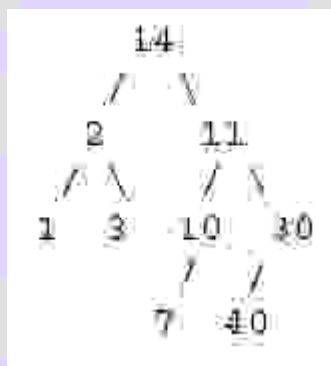
**Question No: 9 ( Marks: 1 ) - Please choose one**

I have implemented the queue with a linked list, keeping track of a front pointer and a rear pointer. Which of these pointers will change during an insertion into an *EMPTY* queue?

- ▶ Neither changes
- ▶ Only front pointer changes.
- ▶ **Only rear pointer changes.**
- ▶ Both change.

**Question No: 10 ( Marks: 1 ) - Please choose one**

Consider the following tree.



How many of the nodes have at least one sibling?

- ▶ 8
- ▶ 7
- ▶ 5
- ▶ 6

**A sibling is an element that shares the same parent with another element**

**Question No: 11 ( Marks: 1 ) - Please choose one**

The nodes with no successor are called \_\_\_\_\_

- ▶ Root Nodes
- ▶ **Leaf Nodes**
- ▶ Both of these
- ▶ None of these

**Question No: 12 ( Marks: 1 ) - Please choose one**

AVL Tree is,

- ▶ **Non Linear data structure** [Click here for detail](#)
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.

**Question No: 13 ( Marks: 1 ) - Please choose one**

We access elements in AVL Tree in,

- ▶ Linear way only
- ▶ **Non Linear way only**
- ▶ Both linear and non linear ways
- ▶ None of the given options.

**Question No: 14 ( Marks: 1 ) - Please choose one**

A binary search tree should have minimum of one \_\_\_\_\_ node/s at each level,

- ▶ One
- ▶ **Two** [Click here for detail](#)
- ▶ Three
- ▶ Four

**Question No: 15 ( Marks: 1 ) - Please choose one**

Consider the following statements.

- (i) A binary tree can contain at least  $2^L$  Nodes at level L.
- (ii) A complete binary tree of depth d is a binary tree that contains  $2^L$  Nodes at each level L between 0 and d, both inclusive.
- (iii) The total number of nodes ( $T_n$ ) in a complete binary tree of depth d is  $2^{d+1} - 1$ .
- (iv) The height of the complete binary tree can be written as  $h = \log_2 (T_n + 1) - 1$  where  $T_n$  is Total number of Nodes.

Which one of the following is correct in respect of the above statements regarding the Binary trees?

- ▶ (i) and (iii) only
- ▶ (i), (ii) and (iii) only
- ▶ (ii) and (iii) only
- ▶ **(ii), (iii) and (iv) only**      [Click here for detail](#)

**Question No: 16**    ( Marks: 1 ) - Please choose one  
“+” is a \_\_\_\_\_ operator.

- ▶ Unary
- ▶ **Binary**      (Page 64)
- ▶ Ternary
- ▶ None of the above

## MIDTERM EXAMINATION

### Spring 2010

**Question No: 1**    ( M a r k s: 1 )

A subscript of an array may be an integer or an integer expression.

- ▶ **True**      [Click here for detail](#)
- ▶ False

**Question No: 2**    ( M a r k s: 1 )

Doubly Linked List always has one NULL pointer.

- ▶ True
- ▶ **False**      (Page 43)

**Question No: 3**    ( M a r k s: 1 )

In which of the traversal method, the recursive calls can be used to traverse a binary tree ?

- ▶ **In preorder traversal only**      (Page 143)
- ▶ In inorder traversal only
- ▶ In postorder traversal only
- ▶ All of the given options

**Question No: 4**    ( M a r k s: 1 )

A tree is an AVL tree if

- ▶ Any one node fulfills the AVL condition
- ▶ At least half of the nodes fulfill the AVL condition
- ▶ **All the nodes fulfill the AVL condition**      (Page 213)
- ▶ None of the given options

**Question No: 5 ( Marks: 1 )**

Suppose currentNode refers to a node in a linked list (using the Node class with member variables called data and nextNode). What boolean expression will be true when cursor refers to the tail node of the list?

- ▶ **(currentNode == null)**
- ▶ (currentNode->nextNode == null)
- ▶ (nextNode.data == null)
- ▶ (currentNode.data == 0.0)

**Question No: 6 ( Marks: 1 ) - Please choose one**

Suppose that the class declaration of SomeClass includes the following function prototype.

bool LessThan( SomeClass anotherObject );

**Which of the following tests in the client code correctly compares two class objects alpha and beta?**

- ▶ if (alpha < beta)
- ▶ **if (alpha.LessThan(beta))** [Click here for detail](#)
- ▶ if (LessThan(alpha, beta))
- ▶ if (LessThan(alpha).beta)

**Question No: 7 ( Marks: 1 )**

In C what is the operation that you can not do with primitive types?

- ▶ Assign a value to primitive type using a literal
- ▶ Declare primitive types to be constant using the Const keyword
- ▶ **Create a new instance of primitive type with New keyword** [Click here for Detail](#)
- ▶ None of these

**Question No: 8 ( Marks: 1 )**

The operation for adding an entry to a stack is traditionally called :

- ▶ add
- ▶ append
- ▶ insert
- ▶ **push** (Page 53)

**Question No: 9 ( Marks: 1 )**

The operation for removing an entry from a stack is traditionally called:

- ▶ delete
- ▶ peek
- ▶ **pop** (Page 53)
- ▶ remove

**Question No: 10 ( Marks: 1 )**

Consider the following sequence of push operations in a stack:

```
stack.push('7');
stack.push('8');
stack.push('9');
stack.push('10');
stack.push('11');
stack.push('12');
```

- ▶ 7 8 9 10 11 12
- ▶ 9 8 11 10 7 12
- ▶ 9 10 8 11 12 7
- ▶ 9 10 8 12 7 11

**Question No: 11 ( Marks: 1 )**

\_\_\_\_\_ is the maximum number of nodes that you can have on a stack-linked list ?

- ▶ Zero
- ▶  $2n$  (where  $n$  is the number of nodes in linked list)
- ▶ **Any Number** [Click here for detail](#)
- ▶ None of these

**Question No: 12 ( Marks: 1 )**

Which of the following can be used to reverse a string value,

- ▶ **Stack** [Click here for detail](#)
- ▶ Queue
- ▶ Both of these
- ▶ None of these

**Question No: 14 ( Marks: 1 )**

AVL Tree is,

- ▶ **Non Linear data structure** [Click here for detail](#)
- ▶ Linear data structure
- ▶ Hybrid data structure (Mixture of Linear and Non Linear)
- ▶ None of the given options.

**Question No: 15 ( Marks: 1 )**

The following are statements related to queues.

- (i) The last item to be added to a queue is the first item to be removed
- (ii) A queue is a structure in which both ends are not used
- (iii) The last element hasn't to wait until all elements preceding it on the queue are removed
- (iv) A queue is said to be a last-in-first-out list or LIFO data structure.

Which of the above is/are related to normal queues?

- ▶ (iii) and (ii) only
- ▶ (i), (ii) and (iv) only
- ▶ (ii) and (iv) only
- ▶ **None of the given options** [Click here for detail](#)

**Question No: 16 ( Marks: 1 )**

An array is a group of consecutive related memory locations.

- ▶ **True** [Click here for detail](#)
- ▶ False

## MIDTERM EXAMINATION

### Spring 2010

#### Question No: 1 ( Marks: 1 ) - Please choose one

In an array we can store data elements of different types.

- ▶ True
- ▶ **False** (Page 7)

#### Question No: 2 ( Marks: 1 ) - Please choose one

In an array list the current element is

- ▶ **The first element** [Click here for detail](#)
- ▶ The middle element
- ▶ The last element
- ▶ The element where the current pointer points to

#### Question No: 3 ( Marks: 1 ) - Please choose one

Which one of the following calling methods does not change the original value of the argument in the calling function?

- ▶ None of the given options
- ▶ **Call by passing the value of the argument** [Click here for detail](#)
- ▶ Call by passing reference of the argument
- ▶ Call by passing the address of the argument

#### Question No: 4 ( Marks: 1 ) - Please choose one

Which one of the following statements is NOT correct?

- ▶ **Array size can be changed after its creation.** [Click here for detail](#)
- ▶ Link List size can be changed after its creation
- ▶ Binary Search Tree size can be changed after its creation
- ▶ AVL Tree size can be changed after its creation

#### Question No: 5 ( Marks: 1 ) - Please choose one

Suppose that the class declaration of SomeClass includes the following function prototype. `bool LessThan( SomeClass anotherObject );`

Which of the following tests in the client code correctly compares two class objects alpha and beta?

- ▶ `if (alpha < beta)`
- ▶ **`if (alpha.LessThan(beta))`** [Click here for detail](#)
- ▶ `if (LessThan(alpha, beta))`
- ▶ `if (LessThan(alpha).beta)`

#### Question No: 6 ( Marks: 1 ) - Please choose one

A queue is a----- data structure, whereas a stack is a -----data structure.

- ▶ **FIFO, LIFO** (Page 161,54)
- ▶ LIFO,FIFO
- ▶ none of these
- ▶ both of these

**Question No: 7 ( Marks: 1 ) - Please choose one**

Which one of the following operators has higher priority than all of others?

- ▶ **Multiplication operator** [Click here for detail](#)
- ▶ Minus operator
- ▶ Plus operator
- ▶ Exponentiation operator

**Question No: 8 ( Marks: 1 ) - Please choose one**

Each node in Binary Search Tree has

- ▶ 1 pointer
- ▶ **2 pointers** [Click here for detail](#)
- ▶ 3 pointers
- ▶ 4 pointers

**Question No: 9 ( Marks: 1 ) - Please choose one**

Four statements about trees are below. Three of them are correct. Which one is INCORRECT?

- ▶ Trees are recursively defined multi-dimensional data structures tree
- ▶ The order of a tree indicates a maximum number of children allowed at each node of the
- ▶ A search tree is a special type of tree where all values (i.e. keys) are ordered
- ▶ **If Tree1's size is greater than Tree2's size, then the height of Tree1 must also be greater than Tree2's height.** [Click here for detail](#)

**Question No: 10 ( Marks: 1 ) - Please choose one**

Which of the following is "TRUE" about arrays,

- ▶ We can increase the size of arrays after their creation.
- ▶ We can decrease the size of arrays after their creation.
- ▶ We can increase but can't decrease the size of arrays after their creation.
- ▶ **We can neither increase nor decrease the array size after their creation.** [Click here for detail](#)

**Question No: 11 ( Marks: 1 ) - Please choose one**

Searching an element in an AVL tree take maximum in AVL tree,

- ▶  $\log_2(n+1)$   
time (where n is no. of nodes)
- ▶  $\log_2(n+1) - 1$
- ▶  **$1.44 \log_2 n$**  (Page 227)
- ▶  $1.66 \log_2 n$

**Question No: 12 ( Marks: 1 ) - Please choose one**

There is/are case/s for rotation in an AVL tree,

- ▶ 1
- ▶ 3
- ▶ 2
- ▶ **4** (Page 229)



**Question No: 13 ( Marks: 1 ) - Please choose one**

Consider the following statements.

- (i) A binary tree can contain at least  $2L$  Nodes at level  $L$ .
- (ii) A complete binary tree of depth  $d$  is a binary tree that contains  $2L$  Nodes at each level  $L$  between 0 and  $d$ , both inclusive.
- (iii) The total number of nodes ( $T_n$ ) in a complete binary tree of depth  $d$  is  $2^{d+1} - 1$ .
- (iv) The height of the complete binary tree can be written as  $h = \log_2(T_n + 1) - 1$  where  $T_n$  is Total number of Nodes.

**Which one of the following is correct in respect of the above statements regarding the Binary trees?**

- ▶ (i) and (iii) only
- ▶ (i), (ii) and (iii) only
- ▶ (ii) and (iii) only
- ▶ (ii), (iii) and (iv) only [Click here for detail](#)

**Question No: 14 ( Marks: 1 ) - Please choose one**

Consider the following infix expression.

$$5 + 6/2$$

If one converts the above expression into postfix, what would be the resultant expression?

- ▶  $56/ + 2$
- ▶ **5 6 2 / +** (Page 66)
- ▶  $5 6 / 2 +$
- ▶  $/62 + 5$

**Question No: 15 ( Marks: 1 ) - Please choose one**

Which of the following is a non linear data structure?

- ▶ Linked List
- ▶ Stack
- ▶ Queue
- ▶ **Tree** (Page 112)

**Question No: 16 ( Marks: 1 ) - Please choose one**

“+” is a operator.

- ▶ Unary
- ▶ **Binary** (Page 64)
- ▶ Ternary
- ▶ None of the above

## MIDTERM EXAMINATION

### Spring 2010

1. Addition of new items in stack make the pointer ----- by 2

- a. Increment, bits
- b. Increment, bytes
- c. Decrement, bits
- d. Decrement, bytes**      [Click here for detail](#)

2. Next item in a linked list is known as

- a. Index
- b. Item
- c. Node**      [Click here for detail](#)
- d. Child

3. What will be the postfix notation of  $5+6/2$ .

- a.  $56+/2$
- b.  $562+/-$
- c.  $562/+$**       (Page 66)
- d.  $5+62/$

4. In an AVL tree to delete a parent with two childs in a straight line following rotations will be required:-

- a. Single
- b. Double**
- c. Triple
- d. None.

5. To check the depth of an AVL tree following time will be taken:-

- a.  $1.66 \log_2 n$
- b.  $1.44 \log_2 n$**       (Page 227)
- c.  $\log_2 (n+1)-1$
- d.  $1.66 \log_2 (n+1)$

6. BST is a Structure:-

- a. Linear
- b. Non Linear**      [Click here for detail](#)
- c. Circular
- d. None of Above

7. After creation of an array:-

- a. Size can be increase but can not be decreased.
- b. Size can be decreased but can not be increased.
- c. Size can neither be increased nor be decreased.**      [Click here for detail](#)
- d. Size can be increased and can also be decreased.

8. Each node in a BST has Pointers:-

- a. 1
- b. 2**      [Click here for detail](#)
- c. 3
- d. 4

9. Highest Operators Precedence is of the following operator:-

- a. Plus
- b. Minus
- c. Multiply**      [Click here for detail](#)
- d. Exponentiation

10. Following are the linear data structures:-

- a. Stacks
- b. Queues
- c. Both a & b (Page 52, 87)**
- d. None of the above

11. Each entry which points to a null value in a Singly Linked List is known as:-

- a. Node
- b. First Node
- c. Last Node**
- d. Head Node

12. Non recursive calls are faster than the Recursive calls.

- a. True (Page 323)**
- b. False

13. Tree data structure is a

- a. Linear
- b. Non Linear (Page 112) rep**
- c. Circular
- d. None of Above

14. What will be the valid postfix notation of  $A+B*C-D$

- a.  $ABC+*D-$
- b.  $ABC*+D-$  (According to rule)**
- c.  $ABCD+-*$
- d.  $AB+D*C$

15. When an operator is used in between two operands this is which type of notation

- a. Prefix
- b. Postfix
- c. Infix (Page 64)**
- d. None of the Above

# MIDTERM EXAMINATION

## Spring 2009

### Question No: 1 ( Marks: 1 ) - Please choose one

Which one of the following is a valid postfix expression?

- ▶  $ab+c*d-$
- ▶  **$abc**d-$  (According to rule)**
- ▶  $abc+*d-$
- ▶  $(abc*)+d-$

### Question No: 2 ( Marks: 1 ) - Please choose one

The tree data structure is a

- ▶ Linear data structure
- ▶ **Non-linear data structure (Page 112)**
- ▶ Graphical data structure
- ▶ Data structure like queue

### Question No: 3 ( Marks: 1 ) - Please choose one

A **Compound Data Structure** is the data structure which can have multiple data items of same type or of different types. Which of the following can be considered compound data structure?

- ▶ **Arrays**      [Click here for detail](#)
- ▶ LinkLists
- ▶ Binary Search Trees
- ▶ All of the given options

### Question No: 4 ( Marks: 1 ) - Please choose one

Suppose a pointer has been declared in main but has not assigned any variable address then

- ▶ That pointer points to First byte in main function
- ▶ **That pointer contains a NULL value**
- ▶ None of these
- ▶ **That pointer points to any memory address**

### Question No: 5 ( Marks: 1 ) - Please choose one

Here is the start of a C++ class declaration:

```
class foo
{
public:
void x(foo f);
void y(const foo f);
void z(foo f) const;
...
```

Which of the three member functions can alter the **PRIVATE** member variables of the foo object that activates the function?

- ▶ **Only x can alter the private member variables of the object that activates the function.**
- ▶ Only y can alter the private member variables of the object that activates the function.
- ▶ Only z can alter the private member variables of the object that activates the function.

► **Two of the functions can alter the private member variables of the object that activates the function.**

Only the x and y can alter the private member variable of the foo class object. Last Option is more correct but not exact. In the last option the two function name are not mentioned

**Question No: 6 ( Marks: 1 ) - Please choose one**

The operation for removing an entry from a stack is traditionally called:

- delete
- peek
- **pop (Page 53)**
- remove

**Question No: 7 ( Marks: 1 ) - Please choose one**

Which statement of the following statements is incorrect?

- Lists can be implemented by using arrays or linked lists
- A list is a sequence of one or more data items
- Stack is a special kind of list in which all insertions and deletions take place at one end
- **Stacks are easier to implement than lists**

**Question No: 8 ( Marks: 1 ) - Please choose one**

Parameters in function call are passed using,

- **Stack (Page 80)**
- Queue
- Binary Search Tree
- AVL Tree

**Question No: 9 ( Marks: 1 ) - Please choose one**

Consider the following sequence of push operations in a stack:

```
stack.push('7');
stack.push('8');
stack.push('9');
stack.push('10');
stack.push('11');
stack.push('12');
```

- **7 8 9 10 11 12**
- 9 8 11 10 7 12
- 9 10 8 11 12 7
- 9 10 8 12 7 11

**Question No: 10 ( Marks: 1 ) - Please choose one**

What is the maximum depth of recursive calls a function may make?

- 1
- 2
- n (where n is the argument)
- **There is no fixed maximum**

**Question No: 11 ( Marks: 1 ) - Please choose one**

**Consider the following function:**

```
void test_a(int n)
{
 cout << n << " ";
 if (n>0)
 test_a(n-2);
}
```

What is printed by the call test\_a(4)?

- ▶ **4 2 0**
- ▶ 0 2 4
- ▶ 0 2
- ▶ 2 4

**Question No: 12 ( Marks: 1 ) - Please choose one**

Queue follows,

- ▶ Last in First out
- ▶ First in Last out
- ▶ **First in First out** (Page 87)
- ▶ None of these

**Question No: 13 ( Marks: 1 ) - Please choose one**

is a binary tree where every node has a value, every node's left subtree contains only values less than or equal to the node's value, and every node's right subtree contains only values that are greater then or equal ?

- ▶ **Strictly Binary Tree**
- ▶ **Binary Search tree** [Click here for detail](#)
- ▶ AVL tree
- ▶ All of these

**Question No: 14 ( Marks: 1 ) - Please choose one**

Four statements about trees are below. Three of them are correct. Which one is INCORRECT?

- ▶ **Trees are recursively defined multi-dimensional data structures** [Click here for detail](#)
- ▶ The order of a tree indicates a maximum number of children allowed at each node of the tree
- ▶ A search tree is a special type of tree where all values (i.e. keys) are ordered
- ▶ If Tree1's size is greater than Tree2's size, then the height of Tree1 must also be greater than Tree2's height.

**Question No: 15 ( Marks: 1 ) - Please choose one**

Below is a binary search tree. If we delete the value 50 using the algorithm we discussed, what value will be in the root of the remaining tree?

- ▶ **50**
- ▶ 60
- ▶ 70
- ▶ 80

**Question No: 16 ( Marks: 1 ) - Please choose one**

Is a data structure that can grow easily dynamically at run time without having to copy existing elements?

- ▶ Array
- ▶ **List**
- ▶ Both of these
- ▶ None of these

## **MIDTERM EXAMINATION**

### **Spring 2009**

**Question No: 1 ( Marks: 1 ) - Please choose one**

Which one of the following statement is NOT correct .

- ▶ **In linked list the elements are necessarily to be contiguous** [Click here for detail](#)
- ▶ In linked list the elements may locate at far positions in the memory
- ▶ In linked list each element also has the address of the element next to it
- ▶ In an array the elements are contiguous

**Question No: 2 ( Marks: 1 ) - Please choose one**

In a program a reference variable, say x, can be declared as

- ▶ **int &x ;** [Click here for detail](#)
- ▶ int \*x ;
- ▶ int x ;
- ▶ None of the given options

**Question No: 3 ( Marks: 1 ) - Please choose one**

Linked lists are collections of data items "lined up in a row" , insertions and deletions can be made only at the front and the back of a linked list.

- ▶ True
- ▶ **False** [Click here for detail](#)

**Question No: 4 ( Marks: 1 ) - Please choose one**

A Linear Data Structure is the data structure in which data elements are arranged in a sequence or a linear list. Which of the following is Non Linear Data Structure?

- ▶ Arrays
- ▶ LinkLists
- ▶ **Binary Search Trees** [Click here for detail](#)
- ▶ None of these

**Question No: 5 ( Marks: 1 ) - Please choose one**

A queue where the de-queue operation depends not on FIFO, is called a priority queue

- ▶ False
- ▶ **True** (Page 101)

**Question No: 6 ( Marks: 1 ) - Please choose one**

Which one of the following statements is correct?

- ▶ **Array size is fixed once it is created** [Click here for detail](#)
- ▶ Link List size is fixed once it is created.
- ▶ Binary Search Tree size is fixed once it is created
- ▶ AVL Tree size is fixed once it is created

**Question No: 7 ( Marks: 1 ) - Please choose one**

Which one of the following is correct about pointers?

- ▶ They always point to different memory locations
- ▶ They may point to a single memory location
- ▶ The address of two pointer variables is same
- ▶ None of these

**Question No: 8 ( Marks: 1 ) - Please choose one**

Which of the following abstract data types are NOT used by Integer Abstract Data type group?

- ▶ short
- ▶ Int
- ▶ **float** [Click here for detail](#)
- ▶ long

**Question No: 9 ( Marks: 1 ) - Please choose one**

The operation for adding an entry to a stack is traditionally called :

- ▶ add
- ▶ append
- ▶ insert
- ▶ **push** (Page 53)

**Question No: 10 ( Marks: 1 ) - Please choose one**

The operation for removing an entry from a stack is traditionally called:

- ▶ delete
- ▶ peek
- ▶ **pop** (Page 53)
- ▶ remove

**Question No: 11 ( Marks: 1 ) - Please choose one**

We can add elements in QUEUE From \_\_\_\_\_

- ▶ Front
- ▶ **Rear** (Page 91)
- ▶ From Both Rare and Front
- ▶ None of these



**Question No: 12 ( Marks: 1 ) - Please choose one**

The difference between a binary tree and a binary search tree is that ,a binary search tree has

▶ **two children per node whereas a binary tree can have none, one, or two children per node** [Click here for detail](#)

- ▶ in binary search tree nodes are inserted based on the values they contain
- ▶ in binary tree nodes are inserted based on the values they contain
- ▶ none of these

**Question No: 13 ( Marks: 1 ) - Please choose one**

Suppose n is the number of nodes in a complete Binary Tree then maximum steps required for a search operation are,

▶  **$\log_2(n+1) - 1$**  (Page 139)

- ▶  $\log_2(n+1)$
- ▶  $\log_2(n) - 1$
- ▶  $\log_2(n)$

**Question No: 14 ( Marks: 1 ) - Please choose one**

The following is a segment of a C program.

```
int pqr(BinaryNode t)
{ if (t == null)
return -1;
else
return 1+max(pqr(t.left),pqr(t.right)) }
```

Identify, what the above program intend(s) to do?

- ▶ Compute the height of a binary tree using an in-order traversal
- ▶ Compute the height of a binary tree using a pre-order traversal
- ▶ Compute the depth of a binary tree using a pre-order traversal
- ▶ Compute the depth of a binary tree using a post-order traversal

**Question No: 15 ( Marks: 1 ) - Please choose one**

**Consider the following infix expression:**

$3 + 5 * 6 - 7 * (8 + 5)$

Which of the following is a correct equivalent expression(s) for the above?

- ▶  $3 \ 6 \ 5 + * \ 7 \ 5 \ 8 + - *$
- ▶  $3 \ 6 \ 5 \ 7 \ 5 \ 8 + * + - *$
- ▶  $3 \ 5 \ 6 + * \ 7 \ 8 \ 5 + - *$
- ▶  **$3 \ 5 \ 6 * + \ 7 \ 8 \ 5 + * -$**

**Question No: 16 ( Marks: 1 ) - Please choose one**

**An array is a group of consecutive related memory locations.**

- ▶ **True** [Click here for detail](#)
- ▶ False

**Question No: 17 ( Marks: 1 )**

Is this a correct statement? Give answer in Yes or No.

A node cannot be deleted, when the node to be deleted has both left and right subtrees.

**False ---- No, it can be deleted.**

**Question No: 18 ( Marks: 1 )**

Deleting a leaf node in binary search tree involves setting \_\_\_\_\_ pointer/s of that node's parent as null.

- 1
- 2
- 3
- 4



# CS301 Data Structures Update MCQS For MidTerm Solve By Vu Topper RM



**80 To 100% Marks**



وَتَعَزُّ مِنْ تَشَاءِ وَتَذُلُّ مِنْ تَشَاءِ

**PROFESSIONAL ONLINE ACADEMY**



**We Offers**

- LMS Handling
- Important Notes
- Online Classes
- Assignments
- Quiz & GDB's
- Projects

**NOTHING  
IS  
IMPOSSIBLE**

**Join Us  
Now**

For More Info  
Contact us at:  
**Rizwan Manzoor**  
☎ **0322-4021365**



بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیک صحبت بہتر ہے

For More Help Contact What's app 03224021365

**Question No:1**

**(Marks:1)**

**Vu-Topper RM**

Which one is the correct function call for the following function of calculating cube?

- A. cube(num)**
- B. cube(\*num)
- C. cube(&num)
- D. cube(&&num)

**Question No:2**

**(Marks:1)**

**Vu-Topper RM**

In a complete binary tree, for 25000 nodes the depth will be-----

- A. 14**
- B. 13
- C. 12
- D. 11

**Question No:3**

**(Marks:1)**

**Vu-Topper RM**

In level-order traversal for Binary Search Tree, \_\_\_\_\_ data structure is used.

- A. Queue**
- B. Stack
- C. Tree
- D. Heap

**Question No:4**

**(Marks:1)**

**Vu-Topper RM**

If both left and right nodes of a node are NULL then this type of node is called a -----node.

- A. Leaf**
- B. Node
- C. Both
- D. None of these

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:5**

**(Marks:1)**

**Vu-Topper RM**

For searching a particular number in Binary Search Tree (if it is not present), the maximum number of comparisons will be \_\_\_\_\_ comparison(s) at each level.

**A. 1**

B. 2

C. 3

D. 4

**Question No:6**

**(Marks:1)**

**Vu-Topper RM**

In a complete binary tree the number of nodes at level 5 are

**A. 32**

B. 30

C. 23

D. 40

**Question No:7**

**(Marks:1)**

**Vu-Topper RM**

While implementing non-recursive traversal for Binary Search Tree, we need to implement \_\_\_\_\_.

**A. Stack**

B. Tree

C. Heap

D. Pointer

**Question No:8**

**(Marks:1)**

**Vu-Topper RM**

----- is used for Reference variables in C++.

**A. &**

B. @

C. #

D. \*

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نيکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:9**

**(Marks:1)**

**Vu-Topper RM**

Left, right, info, and parent are the operations of \_\_\_\_\_ data structure.

**A. Tree**

B. Heap

C. Stack

D. Pointer

**Question No:10**

**(Marks:1)**

**Vu-Topper RM**

If a function has recursive call as the last statement, it is known as \_\_\_\_.

A. Local

B. Last

C. Function

**D. Tail recursive**

**Question No:11**

**(Marks:1)**

**Vu-Topper RM**

The abstract data type refers to the basic mathematical concept that defines the

A. Variable

B. Functions

C. Pointer

**D. Data type**

**Question No:12**

**(Marks:1)**

**Vu-Topper RM**

When a function calls another function, the parameters and return address of the calling function are put in \_\_\_\_\_.

**A. Stack**

B. Tree

C. Heap

D. Pointer

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نيکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:13**

**(Marks:1)**

**Vu-Topper RM**

The balance of a node in a binary tree is defined as the height of its \_\_\_\_\_ sub tree minus height of its right sub tree.

**A. Left**

B. Right

C. Upper

D. Lower

**Question No:14**

**(Marks:1)**

**Vu-Topper RM**

Function signatures are also called :

A. Function definition

**B. Function prototype**

C. Function overriding

D. Function overloading

**Question No:15**

**(Marks:1)**

**Vu-Topper RM**

If we return the reference of a local variable from a function it will cause:

**A. Dangling reference**

B. reference overloading

C. duplication of local variable

D. deletion of local variable from memory

**Question No:16**

**(Marks:1)**

**Vu-Topper RM**

In the perspective of memory organization each process is divided into \_\_\_\_\_ sections.

A. 2

**B. 4**

C. 6

D. 8

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نيکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:17**

**(Marks:1)**

**Vu-Topper RM**

The process of getting the value of a variable using pointers is called:

- A. Referencing
- B. Dereferencing**
- C. Memory allocation
- D. Memory deallocation

**Question No:18**

**(Marks:1)**

**Vu-Topper RM**

We allocate memory dynamically by using \_\_\_\_\_ operator.

- A. New**
- B. This
- C. Increment
- D. Decrement

**Question No:19**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ parameter passing (by value or by reference) is similar to PASCAL.

- A. C++**
- B. JAVA
- C. COBOL
- D. FORTRAN

**Question No:20**

**(Marks:1)**

**Vu-Topper RM**

Which of the following is known as "Last-In, First-Out" or LIFO Data Structure?

- A. Tree
- B. Stack**
- C. Heap
- D. Pointer

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**



**Question No:21**

**(Marks:1)**

**Vu-Topper RM**

Following is a keyword of C++ \_\_\_\_\_

**A. Delete**

B. Update

C. Remove

D. Eliminate

**Question No:22**

**(Marks:1)**

**Vu-Topper RM**

If the root of a tree is at level zero, its two children (subtrees) i.e. nodes will be at \_\_\_\_\_

**A. Level 1**

B. Level 2

C. Level 3

D. Level 4

**Question No:23**

**(Marks:1)**

**Vu-Topper RM**

If the root of a tree is at level three in case of a complete binary tree i.e nodes will be at \_\_\_\_\_.

A. 4

B. 6

**C. 8**

D. 10

**Question No:24**

**(Marks:1)**

**Vu-Topper RM**

Suppose we have the following values to be inserted in constructing AVL

tree, 10, 13, 15, 5, 7, 8 Tell when first rotation will take place,

A. After inserting the node 10

**B. After inserting the node 15**

C. After inserting the node 20

D. After inserting the node 25

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:25**

**(Marks:1)**

**Vu-Topper RM**

Which data structure is needed to convert infix expression to postfix expression?

- A. Tree
- B. Stack**
- C. Heap
- D. Pointer

**Question No:26**

**(Marks:1)**

**Vu-Topper RM**

In AVL tree during insertion, a single rotation can fix the balance in cases \_\_\_\_\_ and.

- A. 1**
- B. 2
- C. 3
- D. 4

**Question No:27**

**(Marks:1)**

**Vu-Topper RM**

Which type of rotation can balance the following AVL tree?

- A. Single left
- B. Single right
- C. Double right-left
- D. Double left-right**

**Question No:28**

**(Marks:1)**

**Vu-Topper RM**

If the root of a tree is at level two in case of a complete binary tree i.e nodes will be at \_\_\_\_\_.

- A. 2
- B. 4**
- C. 6
- D. 8

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:29**

**(Marks:1)**

**Vu-Topper RM**

To search an element in ALV tree, it takes maximum  $1.88 \log_2 n$  time.

**A. False**

B. True

C. In some cases

D. Searching cannot be performed in AVL tree

**Question No:30**

**(Marks:1)**

**Vu-Topper RM**

Local variables of a function are stored in,

**A. Stack**

B. Tree

C. Heap

D. Pointer

**Question No:31**

**(Marks:1)**

**Vu-Topper RM**

For making Binary Search Tree for Strings we need, \_\_\_\_\_ data type.

**A. Int**

B. Char

C. Float

D. Double

**Question No:32**

**(Marks:1)**

**Vu-Topper RM**

We can make a lexicographic order of characters based on their \_\_\_\_.

**A. ASCII values**

B. Binary digits

C. Random choice

D. Memory addresses

**Question No:33**

**(Marks:1)**

**Vu-Topper RM**

A BST generated from the data in ascending order is \_\_\_\_\_.

**A. Linear**

B. Un sorted

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نيکي صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

- C. Balanced
- D. Nonlinear

**Question No:34**

**(Marks:1)**

**Vu-Topper RM**

The \_\_\_\_\_ of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

- A. Width
- B. Level
- C. Height
- D. Balance**

**Question No:35**

**(Marks:1)**

**Vu-Topper RM**

All the objects created using \_\_\_\_\_ operator have to be explicitly destroyed using the delete operator.

- A. New**
- B. Build
- C. Create
- D. Construct

**Question No:36**

**(Marks:1)**

**Vu-Topper RM**

Which of the following is a nonlinear data structure?

- A. Tree**
- B. Heap
- C. Stack
- D. Pointer

**Question No:37**

**(Marks:1)**

**Vu-Topper RM**

Consider the following push operations of a Stack:

- A. Stack.push(4);
- B. Stack.push(6);
- C. Stack.push(5);
- D. Stack.push(8);**

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:38**

**(Marks:1)**

**Vu-Topper RM**

~BinarySearchTree( ) is a \_\_\_\_\_.

**A. Destructor**

B. Constructor

C. Switch case

D. Template method call

**Question No:39**

**(Marks:1)**

**Vu-Topper RM**

HOW many cases of rotation are there in AVL tree?

A. 2

**B. 4**

C. 6

D. 8

**Question No:40**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ tree has been named after two persons Adelson-Velskii and Landis.

**A. AVL**

B. RED

C. Tree

D. Binary search

**Question No:41**

**(Marks:1)**

**Vu-Topper RM**

If there is a strictly complete binary tree of depth 3, the total number of nodes in it will be?

**A. 2K**

B. 0k

C. 1k

D. 3k

**Question No:42**

**(Marks:1)**

**Vu-Topper RM**

There are \_\_\_\_\_cases of Rotation in AVL tree.

A. 2

**B. 4**

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نيکي صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

- C. 6
- D. 8

**Question No:43**

**(Marks:1)**

**Vu-Topper RM**

If one converts the above expression into postfix, what would be the resultant expression?

- A. 78/9+**
- B. 87/9+
- C. +78/\*
- D. 9/78+

**Question No:44**

**(Marks:1)**

**Vu-Topper RM**

In simple or singly linked list there is/are \_\_\_\_\_ pointer/s in each node.

- A. One**
- B. Two
- C. Four
- D. Three

**Question No:45**

**(Marks:1)**

**Vu-Topper RM**

The principal benefit of a linked list over a conventional array is that the order of the linked items may be \_\_\_\_\_ from the order that the data items are stored in memory.

- A. Same**
- B. Different
- C. Both
- D. None of these

**Question No:46**

**(Marks:1)**

**Vu-Topper RM**

The computer memory can be thought of as a/an

- A. Tree
- B. Array**
- C. Heap

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

D. Pointer

**Question No:47**

**(Marks:1)**

**Vu-Topper RM**

Suppose we have a value for Queue: 2 1 4 6 3 Where front is at 2 and rear is at 3, after dequeue one element from that Queue, what will be the resultant Queue?

**A. 2146**

B. 2141

C. 4621

D. 2147

**Question No:48**

**(Marks:1)**

**Vu-Topper RM**

The -- is a decrement operator in C++ that decreases the value of the operand by \_\_\_\_\_.

**A. One**

B. Two

C. Five

D. Six

**Question No:49**

**(Marks:1)**

**Vu-Topper RM**

In Left-Right case of rotation in \_\_\_\_\_ tree. A double rotation is performed.

**A. AVL**

B. VLA

C. EVL

D. LVA

**Question No:50**

**(Marks:1)**

**Vu-Topper RM**

The \_\_\_\_\_ of every node should be 1, 0 or -1 otherwise, it will not be an AVL tree.

A. Tree

B. AVL

C. Pointer

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

## D. Balance

**Question No:51**

**(Marks:1)**

**Vu-Topper RM**

In singly linked list a node comprises of \_\_\_\_\_ field/s.

**A. Two**

B. One

C. Six

D. Five

**Question No:52**

**(Marks:1)**

**Vu-Topper RM**

`int htdiff = height(root->getLeft()) height(root->getRight());` The above line of code is taken from AVL insert method. Complete it by selecting an appropriate symbol.

**A. Minus (-)**

B. Add

C. Sub

D. Multiple

**Question No:53**

**(Marks:1)**

**Vu-Topper RM**

Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

**A. Two**

B. Three

C. Four

D. Five

**Question No:54**

**(Marks:1)**

**Vu-Topper RM**

The \_\_\_\_\_ method of list will position the currentNode and lastCurrentNode at the start of the list.

**A. Start**

B. End

C. Middle

D. None of these

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**



**Question No:55**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ is utilized at the time of memory allocation in dynamic manner.

**A. Heap**

B. Stack

C. Array

D. Function

**Question No:56**

**(Marks:1)**

**Vu-Topper RM**

What will be the result of evaluating following expression?  $5+3*2/(6-3)$

A. 1

B. 3

**C. 7**

D. 5

**Question No:57**

**(Marks:1)**

**Vu-Topper RM**

Which of the following is the correct conversion of infix to postfix expression?

A.  $Z+B-(D-H)/K$

**B.  $ZB+DH-K/-$**

C.  $Z+B-(H-D)/K$

D.  $Z+B-(D-H)$

**Question No:58**

**(Marks:1)**

**Vu-Topper RM**

The \_\_\_\_\_ of a binary tree is the maximum level of its leaves (also called the depth).

A. Width

B. Level

**C. Height**

D. Balance

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:59**

**(Marks:1)**

**Vu-Topper RM**

Memory address is stored in

- A. Array
- B. Int
- C. Pointer**
- D. Function

**Question No:60**

**(Marks:1)**

**Vu-Topper RM**

If numbers 5, 222, 4, 48 are inserted in a queue, which one will be removed first?

- A. 5**
- B. 6
- C. 7
- D. 8

**Question No:61**

**(Marks:1)**

**Vu-Topper RM**

Consider the following infix expression.

- A.  $7/8 + 9$**
- B.  $8/7+9$
- C.  $-7/8+9$
- D.  $78/+9$

**Question No:62**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ is the major factor to see the efficiency of a program.

- A. Time**
- B. Cost
- C. Maintance
- D. None of these

**Question No:63**

**(Marks:1)**

**Vu-Topper RM**

There are four cases of rotation in an \_\_\_\_\_ tree.

- A. AVL**
- B. Tree

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

- C. Binary
- D. Heap

**Question No:64**

**(Marks:1)**

**Vu-Topper RM**

The back() method decreases the value of variable current by \_\_\_\_.

**A. One**

- B. Two
- C. Four
- D. Three

**Question No:65**

**(Marks:1)**

**Vu-Topper RM**

Each node in singly linked list contains\_\_\_\_\_

**A. One Pointers**

- B. Two Pointers
- C. Six Pointers
- D. Five Pointers

**Question No:66**

**(Marks:1)**

**Vu-Topper RM**

The \_\_\_\_\_ symbol is used when we want to get the value of a variable using pointer

**A. \***

- B. @
- C. &
- D. #

**Question No:67**

**(Marks:1)**

**Vu-Topper RM**

A zigzag rotation is performed. In Left-Left case of rotation in AVL tree.

**False**

Ture

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:68**

**(Marks:1)**

**Vu-Topper RM**

The lifetime of a transient object cannot exceed that of the application.

**True**

False

**Question No:69**

**(Marks:1)**

**Vu-Topper RM**

AVL tree is nonlinear data structure.

**True**

False

**Question No:70**

**(Marks:1)**

**Vu-Topper RM**

AVL tree is linear data structure.

**False**

True

**Question No:71**

**(Marks:1)**

**Vu-Topper RM**

AVL tree is a binary search tree

**True**

False

**Question No:72**

**(Marks:1)**

**Vu-Topper RM**

Elements in a queue data structure are added from\_\_\_\_\_ and removed from-----?

**Rear end. front end**

**Question No:73**

**(Marks:1)**

**Vu-Topper RM**

Which of the following traversal method traverses the binary tree in sorted order?

**inOrder() method**

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:74**

**(Marks:1)**

**Vu-Topper RM**

In which traversal method, the recursive calls can be used to traverse a binary tree?

**In preorder traversal**

**Question No:75**

**(Marks:1)**

**Vu-Topper RM**

Which of the following is TRUE for search operations in a binary tree as compared to linked list or an array?

**It increases the number of comparisons**

**Question No:76**

**(Marks:1)**

**Vu-Topper RM**

In a tree, we link the nodes in such a way that it \_\_\_\_\_ a linear structure.

**Does not remain**

**Question No:77**

**(Marks:1)**

**Vu-Topper RM**

When a function calling itself is called as \_\_\_\_\_.

**Recursion**

**Question No:78**

**(Marks:1)**

**Vu-Topper RM**

In a program a reference variable with name "x" can be declared as

**int &x;**

**Question No:79**

**(Marks:1)**

**Vu-Topper RM**

A binary tree is said to be a \_\_\_\_\_ binary tree if every non-leaf node in a binary tree has nonempty left and right subtrees.

**Strictly binary tree**

**Question No:80**

**(Marks:1)**

**Vu-Topper RM**

We can calculate the \_\_\_\_\_ of a subtree by counting its levels from the bottom.

**Height**

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:81**

**(Marks:1)**

**Vu-Topper RM**

In C++, we place the class interface in \_\_\_\_\_file.

**.cpp file**

**Question No:82**

**(Marks:1)**

**Vu-Topper RM**

In which traversal method root node is visited at last step?

**Post-order Traversal**

**Question No:83**

**(Marks:1)**

**Vu-Topper RM**

For a complete binary tree with n numbers of nodes, the depth is calculated as \_\_\_\_\_

**$\log_2(\text{number of nodes}+1)-1$**

**Question No:84**

**(Marks:1)**

**Vu-Topper RM**

A \_\_\_\_\_ is a tree in which every level, except possibly the last, is completely filled.

**Complete binary tree**

**Question No:85**

**(Marks:1)**

**Vu-Topper RM**

Local variables defined inside function body are \_\_\_\_\_ automatically at the end of function execution.

**Destroyed**

**Question No:86**

**(Marks:1)**

**Vu-Topper RM**

BinarySearchTree () is-----?

**Destroyed**

**Question No:87**

**(Marks:1)**

**Vu-Topper RM**

Sub-tree of binary search tree should be \_\_\_\_\_.

**Binary search tree**

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:88**

**(Marks:1)**

**Vu-Topper RM**

Josephus problem is resolved by the implementation of\_\_\_\_\_.

**Circular linked list**

**Question No:89**

**(Marks:1)**

**Vu-Topper RM**

In which of the following tree for each node, the value of root node is greater than left child and smaller than right child ?

**Binary search tree**

**Question No:90**

**(Marks:1)**

**Vu-Topper RM**

If we use singly linked list to implement list, then there is an issue that it gives difficulty when we:

**We will increase its size**

**Question No:91**

**(Marks:1)**

**Vu-Topper RM**

In a list, tail() method of current pointer \_\_\_\_\_

**Moves the "current" pointer to the very last element**

**Question No:92**

**(Marks:1)**

**Vu-Topper RM**

Two common models of simulation are \_\_\_\_\_ and \_\_\_\_\_.

**Time –based simulation and Event- based simulation**

**Question No:93**

**(Marks:1)**

**Vu-Topper RM**

When add() operation of a linked list is called the following action is done

**A new node is made**

**Question No:94**

**(Marks:1)**

**Vu-Topper RM**

What will be the postfix expression of following infix expression?

**A B + C D / + E –**

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:95**

**(Marks:1)**

**Vu-Topper RM**

Stack and Queue can be implemented using \_\_\_\_\_,

**Singly Link List**

**Question No:96**

**(Marks:1)**

**Vu-Topper RM**

What will be postfix expression of the following infix expression?

Infix Expression

**Abc\*+d-**

**Question No:97**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ method returns the top element of the stack without removing it.

**Top()**

**Question No:98**

**(Marks:1)**

**Vu-Topper RM**

In tree, the search operation is \_\_\_\_\_ as compared to the linked list.

**Very fast**

**Question No:99**

**(Marks:1)**

**Vu-Topper RM**

In \_\_\_\_\_, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node.

**Linked list**

**Question No:100**

**(Marks:1)**

**Vu-Topper RM**

A queue is a data structure where elements are

**Inserted and removed from both ends**

**Question No:101**

**(Marks:1)**

**Vu-Topper RM**

The stack implementing an array causes worst case behavior when the insertion and deletion of an element done from

**Beginning of an array**

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**



**Question No:102**

**(Marks:1)**

**Vu-Topper RM**

If a node is inserted in outer side of a node in binary search tree then to make it AVL tree,

**We may have to apply single rotation**

**Question No:103**

**(Marks:1)**

**Vu-Topper RM**

Stack.push(15) will push 15 on \_\_\_\_\_.

**Top of the stack**

**Question No:104**

**(Marks:1)**

**Vu-Topper RM**

During the execution of a process, operating system constructs four things for that process. Which of the following is not part of that process?

**Liked list**

**Question No:105**

**(Marks:1)**

**Vu-Topper RM**

Suppose there are three nodes tree with three numbers 14, 4, 15. Following is not a permutation, or combination for output them.

**(4,4, 15)**

**Question No:106**

**(Marks:1)**

**Vu-Topper RM**

The type of expression in which operator succeeds its operands is \_\_\_\_\_ expression.

**Postfix**

**Question No:107**

**(Marks:1)**

**Vu-Topper RM**

The function calls are made with the help of \_\_\_\_\_.

**Queue**

**Question No:108**

**(Marks:1)**

**Vu-Topper RM**

In case of insertion of right inner node in BST,

**Left rotation then right to make AVL**

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نيکي صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:109**

**(Marks:1)**

**Vu-Topper RM**

Which operation of queue data structure is used to get front element from the queue and then remove it from the queue?

**Remove()**

**Question No:110**

**(Marks:1)**

**Vu-Topper RM**

In \_\_\_\_\_ various cells of memory are not located continuously.

**Linked list**

**Question No:111**

**(Marks:1)**

**Vu-Topper RM**

Which operation of queue data structure is used to insert an element into the Queue?

**Enqueuer()**

**Question No:112**

**(Marks:1)**

**Vu-Topper RM**

From Operating System point of view, the recursive function calls are made with the help of \_\_\_\_\_.

**Queue**

**Question No:113**

**(Marks:1)**

**Vu-Topper RM**

Array cells are \_\_\_\_\_ in computer memory.

**Contiguous**

**Question No:114**

**(Marks:1)**

**Vu-Topper RM**

Leaf node of binary search tree contains \_\_\_\_\_

**One null pointer**

**Question No:115**

**(Marks:1)**

**Vu-Topper RM**

dequeue() operation of queue data structure is used to \_\_\_\_\_

**remove an element from the front and return it**

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نيکي صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:116**

**(Marks:1)**

**Vu-Topper RM**

The depth of a binary tree is

**Maximum level of a leaf**

**Question No:117**

**(Marks:1)**

**Vu-Topper RM**

Which of the following operation returns but do not removes top value of the stack?

**Top**

**Question No:118**

**(Marks:1)**

**Vu-Topper RM**

In doubly linked list a node consists of three parts:

**2 pointer and 1 object**

**Question No:119**

**(Marks:1)**

**Vu-Topper RM**

Last node in circular linked list contains

**No null pointer**

**Question No:120**

**(Marks:1)**

**Vu-Topper RM**

A \_\_\_\_\_ model attempts to model a real-world phenomenon

**Simulation**

**Question No:121**

**(Marks:1)**

**Vu-Topper RM**

Factorial is an example of \_\_\_\_\_ function.

**Recursive**

**Question No:122**

**(Marks:1)**

**Vu-Topper RM**

Which one of the following calling method does not change the original value of the argument in the calling function?

**Call by passing the value of the argument**

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:123**

**(Marks:1)**

**Vu-Topper RM**

In level-order traversal for Binary Search Tree, we visit the nodes at each level before proceeding to the next level, in a \_\_\_\_\_ order.

**Left –to –right**

**Question No:124**

**(Marks:1)**

**Vu-Topper RM**

Binary Search Tree violates the condition of AVL tree when any node has balance equal to

**1or-1**

**Question No:125**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ is when function is calling to itself.

**Recursion**

**Question No:126**

**(Marks:1)**

**Vu-Topper RM**

Which of the following function don't belongs to the stack class?

**Crash()**

**Question No:127**

**(Marks:1)**

**Vu-Topper RM**

copy() method of list data structure \_\_\_\_\_

**Set one list to be a copy of another**

**Question No:128**

**(Marks:1)**

**Vu-Topper RM**

Following is true in case of using Recursive method calls

**The code becomes very short**

**Question No:129**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ is a self-balancing tree.

**Binary Search Tree**

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:130**

**(Marks:1)**

**Vu-Topper RM**

While implementing stack with an array and to achieve LIFO behavior, we used push and pop elements at

**The start of the array**

**Question No:131**

**(Marks:1)**

**Vu-Topper RM**

Generalized code written for a class is called

**Structure**

**Question No:132**

**(Marks:1)**

**Vu-Topper RM**

Which of the following statement is false?

**Pointers store the next data element of a list**

**Question No:133**

**(Marks:1)**

**Vu-Topper RM**

Want and de-allocating memory for linked list nodes does take more than pre-allocated array

**More**

**Question No:134**

**(Marks:1)**

**Vu-Topper RM**

Suppose you are writing a class for Node class and forgot to write the constructor of the class, then

**Compiler will automatically generate a default constructor**

**Question No:135**

**(Marks:1)**

**Vu-Topper RM**

Which of the following is the correct option for priority Queue?

**The type of queues that is FIFO i.e. the person who comes first should leave first.**

**Question No:136**

**(Marks:1)**

**Vu-Topper RM**

add(12) method of linked list class will:

**Add 12 as value in linked list**

بري صحبت سے تنہائی بہتر ہے اور تنہائی سے نيکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:137**

**(Marks:1)**

**Vu-Topper RM**

In singly linked list “next” field of node contains:

**Address of next node**

**Question No:138**

**(Marks:1)**

**Vu-Topper RM**

Which of the following line of code is incorrect?

**The lifetime of a transient object can exceed that of the application which is accessing it.**

**Question No:139**

**(Marks:1)**

**Vu-Topper RM**

A list is the collection of items of the \_\_\_\_\_

**Same type**

**Question No:140**

**(Marks:1)**

**Vu-Topper RM**

back() method of List class is used to:

**Moves the “current” pointer to backward one element.**

**Question No:141**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ is the maximum height of the AVL tree.

**$1.44\log_2 n$**

**Question No:142**

**(Marks:1)**

**Vu-Topper RM**

length() method of List class is used to:

**Return the length of the list**

**Question No:143**

**(Marks:1)**

**Vu-Topper RM**

Allocating and de-allocating memory for linked list nodes does take \_\_\_\_\_ time than pre-allocated array.

**More**

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**

**Question No:144**

**(Marks:1)**

**Vu-Topper RM**

In which case of insertion we require double rotation to make the AVL tree balance.

**None**

**Question No:145**

**(Marks:1)**

**Vu-Topper RM**

In array list the worst case of removing an element is

**To remove the item from start**

**Question No:146**

**(Marks:1)**

**Vu-Topper RM**

Whenever we call a function, the compiler makes a stack, the top element of the stack is \_\_\_\_\_ of the function.

**Return Address**

**Question No:147**

**(Marks:1)**

**Vu-Topper RM**

Whenever we call a function, the compiler makes a \_\_\_\_\_ that it uses to fulfill this function call.

**Stack**

**Question No:148**

**(Marks:1)**

**Vu-Topper RM**

\_\_\_\_\_ rule applies for evaluating operators of same precedence in an expression

**None**

**Question No:149**

**(Marks:1)**

**Vu-Topper RM**

The postfix form of the expression  $A + B * C$  and  $(A + B) * C$  will be \_\_\_\_\_.

**Same**

**Question No:150**

**(Marks:1)**

**Vu-Topper RM**

If we use array to implement list, then there is an issue that it gives difficulty when:

**We will access value randomly**

بری صحبت سے تنہائی بہتر ہے اور تنہائی سے نیکی صحبت بہتر ہے

**For More Help Contact What's app 03224021365**



**Question No:151**

**(Marks:1)**

**Vu-Topper RM**

During in-order traversal using recursive calls, if we found a node is NULL. It means this node will satisfy following condition.

**It will not have left child**

**Question No:152**

**(Marks:1)**

**Vu-Topper RM**

Which one the following is more closer to AVL tree, (chose the best option)

**Binary Search Tree**

**Question No:153**

**(Marks:1)**

**Vu-Topper RM**

In internal memory organization of a process, there is some area of memory for static data that holds \_\_\_\_\_ variables.

**Both Static and Global**

**Question No:154**

**(Marks:1)**

**Vu-Topper RM**

Which one is not the property of binary tree?

**Sibling node should be same parent**

**Question No:155**

**(Marks:1)**

**Vu-Topper RM**

What's wrong with following loop? while( (i < 10) && (i > 24)) { }

**The Condition is always false**

**Visit My YouTube Channel  
For More Important Notes  
Channel Name = #VuTopperRM**

بري صحبت سے تنهائي بهتر ہے اور تنهائي سے نيكي صحبت بهتر ہے

**For More Help Contact What's app 03224021365**



**DOWNLOAD MIDTERM**

**PAST PAPERS BY WAQAR SIDDHU**

**More in PDF From**

**VU Answer**

**Get All Solutions.**



An array x of 100 integers is declared as ,

Answer ( Please select your correct option )



int x = [100] ;



int x[100] ;

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com



integer array x = 100 ;



integer x [] = 100 ;



What is the maximum depth of recursive calls a function may make?

Answer ( Please select your correct option )



1



2



n (where n is the argument)



There is no fixed maximum



A Linear Data Structure is the data structure in which data elements are arranged in a sequence or a linear list. Which of the following is Non Linear Data Structure?

Answer ( Please select your correct option )

☐ Arrays

☐ Linked Lists

☒ Binary Search Trees

☐ Stack

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



The new operation in C++ for dynamically allocating memory returns,

Answer ( Please select your correct option )

☐

Size of the memory it has allocated

☒

Pointer to the memory it has allocated

☐

Both size and pointer to the memory it has allocated

☐

Value of the memory it has allocated



In \_\_\_\_\_, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node.

Answer ( Please select your correct option )



Linked list

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com



Doubly-linked list



Array



Structure

**Made By: Waqar Siddhu**



The expression  $AB+C^*$  is called ?

Answer ( Please select your correct option )



Prefix expression



Postfix expression



Infix expression



Prefix and Infix expression



Consider the function X as under

```
int X (int& Value)
```

```
{
 return Value;
}
```

Now **a** and **b** are integers in a calling function. Which one of the following is a valid call to the above function X ?

Answer ( Please select your correct option )

☐ a = X (b) ;

☐ a = X (&b) ;

☒ a = X (\*b) ;

☐ a = X (&\*b) ;

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**





Each node in Binary Search Tree has

Answer ( Please select your correct option )

☐ 1 pointer

☒ 2 pointers

☐ 3 pointers

☐ 4 pointers



Suppose  $n$  is the number of nodes in a complete Binary Tree, then maximum steps required for a search operation are

Answer ( Please select your correct option )



$\log_2 (n+1) - 1$

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com



$\log_2 (n+1)$



$\log_2 (n) - 1$



$\log_2 (n)$

**Made By: Waqar Siddhu**



The **next** field in the last node of a singly-linked list is set to\_\_\_\_\_.

Answer ( Please select your correct option )

☐ NAN

☐ 1

☒ NULL

☐ -1

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



Consider the following function:

```
void test_a(int n)
{
 cout << n << " ";
 if (n>0)
```

Answer ( Please select your correct option )

4 2 0

☐

0 2 4

☐

4 2

☐

2 4

☐



A queue where the dequeue operation does not depend upon FIFO, is called :

Answer ( Please select your correct option )



enqueue



simple queue



stack



priority queue

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



$a * (b+c) - d$  is an example of ----- expression.

Answer ( Please select your correct option )

☒ infix

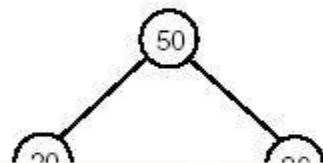
☐ prefix

☐ postfix

☐ allfix



Below is a Binary Search Tree (BST). If we delete the value 50 from the root node, what would be the value in the root of the remaining tree?



Answer ( Please select your correct option )

☐ 50

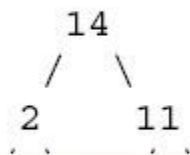
☐ 60

☐ 70

☐ 80



Consider the following tree:



Answer ( Please select your correct option )

2



4



6



9







Which one of the following calling methods does not change the original value of the argument in the calling function?

Answer ( Please select your correct option )



Call by passing the value of the argument

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com



Call by passing reference of the argument



Call by passing the address of the argument



Call by passing pointer of the argument

**Made By: Waqar Siddhu**



Which of the following statement is NOT true for reference variable ?

Answer ( Please select your correct option )



Once a reference is created, it cannot be later made to reference another object.



References cannot be NULL.



References can be uninitialized.



It is not possible to refer directly to a reference object after it is defined.



Searching an element in an AVL tree takes maximum \_\_\_\_\_ time (where n is number of nodes in AVL tree)

Answer ( Please select your correct option )



$\log_2(n+1)$



$\log_2(n+1) - 1$



$1.44 \log_2 n$



$1.66 \log_2 n$

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



In the post-order traversal of a binary search tree, nodes process as:

Answer ( Please select your correct option )



Left-subtree , Right-subtree , Root



Right-subtree , Root , Left-subtree



Left-subtree , Root , Right-subtree



Right-subtree , Left-subtree , Root



The main use of AVL tree is:

Answer ( Please select your correct option )



Searching of data

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com



Storing of data



Insertion of data



Security of data

**Made By: Waqar Siddhu**



Each operator in a postfix expression refers to the previous \_\_\_\_\_ operand(s).

Answer ( Please select your correct option )



One

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com



Two



Three



Four

**Made By: Waqar Siddhu**



Which one of the following statements is correct?

Answer ( Please select your correct option )



Array size is fixed once it is created.



Link List size is fixed once it is created.



Binary Search Tree size is fixed once it is created



AVL Tree size is fixed once it is created



Linked list always contains elements that can be described as ,



Answer ( Please select your correct option )



Redundant



Recursive



Self-referential.



Bidirectional.

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**





One difference between a queue and a stack is:



Answer ( Please select your correct option )

☐ Queues require dynamic memory, but stacks do not.

☐ Stacks require dynamic memory, but queues do not.

☒ Queues use two ends of the structure; stacks use only one.

☐ Stacks use two ends of the structure, queues use only one.

Stack and Queue can be implemented using \_\_\_\_\_.

Answer ( Please select your correct option )

☒ Singly Link List

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

☐ Binary Tree

☐ Binary Search Tree

☐ AVL Tree

**Made By: Waqar Siddhu**



New items are added at the \_\_\_\_\_ of the stack.

Answer ( Please select your correct option )

Bottom

☐

Middle

☐

Top



Center

☐

**Made By: Waqar Siddhu**



\_\_\_\_\_ is the stack characteristic but \_\_\_\_\_ was implemented because of the size limitation of the array.

Answer ( Please select your correct option )

☐ isFull(),isEmpty()

☐ pop(), push()

☒ isEmpty() , isFull()

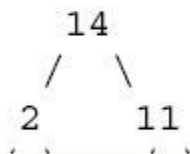
☐ push(),pop()

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



Consider the following tree.



Answer ( Please select your correct option )

☐ 2

☐ 3

☐ 4

☐ 5



The difference between a "Binary Tree (BT)" and a "Binary Search Tree(BST)" is that ,

Answer ( Please select your correct option )



A BST has two children per node whereas a BT can have none, one, or two children per node



In BST, nodes are inserted based on the values they contain

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com



In BT, nodes are inserted based on the values they contain



There is no difference



Which of the following operations returns "most recently entered value" from the stack ?

Answer ( Please select your correct option )



Push



Recent



Top



First

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



Which of the following is correct about AVL Tree?



Answer ( Please select your correct option )



It is identical to BST except height of the left and right subtrees can differ by at least 1.



It is identical to BST except height of the left and right subtrees must differ by at least 1.



It is not identical to BST, its totally different kind of tree.



It is identical to BST except height of the left and right subtrees can differ by at most 1.





What will be postfix expression of the following infix expression?

Infix Expression :  $a+b*c-d$

Answer ( Please select your correct option )



$ab+c*d-$



$abc*+d-$



$abc+*d-$



$abcd+*-$

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



In the call by ..... methodology, a copy of the object is passed to the called function.

Answer ( Please select your correct option )



Reference



Value



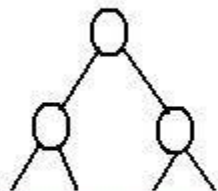
Reference & Value



Copy of the object can not be passed



Consider the following tree, how many levels does it has?



Answer ( Please select your correct option )

One

☐

Two

☐

Three

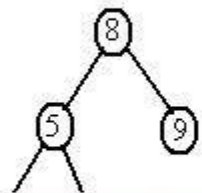
☐

Seven

☐



Consider the following tree, which statement is correct about this tree.



Answer ( Please select your correct option )

☐ 3 and 5 are on the same level

☐ 7 is the parent of 3

☐ 9 is the root of this tree

☐ 5 is on the second level



Recursive call of a function use..... data structure.

Answer ( Please select your correct option )

☐

Linked List

☐

Queue

☒

Stack

☐

Table

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



In the statement `int& a= b;`

Answer ( Please select your correct option )

- ☐ a and b pointing to two different memory location
- ☐ a and b are two different names of the same memory location
- ☐ a and b are two different variable names
- ☐ b hold the address of variable a



The variables which are destroyed automatically when a function's execution ends are:

Answer ( Please select your correct option )

☐ Global variables

☒ Local variables defined inside function body

☐ Variables (objects) defined inside function body dynamically

☐ Variables (objects) defined inside function body statically

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com



The worst case of searching in binary search tree (BST) is:

Answer ( Please select your correct option )



When the data inserted in BST is sorted



When the height of left sub-tree is greater than right sub-tree



When the height of right sub-tree is greater than left sub-tree



When the tree is balanced





In simple implementation of stack, isFull() method is used due to .....



Answer ( Please select your correct option )



Limitation of array

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com



Strength of array



Linked list connectivity



Complexity of linked list

**Made By: Waqar Siddhu**

Each node in singly link list has,

Answer ( Please select your correct option )

☒ 1 pointer

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com

☐ 2 pointers

☐ 3 pointers

☐ 4 pointers

**Made By: Waqar Siddhu**



Parameters in function call are passed using,

Answer ( Please select your correct option )

☒ Stack

☐ Queue

☐ Binary Search Tree

☐ AVL Tree



The \_\_\_\_\_ method of list data structure removes the element residing at the current position.

Answer ( Please select your correct option )

Add



Next



Remove



Find



Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



Insertion in a linked list can be done at

Answer ( Please select your correct option )



Front only



Back only



Somewhere in middle only



Front, back and somewhere in the middle



An array is a group of ----- memory locations.

Answer ( Please select your correct option )



Scattered



Isolated



Random(non-consecutive)



Consecutive

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



Which of the following applications may use a stack?

Answer ( Please select your correct option )



Accessing shared resource



Parentheses balancing program



Buffering messages



Waiting list



Every AVL is \_\_\_\_\_

Answer ( Please select your correct option )



Ternary Tree



Complete Binary Tree



Heap



Binary Search Tree

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**





Consider the following infix expression:

$x - y * a + b / c$

Which of the following is a correct equivalent expression for the above?

Answer ( Please select your correct option )

☐

$x y - a * b + c /$

☐

$x * y a - b c / +$

☒

$x y a * - b c / +$

☐

$x y a * - b / + c$



Non-recursive calls are faster than which of the following calls?

Answer ( Please select your correct option )

☐ Parameterized

☒ Recursive

☐ Function

☐ Non-Function

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



The balance of a node is the result of "height of left subtree" ..... "height of right subtree".

Answer ( Please select your correct option )



Plus



Minus



Multiply



Divided by



Searching an element in an AVL tree takes maximum \_\_\_\_\_ time (where n is number of nodes in AVL tree)

Answer ( Please select your correct option )



$\log_2(n+1)$



$\log_2(n+1) - 1$



$1.44 \log_2 n$



$1.66 \log_2 n$

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



A complete binary tree having "N" nodes consists of ..... Levels.

Answer ( Please select your correct option )

☒  $\log_2(N+1) - 1$

☐  $\log_2(N-1) - 1$

☐  $\log_2(N+1) + 1$

☐  $\log_2(N-1) + 1$



In the post-order traversal of a binary search tree, nodes process as:

Answer ( Please select your correct option )



Left-subtree , Right-subtree , Root

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com



Right-subtree , Root , Left-subtree



Left-subtree , Root , Right-subtree



Right-subtree , Left-subtree , Root

**Made By: Waqar Siddhu**



The simplest case in a BST to delete a node is:

Answer ( Please select your correct option )



When the node, that is to be deleted is root node



When the node, that is to be deleted has both left and right child



When the node, that is to be deleted has only one child



When the node, that is to be deleted is a leaf node



If class A defines class B as its friend, then:

Answer ( Please select your correct option )



Class A can access private members of class B



Class B can access only the public members of class A



Class A can access only the public members of class B



Class B can access private members of class A





In the statement `int& a= b;`

Answer ( Please select your correct option )

- ☐ a and b pointing to two different memory location
- ☐ a and b are two different names of the same memory location
- ☐ a and b are two different variable names
- ☐ b hold the address of variable a

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



The main use of AVL tree is:

Answer ( Please select your correct option )



Searching of data



Storing of data



Insertion of data



Security of data



In simple implementation of stack, isFull() method is used due to .....

Answer ( Please select your correct option )



Limitation of array

Correct answer solved by hadi

Cell No:03228043306

Email: usmanraj20@gmail.com



Strength of array



Linked list connectivity



Complexity of linked list

**Made By: Waqar Siddhu**



Write in postfix form:  $5 * (9 - 7)$

Answer ( Please select your correct option )

☐  $5 (9 7 -) *$

☐  $5 9 * 7 -$

☒  $5 9 7 - *$

☐  $5 (9 7 -)$

Correct answer solved by hadi  
Cell No:03228043306  
Email: usmanraj20@gmail.com

**Made By: Waqar Siddhu**



What will be result of following postfix expression ?

1 2 3 \* + 2 -

Answer ( Please select your correct option )

3



4



(3 + (2 \* 9)) - 6 becomes 3 2 9 \* + 6 -

5



10



**DOWNLOAD MIDTERM**

**PAST PAPERS BY WAQAR SIDDHU**

**More in PDF From**

**VU Answer**

**Get All Solutions.**

How can the dangling reference problem be avoided ?

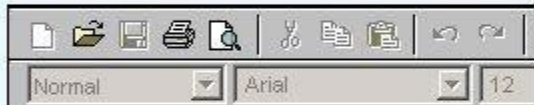
Answer ( [Please click here to Add Answer](#) )



To avoid dangling reference, don't return the reference of a local variable (transient) from a function.

What is "level" in binary tree ?

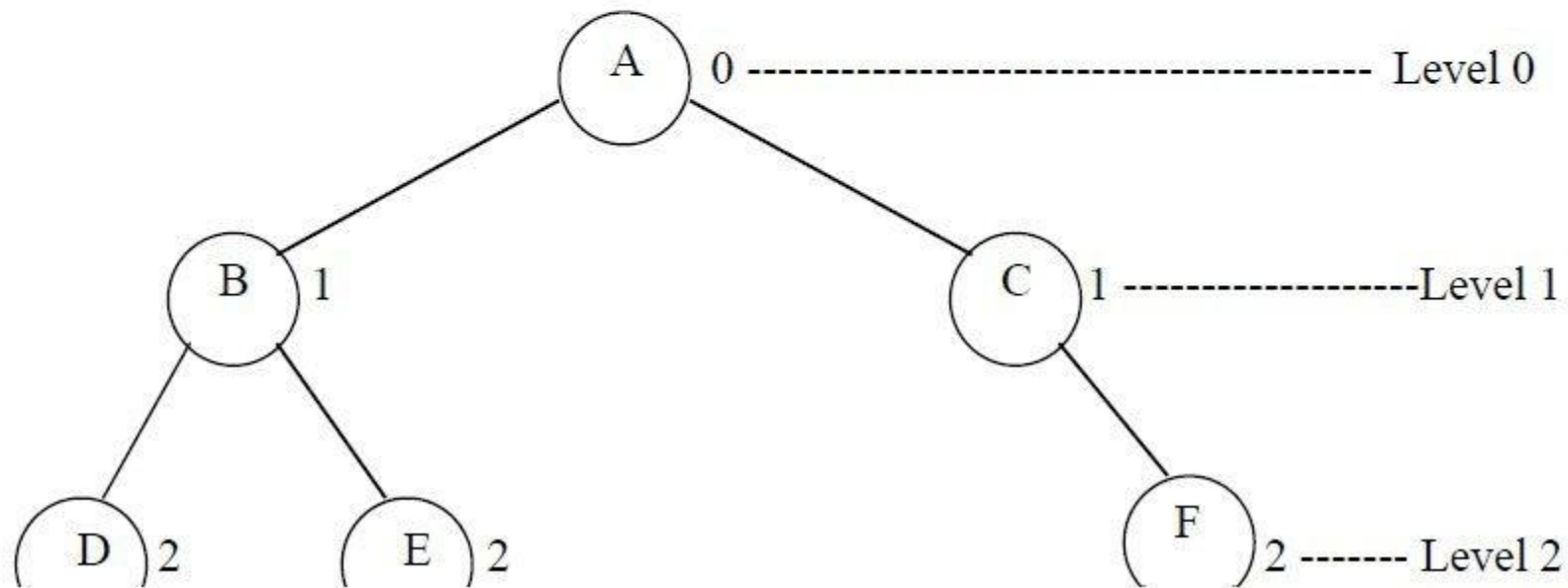
Answer ( [Please click here to Add Answer](#) )



The level of a node in a binary tree is defined as follows:

- Root has level 0,
- Level of any other node is one more than the level its parent (father).
- The *depth* of a binary tree is the maximum level of any leaf in the tree.

To understand level of a node, consider the following figure 11.13. This figure shows the same tree of figure 11.2, discussed so far.

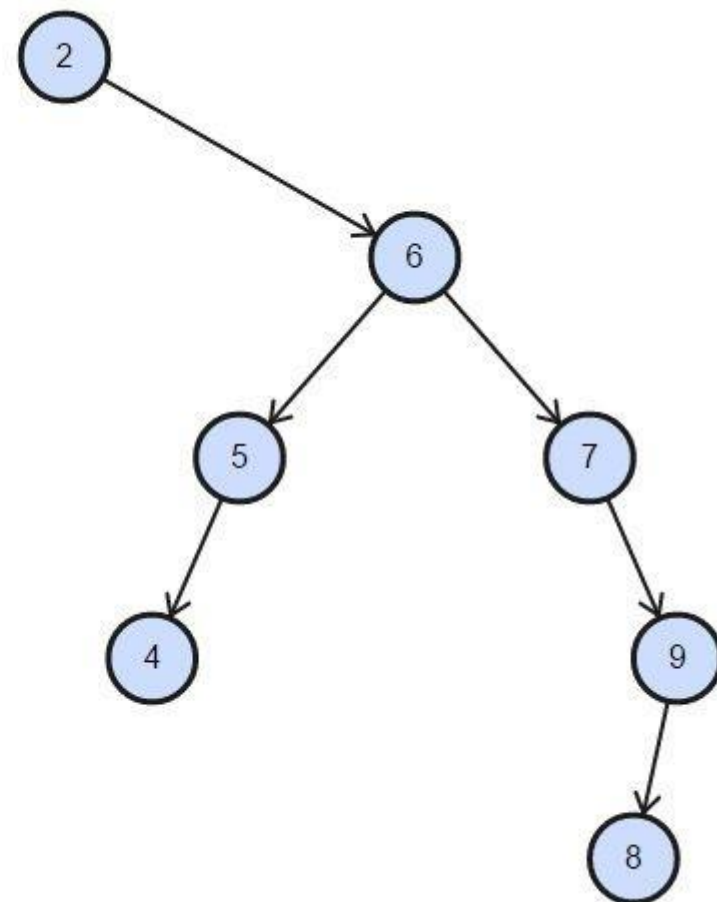




Draw a BST that is as tall as possible and contains the values '2', '6', '5', '4', '7', '9', '8'.

Answer ( [Please click here to Add Answer](#) )

Rich text editor toolbar showing icons for text formatting (bold, italic, underline), alignment, and a zoom level of 100%.



Define the following terms

- 1) Reference variables
- 2) Dangling reference
- 3) Const keyword

Answer ( [Please click here to Add Answer](#) )



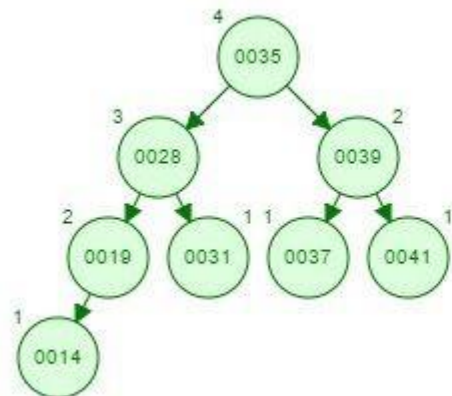
Ref Vari: A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

Dangling: A dangling reference is a reference to an object that no longer exists.

Const: By writing *const*, we are saying that parameter must remain constant for the life of the function.

Show the result of inserting the values 35, 19, 41, 39, 37, 28, 31, 14 into an empty AVL Tree.  
You have to show only the complete AVL tree, steps are not required.

Answer ( [Please click here to Add Answer](#) )





Write down the algorithm (Steps) for level order traversing of a binary tree.

The name of the method is *levelorder* and it is accepting a pointer of type *TreeNode* `<int>`. This method will start traversing tree from the node being pointed to by this pointer. The first line ( line 3) in this method is creating a queue *q* by using the *Queue* class factory interface. This queue will be containing the *TreeNode*`<int>` \* type of objects. Which means the queue will be containing nodes of the tree and within each node the element is of type *int*.

The line 5 is checking to see, if the *treeNode* pointer passed to the function is pointing to NULL. In case it is NULL, there is no node to traverse and the method will return immediately.

Otherwise at line 6, the very first node (the node pointed to by the *treeNode* pointer) is added in the queue *q*.

Next is the *while* loop (at line 7), which runs until the queue *q* does not become empty. As we have recently added one element (at line 6), so this loop is entered.

At line 9, *dequeue()* method is called to remove one node from the queue, the element at *front* is taken out. The return value will be a pointer to *TreeNode*. In the next line (line 10), the *int* value inside this node is taken out and printed. At line 11, we check to see if the left subtree of the tree node (we've taken out in at line 9) is present. In case the left subtree exists, it is inserted into the queue in the next statement at line 12. Next, we see if the right subtree of the node is there, it is inserted into the queue also. Next statement at line 15 closes the *while* loop. The control goes back to the line 7, where it checks to see if there is some element left in the queue. If it is not empty, the loop is entered again until it becomes empty.

**y: Waqar Siddhu**

In which of the traversal methods, recursion can not be applied?

Answer ( [Please click here to Add Answer](#) )



by using an explicit stack.

We cannot avoid stack. The stack will be used to store the tree nodes in the appropriate order.

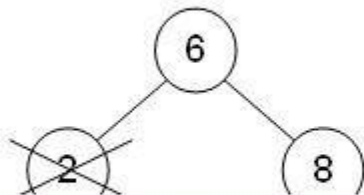
How can we calculate the height of tree ?

Answer ( [Please click here to Add Answer](#) )



It is the longest path from the node to a leaf. So height is the number of edges of the path

What will be the resultant tree after deleting the node 2 from the tree given below.

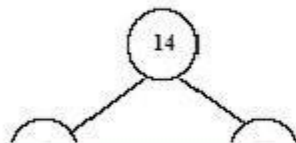


Answer ( [Please click here to Add Answer](#) )





Consider the following AVL tree. Insert a new node with key of **12**. No need to show all steps, just draw the final AVL tree .



Answer ( [Please click here to Add Answer](#) )

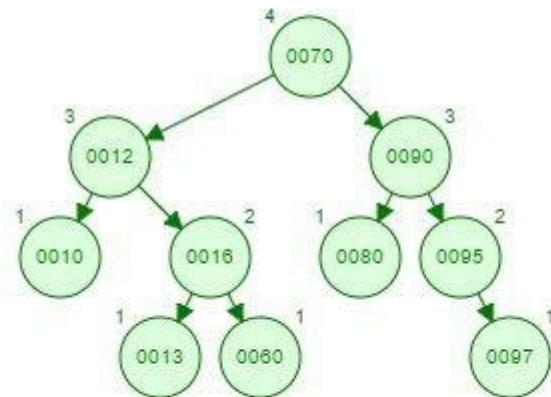


See AVL tree  
youtube



Show the result of inserting the values 10, 80, 12, 70, 90, 60, 13, 16, 95, 97 into an empty AVL Tree. You have to show only the complete AVL tree, steps are not required.

Answer ( Please )



Explain the priority queue with the help of real world examples?

Answer ( [Please click here to Add Answer](#) )



the queue is a FIFO (First in first out) structure. In daily life, you have also seen that it is not true that a person, who comes first, leaves first from the queue. Let's take the example of traffic. Traffic is stopped at the signal. The vehicles are in a queue. When the signal turns green, vehicles starts moving. The vehicles which are at the front of the queue will cross the crossing first. Suppose an ambulance comes from behind. Here ambulance should be given priority. It will bypass the queue and cross the intersection. Sometimes, we have queues that are not FIFO i.e. the person who comes first may not leave first. We can develop such queues in which the condition for leaving the queue is not to enter first. There may be some priority. Here we will also see the events of future like the customer is coming at what time and leaving at what time. We will arrange all these events and insert them in a priority queue. We will develop the queue in such a way that we will get the event which is going to happen first of all in the future. This data structure is known as priority queue. In a sense, FIFO is a special case of priority queue in which priority is given to the time of arrival. That means the person who comes first has the higher priority while the one who comes later, has the low priority. You will see the priority queue being used at many places especially in the operating systems. In operating systems, we have queue of different processes. If some process comes with higher priority will be processed first

**Made By: Waqar Siddhu**

How can we calculate the height of tree ?

Answer ( [Please click here to Add Answer](#) )



It is the longest path from the node to a leaf. So height is the number of edges of the path

What is meant by traversing of binary tree?

# traverse

/ˈtrævəs, trəˈvɜːs/ 

verb

gerund or present participle: **traversing**

1. travel across or through.  
"he traversed the forest"  
*synonyms:* travel over/across, cross, journey over/across, across, negotiate; [More](#)
2. move back and forth or sideways.  
"a probe is traversed along the tunnel"

00% 



"A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree."



Consider the following code snippet,

```
class BST_node
{
 private:
 char ch;
```

Answer ( [Please click here to Add Answer](#) )



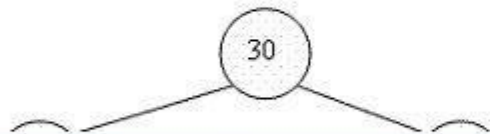
Explain the logic of the following function.

```
int& Fun(int& y)
{
 y = y + 100;
 return y;
```

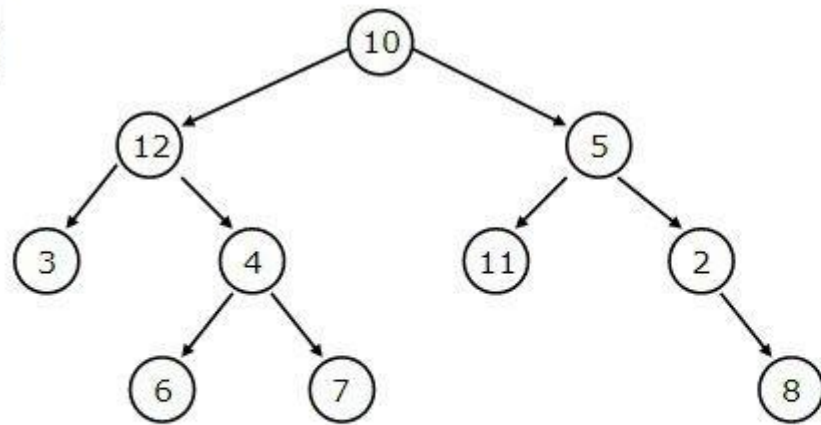
Answer ( [Please click here to Add Answer](#) )



Perform the Preorder and Inorder traversal on the tree given below and show the result in both cases (you have to perform the traversal manually not using any programming language)



Answer ( [Please click here to Add Answer](#) )



**Levelorder tree traversal**

10, 12, 5, 3, 4, 11, 2, 6, 7, 8

**Inorder tree traversal**

3, 12, 6, 4, 7, 10, 11, 5, 2, 8

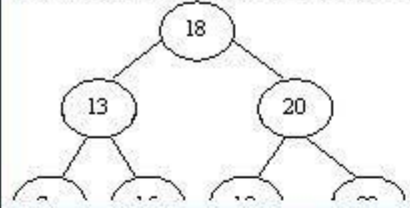
**Preorder tree traversal**

10, 12, 3, 4, 6, 7, 5, 11, 2, 8

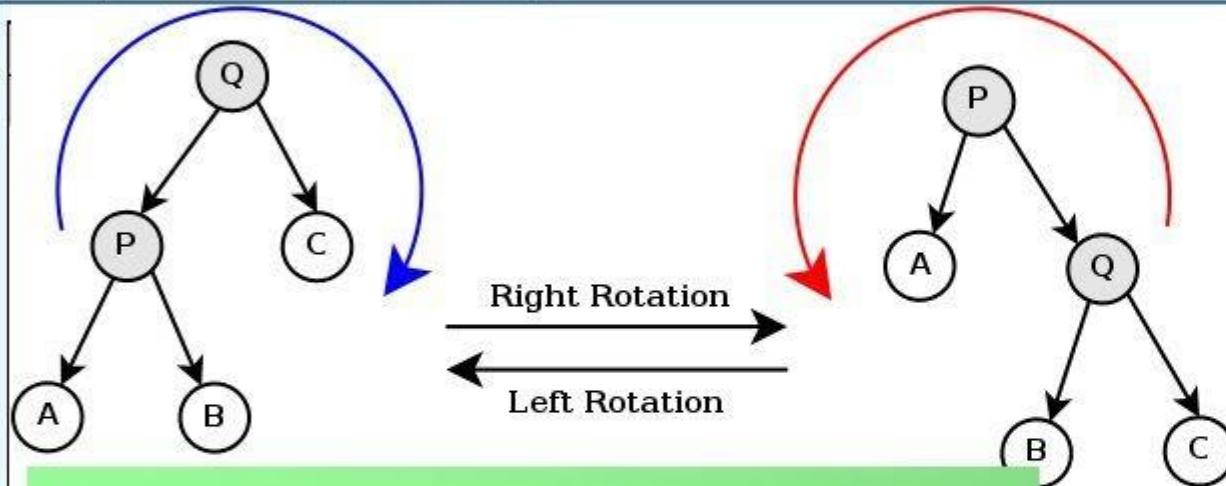
**Postorder tree traversal**

3, 6, 7, 4, 12, 11, 8, 2, 5, 10

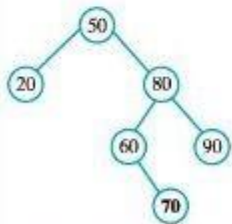
Apply double rotation on the following tree to restore its balance. Show the necessary steps to perform this rotation.



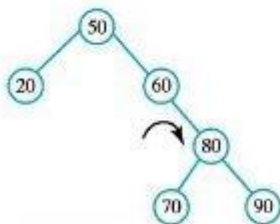
Answer ( [Please click here to Add Answer](#) )



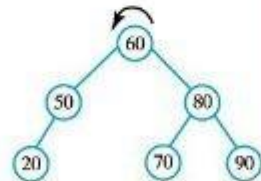
(a) After adding 70



(b) After right rotation



(c) After left rotation



(a) Adding 70 to the previous tree destroys its balance; to restore the balance, perform both (b) a right rotation and (c) a left rotation.

**Made By: Waqar Siddhu**



```
char ch;
BST_node *BST_left,
BST_node *BST_right;
};
BST_node *bst, bt;
```

Answer ( [Please click here to Add Answer](#) )



Correct the following statements,

```
bst.BST_left = NULL;
bst.BST_rifht = NULL;
bt->ch = 'A';
```

Answer ( [Please click here to Add Answer](#) )

