Data Structures

Lecture No. ~~36~~33

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 6, 8
6.3, 8.1

Summary

- Priority Queue Using Heap
- The Selection Problem
- Heap Sort
- Disjoint Set ADT
- Equivalence Relations

Priority Queue Using Heap

~~In~~As discussed in the previous lecture, we ~~discussed a lot about *buildHeap*.~~generally prefer to employ the buildHeap to construct ~~If we have all the data available then~~heap instead of using insert ~~we can use *buildHeap* to construct heap. *buildHeap* is optimized as compared to *insert* method because it takes lesserthan $Nlog_2N$ time. We also proved a theorem that if the number of linkes in the tree are counted then the sum of heights of the nodes is *N-h-1*. We have been iteratinglot of times that the best use of heap is implementing priority queues. Lets try to develop a class of priority queue using heap. You might be remembering, when we did bank simulation, weused a priority queue but that was implementedusing an array. Now, we will implement the same queue using a heap. We will modify the same code to use heap instead of any array. The interface (.h file) will remain the same but the implementation (.cpp file) will be changed. As .cpp file will change, so let's see the .cpp code below:~~

~~The first line is including a file *Event.cpp*, this is containing all the events used for simulation. We are including .cpp files here as we have used templates of C++. At the second line, we have included *heap.cpp*. In the third line, we are defining a constant PQMAX, declaring the maximum size of the priority queue to be 30. In line 4, class *PriorityQueue* is declared. *public* keyword is given at line 6 indicating that all class members below will be of public scope. Line 7 is starting the class constructor's definition. At line 9, a new heap object is being created, this object is a collection of *Event* type objects and the number of elements in the collection is *PQMAX*. The address (pointer) of the newly created heap is stored in the *heap* object pointer. The *heap* is a~~

~~private pointer variable of *PriorityQueue* class. Next comes the destructor of the class, where we are deleting (deallocating the allocated resources) the pointer variable *heap*. Next isthe *remove* method that was sometimes, named as *dequeue* in our previous lectures. *remove* method is returning an *Event* pointer. Inside *remove*, the first statement at line 19 is an if-condition; which is checking whether *heap* is not empty. If the condition is true (i.e. the *heap* is not empty), a local variable (local to the block) *Event** variable *e* is declared. In the next line at line 22, we are calling deleteMin method to delete (remove) an element from the *heap*, the removed element is assigned to the passed in parameter pointer variable *e*. You might have noticed already that in this version of *deleteMin*, the parameter is being passed by pointer, in the previous implementation we used reference. In the next line, the retrieved value is returned, the function returns by returning the pointer variable *e*. But if the heap was empty when checked at line 19, the control is transferred to the line 25. At line 25, a message is displayed to show that the heap is empty and the next line returns a NULL pointer.~~
~~It is important to understand one thing that previously when we wrote the array based *remove* method. We used to take an element out from the start of the array and shifted the remaining elements to left. As in this implementation, we are using heap, therefore, all the responsibilities of maintaining the heap elements lie with the heap. When the deleteMin() is called, it returns the minimum number.~~

????

~~We see the *insert* method now line by line. It is accepting a pointer type parameter of type *Event*. As the heap may become full if we keep on inserting elements into it, therefore, the first line inside the insert function is checking if the heap has gone full. If the heap is not full then the if-block is entered. At line 33 inside the if-block, an element that is passed as a parameter to *insert* method of heap is inserted into the queue by calling the *insert* method of heap. This insert call will internally perform percolate up and down operations to place the new element at its correct position. The returned value 1 indicates the successful insertion in the queue. If the heap has gone full then a message is displayed that '*insert queue is full*'. Note that we have used the word queue not heap in that message. It has to that way otherwise the users of the PriorityQueue class are unaware of the internal representation of the queue (whether it is implemented using a heap or an array). Next line is returning 0indicating that the *insert* operation was not successful. Below to *insert*, we have *full()* method. This method returns an *int* value. Internally, it is calling is*Full()* method of heap. The *full()* method returns whatever is returned by the *isFull()* method of heap. Next is *length()* method, as the size of heap is also the size of the queue, therefore, *length()* is internally calling the *getSize()* method of heap.~~

~~In this new implementation the code is better readable than the PriorityQueue's implementation with array. You must be remembering when we implemented the PriorityQueue with an array, at each insertion in the~~ if we have all the required data. *buildHeap* is optimized as compared to *insert* method as it takes lesser time than $Nlog_2N$. In the previous discussion, we had proved a theorem that if the number of links in the tree are counted, the sum of all is *N-h-1*. We have been iterating a lot of times that the best use of heap is priority queue's implementation. Let's try to develop a class of priority

queue with the help of a heap. You are very familiar with the concept of bank simulation. While explaining the bank simulation, we had used a priority queue that was implemented by using an array. Now, we will implement the same queue with the help of a heap. For this purpose, the code will be modified so that heap is used in place of array. The interface (.h file) will remain the same. However, the implementation (.cpp file) will be changed. Let's see the following cpp code, which also shows the change in the .cpp file:

```
1.      #include  "Event.cpp"
2.      #include  "Heap.cpp"
3.      #define  PQMAX  30
4.      class  PriorityQueue
5.      {
6.              public:
7.                      PriorityQueue()
8.                      {
9.                              heap  =  new  Heap <Event> ( PQMAX );
10.                     };
11.
12.                     ~PriorityQueue()
13.                     {
14.                             delete  heap;
15.                     };
16.
17.                     Event *  remove()
18.                     {
19.                             if( !heap->isEmpty() )
20.                             {
21.                                     Event *  e;
22.                                     heap->deleteMin( e );
23.                                     return  e;
24.                             }
25.                             cout  <<  "remove - queue is empty."  <<  endl;
26.                             return  (Event *)  NULL;
27.                     };
28.
29.                     int  insert(Event *  e)
30.                     {
31.                             if( !heap->isFull() )
1.                              {
33.                                     heap->insert( e );
34.                                     return  1;
35.                             }
36.                             cout  <<  "insert queue is full."  <<  endl;
37.                             return  0;
38.                     };
```

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

```
39.
40.        int full(void)
41.        {
42.            return heap->isFull();
43.        };
44.
45.        int length()
46.        {
47.            return heap->getSize();
48.        };
49.    };
```

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

The first line has a file *Event.cpp that contains* all the events used for simulation. We are including .cpp files here as done in case of templates of C++. In the second line, there is heap.cpp *while* a constant PQMAX has been defined in third line, declaring the maximum size of the priority queue to be 30. In line 4, class *PriorityQueue* is declared. *public* keyword is given at line 6 indicating that all class members below will be of public scope. Line 7 starts the class constructor's definition while in the line 9; a new heap object is being created. This object is a collection of *Event* type objects and the number of elements in the collection is *PQMAX*. The address (pointer) of the newly created heap is stored in the *heap* object pointer. The *heap* is a private pointer variable of *PriorityQueue* class. Now there is the destructor of the class, where we are deleting (deallocating the
The first line is including a file *Event.cpp*, this is containing all the events used for simulation. At the second line, we have included *heap.cpp*. In the third line, we are defining a constant PQMAX, declaring the maximum size of the priority queue to be 30. In line 4, class *PriorityQueue* is declared. *public* keyword is given at line 6 indicating that all class members below will be of public scope. Line 7 is starting the class constructor's definition. At line 9, a new heap object is being created, this object is a collection of *Event* type objects and the number of elements in the collection is *PQMAX*. The address (pointer) of the newly created heap is stored in the *heap* object pointer. The *heap* is a private pointer variable of *PriorityQueue* class. Next comes the destructor of the class, where we are deleting (deallocating the allocated resources) the pointer variable *heap*. Next is the *remove* method that was sometimes, named as *dequeue* in our previous lectures. *remove* method is returning an *Event* pointer. Inside *remove*, the first statement at line 19 is an if-condition; which is checking whether *heap* is not empty. If the condition is true (i.e. the *heap* is not empty), a local variable (local to the block) *Event** variable *e* is declared. In the next line at line 22, we are calling deleteMin method to delete (remove) an element from the *heap*, the removed element is assigned to the passed in parameter pointer variable *e*. You might have noticed already that in this version of *deleteMin*, the parameter is being passed by pointer, in the previous implementation we used reference. In the next line, the retrieved value is returned, the function returns by returning the pointer variable *e*. But if the heap was empty when checked at line 19, the control is transferred to the line 25. At line 25, a message is displayed to show that the heap is empty and the next line returns a NULL pointer.

It is important to understand one thing that previously when we wrote the array based *remove* method. We used to take an element out from the start of the array and shifted the remaining elements to left. As in this implementation, we are using heap, therefore, all the responsibilities of maintaining the heap elements lie with the heap. When the deleteMin() is called, it returns the minimum number.

We now observe the *insert* method line by line. It is accepting a pointer type parameter of type *Event*. As the heap may become full, if we keep on inserting elements into it, so the first line inside the insert function is checking whether the heap has gone full. If the heap is not full, the if-block is entered. At line 33 inside the if-block, an element that is passed as a parameter to *insert* method of heap is inserted into the queue by calling the *insert* method of heap. This insert call will internally perform percolate up and down operations to place the new element at its correct position. The returned value 1 indicates the successful insertion in the queue. If the heap has gone full, a message is displayed i.e. '*insert queue is full*'. Note that we have used the word queue, not heap in that message. It needs to be done this way. Otherwise, the users of the PriorityQueue class are unaware of the internal representation of the queue (whether it is implemented using a heap or an array). Next line is returning 0 while indicating that the *insert* operation was not successful. Below to *insert*, we have *full()* method. This method returns an *int* value. Internally, it is calling is*Full()* method of heap. The *full()* method returns whatever is returned by the *isFull()* method of heap. Next is *length()* method as the size of heap is also that of the queue. Therefore, *length()* is internally calling the *getSize()* method of heap.

In this new implementation, the code is better readable than the PriorityQueue's implementation with array. While implementing the PriorityQueue with an array,~~array,~~ we had to sort the internal array every time~~. Clearly then this new implementation is more efficient because the heap can readjusts itself in $\log_2 N$ times. Gain in performance is the major benefit of implementing PriorityQueue using heap over implementing with array and very important for you to remember as a student of Data Structures.~~
~~There are other significant benefits of using heap and there are number of other uses of heap that would be covered in this course time to time. At the moment, we willsome common example usages of heap in order to make you clear about other uses of heap. The heap data structure will be covered in the Algorithms course also.~~

~~The Selection Problem~~

- ~~Given a list of *N* elements (numbers, names etc.), which can be totally ordered, and an integer k, find the *k*th smallest (or largest) element.~~

Formatted: Bullets and Numbering

~~Suppose we have list of N names (names of students or names of motor vehicles or a list of numbers of motor vehicles or list of roll numbers of students or id card numbers of the students, whatever). The problem is to findout the kth smallest element. So let's say we have a list of 1000 numbers and we want to find the 10th smallest number in it. The sorting is applied in order to make the elements ordered. After sorting out list of numbers~~

(in any order ascending or decending, let's do in ascending in this as we are finding the smallest number), it will be very easy to findany desired smallest number.

- One way is to put these *N* elements in an array and sort it. The kth smallest of these is at the kth position.

Formatted: Bullets and Numbering

It will take $N\log_2 N$ time in case we use array data structure. Now, we want to see if it is possible to reduce the time from $N\log_2 N$ by using some other data structure or by improving the algorithm? Yes, we can apply heap data structure in order to make this operation more efficient.

- A faster way is to put the *N* elements into an array and apply the *buildHeap* algorithm on this array.
- Finally, we perform *k deleteMin* operations. The last element extracted from the heap is our answer.

Formatted: Bullets and Numbering

The *buildHeap* algorithm is used to construct a heap of given N elements. If we constructed min-heap, the minimum of the N elements will be positioned in the root node of the heap. If we take out (*deleteMin*) k elements from the heap, we can get the kth smallest element. *BuildHeap* works in linear time to make a min or a max-heap.

- The interesting case is $k = \lceil N/2 \rceil$, since this is known as the *median*.

Formatted: Bullets and Numbering

In Statistics, we take the average of numbers, find the minimum, maximum and median as well. Median is defined as a number in the sequence where the half of the numbers are greater than this number and half of the numbers are smaller. Now, we can come up with our mechanism to find median from a given N numbers. Let's say , we want to compute the median of final marks of students of our class. Further suppose that the maximum aggregate marks for a student are 100. We use the *buildHeap* to construct a heap for N number of students. By calling *deleteMin* for N/2 times, the minimum marks of the half number students will be taken out. The $N/2^{th}$ marks would be the median of the marks of our class. The alternates methods are there to calculate median, however, we are discussing here about the possible uses of heap. Let's see another use of heap.

Heap Sort

As discussed above in The Selection Problem that to take the 100th minimum element out from the min-heap, we will call *deleteMin* to take out 1st element, 2nd element, 3rd element and eventually we will call *deleteMin* 100th time to take out our required 100th minimum element. Suppose, if the size of the heap is 100 elements then we have taken out all the elements out from the heap. Interestingly the elements are sorted in increasing order. If somehow, we can store these taken out numbers, let's say in an array we can have all the elements sorted (in ascending order in this min-heap case).
Hence,

- If $k = N$, and we record the *deleteMin* elements as they come off the heap, we will have essentially sorted the *N* elements.

Formatted: Bullets and Numbering

- Later in the course, we will refine this idea to obtain a fast sorting algorithm called *heapsort*.

We finish our discussion above heap here but it will be discussed in the forthcoming courses and a little bit when we will discuss about sort in the next lectures of this course. At the moment, let' see another Abstract Data Type.

Disjoint Set ADT

As always, before we actually move to an Abstract Data Type (ADT), we like to see what that ADT is, how it works and in what situations it can be helpful. We are going to cover Disjoint Set ADT. Firstly, we will have its introduction, examples and later on, we will discuss about its implementation.

- Suppose we have a database of people.

We want to figure out who is related to whom.Initially, we only have a list of people, and information about relations is gained by updates of the form "Haaris is related to Saad".Let's say we have a list of names of all people in a locality but we are not aware of their relationships to each other, if they are related. After having the list of all people we start getting some information about their relationships slowly, one relationship at a time, for example, "Ali Abbas isson of Abbas".

The situation becomes interesting when we have relationships like "Ali Abbas is first cousin of Ayesha Ali (i.e. fathers of both are brothers) but Ayesha Ali has other immediate cousins also from her mother's side, therefore, other immediate cousins of Ayesha Ali also get related to Ali Abbas, although they are not immediate to Ali Abbas."

So as we keep on getting relationships details of those people, these direct and indirect relationships can be establishded.

Key property: If Haaris is related to Saad and Saad is related to Ahmad, then Haaris is related to Ahmad.

See the key property line's first part above "Harris is related to Saad and Saad is related to Ahmad". Suppose we have a program to handle this list of people and their relationships. After we have provided all the name "Harris, Saad and Ahmad" to that program and then this part of information about their relationships, the program might be able to determine the remaining part "Harris related to Ahmad".

The same problem (the intelligence required in the program) is describe in the sentence below:

- Once we have relationships information, we would like to answer queries like "Is Haaris related to Ahmad?"

To answer this kind of queries and to have that intelligence in our programs *Disjoint Set ADT* is used. Before we see what *Disjoint Set ADT* is, we see another application of this ADT in image analysis. This problem is known as *Blob Coloring*.

Blob Coloring

Formatted: Bullets and Numbering

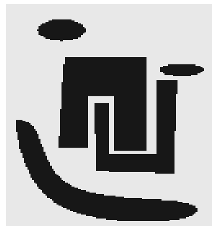
Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

A well known low level computer vision problem for black and white images is the following: Gather together all the picture elements (pixels) that belong to the same "blobs", and give each pixel in each different blob an identical label.
You must have heard of robots that perform certain tasks automatically. How do they know from the images provided to them that what in which direction should they move, they can catch things and carry them. They do different things like humans. Obviously, there is a software working internal to robots body that is doing all this controlling part and vision to the robot. This is very complex problem and broad area of *Computer Science* and *Electrical Engineering* called *Robotics* (*Computer Vision* in particular).

Consider the image below:



`` **Fig 33.1**

This image is black and white, which consist of five non-overlapping black colored regions of different shapes called blobs. These blobs are two elliptic, n and u shaped (two blobs) and one arc at the bottom. From human eye we can see these five blobs but how a robot can identify these. So the problem is:

We want to *partition* the pixels into *disjoint sets*, one set per blob.

Formatted: Bullets and Numbering

Now if we make one set per blob then for five blobs in the image above, we have to have five sets. To understand the concept of disjoint sets, we can take an analogy where we have two sets A and B (as in Mathematics) where none of the elements inside set A is present in set B. The sets A and B are called disjoint sets.

Another problem relatedthe Computer Vision is the *image segmentation problem*. See the image below on the left of an old ship in gray scales.

~~Fig 33.2~~

~~We want to find regions of different colors in this picture, for example all regions in the picture of color black, all regions in the image of color gray. The image on the right represents the resultant image.~~

~~Different scanning done in hospitals *mri scan*, *cat scan* or *ct scan* scans the inner parts of the whole body of humans. Those scans images in gray scales represent organs of the human body. All these are application of *Disjoint Set ADT*.~~

at each insertion in the array. This new implementation is more efficient as the heap can readjust itself in $\log_2 N$ times. Gain in performance is the major benefit of implementing PriorityQueue with heap as compared to implementation with array.
There are other significant benefits of the heap that will be covered in this course time to time. At the moment, we will have some common example usages of heap to make you clear about other uses of heap. The heap data structure will be covered in the Algorithms course also.

The Selection Problem

- Given a list of *N* elements (numbers, names etc.) which can be totally ordered and an integer k, find the kth smallest (or largest) element.

Suppose, we have list of N names (names of students or names of motor vehicles or a list of numbers of motor vehicles or list of roll numbers of students or id card numbers of the students, whatever). However, we are confronting the problem of finding out the kth smallest element. Suppose we have a list of 1000 numbers and want to find the 10th smallest number in it. The sorting is applied to make the elements ordered. After sorting out list of numbers, it will be very easy to find out any desired smallest number.

- One way is to put these *N* elements in an array and sort it. The kth smallest of these is at the kth position.

It will take $N\log_2 N$ time ,in case we use array data structure. Now, we want to see if it is possible to reduce the time from $N\log_2 N$ by using some other data structure or by improving the algorithm? Yes, we can apply heap data structure to make this operation more efficient.

- A faster way is to put the *N* elements into an array and apply the *buildHeap* algorithm on this array.
- Finally, we perform *k deleteMin* operations. The last element extracted from the heap is our answer.

The *buildHeap* algorithm is used to construct a heap of given N elements. If we construct

'min-heap, the minimum of the N elements, will be positioned in the root node of the heap. If we take out (*deleteMin*) k elements from the heap, we can get the Kth smallest element. *BuildHeap* works in linear time to make a min or a max-heap.

– The interesting case is $k = \lceil N/2 \rceil$ as it is also is known as the *median*.

In Statistics, we take the average of numbers to find the minimum, maximum and median. Median is defined as a number in the sequence where the half of the numbers are greater than this number while the remaining half are smaller ones. Now, we can come up with the mechanism to find median from a given N numbers. Suppose, we want to compute the median of final marks of students of our class while the maximum aggregate marks for a student are 100. We use the *buildHeap* to construct a heap for N number of students. By calling *deleteMin* for N/2 times, the minimum marks of the half number students will be taken out. The $N/2^{th}$ marks would be the median of the marks of our class. The alternate methods are there to calculate median. However, we are discussing the possible uses of heap. Let's see another use of heap.

Heap Sort

To take the 100th minimum element out from the min-heap, we will call *deleteMin* to take out 1st element, 2nd element, 3rd element. Eventually we will call *deleteMin* 100th time to take out our required 100th minimum element. Suppose, if the size of the heap is 100 elements, we have taken out all the elements out from the heap. Interestingly the elements are sorted in ascending order. If somehow, we can store these numbers, let's say in an array, all the elements sorted (in ascending order in this min-heap case) can be had.
Hence,

– If $k = N$, and we record the *deleteMin* elements as they come off the heap. We will have essentially sorted the *N* elements.

– Later in the course, we will fine-tune this idea to obtain a fast sorting algorithm called *heapsort*.

We conclude our discussion on the heap here. However, it will be discussed in the forthcoming courses. At the moment, let' see another Abstract Data Type.

Disjoint Set ADT

Before actually moving to an Abstract Data Type (ADT), we need to see what that ADT is, how it works and in which situations it can be helpful. We are going to cover Disjoint Set ADT. Firstly, we will have its introduction, examples and later the ways of its implementation.

– Suppose we have a database of people.
– We want to figure out who is related to whom.Initially, we only have a list of people, and information about relations is obtained by updating the form "Haaris is related to Saad".Let's say we have a list of names of all people in a locality but are not aware of their relationships to each other. After having the list of all people, we start getting some information about their relationships gradually. For example, "Ali Abbas is the son of

Formatted: Bullets and Numbering

Abbas".
The situation becomes interesting when we have relationships like "Ali Abbas is first cousin of Ayesha Ali (i.e. fathers of both are brothers) but Ayesha Ali has other immediate cousins also from her mother's side. Therefore, other immediate cousins of Ayesha Ali also get related to Ali Abbas despite the fact that they are not immediate to him".
So as we keep on getting relationship details of the people, the direct and indirect relationships can be established.

– Key property: If Haaris is related to Saad and Saad is related to Ahmad, then Haaris is related to Ahmad.

See the key property line's first part above "Harris is related to Saad and Saad is related to Ahmad". Suppose we have a program to handle this list of people and their relationships. After providing all the names "Harris, Saad and Ahmad" to that program and their relationships, the program might be able to determine the remaining part "Harris related to Ahmad".

The same problem (the intelligence required in the program) is described in the sentence below:

– Once we have relationships information, it will be easy for us to answer queries like "Is Haaris related to Ahmad?"

To answer this kind of queries and have that intelligence in our programs, *Disjoint Set ADT* is used. Before going for more information about *Disjoint Set ADT* , we see another application of this ADT in image analysis. This problem is known as *Blob Coloring*.

Blob Coloring

A well-known low-level computer vision problem for black and white images is the following:

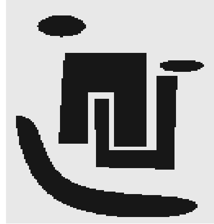
Put together all the picture elements (pixels) that belong to the same "blobs", and give each pixel in each different blob an identical label.

You must have heard of robots that perform certain tasks automatically. How do they know from the images provided to them that in which direction should they move? They can catch things and carry them. They do different things the way human beings do. Obviously, there is a software working internally in robots' body, which is doing all this controlling part and vision to the robot. This is very complex problem and broad area of *Computer Science* and *Electrical Engineering* called *Robotics* (*Computer Vision* in particular).

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

Consider the image below:



``

Fig 33.1

This image is black and white, consisting of five non-overlapping black colored regions of different shapes, called blobs. These blobs are two elliptics- n and u shaped (two blobs) and one arc at the bottom. We can see these five blobs. How can robot identify them? So the problem is:

– We want to *partition* the pixels into *disjoint sets*, one set per blob.

Formatted: Bullets and Numbering

If we make one set per blob, there will be five sets for the above image. To understand the concept of disjoint sets, we can take an analogy with two sets- A and B (as in Mathematics) where none of the elements inside set A is present in set B. The sets A and B are called disjoint sets.

Another problem related to the Computer Vision is the *image segmentation problem*. See the image below on the left of an old ship in gray scales.

Fig 33.2

We want to find regions of different colors in this picture e.g. all regions in the picture of color black and all regions in the image of color gray. The image on the right represents

the resultant image.

Different scanning processes carried out in hospitals through MRI (Magnetic Resonance Imaging), CAT *Scan* or CT *Scan views* the inner parts of the whole body of human beings. These scanned images in gray scales represent organs of the human body. All these are applications of *Disjoint Set ADT*.

Equivalence Relations

Let's discuss some Mathematics about sets. You might have realized that Mathematics is handy whenever we perform analysis of some data structure.

– A binary relation R over a set S is called an *equivalence relation* if it has following propertiesReflexivity: for all element x ∈ S, x R x
  2. Symmetry: for all elements x and y, x R y if and only if y R x
  3. Transitivity: for all elements x, y and z, if x R y and y R z then x R z

– The relation "is related to" is an equivalence relation over the set of people.

You are advised to read about equivalence relations yourself from your text books or from the internet.

Data Structures

Lecture No. 34

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 8
8.1, 8.2, 8.3

Summary

- Equivalence Relations
- Disjoint Sets
- Dynamic Equivalence Problem

Equivalence Relations

We will continue discussion on the abstract data structure, 'disjointSets' in this lecture with special emphasis on the mathematical concept of Equivalence Relations. You are aware of the rules and examples in this regard. Let's discuss it further and define the concept of Equivalence Relations.

'A binary relation R over a set S is called an *equivalence relation* if it has following properties':

1. Reflexivity: for all element x ξ S, x R x

2. Symmetry: for all elements x and y, x R y if and only if y R x
3. Transitivity: for all elements x, y and z, if x R y and y R z then x R z

The relation "is related to" is an equivalence relation over the set of people. This is an example of Equivalence Relations. Now let's see how the relations among people satisfy the conditions of Equivalence Relation. Consider the example of Haris, Saad and Ahmed. Haris and Saad are related to each other as brother. Saad and Ahmed are related to each other as cousin. Here Haris "is related to" Saad and Saad "is related to" Ahmed. Let's see whether this binary relation is Equivalence Relation or not. This can be ascertained by applying the above mentioned three rules.

First rule is reflexive i.e. for all element *x ξ S, x R x*. Suppose that *x* is Haris so *Haris R Haris*. This is true because everyone is related to each other. Second is Symmetry: for all elements *x* and *y, x R y* if and only if *y R x*. Suppose that *y* is Saad. According to the rule, *Haris R Saad* if and only if *Saad R Haris*. If two persons are related, the relationship is symmetric i.e. if I am cousin of someone so is he. Therefore if Haris is brother of Saad, then Saad is certainly the brother of Haris. The family relationship is symmetric. This is not the symmetric in terms of respect but in terms of relationship. The transitivity is: 'for all elements *x, y* and *z*. If *x R y* and *y R z*, then *x R z*'. Suppose *x* is Haris, *y* is Saad and *z* is Ahmed. If Haris "is related to" Saad, Saad "is related to" Ahmed. We can deduce that Haris "is related to" Ahmed. This is also true in relationships. If you are cousin of someone, the cousin of that person is also related to you. He may not be your first cousin but is related to you.

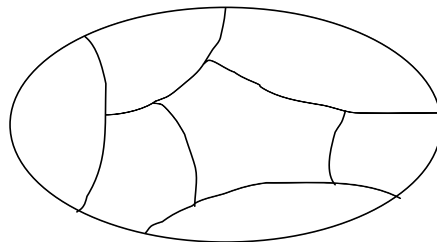
Now we will see an example of binary relationship that is not based on equivalence relationship. The ≤ relationship is *not* an equivalence relation. We will prove this by applying the three rules. The first rule is reflexive. It is reflexive, since $x \leq x$, as x is not less than x but surely is equal to x. Let's check the transitive condition. Since $x \leq y$ and $y \leq z$ implies $x \leq z$., it is also true. However it is *not symmetric* as $x \leq y$ does not imply $y \leq x$. Two rules are satisfied but symmetric rule does not. Therefore ≤ is not an equivalence relation.

Let's see another example of binary relationship that is also equivalence relation. This is from the electric circuit domain. Electrical connectivity, where all connections are by metal wires, is an equivalence relation. You can make the circuit diagram on the paper including transistor, resistors and capacitors etc. These parts are connected to each other with the metal wire. Now let's apply the rules of equivalence relations on it. It is clearly reflexive, since the component is connected to itself. In a circuit, a transistor is connected to itself. It is symmetric due to the fact that if component *a* is connected to component *b,* then *b* must be electrically connected to *a*. Suppose we have two capacitors *a* and *b*. If capacitor *a* is connected to *b*, capacitor *b* is also connected to *a*. It is also transitive. If component a is connected to component *b* and *b* is connected to *c,* then *a* is connected to *c*. This is also true in electrical connectivity. All the three rules of equivalence relations satisfy in this case. Therefore this is equivalence relation.

Disjoint Sets

In the beginning of the lecture, it was told that equivalence relationship partitioned the set. Suppose we have a set of elements and a relation which is also equivalence relation. Let's see what it does mathematically to the set. An equivalence relation R over a set S can be viewed as a partitioning of S into disjoint sets. Here we have an equivalence relationship which satisfies the three conditions. Keep in mind the examples of family relationships or electrical circuits. Here the second point is that each set of the partition is called an *equivalence class* of R (all elements that are related).

Consider the diagram below. We have a set in the shape of an ellipse.



This set has been partitioned into multiple sets. All these parts are disjoint sets and belong to an equivalence class.

Let's discuss an example to understand it. Suppose there are many people around you. How can we separate those who are related to each other in this gathering? We make groups of people who are related to each other. The people in one group will say that they are related to each other. Similarly we can have many groups. So every group will say that they are related to each other with family relationship and there is no group that is related to any other group. The people in the group will say that there is not a single person in the other group who is related to them. There is a possibility that a boy from one group marries to a girl from the other group. Now a relation has been established between these two groups. The people in both groups are related to each other due to this marriage and become a bigger family. With the marriages people living in different families are grouped together. We will do operations like this in case of disjoint sets by combining two sets. You must be aware of the union and intersection operations in sets.

Every member of S appears in exactly one equivalence class. We have divided a set into many disjoint sets. One member of the set can appear in only one equivalence class. Keeping in mind the example of family relations, if a person belongs to a group he cannot go to some other group. The second point is to decide if *a R b.* For this purpose, we need only to check whether *a* and *b* are in the same equivalence class. Consider the example of pixels. If a pixel *p1* is in a region and want to know the relation of pixel *p2* with it, there is only the need to confirm that these two pixels are in the same region. Lastly, this provides a strategy to solve the equivalence problem. With the help of second point we can get help to solve the equivalence problem. So far, we have not seen the disjoint sets and its data structure.

Dynamic Equivalence Problem

Let's talk about the data structure. Suppose we have a set and converted into disjoint sets. Now we have to decide if these disjoint sets hold equivalence relation or not. Keep in mind the example of family relations in which we had divided the people into groups depending upon their relations. Suppose we have to decide that two persons belong to the same family or not. You have to make this decision at once. We have given a set of people and know the binary relation between them i.e. a person is a cousin of other, a person is brother of other etc. So the relation between two persons is known to us. How we can take that decision? What is the data available to us? We have people (with their names) and the binary relation among them. So we have names and the pair of relations. Think that Haris and Ahmed are related to each other or not i.e. they belong to the same family or not. How can we solve this? One way to find out the relationship is given below.

If the relation is stored as a two-dimensional array of booleans, this can be done in constant time. The problem is that the relation is usually not explicitly defined, but shown in implicit terms. Suppose we are given 1000 names of persons and relations among them. Here we are not talking about the friendship as it is not a family relation. We are only talking about the family relations like son-father, brother-brother, cousin-cousin, brother-sister, etc. Can we deduce some more relations from these given relations? Make a two dimensional array, write the names of persons as the columns and rows headings. Suppose Haris is the name of first column and first row. Saad is the name of 2nd col and 2nd row. Similarly Ahmed, Omar, Asim and Qasim are in the 3rd, 4th, 5th, and 6th columns and rows respectively. The names are arranged in the same way in columns and rows.

	Haaris	Saad	Ahmed	Omar	Asim	Qasim		
Haaris	*T*	*T*	*T*					
Saad		*T*	*T*					
Ahmed			*T*					
Omar				*T*				
Asim					*T*	*T*		
Qasim						*T*		

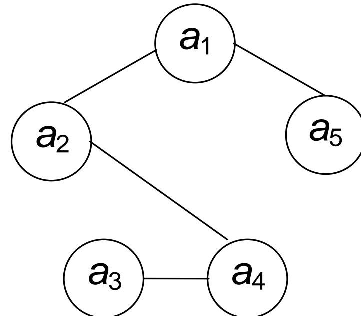
Now visit each cell of matrix. If two persons are related, store T(true) in the corresponding column and row. As Haris and Saad are related to each other, so we take the row of Haris and column of Saad and store T in that cell. Also take the row of Saad and column of Ahmed and mark it with T. By now, we do not know that Haris and Ahmed are related so the entry in this regard will be false. We cannot write T(True) here as this relationship has not be stated explicitly. We can deduce that but cannot write it in the matrix. Take all the pairs of relations and fill the matrix. As Haris is related to Haris so we will write T in the Haris row and Haris column. Similarly, the same can be done in the Saad row and Saad column and so on. This will be square matrix as its rows and columns are equal. All of the diagonal entries are T because a person is related to himself. This is a self relation. Can we find out that Haris and Ahmed are related? We can find the answer of this question with the help of two relations between Haris & Saad and Saad & Ahmed. Now we can write T in the Haris row and Ahmed column. The problem is that this information was not given initially but we deduced this from the given information. We have 1000 people so the size of the matrix will be 1000 rows * 1000 columns. Suppose if we have 100,000 people, then the size of the array will be 100,000*100,000. If each entry requires one byte then how much storage will be required to store the whole array? Suppose if we have one million people, the size will be 10 to the power 14. Do we have as much memory available in the computer? We have one solution but it is not efficient in terms of memory. There is always need of efficient solution in terms of space and time. For having a fast algorithm, we can compromise on the space. If we want to conserve memory, a slow algorithm can be used. The algorithm we discussed is fast but utilized too much space.

Let's try to find out some more efficient solution of this problem. Consider another example. Suppose the equivalence relation is defined over the five-element set {*a*1, *a*2, *a*3, *a*4, *a*5}. These elements can be the name of people, pixels, electrical components etc. How many pairs we can have in this set? We have a pair *a1-a1*, *a1-a2*, *a1-a3, a1-a4, a1-a5, a2-a2, a2-a3* etc. The pair *a1-a2* and *a2-a1* are equal due to symmetric. We also find some self-pairs (i.e. *a1-a1, a2-a2* etc). There are 25 pairs of elements, each of which is related or not (30 pairs – 5 self-pairs = 25). These are the total pairs and we may not have as much relations. What will be size of array with five elements? The size will be 5*5 = 25. We are also given relations i.e.

- *a*1 R *a*2,
- *a*3 R *a*4,
- *a*5 R *a*1,
- *a*4 R *a*2,

We have five people in the example and their relation is given in four pairs. If two persons are brothers, then cousin of one brother is also the cousin of other. We can find out this information with the help of a matrix. We would like to be able to infer this information quickly.

We made nodes of each element. These five nodes are as *a1, a2, a3, a4, a5.*



As *a1 R a2*, so we link the nodes *a1* and *a2*. Similarly *a3 R a4*, these two nodes are also linked. Following this pattern, it is established that *a5 R a1* and *a4 R a2*. So we connect these nodes too. It is clear from the figure that all of these nodes are related to each other. If they are related to each other as cousin, then all of these five persons belong to the same family. They need to be in the same set. They are not in disjoint sets. Is *a3 R a5*? You can easily say yes. How you get this information? This relation is not provided in the given four relations. With the above tree or graph we can tell that *a3* is related to *a5*. We can get the information of relationship between different persons using these nodes and may not need the matrix.

We want to get the information soon after the availability of the input data. So the data is

processed immediately. The input is initially a collection of *n* sets, each with one element. Suppose if we have 1000 people, then there will be need of 1000 sets having only one person. Are these people related to each other? No, because every person is in different set. This initial representation is that all relations (except reflexive relations) are false. We have made 1000 sets for 1000 people, so only the reflexive relation (every person is related to himself) is true. Now mathematically speaking, each set has a different element so that $S_i \cap S_j = \emptyset$ which makes the sets *disjoint*. A person in one set has no relation with a person in another set, therefore there intersection is null. Now here we have 1000 sets each containing only one person. Only the reflexive relation is true and all the 1000 sets are disjoint. If we take intersection of any two sets that will be null set i.e. there is no common member in them.

Sometimes, we are given some binary relations and are asked to find out the relationship between two persons. In other words, we are not given the details of every pair of 1000 persons. The names of 1000 persons and around 50 relations are provided. With the help of these 50 relations, we can find out more relations between other persons. Such examples were seen above while find out the relationship with the help of graph.

There are two permissible operations in these sets i.e. *find* and *union*. In the *find* method, we are given one element (name of the person) and asked to find which set it belongs to. Initially, we have 1000 sets and asked in which set person 99 is? We can say that every person is in a separate set and person 99 is in set 99. When we get the information of relationships between different persons, the process of joining the sets together can be started. This is the union operation. When we apply union operation on two sets, the members of both sets combined together and form a new set. In this case, there will be no duplicate entry in the new sets as these were disjoint. The definitions of *find* and *union* are:

- *Find* returns the name of the set (equivalence class) that contains a given element, i.e., $S_i = \text{find}(a)$
- *Union* merges two sets to create a new set $S_k = S_i \cup S_j$.

We give an element to the *find* method and it returns the name of the set. The method *union* groups the member of two sets into a new set. We will have these two operations in the disjoint abstract data type. If we want to add the relation *a R b*, there is need to see whether *a* and *b* are already related. Here *a* and *b* may be two persons and a relation is given between them. First of all we will see that they are already related or not. This is done by performing *find* on both *a* and *b* to check whether they are in the same set or not. At first, we will send *a* to the *find* method and get the name of its set before sending b to the *find* method. If the name of both sets is same, it means that these two belong to the same set. If they are in the same set, there is a relation between them. We did not get any useful information with this relation. Let's again take the example of the Haris and Saad. We know that Haris is the brother of Saad, so they can be placed into a new set. Afterwards, we get the information that Saad is the brother of Haris.

Now the question arises, if they are not in the same set, what should we do? We will not

waste this information and apply *union* which merges the two sets *a* and *b* into a new set. This information will be helpful later on. We keep on building the database with this information. Let's take the example of pixels. Suppose we have two pixels and are told that these two belong to the same region, have same color or these two pixels belong to the liver in CT scan. We will keep these two pixels and put them in one set. We can name that set as *liver*. We will use union operation to merge two sets. The algorithm to do this is frequently known as *Union*/*Find* for this reason.

There are certain points to note here. We do not perform any operations comparing the relative values of set elements. That means that we are not considering that one person is older than the other, younger or one electrical component is bigger than other etc. We merely require knowledge of their location, i.e., which set an element, belongs to.

Let's talk about the algorithm building process. We can assume that all elements are numbered sequentially from 1 to *n*. Initially we have $Si = \{i\}$ for $i = 1$ through *n*. We can give numbers to persons like person1, person2 etc. Consider the example of jail. In a jail, every prisoner is given a number. He is identified by a number, not by the name. So you can say that persons in a cell of jail are related. We will use these numbers to make our algorithm run faster. We can give numbers to every person, every pixel, each electrical component etc. Under this scheme, the member of *S10* is number 10. That number 10 can be of a person, a pixel or an electrical component. We are concerned only with number.

Secondly, the name of the set returned by *find* is fairly arbitrary. All that really matters is that *find*(*x*) = *find*(*y*) if and only if *x* and *y* are in the same set. We will now discuss a solution to the *union*/*find* problem that for any sequence of at most *m* finds and up to *n*-1 unions will require time proportional to $(m + n)$. We are only storing numbers in the sets and not keeping any other relevant information about the elements. Similarly we are using numbers for the names of the sets. Therefore we may want that the method *find* just returns the number of the set i.e. the person 10 belongs to which set number. The answer may be that the person 10 belongs to set number 10. Similarly the set number of person 99 may be set 99. So if the set numbers of two persons is not equal, we can decide that these two persons do not belong to the same set. With the help of this scheme, we will develop our algorithm that will run very fast. In the next lecture, we will discuss this algorithm and the data structure in detail.

Data Structures

Lecture No. 35

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 8

8.2, 8.3

Summary

- Dynamic Equivalence Problem
- Example 1
- Parent Array
  - Initialization
  - Find (i)
  - Union (i, j)
- Example 2
  - Initialization
  - union operation
  - find Operation
- Running Time Analysis

Before talking about the data structure implementation in detail, we will recap the things discussed in the previous lecture. The concepts that came under discussion in the last lecture included the implementation of the disjoint set. It was observed that in case of data elements, we assign a unique number to each element to make sets from these numbers. Moreover, techniques of merger and searching in these sets by using the operations union and find were, talked about. It was seen that if we send a set item or a number to *find* method, it tells us the set of which this item is a member. It will return the name of the set. This name of set may also be a number. We talked about the *union* method in which members of two sets join together to form a new set. The *union* method returns the name or number of this new set.
Now we will discuss the data structure used to implement the disjoint sets, besides ascertaining whether this data structure implements the find and union operations efficiently.

Dynamic Equivalence Problem
We are using sets to store the elements. For this purpose, it is essential to remember which element belongs to which set. We know that the elements in a set are unique and a tree is used to represent a set. Each element in a tree has the same root, so the root can be used to name the set. The item (number) in the root will be unique as the set has unique values of items. We use this root as the name of set for our convenience. Otherwise, we can use any name of our choice. So the element in the root (the number) is used as the name of the set. The *find* operation will return this name. Due to the presence of many sets, there will be a collection of trees. In this collection, each tree will be representing one set. If there are ten elements, we will have ten sets initially. Thus there will be ten trees at the beginning. In general, we have N elements and initially there will be N trees. Each tree will have one element. Thus there will be N trees of one node. Now here comes a definition for this collection of trees, which states that a collection of trees is called a *forest*.
The trees used for the sets are not necessarily binary trees. That means it is not necessary that each node should have a maximum of two children nodes. Here a node may have more than two children nodes.

To execute the union operation in two sets, we merge the two trees of these sets in such a manner that the root of one tree points to the root of other. So there will be one root, resulting in the merger of the trees. We will consider an example to explain the union operation later.
In the find operation, when we call *find (x)*, it helps us to know which set this *x* belongs to. Internally, we find this *x* in a tree in the *forest*. When this *x* is found in a tree the *find* returns the number at root node (the name of the set) of that tree.

Example 1
Now let's consider an example to understand this phenomenon. Suppose, we have developed a data structure and apply the union and find operations on it. There are eight elements in this data structure i.e.1 to 8. These may be the names of people to which we have assigned these numbers. It may be other distinct eight elements. We will proceed with these numbers and make a set of each number. The following figure shows these numbers in different sets.

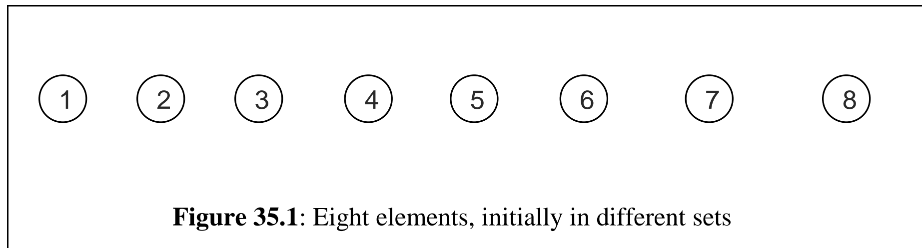


Figure 35.1: Eight elements, initially in different sets

Now we carry out union operation on the sets 5 and 6. The call *union(5,6)* means, merge the sets 5 and 6 and return the new set developed due to the merger of the sets- 5 and 6. In the following figure, we see this union. We put the set 6 below set 5, which join together.

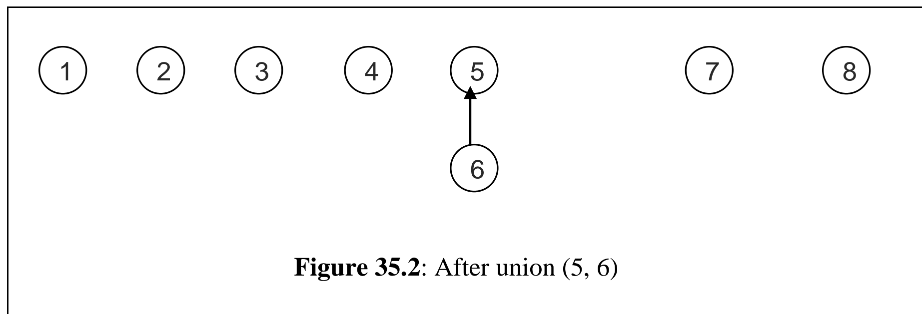


Figure 35.2: After union (5, 6)

After the merger, the new set contains two members 5 and 6. Now the question arises what will be the name of this new set. In the above union operation, the set 6 becomes the node of 5. It may be in reverse i.e. 5 could be a node of 6. In the above figure, we put 6 as node of 5. Moreover the arrow joining these is from 6 to 5. In the new set formed due to the merger of 5 and 6, it seems that 5 has some superiority. So the name of the new set is 5. We passed two arguments 5 and 6 to the *union* function. And the union made 6 a

member of 5. Thus, if we pass S1 and S2 to the *union* function, the *union* will make S2 a member of S1. And the name of the new set will be S1. That means the name of first argument will be the name of the new set.
Now we call *union (7,8)*. After this call, 7 and 8 form a new set in which 8 is a member of 7 and the name of this new set is 7 (that is the first argument in the call). In other words, 7 is root and 8 its child. This is shown in the following figure.

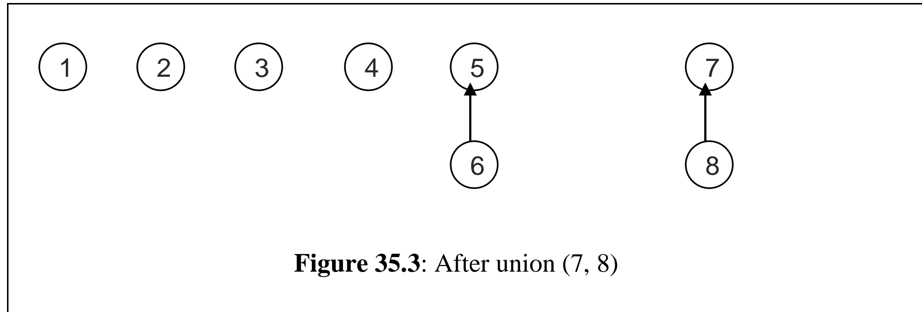


Figure 35.3: After union (7, 8)

Now we call the union function by passing it set 5 and set 7 as arguments i.e. we call *union (5,7)*. Here the sets 5 and 7 have two members each. 5 and 6 are the members of 5 and similarly the two members of 7 are 7 and 8. After merging these sets, the name of the new set will be 5 as stated earlier that the new set will be named after the first argument. The following figure (figure 35.4) shows that now in the set 5, there are four members i.e. 5, 6, 7 and 8.

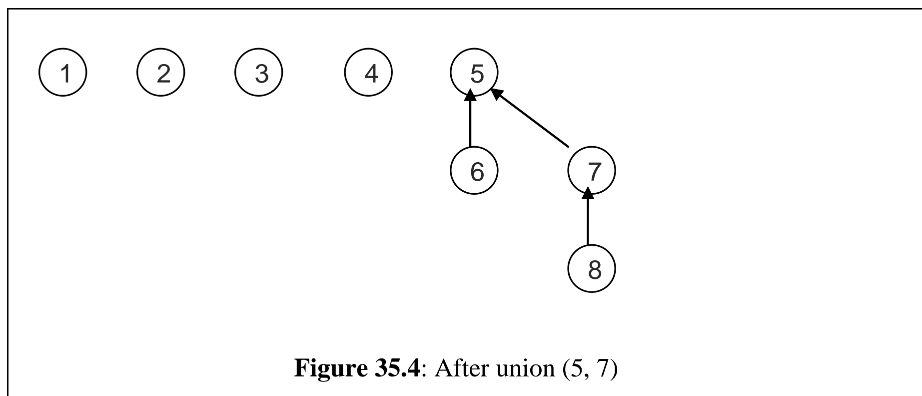


Figure 35.4: After union (5, 7)

We can see that there are four unique set members in this set (i.e. 5).
We will demonstrate the union operation with the help of another example. Suppose, we have made another union that is *union (3,4)*. This call merges the set 4 with the set 3. The following figure shows this.

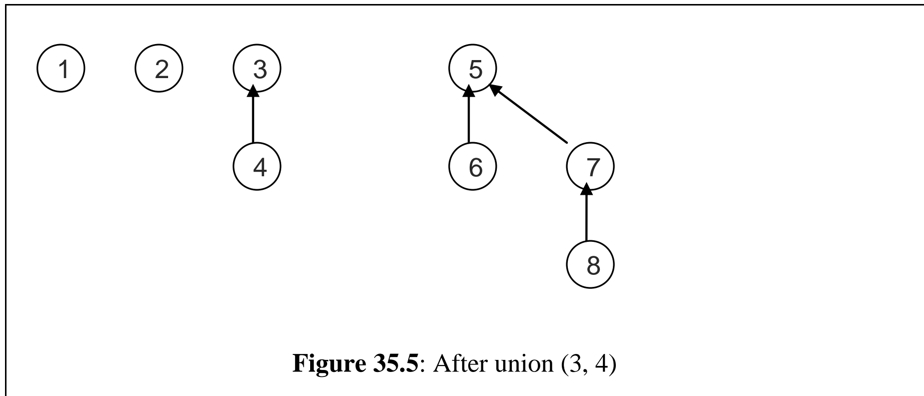


Figure 35.5: After union (3, 4)

In this figure, we see that there are four sets that are 1, 2, 3 and 5. Now we unite the sets 4 and 5 and make a call *union (4,5)*. The following figure shows the outcome of this call. In this figure, the set 5 points to 3 whereas we made a call *union (4,5)*.

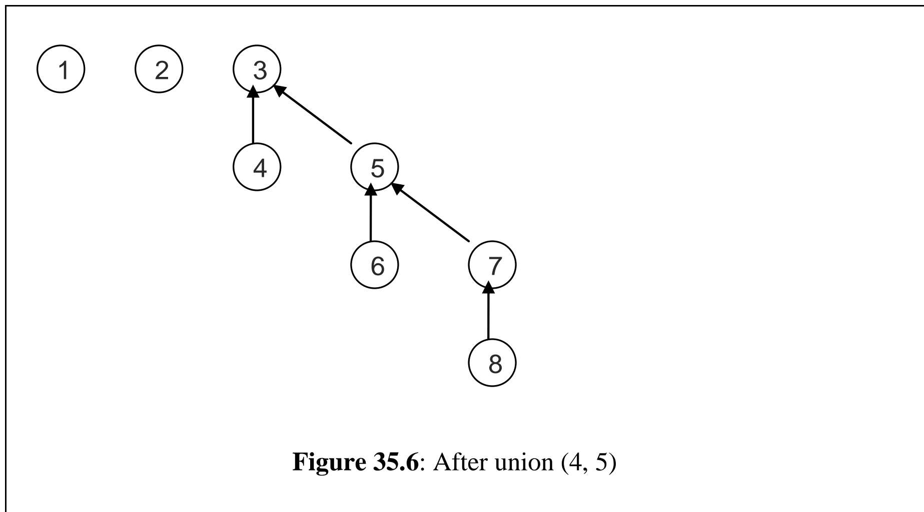


Figure 35.6: After union (4, 5)

So we conclude that it is not necessary that the caller should send the roots to union. It is necessary that the union function should be such that if a caller sends elements of two sets, it should find the sets of those elements, merge them and return the name of new set. Thus our previous call i.e. *union (4,5)* was actually carried out in the way that first the *union* finds the root of 4 that is 3. Then it looks for 5 that itself is a root of its set. After this, it merges both the trees (sets). This merger is shown in the above figure i.e. Figure 35.6.

Up to now, we have come to know that the formation of this tree is not like a binary tree in which we go down ward. Moreover, a binary tree has left and right children that are not seen yet in the tree we developed. This is a tree like structure with some properties.

Let's talk about these properties.
Here we see that typical tree traversal (like inorder, preorder or postorder) is not required. So there is no need for pointers to point to the children. Instead, we need a pointer to parent, as it's an up-tree. We call it up-tree due to the fact that it is such a tree structure in which the pointers are upward. These parent pointers can be stored in an array. Here we have to keep (and find) pointer to the parent of a node unlike a binary tree in which we keep pointer to child nodes. In the array, we will set the parent of root to –1. We can write it as

```
Parent[i] = -1          // if i is the root
```

Now we will keep these tree structures (forest) in an array in the same manner. With the merger of the trees, the parents of the nodes will be changed. There may be nodes that have no parent (we have seen this in the previous example). For such nodes, we will keep –1 in the array. This shows that this node has no parent. Moreover, this node will be a root that may be a parent of some other node.

Now we will develop the algorithm for *find* and *union*. Let's consider an example to see the implementation of this disjoint set data structure with an array.

Parent Array

Initialization
We know that at the start we have *n* elements. These *n* elements may be the original names of the elements or unique numbers assigned to them. Now we take an array and with the help of a *for* loop, keep these *n* elements as root of each set in the array. These numbers are used as the index of the array before storing –1 at each location from index zero to *n*. We keep –1 to indicate a number as root. In code, this *for* loop is written as under.

```
for ( i = 0; i < n ; i ++)
        Parent [i] = -1 ;
```

Find (i)

Now look at the following code of a loop. This loop is used to find the parent of an element or the name of the set that contains that element.

```
// traverse to the root (-1)
for(j=i; parent[j] >= 0; j=parent[j])
        ;
return j;
```

In this loop, *i* is an argument passed to the find function. The execution of the loop starts from the value, passed to the find function. We assign this value to a variable *j* and check whether its parent is greater than zero. It means that it is not –1. If it is greater than zero, its parent exists, necessitating the re-initialization of the *j* with this parent of *j* for the next iteration. Thus this loop continues till we find parent of *j* (parent [*j*]) less than zero (i.e. -1). This means that we come to the root before returning this number *j*.

Union (i, j)

Now let's see the code for the function of union. Here we pass two elements to the function union. The union finds the roots of i and j. If *i* and *j* are disjoint sets, it will merge them. Following is the code of this function.

```
root_i = find(i);
root_j = find(j);
if (root_i != root_j)
      parent[root_j] = root_i;
```

In the code, we see that at first it finds the root of tree in which *i* exists by the find(*i*) method and similarly finds the root of the set containing j by find(*j*). Then there is a check in if statement to see whether these sets (roots) are same or not. If these are not the same, it merges them in such a fashion that the root *i* is set as the parent of root *j*. Thus, in the array where the value of root *j* exists, the value of root *i* becomes there.

Example 2

To understand these concepts, let's consider an example. We re-consider the same previous example of eight numbers and see how the initialization, union and find work.

Initialization

In the following figure (figure 35.7), we have shown the initialization step. Here we make an array of eight locations

and have initialized them with –1 as these are the roots of the eight sets. This –1 indicates that there is no parent of this number. We start the index of array from 1 for our convenience. Otherwise, we know that the index of an array starts from zero.

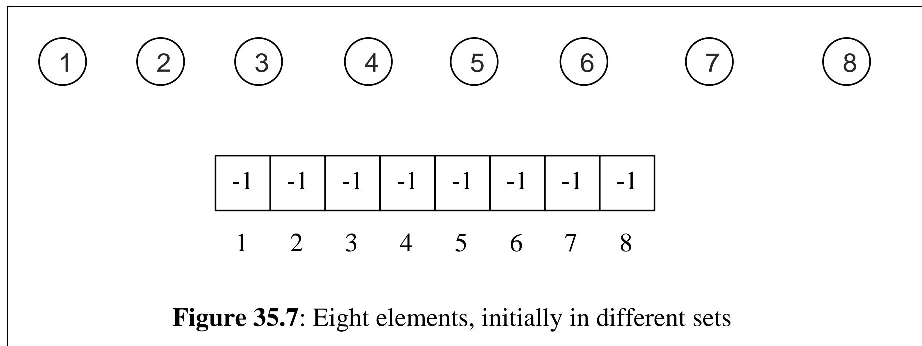


Figure 35.7: Eight elements, initially in different sets

Union Operation

Now we come to the union operation. First of all, we call union (5,6). This union operation forms an up-tree in which 5 is up and 6 down to it. Moreover 6 is pointing to 5. In the array, we put 5 instead of –1 at the position 6. This shows that now the parent of 6 is 5. The other positions have –1 that indicates that these numbers are the roots of some tree. The only number, not a root now is 6. It is now the child of 5 or in other words, its parent is 5 as shown in the array in the following figure.

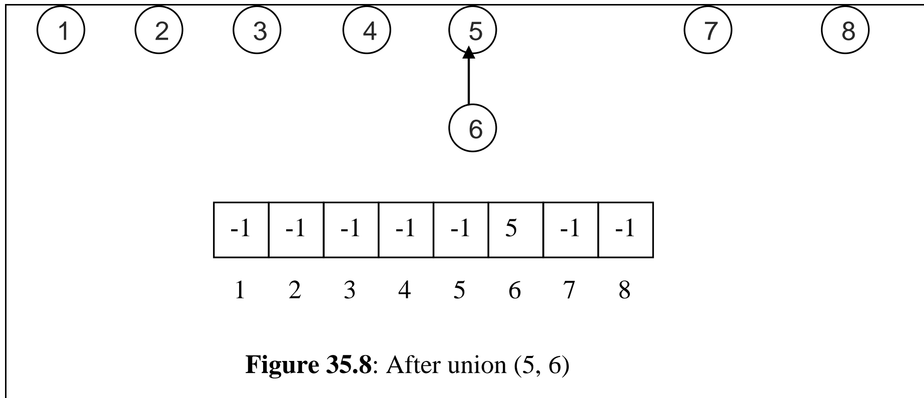


Figure 35.8: After union (5, 6)

Now we carry out the same process (i.e. union) with the numbers 7 and 8 by calling union (7,8). The following figure represents this operation. Here we can see that the parent of 8 is 7 and 7 itself is still a root.

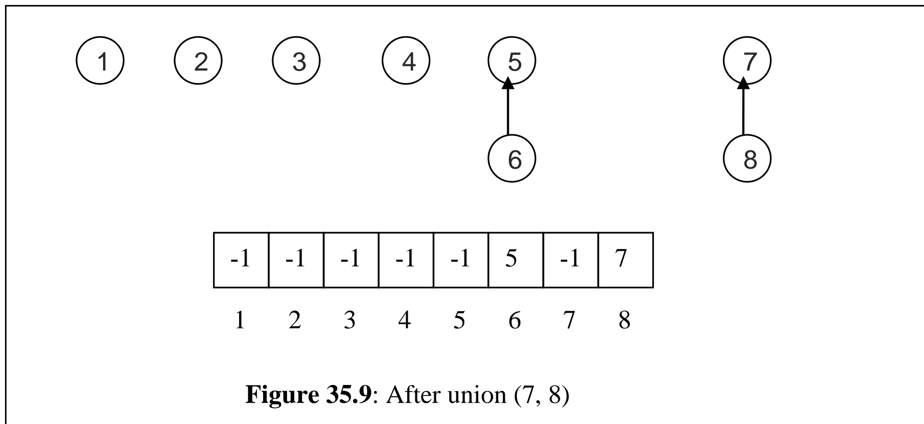


Figure 35.9: After union (7, 8)

Now we execute the *union (5,7)*. The following figure represents the tree and array status after the performance of this union operation.

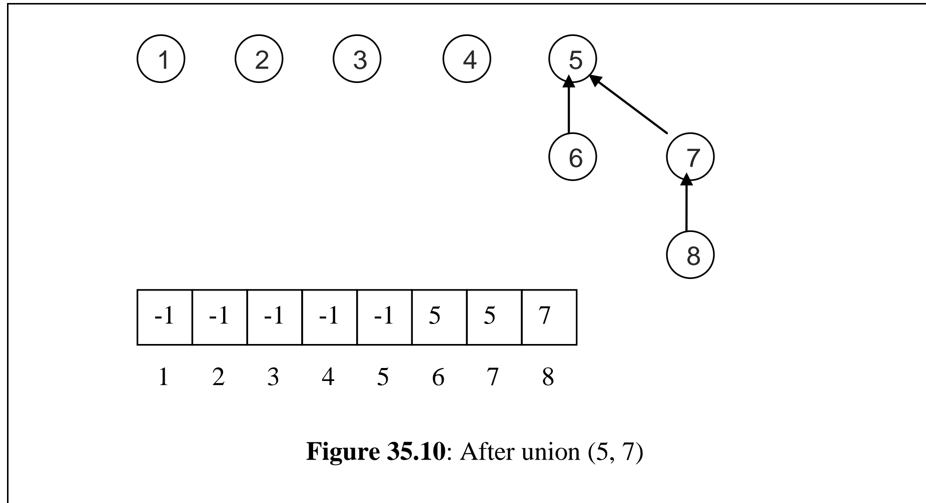
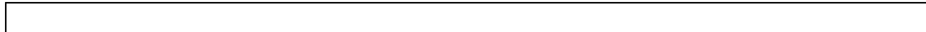
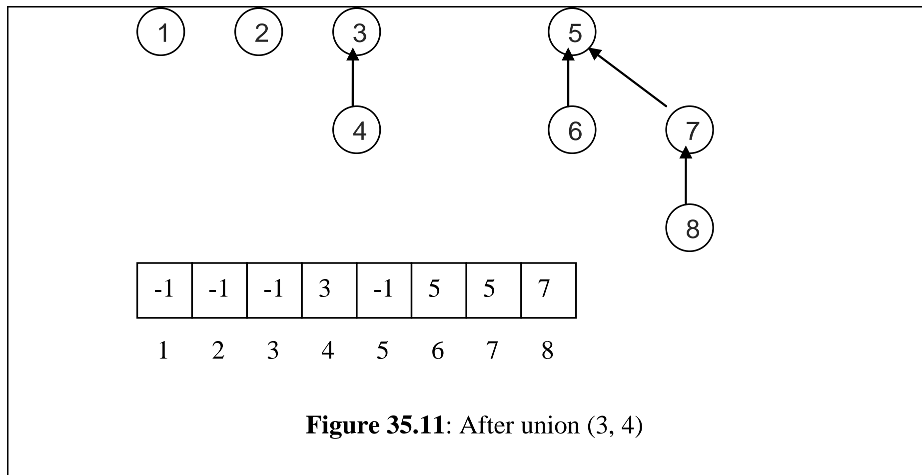


Figure 35.10: After union (5, 7)

The tree in the figure shows that 7 points to 5 now. In the array, we see that the value at position 7 is 5. This means that the parent of 7 is 5 now. Whereas the parent of 5 is still –1 that means it is still a root. Moreover, we can see that the parent of 8 is 7 as before.

Afterwards, we call *union (3,4)*. The effect of this call is shown in the following figure. Here in the array at position 4, there is 3 instead of –1. This shows that the parent of 4 is 3.





-1	-1	-1	3	-1	5	5	7
1	2	3	4	5	6	7	8

Figure 35.11: After union (3, 4)

By looking at the array only, we can know that how many trees are there in the collection (forest). The number of trees will be equal to the number of –1 in the array. In the above figure, we see that there are four –1 in the array so the number of trees is four and the trees in the figure confirms this. These four trees are with the roots 1, 2 3 and 5 respectively.

Now we carry out another union operation. In this operation, we do the union of 4 and 5 i.e. union (4, 5). Here the number 4 is not a root. Rather, it is an element of a set. We have already discussed that the union operation finds the set (root) of the element passed to it before putting together those sets. So here the union finds the set containing 4 that is 3. The element 5 itself is a name (root) of a set. Now the union operation merges the two sets by making the second set a member of the first set. In this call (union (4, 5)), the first set is 3 (that is found for the element 4) and second is 5. So the union joins 5 with 3 while 5 is pointing to 3. Due to this union operation, now, in the array, there is 3 at position 5 instead of –1. The 3 at this position 5 indicates that the parent of 5 is 3. The following figure shows the tree and array after the operation union (4,5).

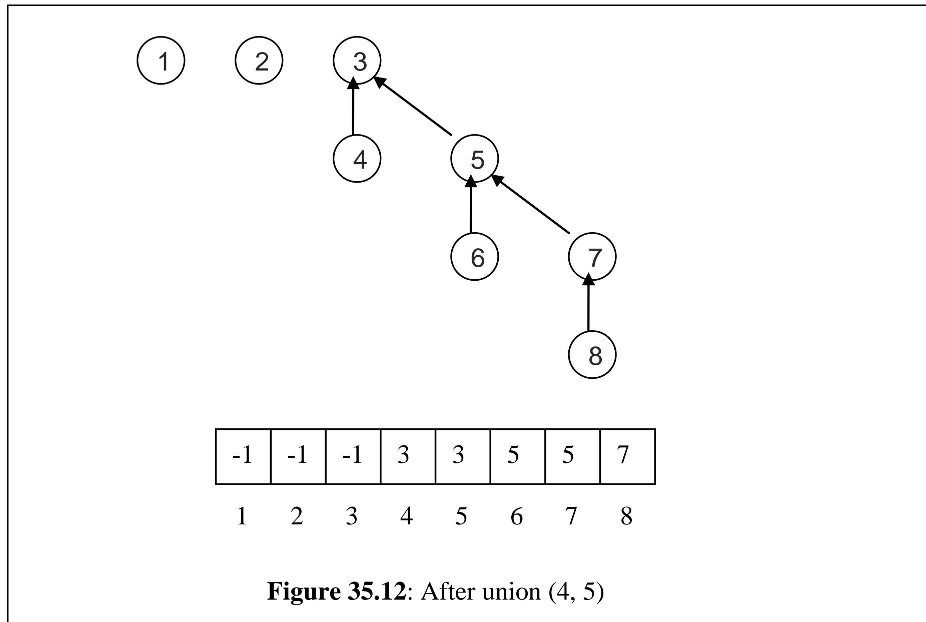


Figure 35.12: After union (4, 5)

These were the union operations while using parent array. Now let's look at the find operation.

Find Operation

To understand the find operation, we make a call *find (8)* in the above forest. This call means that the caller wants to know the set in which number 8 is lying. We know that the root of the tree is also the name of the set. By looking at the trees in the figure below, we come to know that 8 is in the tree with 3 as root. This means 8 is in set 3.

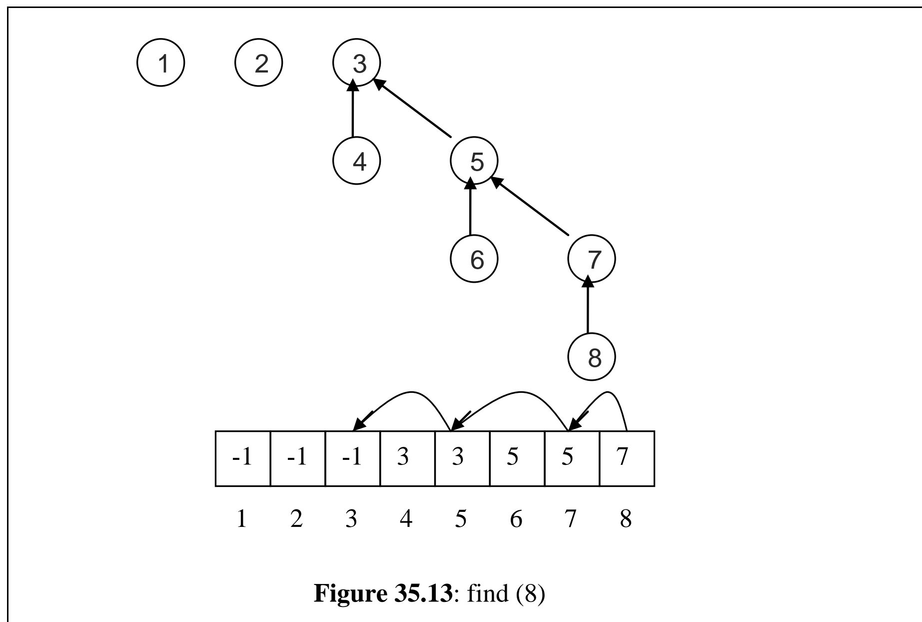


Figure 35.13: find (8)

Thus from the figure, we find the set containing 8. This is being implemented with the parent array to ultimately find it in the array. For this find operation in the array, we have discussed the algorithm in which we used a '*for* loop. This 'for loop' starts from the position that given to the *find* function. Here it is 8. The condition in the for loop was that as long as parent [*j*] is greater than zero, set this parent [*j*] to *j*. Now we execute the loop with the value of *j* equal to 8. First of all, the loop goes to position 8 and looks for the value of parent of 8. This value is 7, which sets the value of *j* to 7 and goes to position 7. At that position, the value of parent of 7 is 5. It goes to position 5. At position 5, the value of parent is 3. So the loop sets the value of *j* equal to 3 and goes to the position 3. Here it finds that the parent of 3 is –1 i.e. less than zero so the loop ends. This position 3 is the root and name of the set as parent of it is –1. Thus the name of the

set that contains 8 is 3. The find will return 3 which mean that 8 is a member of set 3.
Similarly, in the above array, we can execute the operation *find (6)*. This will also return 3 as the loop execution will be at the positions 6 before going to 5 and finally to 3. It will end here, as the parent of 3 is –1. Thus *find (6)* returns 3 as the set that contains 6.

Running Time analysis

Now we will discuss how the implementation of disjoint set is better. We must remember that while discussing the implementation of disjoint set, we talked about Boolean matrix. This is a two dimensional structure in which the equivalence relations is kept as a set of Boolean values. Here in the parent array, we also are keeping the same information. The union will be used when the two items are related. In the two-dimensional matrix, we can easily find an item by its index. Now we are using an array that is a single dimensional structure and seems better with respect to space. We keep all the information in this array which is at first kept in a matrix. Now think about a single dimension array versus a two dimensional array. Suppose we have 1000 set members i.e. names of people. If we make a Boolean matrix for 1000 items, its size will be 1000 x 1000. Thus we need 1000000 locations for Boolean values. In case of an array, the number of locations will be 1000. Thus this use of array i.e.tree like structure is better than two-dimensional array in terms of space to keep disjoint sets and doing union and find operations. Moreover, we do not use pointers (addresses) that are used in C++. We use array indices as pointers. In case of find, we follow some indices (pointers) to find the set containing a particular member. From these points of discussion, we conclude that

- *union* is clearly a constant time operation.
- Running time of *find*(*i*) is proportional to the height of the tree containing node *i*.
- This can be proportional to n in the worst case (but not always)
- Goal: Modify *union* to ensure that heights stay small

We will discuss these points in detail in the next lecture.

Data Structures

Lecture No. 36

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 8
8.4, 8.5, 8.6

Summary

- Running Time Analysis
- Union by Size
- Analysis of Union by Size
- Union by Height
- Sprucing up find
- Timing with Optimization

Running Time Analysis

In the previous lecture, we constructed up trees through arrays to implement disjoint sets. u*nion* and *find* methods along with their codes were discussed with the help of figures. We also talked about the ways to improve these methods. As far as the *union* operation is concerned, considering the parent array, it was related to changing the index of the array. The root value of the merging tree was put into that. In case of find, we have to go to the root node following the up pointers.

While studying the balanced trees, we had constructed a binary search tree from the already sorted data. The resultant tree becomes a linked list with height N. Similar kind of risks involved here. Suppose, we have eight members i.e.1, 2, 3, 4, 5, 6, 7 and 8 as discussed in an example in the previous lecture. Suppose their unions are performed as union(1,2), union(2,3), union(3,4) and so on up to union(7,8). When the trees of these unions are merged together, there will be a longer tree of height 8. If we perform a find operation for an element that is present in the lowest node in the tree then N-1 links are traversed. In this particular case, 8-1=7 links.
We know that with reduced tree (with lesser height), the time required for *find* operation will also be reduced. We have to see whether it is possible to reduce the size (in terms of height) of the resultant tree (formed after unions of trees). Therefore, our goal is:

- ***Goal***: Modify *union* to ensure that heights stay small

We had already seen this goal in the last slide of our previous lecture, which is given below in full:

- *union* is clearly a constant-time operation.
- Running time of *find*(*i*) is proportional to the height of the tree containing node *i*.
- This can be proportional to n in the worst case (but not always)
- Goal: Modify *union* to ensure that heights stay small

You might be thinking of employing the balancing technique here. But it will not be sagacious to apply it here. Due to the use of array, the 'balancing' may prove an error-prone operation and is likely to decrease the performance too. We have an easier and appropriate method, called *Union by Size* or *Union by Weight*.

Union by Size

Following are the salient characteristics of this method:

- Maintain sizes (number of nodes) of all trees, and during *union.*
- Make smaller tree, the subtree of the larger one.
- Implementation: for each root node `i`, instead of setting `parent[i]` to `-1`, set it to `-k` if tree rooted at `i` has `k` nodes.
- This is also called *union-by-weight*.

We want to maintain the sizes of the trees as given in the first point. Until now, we are not maintaining the size but only the *parent* node. If the value for *parent* in a node is –1, then this indicates that the node is the *root* node. Consider the code of the *find* operation again:

```
// find(i):
// traverse to the root (-1)
for(j=i; parent[j] >= 0; j=parent[j])
          ;
return j;
```

The terminating condition for loop is checking for non-negative values. At any point when the value of *parent[j]* gets negative, the loop terminates. Note that the condition does not specify exactly the negative number. It may be the number –1, –2 , –3 or some other negative number that can cause the loop to be terminated. That means we can also put the number of nodes (size) in the tree in this place. We can put the number of nodes in a tree in the *root* node in the negative form. It means that if a tree consists of six nodes, its *root* node will be containing –6 in its *parent*. Therefore, the node can be identified as the *root* being the negative number and the magnitude (the absolute value) of the number will be the size (the number of nodes) of the tree.

When the two trees would be combined for union operation, the tree with smaller size will become part of the larger one. This will cause the reduction in tree size. That is why it is called union-by-size or union-by-weight.

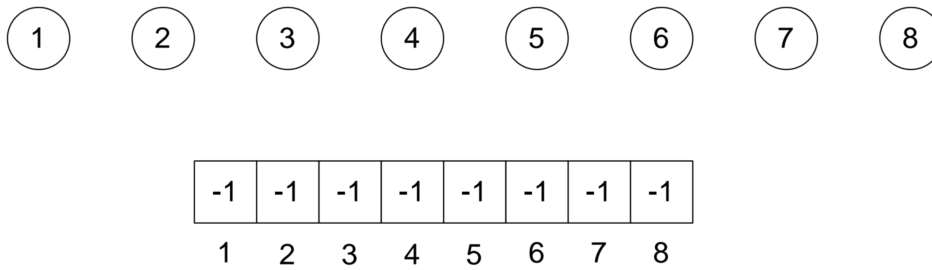
Let's see the pseudo-code of this union operation (quite close to C++ language). This contains the logic for reducing tree size as discussed above:

```
        //union(i,j):
1.      root1 = find(i);
2.      root2 = find(j);
3.      if (root1 != root2)
4.               if (parent[root1] <= parent[root2])
```

```
5.          {
6.                  // first tree has more nodes
7.                  parent[root1] += parent[root2];
8.                  parent[root2] = root1;
9.          }
10.         else
11.         {
12.                 // second tree has more nodes
13.                 parent[root2] += parent[root1];
14.                 parent[root1] = root2;
15.         }
```

This operation performs the *union* of two sets; element i's set and element j's set. The first two lines are finding the *root*s of the sets of both the elements i.e.i and j. Line 3 is performing a check that the *root1* is not equal to *root2*. In case of being unequal, these are merged together. In next statement (line 4), the numbers in *parent* variables of *root* nodes are compared. Note that, the comparison is not on absolute values. So the tree with greater number of nodes would contain the lesser number mathematically. The condition at line 4 will be true, if the tree of *root1* is greater in size than that of *root2*. In line 7, the size of the smaller tree is being added to the size of the greater one. Line 8 is containing the statement that causes the trees to merge. The *parent* of *root2* is being pointed to *root1*. If the *root2* tree is greater in size than the *root1* tree, the if–condition at line 4 returns false and the control is transferred to *else* part of the if-condition. The *parent* of *root1* (the size of the *root1* tree) is added in the *parent* of *root2* in line 13 and the *root1* tree is merged into the *root2* tree using the statement at line 14.

Let's practice this approach using our previous example of array of eight elements.



Eight elements, initially in different sets.

Fig 36.1

These are eight nodes containing initially –1 in the *parent* indicating that each tree contains only one node. Let's unite two trees of 4 and 6. The new tree will be like the one shown in Fig 36.2.

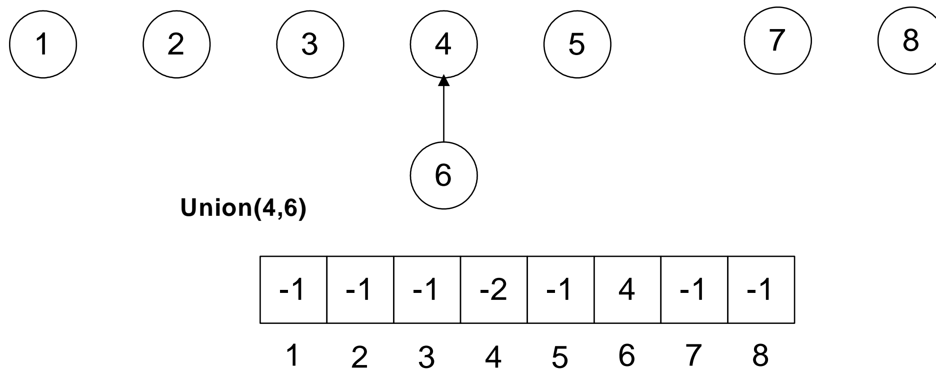


Fig 36.2

From the figure, you can see the node 6 has come down to node 4. The array also depicts that the *parent* at position 4 is containing –2. The number of nodes has become 2. The position 6 is set to 4 indicating that *parent* of 6 is 4. Next, we similarly perform the *union* operation on nodes 2 and 3.

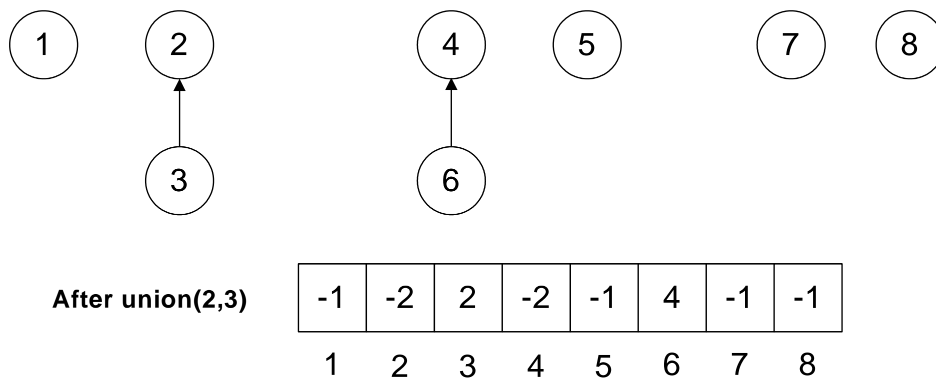


Fig 36.3

Let's perform the *union* operation further and merge the trees 1 and 4.

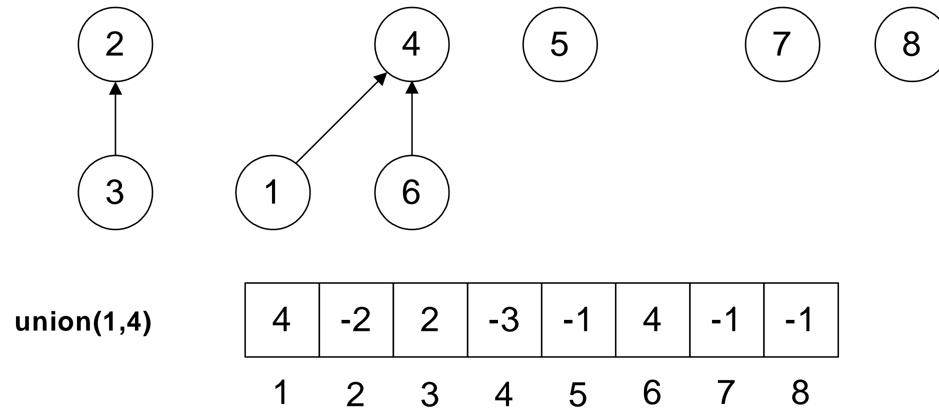


Fig 36.4

1 was a single node tree (-1 in the array at position 1) and 4 was the root node of the tree with two elements (-2 in the array at position 4) 4 and 6. As the tree with root 4 is greater, therefore, node 1 will become part of it. Previously (when the union was not based on weight), it happened contrary to it with second argument tree becoming the part of the first tree.
In this case, the number of levels in the tree still remains the same (of two levels) as that in the greater tree (with root 4). But we apply our previous logic i.e. the number of levels of the trees would have been increased from two to three after the union. Reducing the tree size was our goal of this approach.

In the next step, we merge the trees of 2 and 4. The size of the tree with root 2 is 2 (actually –2 in the parent array) while the size of the tree with root 4 is 3 (actually –3). So the tree of node 2 will be joined in the tree with root 4.

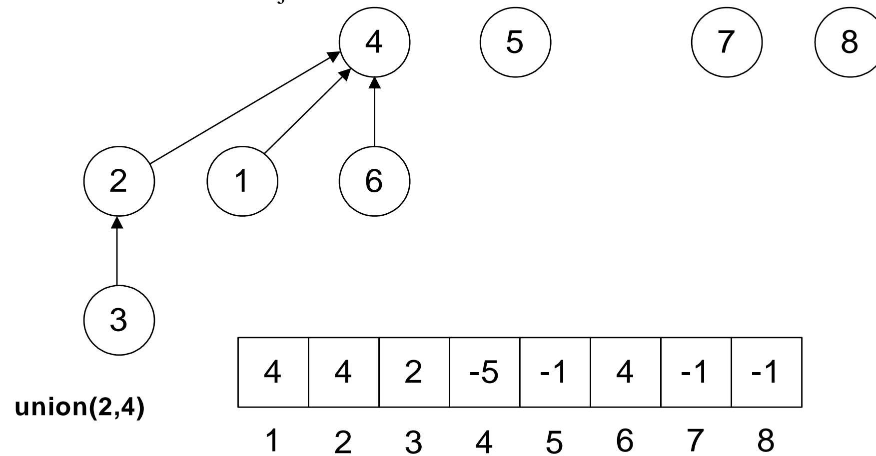


Fig 36.5

The latest values in the array i.e. at position 4, have been updated to –5 (indicating the size of the tree has reached 5). At position 2, the new value is 4, which indicates that the up node of 2 is 4.
Next, we perform the *union(5,4)* operation. We know that the size of the tree with node 4 is 5 (-5 in the array) and node 5 is single node tree. As per our rules of union by size, the node 5 will become part of the tree 4.

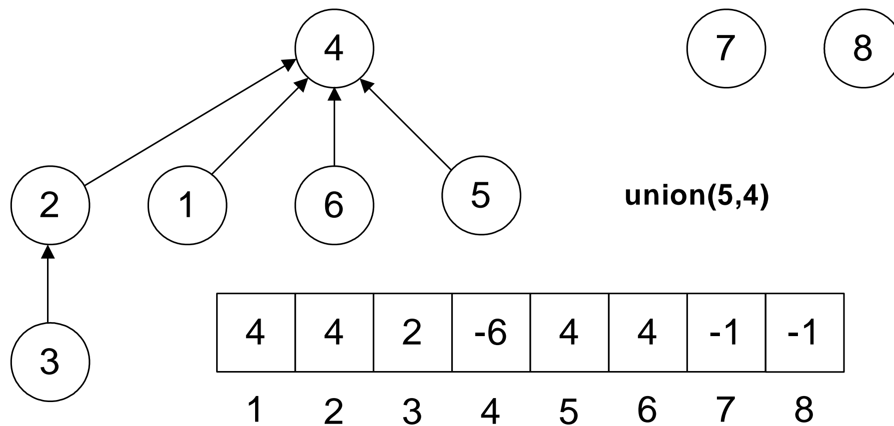


Fig 36.6

The updates inside the array can be seen as the value at position 4 has been changed to –6 and new value at position 5 is 4. This is the tree obtained after applying the *union-by-size approach*. Consider, if we had applied the previous method of union, tree's depth would have been increased more.
It seems that we have achieved our goal of reducing the tree size up to certain extent. Remember, the benefit of reducing the tree height is to increase performance while finding the elements inside the tree.

Analysis of Union by Size

– If unions are done by weight (size), the depth of any element is never greater than $\log_2 n$.

By following the previous method of union where second tree becomes the part of the first tree, the depth of the tree may extend to N. Here N is the number of nodes in the tree. But if we take size into account and perform union by size, the depth of tree is $\log_2 n$ maximum. Suppose the N is 100,000 i.e.for 100,000 nodes. The previous methods may give us a tree with depth level as 100,000. But on the other hand, $\log_2 100000$ is approximately 20.U*nion-by-size* gives us a tree of 20 levels of depth maximum. So this is

a significant improvement.

Mathematical proof of this improvement is very complex .We are not covering it here but only providing the logic or reasoning in intuitive proof.

Intuitive Proof:
- Initially, every element is at depth zero.
- When its depth increases as a result of a union operation (it's in the smaller tree), it is placed in a tree that becomes at least twice as large as before (union of two equal size trees).
- How often can each union be carried out? -- $\log_2 n$ times, because after at most $\log_2 n$ unions, the tree will contain all n elements.

Union by Height

- Alternative to *union-by-size* strategy: maintain heights,
- During *union*, make a tree with smaller height a subtree of the other.
- Details are left as an exercise.

This is an alternate way of *union-by-size* that we maintain heights of the trees and join them based on their heights. The tree with smaller height will become part of the one with greater height. This is quite similar with the *union-by-size*. In order to implement Disjoint Set ADT, any of these solutions can work. Both the techniques i.e. *union-by-size* and *union-by-height* are equivalent, although, there are minor differences when analyzed thoroughly.

Now, let's see what can we do for *find* operation to make it faster.

Sprucing up Find

- So far we have tried to optimize *union*.
- Can we optimize *find*?
- Yes, it can be achieved by using *path compression* (or compaction).

Considering performance, we can optimize the *find* method. We have already optimized the trees *union* operation and now we will see how can optimize the *find* operation using the *path compression*.

- During *find*(*i*), as we traverse the path from *i* to *root*, update *parent* entries for all these nodes to the *root*.
- This reduces the heights of all these nodes.
- Pay now, and reap the benefits later!
- Subsequent *find* may do less work.

To understand the statements above, let's see the updated code for *find* below:

```
find (i)
{
        if (parent[i] < 0)
                return i;
        else
                return parent[i] = find(parent[i]);
}
```

We have modified the code in the body of the *find*. This implementation is of recursive nature. *parent[i]* negative means that we have reached the *root* node. Therefore, we are returning i. If this is not the *root* node, then find method is being called recursively. We know that *parent[i]* may be positive or negative. But in this case it will not be negative due to being an index. To understand this recursive call, we elaborate it further with the help of figure. We call the *find* method with argument 1.

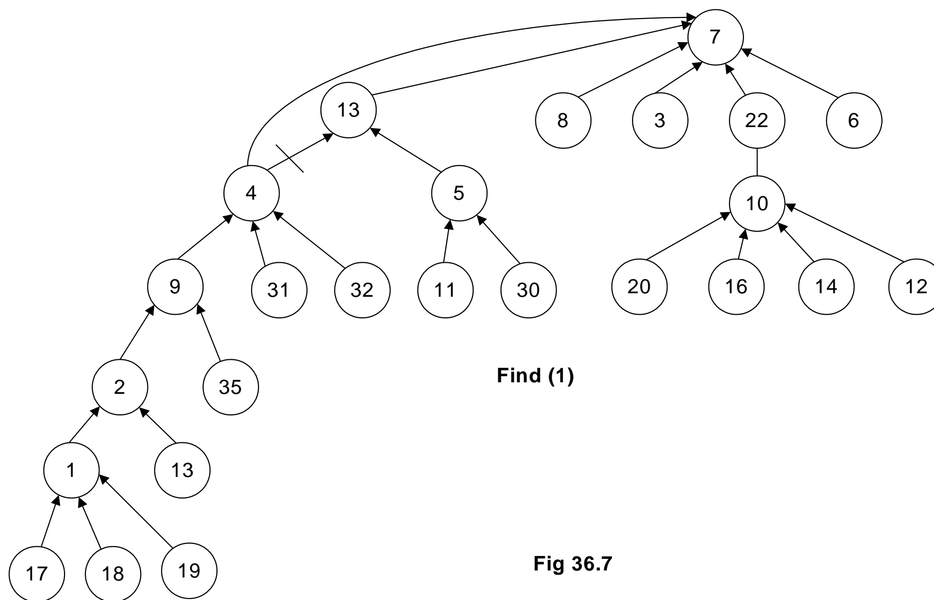


Fig 36.7

This tree is bigger as compared to the ones in our previous examples, it has been constructed after performing some *union* operations on trees. Now we have called *find* method with argument 1. We want to find the set with node 1. By following the previous method of *find*, we will traverse the tree from 1 to 2, from 2 to 9, 9 to 4, 4 to 13 and finally from 13 to 7, which is the *root* node, containing the negative number. So the find will return 7.

If we apply recursive mechanism given in the above code, the recursive call will start from *find(1). The parent* (*find(1))* will return 2. The recursive call for 2 will take the control to node 9, then to 4, 13 and then eventually to 7. When *find(7)* is called, we get the negative number and reach the recursion stopping condition. Afterwards, all the recursive calls, on the stack are executed one by one. The call for 13, *find(13)* returns 7. *find(4)* returns 7 because *find(parent(4))* returns 7. That is why the link between 4 and 13 has been dropped and a new link is established between 4 and 7 as shown in the Fig 36.7. So the idea is to make the traversal path shorter from a node to the *root*. The remaining recursive calls will also return 7 as the result of find operation. Therefore, for subsequent calls we establish the links directly from the node to the *root*.

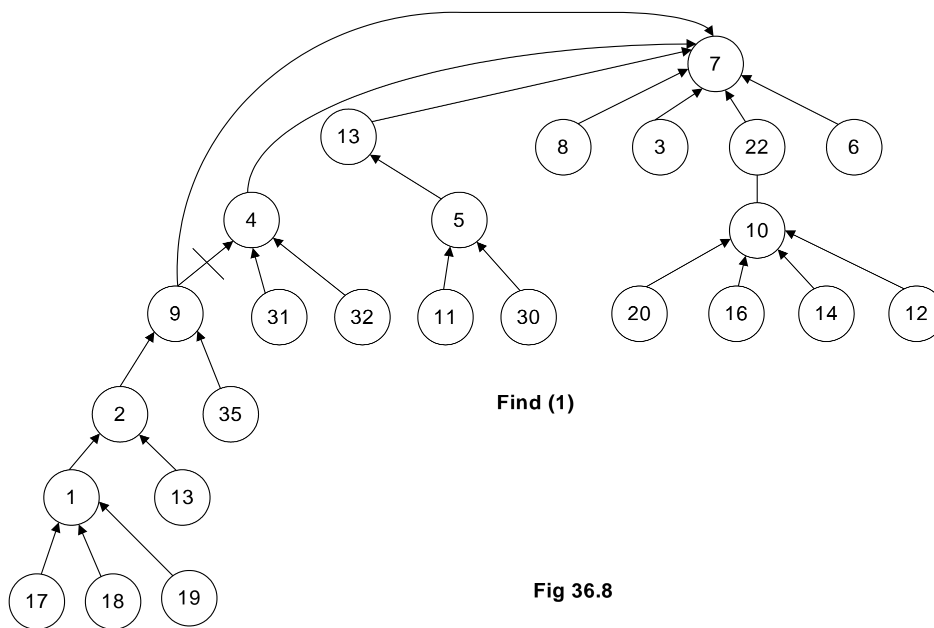


Fig 36.8

Similarly the node 2 is directly connected to the root node and the interlink between the node 2 and 9 is dropped.

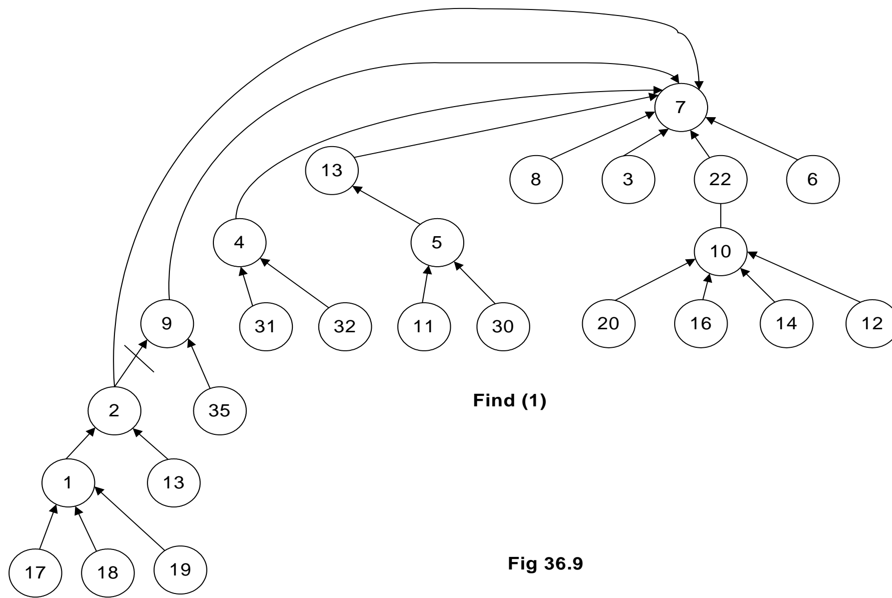


Fig 36.9

The same will happen with 1.

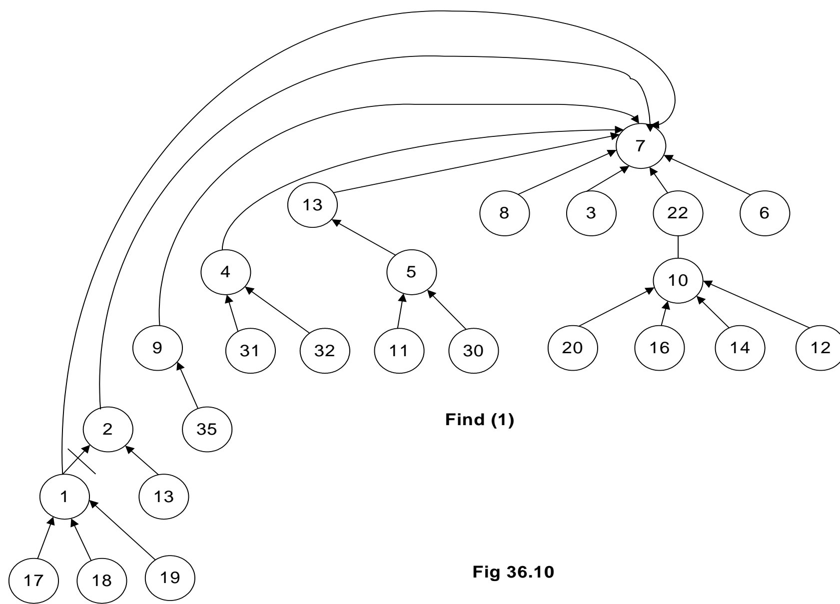


Fig 36.10

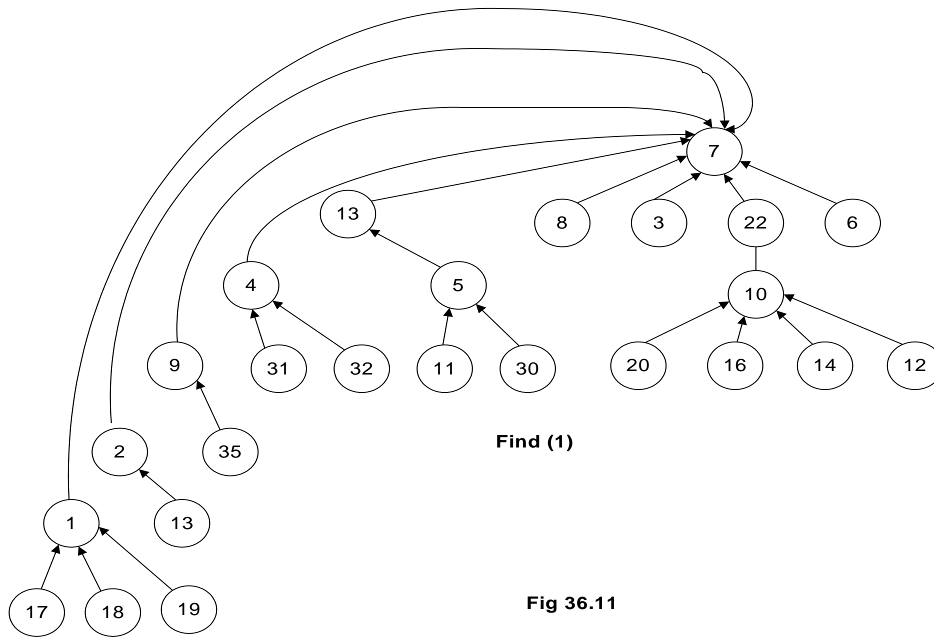


Fig 36.11

The *union* operation is based on size or weight but the reducing the in-between links or path compression from nodes to the *root* is done by the *find* method. The *find* method will connect all the nodes on the path to the *root* node directly.
The *path compression* is depicted by the following tree:

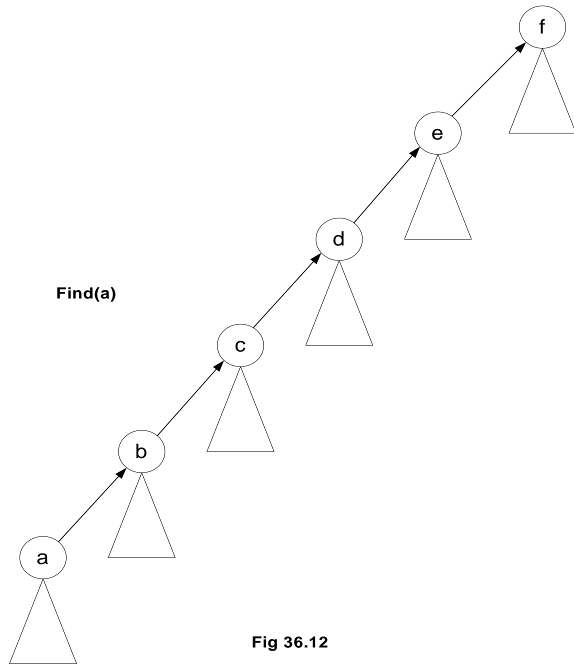


Fig 36.12

If we have called find(a) then see the path from node a to the *root* node f. a is connected to root node f through b, c, d and e nodes. Notice that there may be further subtrees below these nodes. After we apply this logic of *path compression* for *find* operation, we will have the tree as follows:

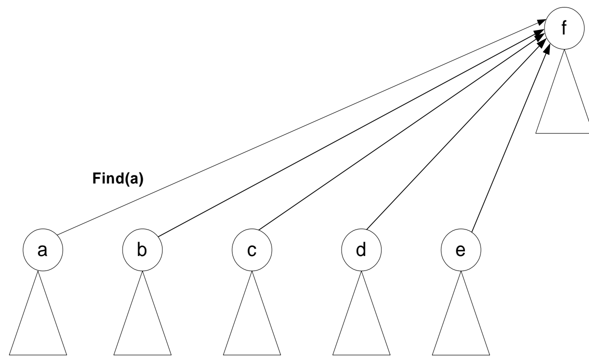


Fig 36.13

We see, how much is the impact on performance in *find* by a theorem.

Timing with Optimization

– Theorem: A sequence of *m union* and *find* operations, *n* of which are *find* operations, can be performed on a disjoint-set forest with union by rank (weight or height) and

path compression in worst case time proportional to $(m\tilde{N}(n))$.
- $\tilde{N}(n)$ is the inverse Ackermann's function which grows extremely slowly. For all practical purposes, $\tilde{N}(n)$ R 4.
- Union-find is essentially proportional to *m* for a sequence of *m* operations, linear in *m*.

There are number of things present in this theorem regarding analysis, which are difficult to cover in this course. We sill study these in course of Algorithm Analysis. At the moment, consider if there are m *union* operations out of which n are *find*s. The average time for union-find will be linear in m and it will grow not quadratically or exponentially. *union* and *find* operations using this data structure are very efficient regarding both space and time.

Data Structures

Lecture No. 37

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 8

Summary

- Review
- Image Segmentation
- Maze Example
- Pseudo Code of the Maze Generation

Review

In the last lecture, we talked about *union* and *find* methods with special reference to the process of optimization in *union*. These methods were demonstrated by reducing size of tree with the help of the techniques-union by size or union by weight. Similarly the tree size was reduced through path optimization in the *find* method. This was due to the fact that we want to reduce the tree traversal for the *find* method. The time required by the *find/union* algorithm is proportional to *m*. If we have *m* union and *n* find, the time required by find is proportional to *m+n*. Union is a constant time operation. It just links two trees whereas in the *find* method tree traversal is involved. The disjoint sets are increased and forest keeps on decreasing. The *find* operation takes more time as unions are increased. But on the average, the time required by *find* is proportional to *m+n*.

Now we will see some more examples of disjoint sets and *union/find* methods to understand their benefits. In the start, it was told that disjoint sets are used in image segmentation.

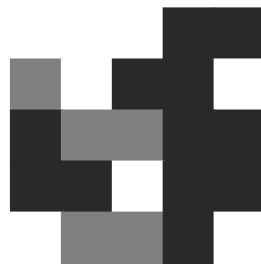
We have seen the picture of a boat in this regard. We have also talked about the medical scanning to find the human organs with the help of image segmentation. Let's see the usage of *union/find* in image segmentation.

Image segmentation is a major part of image processing. An image is a collection of pixels. A value associated with a pixel can be its intensity. We can take images by cameras or digital cameras and transfer it into the computer. In computer, images are represented as numbers. These numbers may represent the gray level of the image. The zero gray level may represent the black and 255 gray level as white. The numbers between 0 and 255 will represent the gray level between black and white. In the color images, we store three colors i.e. RGB (Red, Green, Blue). By combining these three colors, new ones can be obtained.

Image Segmentation

In image segmentation, we will divide the image into different parts. An image may be segmented with regard to the intensity of the pixels. We may have groups of pixels having high intensity and those with pixels of low intensity. These pixels are divided on the basis of their threshold value. The pixels of gray level less than 50 can be combined in one group, followed by the pixels of gray level less between 50 and 100 in another group and so on. The pixels can be grouped on the basis of threshold for difference in intensity of neighbors. There are eight neighbors of the pixel i.e. top, bottom, left, right, top-left, top-right, bottom-left and bottom-right. Now we will see the difference of the threshold of a pixel and its neighbors. Depending on the difference value, we will group the pixels. The pixels can be grouped on the basis of texture (i.e. a pattern of pixel intensities). You will study all these in detail in the image processing subject.

Let's see an example. Consider the diagram below:



It seems a sketch or a square of black, gray and white portions. These small squares represent a pixel or picture element. The color of these picture elements may be white, gray or black. It has five rows and columns each. This is a 5 * 5 image. Now we will have a matrix of five rows and five columns. We will assign the values to the elements in the matrix depending on their color. For white color, we use 0, 4 for black and 2 for gray

color respectively.

	0	1	2	3	4
0	0	0	0	4	4
1	2	0	4	4	0
2	4	2	2	4	4
3	4	4	0	4	4
4	0	2	2	4	0

When we get the image from a digital device, it is stored in numbers. In the above matrix, we have numbers from 0 to 4. These can be between 0 and 255 or may be more depending upon the digital capturing device. Now there is a raw digital image. We will apply some scheme for segmentation. Suppose we need the pixels having value 4 or more and the pixels with values less than 4 separately. After finding such pixels, put 1 for those that have values more than 4 and put 0 for pixels having less than 4 values. We want to make a binary array by applying the threshold. Let's apply this scheme on it.

	0	1	2	3	4
0	0	0	0	1	1
1	0	0	1	1	0
2	1	0	0	1	1
3	1	1	0	1	1
4	0	0	0	1	0

We have applied threshold equal to 4. We have replaced all 4's with 1 and all the values less than 4, have been substituted by zero. Now the image has been converted into a binary form. This may represent that at these points, blood is present or not. The value 4 may represent that there is blood at this point and the value less than 4 represents that there is no blood. It is very easy to a program for this algorithm.

Our objective was to do image segmentation. Suppose we have a robot which captures an image. The image obtained contains 0, 2 and 4 values. To know the values above and below the threshold, it may use our program to get a binary matrix. Similarly, for having the knowledge regarding the region of 1's and region of 0's, the robot can use the algorithm of disjoint sets. In the figure above, you can see that most of the 1's are on the right side of the matrix.

We will visit each row of the matrix. So far, there are 25 sets containing a single element each. Keep in mind the example discussed in *union/find* algorithm. In the zeroth row, we have three 0's. We can apply union on 0's. Yet we are interested in 1's. In the 3rd column, there is an entry of 1. Now the left neighbor of this element is 0 so we do not apply union here and move to the next column. Here, the entry is 1. Its left neighbor is also 1. We apply union here and get a set of two elements. We have traversed the first row completely and moved to the next row. There is 0 in the 0th and 1st column while in

the 2nd column; we have an entry of 1. Its left neighbor is 0 so we move to the next column where entry is also 1. We apply union here. There is a set of 1 at the top of this set also. So we take union of these two sets and combine them. There is a set of four 1's, two from row 0 and two from row 1. In the next row, we have 1, 0, 0, 1, 1. In the row 2 column 3, there is 1. At the top of this element, we have 1. We apply union to this 1 and the previous set of four 1's. Now we have a set of five 1's. In the next column again, there is 1, also included in the set as the sixth element. In the row 3, we have 1 at the 0 column. There is 1 at the top of this so we have set of these two 1's. The next element at the row 3 is again 1, which is included in the left side set of 1's. Now we have another set of 1's having 3 elements. The 1's at column 3 and column 4 are included in the set of six 1's making it a set of eight elements. In the last row, we have single 1 at the column 3 which will be included with the eight element set. So there are two sets of 1's in this matrix.

	0 1 2 3 4
0	0 0 0
1	1 1
2	0 0 1
3	1 0
4	1 0 0
	1 1
	1 1 0
	1 1
	0 0 0
	1 0

We have two disjoint sets and it is clearly visible that where are the entries of 1's and 0's. This is the situation when we have set the threshold of 4 to obtain a binary image. We can change this threshold value.

Let's take the value 2 as threshold value. It means that if the value of pixel is 2 or more than 2, replace it by 1 otherwise 0. The new image will be as:

	0	1	2	3	4
0	0	0	0	1	1
1	1	0	1	1	0
2	1	1	1	1	1
3	1	1	0	1	1
4	0	1	1	1	0

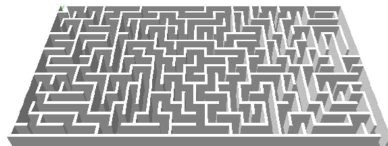
We can apply the union scheme on it to find the region of 1's. Here we have a blob of 1.

	0	1	2	3	4
0	0	0	0	1	1
1	1	0	1	1	0
2	1	1	1	1	1
3	1	1	0	1	1
4	0	1	1	1	0

The region is shaded. The image has been segmented. With the help of *union/find* algorithm, we can very quickly segment the image. The *union/find* algorithm does not require much storage. Initially, we have 25 sets that are stored in an array i.e. the up-tree. We do all the processing in this array without requiring any extra memory. For the image segmentation, disjoint sets and *union-find* algorithm are very useful. In the image analysis course, you will actually apply these on the images.

Maze Example

You have seen the maze game in the newspapers. This is a puzzle game. The user enters from one side and has to find the path to exit. Most of the paths lead to blind alley, forcing the user to go back to square one.



This can be useful in robotics. If the robot is in some room, it finds its path between the different things. If you are told to build mazes to be published in the newspaper, a new maze has to be developed daily. How will you do that? Have you done like this before? Did anyone told you to do like this? Now you have the knowledge of disjoint sets and *union-find* algorithm. We will see that the maze generation is very simple with the help of this algorithm.

Let's take a 5 * 5 grid and generate a 5 * 5 maze. A random maze generator can use *union-find* algorithm. Random means that a new maze should be generated every time.

Consider a 5x5 maze:

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

Here we have 25 cells. Each cell is isolated by walls from the others. These cells are numbered from 0 to 24. The line between 0 and 1 means that there is a wall between them, inhibiting movement from 0 to 1. Similarly there is a wall between 0 and 5. Take the cell 6, it has walls on 4 sides restricting to move from it to anywhere. The internal cells have walls on all sides. We will remove these walls randomly to establish a path from the first cell to the last cell.

This corresponds to an equivalence relation i.e. two cells are equivalent if they can be reached from each other (walls been removed so there is a path from one to the other). If we remove the wall between two cells, then there can be movement from one cell to the other. In other way, we have established an equivalence relationship between them. This can be understood from the daily life example that when the separation wall of two persons is removed, they become related to each other. In maze generation, we will remove the walls, attaching the cells to each other in some sequence. In the end, we will have a path from the start cell to the end cell.

First of all, we have to decide an entrance and an exit. From the entrance, we will move into the maze and have to reach at the exit point. In our example, the entrance is from the cell 0 and the exit will be at cell 24.

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

How can we generate maze? The algorithm is as follows:

- Randomly remove walls until the entrance and exit cells are in the same set.
- Removal of the wall is the same as doing a union operation.
- Do not remove a randomly chosen wall if the cells it separates are already in the same set.

We will take cells randomly. It means that the probability of each cell is equal. After selecting a cell, we will choose one of its surrounding walls. The internal cells have four walls around them. Now we will randomly choose one wall. In this way, we will choose

the neighboring cell. Initially we have 25 sets. Now after removing a wall between 2 cells, we have combined these two cells into one set. Here the union method has been applied to combine these two cells. If these cells are already in the same set, we will do nothing.

We will keep on randomly choosing the cells and removing the walls. The cells will merge together. The elements of the set will keep on growing and at some point, we may have the entrance cell (cell 0) and the exit cell (cell 24) in the same set. When the entrance cell and the exit cell are in the same set, it means that we have a set in which the elements are related to each other and there is no wall between them. In such a situation, we can move from start to the end going through some cells of this set. Now we have at least one path from entrance to exit. There may be other paths in which we have the entrance cell but not the exit. By following this path, you will not reach at the exit as these paths take you to the dead end. As these are disjoint sets, so unless the entrance and exit cell are in the same set, you will not be able to find the solution of the maze.

Pseudo Code of the Maze Generation

Let's take a look at the pseudo code of the maze generation. We will pass it to the size argument to make a maze of this size. We will use the entrance as 0 and exit as *size-1* by default.

```
MakeMaze(int size) {
    entrance = 0; exit = size-1;
    while (find(entrance) != find(exit)) {
        cell1 = randomly chosen cell
        cell2 = randomly chosen adjacent cell
        if (find(cell1) != find(cell2) {
            knock down wall between cells
            union(cell1, cell2)
        }
    }
}
```

After initializing the entrance and exit, we have a while loop. The loop will keep on executing till the time the *find(entrance)* is not equal to *find(exit)*. It means that the loop will continue till the set returned by the *find(entrance)* and *find(exit)* are not same. When the entrance and exit cells are in the same set, the loop will stop. If these cells are not in the same set, we will enter into the loop. At first, we will randomly choose a cell from the available cells e.g. from 0 to 24 from our example. We are not discussing here how to randomly choose a cell in C++. There may be some function available to do this. We store this value in the variable *cell1 and* choose randomly its neighboring cell and store it in *cell2*. This cell may be its top, bottom, left or right cell, if it is internal cell. If the cell is at the top row or top bottom, it will not have four neighbors. In case of being a corner cell, it will have only two neighboring cells. Then we try to combine these two cells into one set. We have an if statement which checks that the set returned by *find(cell1)* is different than the set returned by *find(cell2)*. In this case, we remove the wall between

them. Then we apply union on these two cells to combine them into a set.

We randomly choose cells. By applying union on them, the sets are joined together and form new sets. This loop will continue till we have a set which contains both entrance and exit cell. This is a very small and easy algorithm. If we have the *union-find* methods, this small piece of code can generate the maze. Let's take a pictorial look at different stages of maze generation. You can better understand this algorithm with the help of these figures. We are using the disjoint sets and *union-find* algorithm to solve this problem.

Initially, there are 25 cells. We want to generate a maze from these cells.

→ 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 →

Apply the algorithm on it. We randomly choose a cell. Suppose we get *cell 11*. After this, we randomly choose one of its walls. Suppose we get the right wall. Now we will have *cell 11* in the variable *cell1* and *cell 12* in the variable *cell2*.

→ 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 →

By now, each cell is in its own set. Therefore *find(cell 11)* will return *set_11* and *find(cell 12)* will return *set_12*. The wall between them is removed and union is applied on them. Now we can move from *cell 11* to *cell 12* and vice versa due to the symmetry condition of disjoint sets. We have created a new set (set_11 = {11,12}) that contains *cell 11* and *cell 12* and all other cells are in their own cells.

→ 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

Now we randomly choose the *cell 6* and its bottom wall. Now the *find(cell 6)* will return *set_6*. The *find(cell 11)* will return *set_11* which contains *cell 11* and *cell 12*. The sets returned by the two *find* calls are different so the wall between them is removed and union method applied on them. The set *set_11* now contains three elements *cell 11*, *cell 12* and *cell 6*. These three cells are joined together and we can move into these cells.

Now we randomly select the *cell 8*. This cell is not neighbor of *set_11* elements. We can randomly choose the *cell 6* again and its bottom wall. However, they are already in the same set so we will do nothing in that case. We randomly choose the *cell 8* and its top wall (cell 3). As these two cells are in different sets, the wall between them is removed so that union method could be applied to combine them into a set (set_8 = {8, 3}).

→ 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 →

Now we randomly choose the *cell 14* and its top i.e. *cell 9*. Now we keep on combining cells together but so far entrance and exit cells are not in the same set.

→ 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 →

We randomly choose *cell 0* and its bottom wall so the *cell 0* and *cell 5* are combined together.

→ **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

If you keep on using this algorithm, the walls between cells will keep on vanishing, resulting in the growth of set elements. At some point, the entrance and exit cell will come into one set and there will be a path taking us from the start to exit. There may be some other sets which do not contain the exit cell. Take it as a programming exercise and do it. This is very interesting exercise. Find some information on the internet regarding maze generation. In the next lecture, we will talk about the new data type i.e. table.

Data Structures

Lecture No. 38

Summary

- Table and Dictionaries
- Operations on Table ADT
- Implementation of Table
  - Unsorted Sequential Array
  - Sorted Sequential Array
- Binary Search

We will discuss the concepts of Tables and Dictionaries in this lecture with special reference to the operations on Table ADT and the implementation.

Tables and Dictionaries

The table, an abstract data type, is a collection of rows and columns of information. From rows and columns, we can understand that this is like a two dimensional array. But it is not always a two dimensional array of same data type. Here in a table, the type of information in columns may be different. The data type of first column may be integer while the type of second column is likely to be string. Similarly there may be other different data types in different columns. So the two-dimensional array used for table will have different data types.

A table consists of several columns, known as fields. These fields are some type of information. For example a telephone directory may have three fields i.e. *name*, *address* and *phone number*. On a computer system, the user account may have fields- *user ID*, *password* and *home folder*. Similarly a bank account may have fields like *account number*, *account title*, *account type* and *balance of account*.

Following is a table that contains three fields (We are calling columns as fields). These three fields are *name*, *address* and *phone*. Moreover, there are three rows shown in the table.

Name	Address	Phone
Sohail Aslam	50 Zahoor Elahi Rd, Gulberg-4, Lahore	567-4567
Imran Ahmad	30-T Phase-IV, LCCHS, Lahore	517-2349

Salman Akhtar	131-D Model Town, Lahore	756-5678

Figure 38.1: A table having three fields.

Each row of the table has a set of information in the three fields. The fields of a row are linked to each other. The row of a table is called a record. Thus the above table contains three records. It resembles to the telephone directory. There may be other fields in the phone directory like *father's name* or the *occupation* of the user. Thus the data structure that we form by keeping information in rows and columns is called table.
During the discussion on the concept of table, we have come across the word, dictionary. We are familiar with the language dictionary, used to look for different words and their meanings. But for a programmer, the dictionary is a data type. This data is in the form of sets of fields. These fields comprise data items.
A major use of table is in Databases where we build and use tables for keeping information. Suppose we are developing payroll software for a company. In this case, a table for employees will be prepared. This table of employees has the name, id number, designation and salary etc of the employees. Similarly there may be other tables in this database e.g. to keep the leave record of the employees. There may be a table to keep the personal information of an employee. Then we use SQL (Structured Query Language) to append, select, search and delete information (records) from a table in the database.
Another application of table is in compilers. In this case, symbol tables are used. We will see in compilers course that symbol table is an important structure in a compiler. In a program, each variable declared has a, type and scope. When the compiler compiles a program, it makes a table of all the variables in the program. In this table, the compiler keeps the name, type and scope of variables. There may be other fields in this table like function names and addresses. In the compiler construction course, we will see that symbol table is a core of compiler.
When we put the information in a table, its order will be such that each row is a record of information. The word *tuple* is used for it in databases. Thus a row is called a record or tuple. As we see there is a number of rows (records) in a table. And a record has some fields.

Now here are some things about tables.
To find an *entry* in the table, we only need to know the contents of one of the fields (not all of them). This field is called *key*. Suppose we want to see information (of a record) in dictionary. To find out a record in the dictionary, we usually know the name of the person about whom information is required. Thus by finding the name of the person, we reach a particular record and get all the fields (information) about that person. These fields may be the address and phone number of the person. Thus in telephone directory, the *key* is the *name* field. Similarly in a user account table, the key is usually *user id*. By getting a user id, we can find its information that consists of other fields from the table. The *key* should be unique so that there is only one record for a particular value of *key*. In the databases, it is known as *primary key*. Thus, in a table, a key uniquely identifies an entry. Suppose if the name is the *key* in telephone book then no two entries in the telephone book have the same name. The *key* uniquely identifies the entries. For example if we use the *name* "imran ahmed" in the following table that contains three entries, the name

"imran ahmed" is there only at one place in the following table. In this record, we can also see the address and phone number of "imran ahmed".

Name	Address	Phone
Sohail Aslam	50 Zahoor Elahi Rd, Gulberg-4, Lahore	567-4567
Imran Ahmad	30-T Phase-IV, LCCHS, Lahore	517-2349
Salman Akhtar	131-D Model Town, Lahore	756-5678

Figure 38.2: A table having three fields.

Similarly if we use "salman akhtar" as key we find out one record for this key in the table.
The purpose of this key is to find data. Before finding data, we have to add data. On the addition of the data in the table, the finding of data will be carried through this *key*. We may need to delete some data from the table. For example, in the employees' table, we want to delete the data of an employee who has left the company. To delete the data from the table, the *key* i.e. *name* will be used. However in the employees' table, the *key* may be the *employee id* and not the *name*.

Operations on Table ADT

Now we see the operations (methods) that can be performed with the table abstract data type.

insert

As the name shows this method is used to insert (add) a record in a table. For its execution, a field is designated as *key*. To insert a record (entry) in the table, there is need to know the *key* and the entry. The insert method puts the *key* and the other related fields in a table. Thus we add records in a table.

find

Suppose we have data in the table and want to find some particular information. The find method is given a *key* and it finds the entry associated with the key. In other words, it finds the whole record that has the same *key* value as provided to it. For example, in employees table if *employee id* is the key, we can find the record of an employee whose *employee id* is, say, 15466.

remove

Then there is the remove method that is given a value of the *key to* find and remove the entry associated with that key from the table.

Implementation of Table

Let's talk about why and how we should implement a table. Our choice for implementation of the Table ADT depends on the answers to the following.

- How often entries are inserted, found and removed?
- How many of the possible key values are likely to be used?

- What is the likely pattern of searching for keys? Will most of the accesses be to just one or two key values?
- Is the table small enough to fit into the memory?
- How long will the table exist?

In a table for searching purposes, it is best to store the key and the entry separately (even though the key's value may be inside the entry). Now, suppose we have a record of a person 'Saleem' whose address is '124 Hawkers Lane' while the phone number is '9675846'. Similarly we have another record of a person 'Yunus'. The address and phone fields of this person are '1 Apple crescent' and '622455' respectively. For these records in the table, we will have two parts. One part is the complete entry while the other is the key in which we keep the unique item of the entry. This unique item is twice in a record one as part of the entry and the second in a field i.e. the key field. This key will be used for searching and deletion purposes of records. With the help of key, we reach at the row and can get other fields of it. We also call *TableNode* to row of the table. Its pictorial representation is given below.

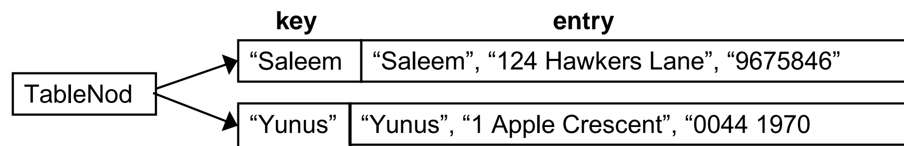


Figure 38.3: key and entry in table

Now we will discuss the implementation of table with different data structures. The first implementation is the unsorted sequential array.

Unsorted Sequential Array

In this implementation, we store the data of table in an array such that TableNodes are stored consecutively in any order. Each element of the row will have a key and entry of the record.

Now let's think how we can store the data of a table in an array. For a telephone directory, there may be a field name i.e. a string. Then there is 'address', which is also a string but of large size as compared to the name. The phone number may be a string or integer. Each record in the directory has these three fields with different types. How can we store this information in an array? Here comes the class. Suppose we make a class *Employee*, with an ultimate aim of developing objects. An object of *Employee* will contain the name, address, designation and salary. Similarly we can make a class of *PhoneDirectoryEntry* that will have name, address and phone number. We can add other fields to these classes. Thus we have *Employee* and *PhoneDirectoryEntry* objects. Now we make an array of objects. In this array, every object is a table entry. To insert a new object (of Employee or PhoneDirectoryEntry), we add it to the back of the array. This insert operation in the array is fast as we directly put the data at the last available

position. Thus we add the data to the array as soon as it is available. In other words, we don't have an order of the data. Now if there are *n* entries in the array, the find operation searches through the keys one at a time and potentially all of the keys to find a particular key. It has to go through all the entries (i.e. n) if the required key is not in the table. Thus the time of searching will be proportional to the number of entries i.e. *n*. Similarly the remove method also requires time proportional to *n*. The remove method, at first, has to find the key, needed to be removed. It consumes the time of find operation that is proportional to the number of entries i.e. *n*. If the entry is found, the remove method removes it. Obviously it does nothing if the entry is not found.
Here we see that in unsorted sequential array, the insertion of data is fast but the find operation is slow and requires much time. Now the question arises if there is any way to keep an array in which the search operation can be fast? The use of sorted sequential array is a solution to this problem.

Sorted Sequential Array

We have studied in the tree section that binary search tree is used to search the information rapidly. Equally is true about the sorted sequential array. But in this case, we want to put the data in an array and not in a tree. Moreover, we want to search the data in this array very fast. To achieve this objective, we keep the data in the array in a sorted form with a particular order. Now suppose we are putting the information in the *telephoneDirectory* table into an array in a sorted form in a particular order. Here, for example, we put the data alphabetically with respect to name. Thus data of a person whose name starts with 'a' will be at the start of the table and the name starting with 'b' will be after all the names that start with 'a'. Then after the names starting with 'b' there will be names starting with 'c' and so on. Suppose a new name starting from 'c' needs to be inserted. We will put the data in the array and sort the array so that this data can be stored at its position with respect to the alphabetic order. Later in this course, we will read about sorting.
Let's talks about the insert, find and remove methods for sorted data.

insert

For the insertion of a new record in the array, we will have to insert it at a position in the array so that the array should be in sorted form after the insertion. We may need to shift the entries that already exist in the array to find the position of the new entry. For example, if a new entry needs to be inserted at the middle of the array, we will have to shift the entries after the middle position downward. Similarly if we have to add an entry at the start of the array, all the entries will be moved in the array to one position right (down). Thus we see that the *insert* operation is proportional to *n* (number of entries in the table). This means *insert* operation will take considerable time.

find

The *find* operation on a sorted array will search out a particular entry in *log n* time by the binary search. The binary search is a searching algorithm. Recall that in the tree, we also find an item in *log n* time. The same is in the case of sorted array.

remove

The *remove* operation is also proportional to *n*. The *remove* operation first finds the entry that takes *log n* time. While removing the data, it has to shuffle (move) the elements in the array to keep the sorted order. This shuffling is proportional to *n*. Suppose, we remove the first element from the array, then all the elements of the array have to be moved one position to left. Thus *remove* method is proportional to *n*.

Binary Search

The binary search is an algorithm of searching, used with the sorted data. As we have sorted elements in the array, binary search method can be employed to find data in the array. The binary search finds an element in the sorted array in *log n* time. If we have 100000 elements in the array, the log 1000000 will be 20 i.e. very small as compared to 100000. Thus binary search is very fast.

The binary search is like looking up a phone number in the directory or looking up a word in the dictionary. For looking a word in the dictionary, we start from the middle in the dictionary. If the word that we are looking for comes before words on the page, it shows that the word should be before this page. So we look in the first half. Otherwise, we search for the word in the second half of the dictionary. Suppose the word is in the first half of the dictionary, we consider first half for looking the word. We have no need to look into the second half of the dictionary. Thus the data to be searched becomes half in a step. Now we divide this portion into two halves and look for the word. Here we again come to know that the word is in the first half or in the second half of this portion. The same step is repeated with the part that contains the required word. Finally, we come to the page where the required word exists. We see that in the binary search, the data to be searched becomes half in each step. And we find the entry very fast. The number of maximum steps needed to find an entry is *log n*, where *n* is the total number of entries. Now if we have 100000 entries, the maximum number of attempts (steps) required to find out the entry will be 20 (i.e. log 1000000).

Data Structures

Lecture No. 39

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 10
10.4.2

Summary

- Searching an Array: Binary Search
- Binary Search - Example 1
- Binary Search - Example 2
- Binary Search - Example 3
- Binary Search – C++ Code
- Binary Search – Binary Tree
- Binary Search - Efficiency
- Implementation 3 (of Table ADT): Linked List

- Implementation 4 (of Table ADT): Skip List
- Skip List - Representation
- Skip List - Higher Level Chains
- Skip List - Formally

Searching an Array: Binary Search

In the previous lecture, we had started discussion on Binary Search Tree algorithm. The discussion revealed that if already sorted data is available, then it is better to apply an algorithm of binary search for finding some item inside instead of searching from start to the end in sequence. The application of this algorithm will help get the results very quickly. We also talked about the example of directory of employees of a company where the names of the employee were sorted. For efficient searching, we constructed the binary search tree for the directory and looked for information about an employee named 'Ahmed Faraz'.
We also covered:

- Binary search is like looking up a phone number or a word in the dictionary
- Start in middle of book
- If the name you're looking for, comes before names on the page, search in the first half
- Otherwise, look into the second half

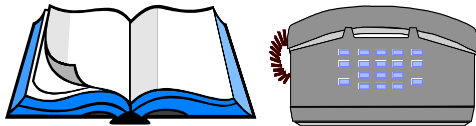


Fig 39.1

The telephone directory is the quotable example to understand the way the binary search method works.

In this lecture, we will focus on data structures for performing search operation. Consider the data is present in an array as we discussed in the previous lecture. For the first implementation, we supposed that the data is not sorted in the array. For second implementation, we considered that the data inside the array is put in sorted array. The advantage of the effort of putting the data in the array in sorted order pays off when the searches on data items are performed.
Now, let's first see the algorithm (in pseudo code) for this operation below. It is important to mention that this algorithm is independent of data type i.e. the data can be of any type numeric or string.

```
if ( value == middle element )
    value is found
else if ( value < middle element )
```

search left half of list with the same method

else

*search right half of list with the same method*The item we are searching for in this algorithm is called *value*. The first comparison of this *value* is made with the *middle element* of the array. If both are equal, it means that we have found our desired search item, which is present in the middle of the array. If this is not the case, then the *value* and the *middle element* are not the same. The *else-if* part of the algorithm is computed, which checks if the *value* is less than the *middle element*. If so, the left half part of the array is searched further in the same fashion (of logically splitting that half part of array into two further halves and applying this algorithm again). This search operation can also be implemented using the recursive algorithm but that will be discussed later. At the moment, we are discussing only the non-recursive algorithm. The last part of the algorithm deals with the case when the *value* is greater than the *middle element.* This processes the right half of the array with the same method.

Let's see this algorithm in action by taking an example of an array of elements:

Binary Search – Example 1

<u>Case 1:</u> ***val == a[mid]***
val = 10
low = 0, high = 8
mid = (0 + 8) / 2 = 4

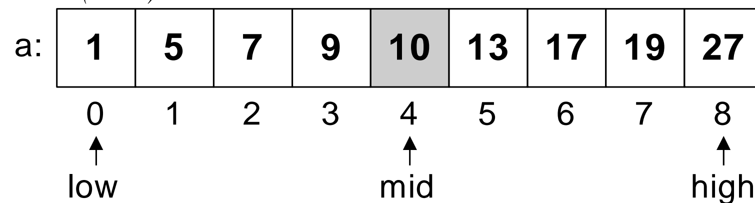


Fig 39.2

You can see an array *a* in the Fig 39.2 with indexes 0 to 8 and values 1, 5, 7, 9, 10, 13, 17, 19 and 27. These values are our data items. Notice that these are sorted in ascending (increasing) order.
You can see in the first line of case 1 that *val = 10*, which indicates that we are searching for value 10. From second line, the range of data to search is from 0 to 8. In this data range, the middle position is calculated by using a simple formula *(low + high)/2*. In this case, it is *mid =(0+8)/2=4*. This is the middle position of the data array. See the array in the above figure Fig 39.2, which shows that the item at array position 4 is 10, exactly the value we are searching for. So, in this case, we have found the value right away in the middle position of the array. The search operation can stop here and an appropriate value can be returned back.

Let's see the case 2 now:

Binary Search – Example 2

Case 2: ***val > a[mid]***
val = 19
low = 0, high = 8
mid = (0 + 8) / 2 = 4
new low = mid + 1 = 5

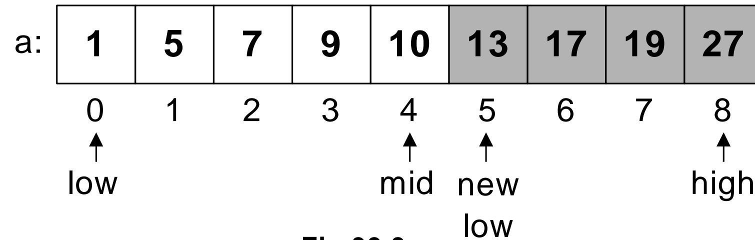


Fig 39.3

The second case is about the scenario when value (*val*) is greater than the middle value (*a[mid]*). The range of data items (*low* and *high*) is the same as that in case 1. Therefore, the middle position (*mid*) is also the same. But the value (*val*) 19 is greater than the value at the middle (mid) 10 of the array. As this array is sorted, therefore, the left half of the array must not contain value 19. At this point of time, our information about val 19 is that it is greater than the middle. So it might be present in the right half of the array. The right half part starts from position 5 to position 8. It is shown in Fig 39.3 that the *new low* is at position 5. With these new low and high positions, the algorithm is applied to this right half again.

Now, we are left with one more case.

Binary Search – Example 3

Case 3: ***val < a[mid]***
val = 7
low = 0, high = 8
mid = (0 + 8) / 2 = 4
new high = mid - 1 =5

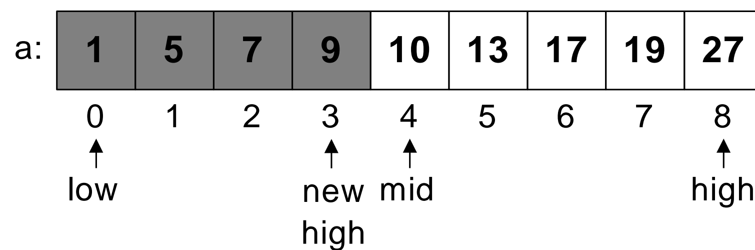


Fig 39.4

The value to be searched (*val*) in this case is 7. The data range is the same starting from *low=0* to *high=8*. Middle is computed in the same manner and the value at the middle position (*mid*) is compared with the *val*. The *val* is less than the value at *mid* position. As

the data is sorted, therefore, a value lesser than the one at mid position should be present in the lower half (left half) of the array (if it is there). The left half of the array will start from the same starting position low=0 but the *high* position is going to be changed to *mid-1 i.e. 3*. Now, let's execute this algorithm again on this left half.

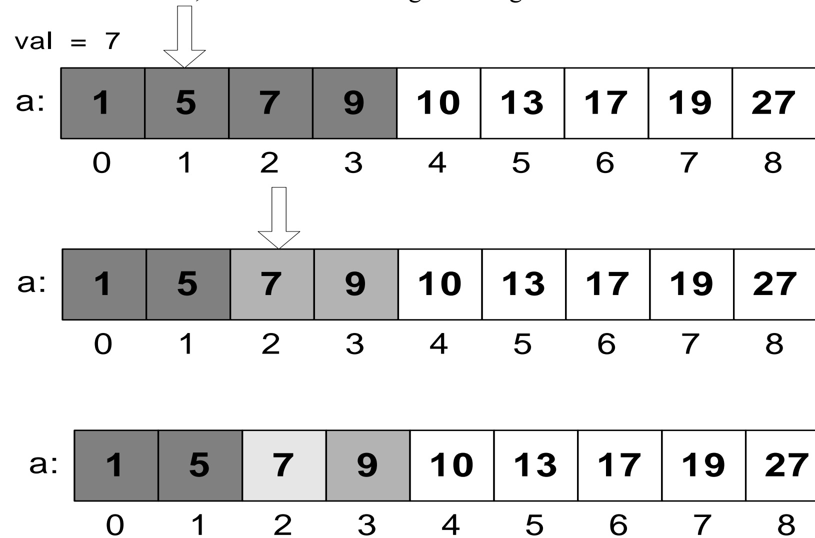


Fig 39.5

Firstly, we will compute the middle of 0 and 3 that results in 1 in this integer division. This is shown in the top array in Fig 39.5. Now, the *val* 7 is compared with the value at the middle position (mid) i.e.5. As 7 is greater than 5, we will process the right half of this data range (which is positioned from *low=0* to *high=3*). The right half of this data range starts from position 2 and ends at position 3. The new data range is *low=2* and *high=3*. The middle position (*mid*) is computed as (2+3)/2=2. The value at the *mid* position is 7. We compare the value at *mid* position (7) to the *val* we are looking for. These are found to be equal and finally we have the desired value.
Our desired number is found within positions- 0 to 8 at position 2. Without applying this binary search algorithm, we might have performed lot more comparisons. You might feel that finding this number 7 sequentially is easier as it is found at position 2 only. But what will happen in case we are searching for number 27. In that case, we have to compare with each element present in the array to find out the desired number. On the other hand, if this number 27 is searched with the help of the binary search algorithm, it is found in third comparison.
Actually, we have already seen binary search while studying the binary search tree. While comparing the number with *the root* element of the tree, we had come to know that if the number was found smaller than the number in the *root*, we had to switch to left-subtree of the *root* (ensuring that it cannot be found in the right subtree).

Now, let's see the C++ code for this algorithm:

Binary Search – C++ Code

```
int isPresent(int *arr, int val, int N)
{
  int low = 0;
  int high = N - 1;
  int mid;
  while ( low <= high )
  {
        mid = ( low + high )/2;
        if (arr[mid] == val)
          return 1;     // found!
        else if (arr[mid] < val)
          low = mid + 1;
        else
          high = mid - 1;
  }
  return 0; // not found
```

The name of the routine is *isPresent*, which expects an *int* pointer (an array in actual); an *int* value *val* is required to be searched. Another int value is N that indicates the maximum index value of the array. Inside the body of the function, *int* variables *low*, *high* and *mid* are declared. *low* is initialized to 0 and high is initialized to *N-1*. *while* loop is based on the condition that executes the loop until *low <= high*. Inside the loop, very first thing is the calculation of the middle position (*mid*). Then comes the first check inside the loop, it compares the *val* (the value being searched) with the number at middle position (*arr[mid]*). If they are equal, the function returns 1. If this condition returns false, it means that the numbers are unequal. Then comes the turn of another condition. This condition (*arr[mid] < val*) is checking if the value at the middle is less than the value being searched. If this is so, the right half of the tree is selected by changing the position of the variable *low* to *mid+1* and processed through the loop again. If both of these conditions return false, the left half of the array is selected by changing the variable *high* to *mid-1*. This left half is processed through the loop again. If the loop terminates and the required value is not found, then 0 is returned as shown in the last statement of this function.
You change this function to return the position of the value if it is found in the array otherwise return –1 to inform about the failure. It is important to note that this function requires the data to be sorted to work properly. Otherwise, it will fail.

This algorithm is depicted figurative in Fig 39.6.

Binary Search – Binary Tree

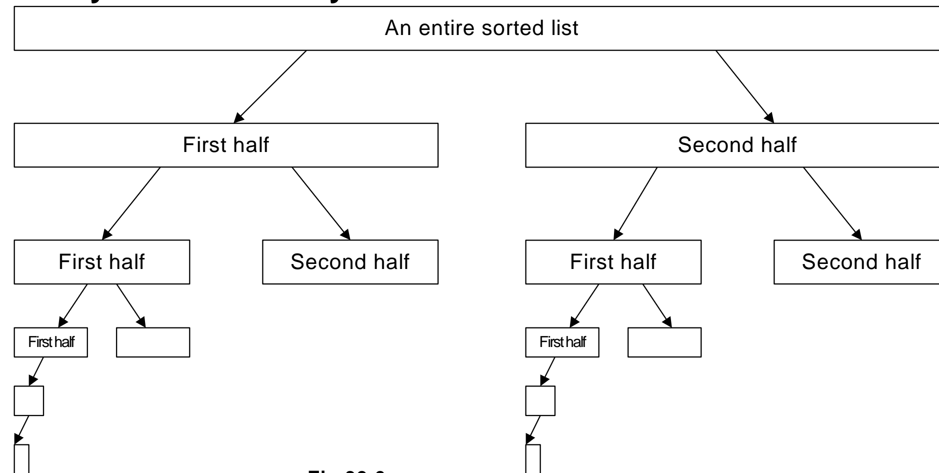


Fig 39.6

– The search divides a list into two small sub-lists till a sub-list is no more divisible.
You might have realized about the good performance of binary trees by just looking at these if you remember the fully balanced trees of N items discussed earlier.

Binary Search - Efficiency

To see the efficiency of this binary search algorithm, consider when we divide the array of N items into two halves first time.

After 1 bisection $N/2$ *items*
After 2 bisections $N/4 = N/2^2$ *items*
. . .
After i bisections $N/2^i = 1$ *item*
____
$i = \log_2 N$

First half contains N/2 items while the second half also contains around N/2 items. After one of the halves is divided further, each half contains around N/4 elements. At this point, only one of N/4 items half is processed to search further. If we carry out three bisections, each half will contain N/8 items. Similarly for i bisections, we are left with $N/2^i$, which is at one point of time is only one element of the array. So we have the equation here:

$$N/2^i = 1$$

Computing the value of i from this gives us:

$$i = \log_2 N$$

Which shows that after maximum $\log_2 N$ bisections, either you will be successful in finding your item or fail to do so.

This was our second implementation of table or dictionary abstract data type using sorted sequential array. As discussed at start of it that if we implement table or dictionary abstract data type using an array, we have to keep the array elements in sorted order. An easier way to sort them can be that whenever we want to *insert* an element in the array, firstly we find its position (in sorted order) in the array and then shift the right side (of that position) elements one position towards right to insert it. In worst case, we might have to shift all the elements one position towards right just to keep the data sorted so this will be proportional to N. Similarly the *remove* operation is also proportional to N. But after keeping the data sorted, the search operation is returned within maximum $log_2 N$ bisections.

Implementation 3 (of Table ADT): Linked List

We might also use linked list to implement the table abstract data type. We can keep the data unsorted and keep on inserting the new coming elements to *front* in the list. It is also possible to keep the data in sorted order. For that, to insert a new element in the list, as we did for array, we will first find its position (in sorted order) in the list and then insert it there. The search operation cannot work in the same fashion here because binary search works only for arrays. Because the linked list may not be contiguous in memory, normally its nodes are scattered through, therefore, binary search cannot work with them.

– *TableNodes are again stored consecutively (unsorted or sorted)*
– ***insert****: add to front; (1or n for a sorted list)*
– ***find****: search through potentially all the keys, one at a time; (n for unsorted or for a sorted list*
– ***remove****: find, remove using pointer alterations; (n)*

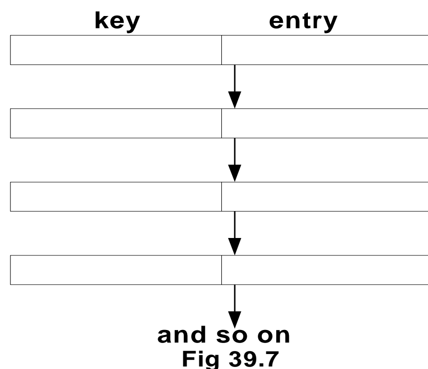


Fig 39.7

Well, linked list is one choice to implement table abstract data type. For unsorted elements, the insertion at *front* operation will take constant time. (as each element is inserted in one go). But if the data inside the list is kept in sorted order then to insert a new element in the list, the entire linked list is traversed through to find the appropriate position for the element.

For *find* operation, all the keys are scanned through whether they are sorted or unsorted. That means the time of *find* is proportional to N.

For *remove* operation, we have to perform *find* first. After that the element is removed and the links are readjusted accordingly.

We know that when we used sorted array, the *find* operation was optimized. Let's compare the usage of array and linked list for table abstract data type. The fixed size of the array becomes a constraint that it cannot contain elements more than that. Linked list has no such constraint but the *find* operation using linked list becomes slower. Is it possible to speed up this operation of *find* while using linked list? For this, a professor of University of Maryland introduced a new data structure called skip list. Let's discuss little bit about *skip list*.

Implementation 4 (of Table ADT): Skip List

- Overcome basic limitations of previous lists
    - Search and update require linear time
- Fast Searching of Sorted Chain
- Provide alternative to BST (binary search trees) and related tree structures. Balancing can be expensive.
- Relatively recent data structure: Bill Pugh proposed it in 1990.

Important characteristics of skip list are stated above. Now, we see skip list in bit more detail.

Skip List - Representation

Can do better than n comparisons to find
element in chain of length *n*

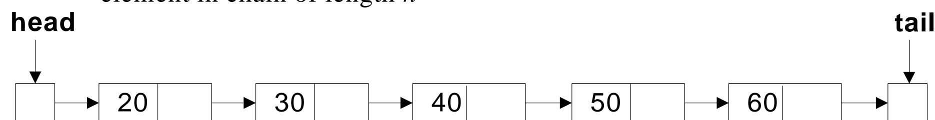


Fig 39.8

As shown in the figure. The *head* and *tail* are special nodes at start and end of the list respectively. If we have to find number 60 in the list then we have no other choice but starting from *head* traverse the subsequent nodes using the *next* pointer until the required node is found or the *tail* is reached. To find 70 in the list, we will scan through the whole list and then get to know that it is not present in it. The professor Pugh suggested something here:

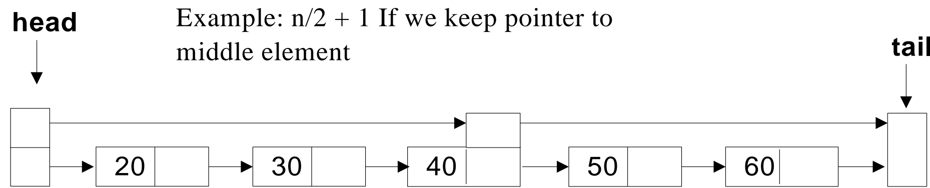


Fig 39.9

Firstly, we use two pointers *head* and *tail*. Secondly, the node in the middle has two *next* pointers; one is the old linked list pointer leading to next node 50 and the second is leading to the *tail* node. Additionally the *head* node also has two pointers, one is the old liked list pointer pointing to the next node 20 and second one is pointing to the middle element's *next* pointer, which is (as told above) further pointing to the *tail* node.
Now, if we want to find element 60 in the above list. Is it possible to search the list in relatively quick manner than the normal linked list shown in Fig 39.8? Yes, it is with the help of the additional pointers we have placed in the list. We will come to the middle of the list first and see that the middle element (40) is smaller than 60, therefore the right half part of the list is selected to process further (as the linked list is sorted). Isn't it the same we did in binary search? It definitely is.

What if we can add additional pointers (links) and boost the performance. See the figure below.

Skip List - Higher Level Chains

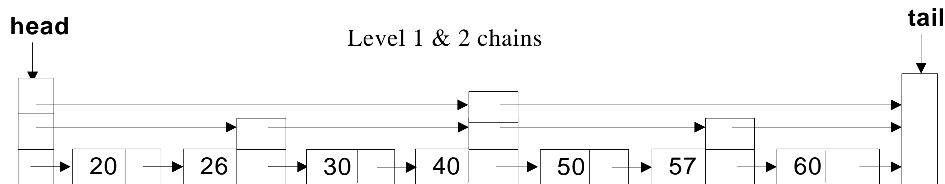


Fig 39.10

– For general n, level 0 chain includes all elements
– level 1 every other element, level 2 chain every fourth, etc.
– level *i*, every 2^i th elementLevel 0 chain is our old linked list chain as shown in Fig 39. Level 1 is new chain added to contain the link of every other node (or alternate node). Level 2 chain contains links to every 4th node. We keep on adding levels of chains so that we generalize that for level i chain includes 2^ith elements. After adding these pointers, the skip list is no more our old linked list; it has become sort of binary tree. Still, there are still few important things to consider.

– *Skip list contains a hierarchy of chains*

– *In general level i contains a subset of elements in level i-1*Skip list becomes a hierarchy of chains and every level contains a subset of element of previous level. Using this kind of skip list data structure, we can find elements in $\log_2 n$ time. But the problem with this is that the frequency of pointers is so high as compared to the size of the data items that it becomes difficult to manage them. The *insert* and *remove* operations on this kind of skip list become very complex because single insertion or removal requires lot of pointers to readjust.
Professor Pugh suggested here that instead of doing leveling in powers of 2, it should be done randomly. Randomness in skip lists is a new topic for us. Let's see a formal definition of skip list.

Skip List - Formally

– A skip list for a set S of distinct (key, element) items is a series of lists S_0, S_1 , … , S_h such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in non-decreasing order Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$
  - List S_h contains only the two special keys

You are advised to study skip list from your text books. The idea of randomness is new to us. We will study in the next lecture, how easy and useful becomes the skip list data structure after employing randomness.

Data Structures

Lecture No. 40

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 10

10.4.2

Summary

- Skip List
- Skip List Search
- Insertion in Skip List
- Deletion from Skip List

In the previous lecture, we had started the discussion on the concept of the skip lists. We came across a number of definitions and saw how the use of additional pointers was effective in the list structures. It was evident from the discussion that a programmer prefers to keep the data in the linked list sorted. If the data is not sorted, we cannot use

binary search algorithm. And the insert, find and remove methods are proportional to n. But in this case, we want to use the binary search on linked list. For this purpose, skip list can be used. In the skip list, there is no condition of the upper limit of the array.

Skip List

A skip list for a set S of distinct (key, element) items is a series of lists S_0, S_1 , … , S_h such that

- Each list Si contains the special keys +∞ and -∞
- List S_0 contains the keys of S in non-decreasing order
- Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$
- List S_h contains only the two special keys

Now let's see an example for the skip list. First of all, we have S0 i.e. a linked list. We did not show the arrows in the list in the figure. The first and last nodes of S0 contain -∞ and +∞ respectively. In the computer, we can put these values by –max(int) and max(int). The values can also be used about which we are sure that these will not be in the data items. However, for the sake of discussion to show these values, the -∞ and +∞ are the best notations. We can see that -∞ is less than any value of data item while +∞ is greater than any value of data item. If we insert any value much ever, it is large the +∞ will be greater than it. Moreover, we see that the numbers in S0 are in the non- decreasing order. This S0 is the first list in which all keys are present.
Now we will take some nodes from this list and link them. That will be not every other node or every fourth node. It may happen this way. However, we will try that the node should not be every other or fourth node. It will be a random selection. Now we see S1 i.e. a subset of S0. In S1 we selected the nodes 23, 31, 34 and 64. We have chosen these nodes randomly with out any order or preference. Now from this S1, we make S2 by selecting some elements of S1. Here we select only one node i.e. 31 for the list S2. The additional pointer, here, has to move from -∞ to 31 and from 31 to +∞. Now the next list i.e. S3 will be subset of S2. As there is only one node in S2, so in S3, there will only the special keys. In these lists, we use pointers to link all the nodes in S0. Then with additional pointers, we linked these nodes additionally in the other lists. Unlike 2i, there is not every other node in S1 or every fourth node in S2. The following figure represents these lists.

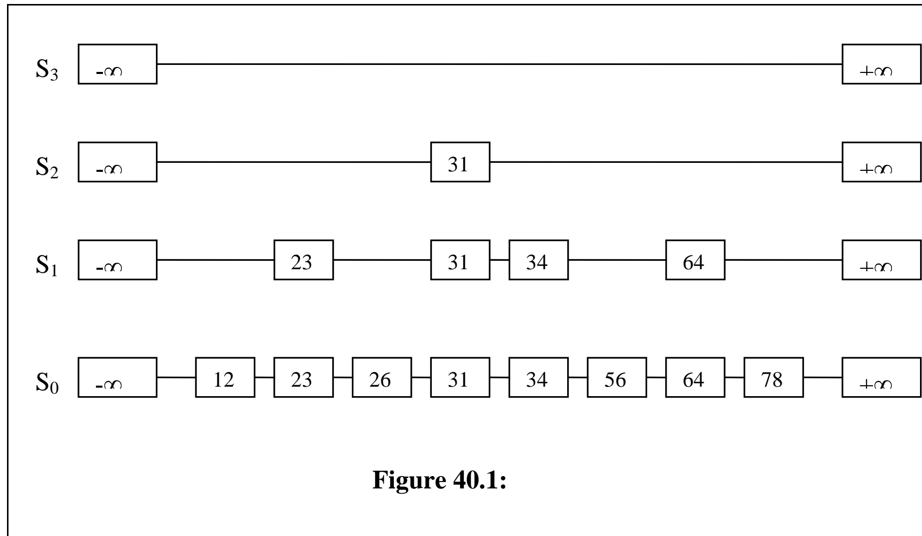


Figure 40.1:

Now we have the list i.e. from S0 to S3. Actually, these list are made during insert operation. We will see the insert method later. Let's first talk about the search method.

Skip List Search

Suppose we have a skip list structure available. Let's see what is its benefit. We started our discussion from the point that we want to keep linked list structure but do not want to search n elements for finding an item. We want to implement the algorithm like binary search on the linked list.

Now a skip list with additional pointers is available. Let's see how search will work with these additional pointers. Suppose we want to search an item x. Then search operation can be described as under.

We search for a key *x* in the following fashion:

- We start at the first position of the top list
- At the current position *p*, we compare *x* with *y* ← *key*(*after*(*p*))
- *x* = *y*: we return *element*(*after*(*p*))
- *x* > *y*: we "scan forward"
- *x* < *y*: we "drop down"
- If we try to drop down past the bottom list, we return *NO_SUCH_KEY*

To search a key *x,* we start at the first position of the top list. For example, we are discussing the top list is S3. We note the current position with *p*. We get the key in the list after the current list (i.e. in which we are currently looking for key) by the *key (after(p))* function. We get this key as *y*. Then we compare the key to be searched for i.e. *x* with this *y*. If *x* is equal to *y* then it means *y* is the element that we are searching for so

we return *element(after(p))*. If *x* is greater than *y,* we scan forward and look at the next node. If *x* is less than *y,* we drop down and look in the down lists. No if we drop down and past the bottom list, it means that the element (item) is not there and we return *NO_SUCH_KEY*.
To see the working of this search strategy let's apply it on the skip list, already discussed and shown in the figure 40.1. This figure shows four lists. Remember that these four lists are actually in the one skip list and are made by the additional pointers in the same skip list. There is not such situation thatS1 is developed by extracting the data from S0 and S1 duplicates this data. Actually every node exists once and is pointed by additional pointers. For example, the node 23 exists once but has two next pointers. One is pointing to 26 while the other pointing to 31. In the same way, there are three pointers in node 31, two are to the node 34 and the third is toward the +∞.

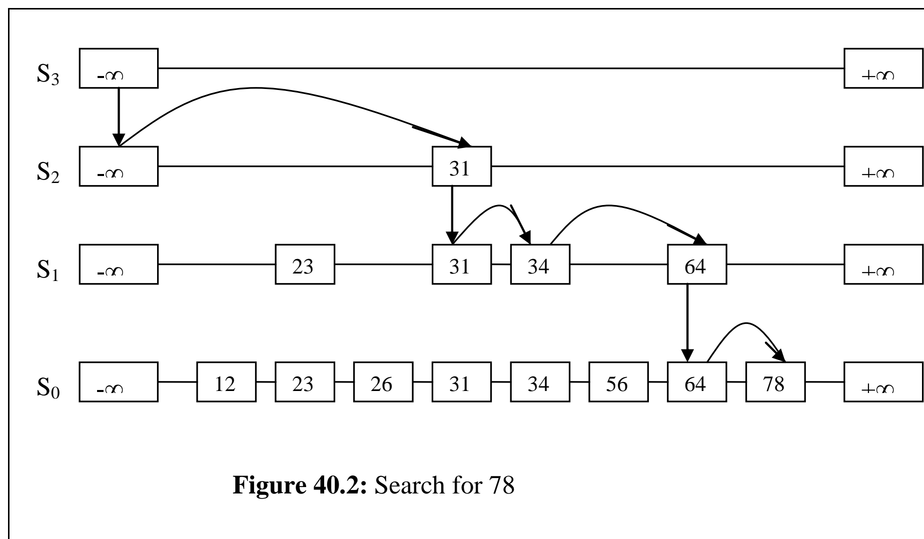


Figure 40.2: Search for 78

Suppose we want to search 78. We start from the first node of the top list i.e. S3. The 78 will be the current node and we denote it with *p*. Now we look at the value in the node after *p*. In the figure, it is +∞. Now as the +∞ is greater than 78, we drop down to S2. Note in the figure that we drop down vertically. We don't go to the first element p in the down list. This process of going down will be discussed later. Now we drop from S3 to S2. This is our current pointer. Now we look for the value in the next node to it. We see that this value is 31. Now 31 is less than 78, so we will do scan forward. The next node is +∞ that is obviously greater than 78. So we drop from here and go to the list S1. In this list, the current position is 34. We compare 78 with this node. As 34 is less than 78, we scan forward in the list. The next node in the list is 64 that is also less than 78. So we look at the next node and note that the next node is +∞ that is greater than 78. Due to this we drop down to list S0. Here we look at the next node of 64 and find that this is 78. Thus at last we reach at the node that we are searching for. Now if we look at the arrows that are actually the steps to find out the value 78 in the skip list. Then we come to know

that these are much less than the links that we have to follow while starting from the -+∞ in the list S0. In S0 we have to traverse the nodes 12, 23, 26, 31, 34, 44, 56, and 64 to reach at 78. This traversal could be larger if there were more nodes before 31. But in our algorithm, we reach at node 31 in one step. Thus we see that the search in the skip list is faster than the search in a linear list. By the steps that we do in searching an element and see that this search is like the binary search scheme. And we will see that this skip list search is also log2 N as in binary search tree. Though the skip list is not a tree, yet its find method works like the binary search in trees. As we know that find method is also involved in the remove method, so remove method also becomes fast as the find method is fast.

Insertion in Skip List

When we are going to insert (add) an item (*x,0*) into a skip list, we use a randomized algorithm. Note that here we are sending the item in a pair. This is due to the fact that we keep the data and the key separately to apply the find and remove methods on table easily. In this pair, *x* is the key, also present in the record. The *0* denotes the data (i.e. whole record). Now the randomized algorithm for insertion of this value is described below.

The first step of the algorithm is that

- We repeatedly toss a coin until we get tails, and we denote with ***i*** the number of times the coin came up heads.

This first step describes that we toss a coin while knowing a 50 percent probability for both head and tail. We keep a counter denoted with *i* to count the heads that come up. Now if the head comes up, *i* becomes 1. We again toss the coin if again the head comes up we add 1 to the counter i. We continue this counting until the tail comes up. When tail comes up, we stop tossing and note the value of *i*.

After this, the second step of algorithm comes, stating that

- If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys

Here we compare *i* (that is the count of heads came up) with *h* (that is the number of list) if *i* is greater than or equal to *h* then we add new lists to the skip list. These new lists are $S_{h+1}, \ldots, S_{i+1}$. Suppose if *i* is 8 and *h* is 4 , we add additional lists S5, S6, S7, S8 and S9. These lists initially will contain the only two special keys that means -∝ and +∝. The next steps are:

- We search for x in the skip list and find the positions p_0, p_1, …, p_i of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
- For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j

Now we will start from the left most node of the top list i.e. Si and will find the position for the new data item. We find this position as we want to keep the data in the list sorted. Let's see this insert position with the help of figures. Suppose we have a skip list , shown

in the figure below. Here are only three items 10, 23 and 36 in it. There are also additional layers S1 and S2. These were made when 10, 23 and 36 were inserted.

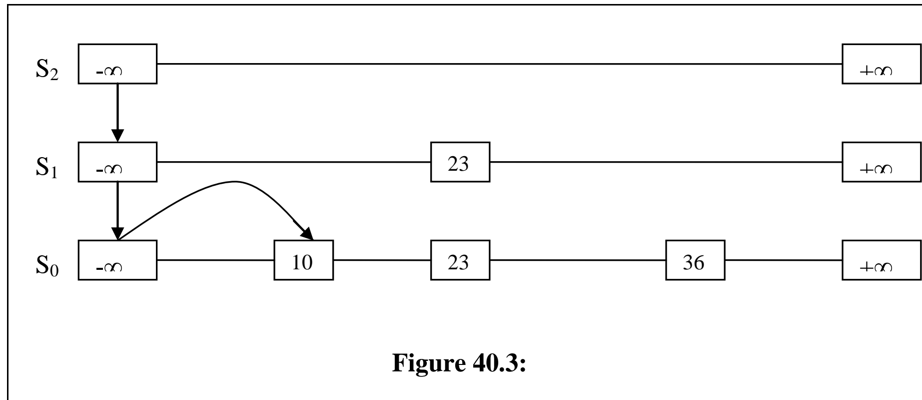


Figure 40.3:

Now we proceed with this list and insert a value in it. The value that we are going to insert is 15. We have tossed the coin and figured out that the value of i is 2. As it is randomized algorithm so in this case, the value of i has become 2. The value of I is the count number of heads before the tail comes up by tossing the coin. From the above figure, we see that the value of h is 2. As h and i are equal, we are not adding S3, S4 and S5 to the list. Rather, we will apply the search algorithm. We start from the left most node of the top list. We call this position p2. We label these positions for their identification. We see that the item being added i.e.15 is less than the +∝, so we drop down to list S1. We denote this step with p1. In this list, the next value is 23 that is also greater than 15. So we again drop down and come in the list S0. Our current position is still the first node in the list as we did not have any scan forward. Now the next node is 10 that is less than 15. So we skip forward. We note this skip forward with p0. Now after p0 the value in next node is 23 that is greater than 15. As we are in the bottom list, there is no more drop down. So here is the position of the new node. This position of new node is after the node 10 and before the node 23. We have labeled the positions po, p1 and p2 to reach there. Now we add the value 15 additionally to the list that we have traversed to find the positions and labeled them to remember. After this we add a list S3 that contains only two keys that are -∝ and +∝, as we have to keep such a list at the top of the skip list. The following figure shows this insertion process.

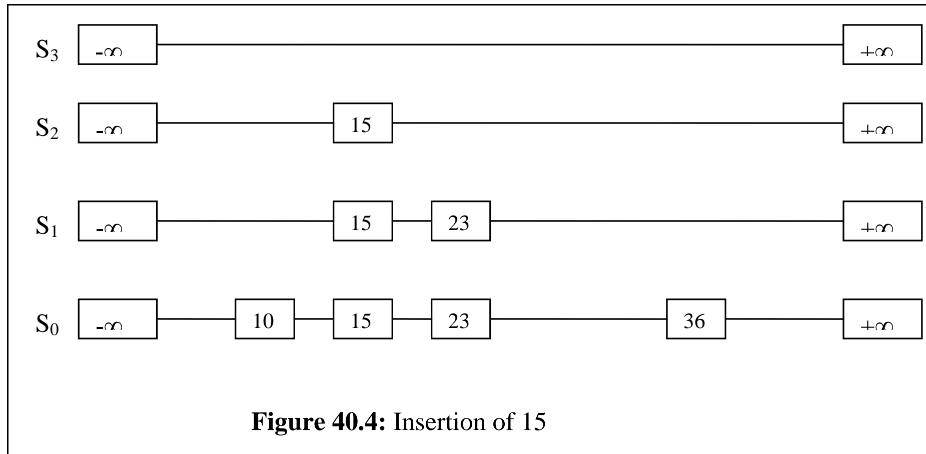


Figure 40.4: Insertion of 15

Here we see that the new inserted node is in three lists. The value that we insert must will be in S0 as it is the data list in which all the data will reside. However, it is also present in the lists that we have traversed to find its position. The value of i in the insertion method is randomly calculated by tossing the coin. How can we toss a coin in computers? There is routine library available in C and C++ that generates a random number. We can give it a range of numbers to generate a number in between them. We can ask it to give us only 0 and 1 and assign 1 to head and 0 to tail respectively. Thus the count number of 1's will give us the number of heads that come up. We can also use some other range like we can say that if the random number is less than some fixed value (whatever we fixed) that it means head otherwise it will mean tail. The algorithms that use random numbers are generally known as randomized algorithms.

So

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- It contains statements of the type

```
b ← random()
if b <= 0.5    // head
        do A …
else           // tail
        do B …
```

- Its running time depends on the outcome of the coin tosses, i.e, head or tail

In the randomized algorithm, we take a number from the random() function between 0 and 1. This number is up to one decimal place. However, we can keep it up to nay decimal places. As stated above ,after getting a number we check its value. If this is less than or equal to 0.5, we consider it as head and execute the process A. In our algorithm, we increase the value of i by 1. However, if value is greater than 0.5, we consider it as tail and do the process B. In our algorithm, the process B takes place when we note the value of I and stop the tossing. We do this process of tossing in a while loop. The while condition comes false when the tail (i.e. number greater than 0.5) comes. We cannot

predict how many times this loop will execute as it depends upon the outcome of the toss. It is also a random number. There may be only one or a large number of heads before the tail comes up.

Deletion from Skip List

In the remove method, we find the item to be removed with the find item and remove it from the list. In the lists where ever this item has links, we bypass them. Thus, the procedure is quite easy. Now let's talk about this method.

To remove an item with key ***x*** from a skip list, we proceed as follows:

We search for ***x*** in the skip list and find the positions ***p***0, ***p***1 , …, ***pi*** of the items with key ***x***, where position ***pj*** is in list ***Sj.*** This means that we look for the links of the item to be removed . We know that an item in the list has necessarily link in the list S0. Moreover it may have links in other lists up to Si or say Sj. After this, we remove positions ***p***0, ***p***1 , …, ***pi*** from the lists ***S***0, ***S***1, … , ***Si.*** We remove all the lists except the list containing only the two special keys.

Let's consider the skip list shown in the figure below.

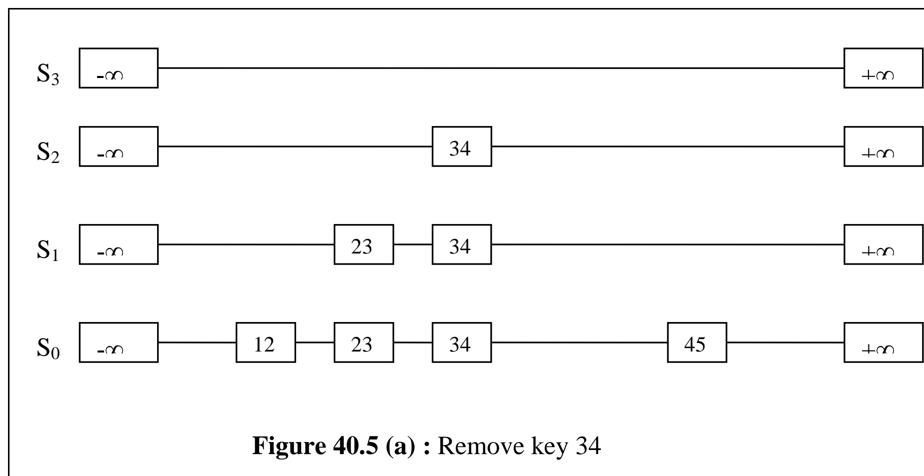


Figure 40.5 (a) : Remove key 34

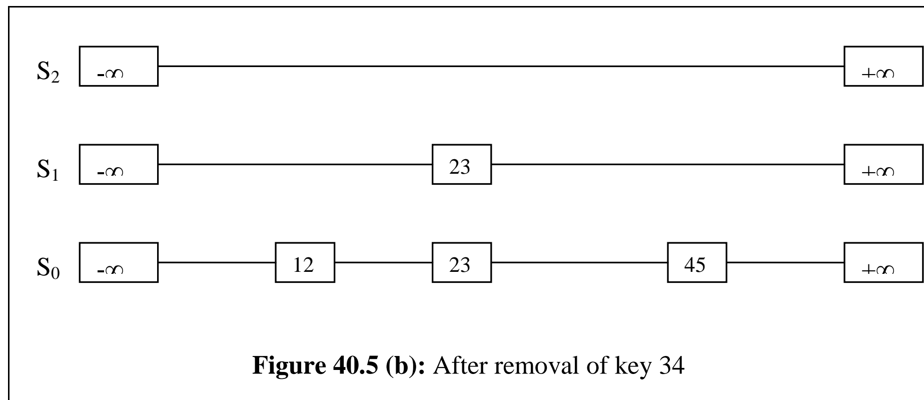


Figure 40.5 (b): After removal of key 34

Suppose we want to remove the node 34 from the list. When this node 34 was inserted, it came not only in list S0 but there were its additional links o in S1 and S2 lists respectively. The list S3 has only the two special keys. Now for finding 34, we start from the top list. There is +∝ as the next node, so we drop down to the list S2 as +∝ is greater than 34. In S2, we are at node 34. Now we are at the point to remove 34. From here, we go to the remaining lists and reach list S0. We also label the links being traversed. It is evident that we have labeled the links as p2, p1 and p0. Now we remove the node 34 and change the pointers in the lists. We see that in S3, this was the single node. After removing this node, there is only one link that is from -∝ to +∝. The list S3 already has link from -∝ to +∝. Instead of keeping these two i.e. S2 and S3, we keep only S2. Thus we have removed the node 34 from the skip list. We see that the remove method is simple. We don't have randomness and need not tossing. The tossing and random number was only in the case of insertion. In the remove method, we have to change some pointers.

Thus in this data structure, we need not to go for rotations and balancing like AVL tree. In this data structure- Skip-list, we get rid of the limitation of arrays. This way, the search gets very fast. Moreover, we have sorted data and can get it in a sorted form.

Data Structures

Lecture No. 41

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 10
10.4.2

Summary

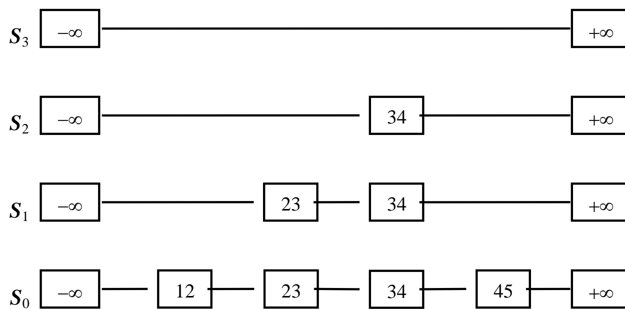
- Review

- Quad Node
- Performance of Skip Lists
- AVL Tree
- Hashing
- Examples of Hashing

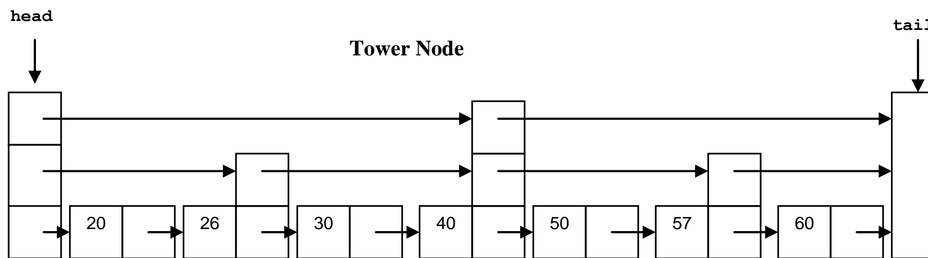
Review

In the previous lecture, we studied three methods of skip list i.e. *insert*, *find* and *remove* and had their pictorial view. Working of these methods was also discussed. With the help of sketches, you must have some idea about the implementation of the extra pointer in the skip list.

Let's discuss its implementation. The skip list is as under:



We have some nodes in this skip list. The data is present at 0, 1st and 2nd levels. The actual values are 12, 23, 34 and 45. The node 34 is present in three nodes. It is not necessary that we want to do the same in implementation. We need a structure with next pointers. Should we copy the data in the same way or not? Let's have a look at the previous example:



Here, the data is 20, 26, 30, 40, 50, 57, 60. At the lowest level, we have a link list. A view of the node 26, node 40 and node 57 reveals that there is an extra next 'pointer'. The head pointer is pointing to a node from where three pointers are pointing at different nodes.

We have seen the implementation of link list. At the time of implementation, there is a data field and a *next* pointer in it. In case of doubly link list, we have a *previous* pointer too. If we add an extra pointer in the node, the above structure can be obtained. It is not necessary that every node contains maximum pointers. For example, in the case of node 26 and node 57, there are two *next* pointers and the node 40 has three next pointers. We will name this node as 'TowerNode'.

TowerNode will have an array of *next* pointers. With the help of this array of pointers, a node can have multiple pointers. Actual number of *next* pointers will be decided by the random procedure. We also need to define *MAXLEVEL* as an upper limit on number of levels in a node. Now we will see when this node is created. A node is created at a time of calling the *insert* method to insert some data in the list. At that occasion, a programmer flips the coin till the time he gets a tail. The number of heads represents the levels. Suppose we want to insert some data and there are heads for six times. Now you know how much next pointers are needed to insert which data. Now we will create a *listNode* from the *TowerNode* factory. We will ask the factory to allocate the place for six *next* pointers dynamically. Keep in mind that the next is an array for which we will allocate the memory dynamically. This is done just due to the fact that we may require different number of *next* pointers at different times. So at the time of creation, the factory will take care of this thing. When we get this node from the factory, it has six next pointers. We will insert the new data item in it. Then in a loop, we will point all these next pointers to next nodes. We have already studied it in the separate lecture on *insert* method.

If your random number generation is not truly so and it gives only the heads. In this case, we may have a very big number of heads and the Tower will be too big, leading to memory allocation problem. Therefore, there is need to impose some upper limit on it. For this purpose, we use *MAXLEVEL*. It is the upper limit for the number of *next* pointers. We may generate some error in our program if this upper limit is crossed.

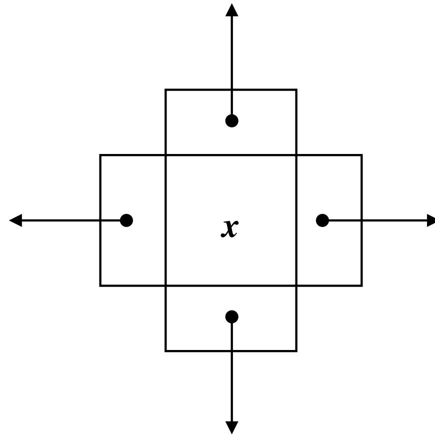
The *next* pointers of a node will point at their own level. Consider the above figure. Suppose we want to insert node 40 in it. Its 0 level pointer is pointing to node 50. The 2nd pointer is pointing to the node 57 while the third pointer pointing to tail. This is the case when we use *TowerNode*. This is one of the solutions of this problem.

Quad Node

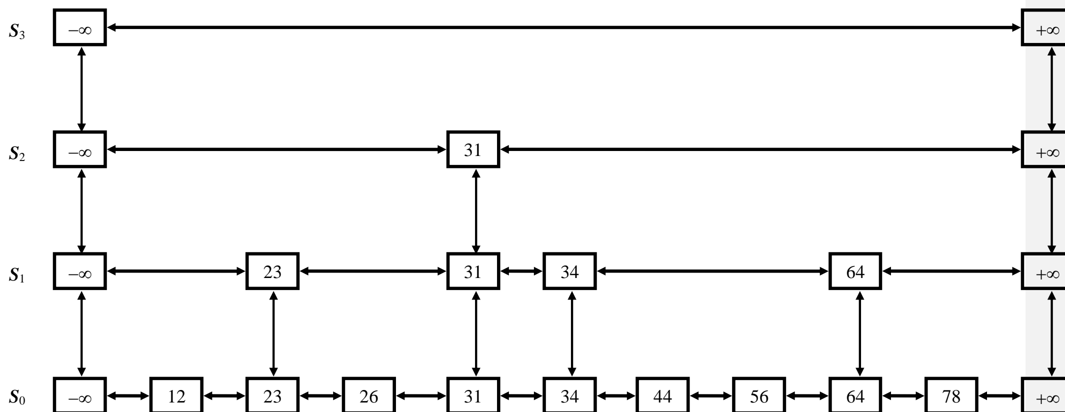
Let's review another method for this solution, called *Quad node*. In this method, we do not have the array of pointers. Rather, there are four *next* pointers. The following details can help us understand it properly.

A quad-node stores:

- item
- link to the node before
- link to the node after
- link to the node below
- link to the node above



This will require copying of the key (item) at different levels. We do not have an array of next pointers in it. So different ways are adopted to create a multilevel node of skip list. While requiring six levels, we will have to create six such nodes and copy the data item *x* in all of these nodes and insert these in link list structure. The following figure depicts it well.



You can see *next* and *previous* and *down* and *up* pointers here. In the bottom layer, the *down* pointer is nil. Similarly the right pointers of right column are nil. In the top layer, the top pointers are nil. You can see that the values 23, 34, and 64 are copied two times and the value 31 is copied three times. What are the advantages of *quad node*? In *quad node*, we need not to allocate the array for next pointers. Every list node contains four pointers. The *quad node* factory will return a node having four pointers in it. It is our responsibility to link the nodes up, bottom, left and right pointers with other nodes. With the help of previous pointer, we can also move backward in the list. This may be called as doubly skip list.

Performance of Skip Lists

Let's analyze this data structure and see how much time is required for search and deletion process. The analysis is probability-based and needs lot of time. We will do this in some other course. Let's discuss about the performance of the skip list regarding insert, find and remove methods.

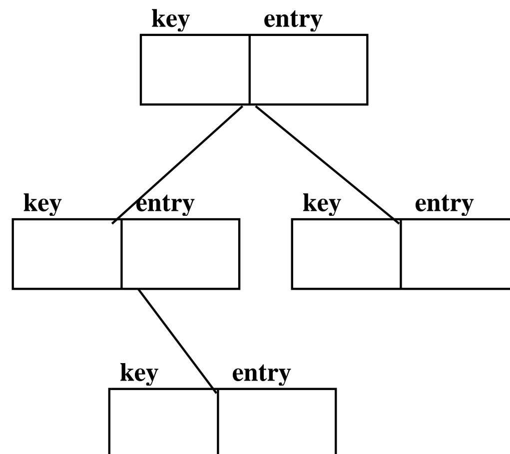
In a skip list, with *n* items the expected space used is proportional to *n*. When we create a skip list, some memory is needed for its items. We also need memory to create a link list at lowest level as well as in other levels. This memory space is proportional to *n* i.e. number of items. For *n* items, we need *n* memory locations. The items may be of integer data type. If we have 100 items, there will be need of 100 memory locations. Then we need space for *next* pointers that necessitates the availability of *2n* memory locations. We have different layers in our structure. We do not make every node as *towerNode* neither we have a formula for this. We randomly select the *towerNode* and their height. The *next* pointers can be up to *maxLevel* but normally these will be few. We do not require pointers at each level. We do not need *20n* or *30n* memory locations. If we combine all these the value will be *15n* to *20n*. Therefore the proportionality constant will be around 15 or 20 but it cant be n^2 or n^3. If this is the case then to store 100 items we do need 100^2 or 100^3 memory locations. There are some algorithms in which we require space in square or cubic times. This is the space requirement and it is sufficient in terms of space requirements. It does not demand too much memory.

Let's see the performance of its methods. The expected search, insertion and deletion time is proportional to *log n*. It looks like binary tree or binary search tree (BST). This structure is proposed while keeping in mind, the binary search tree. You have witnessed that if we have extra nodes, search process can be carried out very fast. We can prove it with the probabilistic analyses of mathematics. We will not do it in this course. This information is available in books and on the internet. All the searches, insertions and deletions are proportional to *log n*. If we have 100,000 nodes, its *log n* will be around 20. We need 20 steps to insert or search or remove an element. In case of insert, we need to search that this element already exists or not. If we allow duplicate entries then a new entry would be inserted after the previous one. In case of delete too, we need to search for the entry before making any deletion. In case of binary search tree, the insertion or deletion is proportional to *log n* when the tree is a balanced tree. This data structure is very efficient. Its implementation is also very simple. As you have already worked with link list, so it will be very easy for you to implement it.

AVL Tree

The insertion, deletion and searches will be performed on the basis of key. In the nodes, we have key and data together. Keep in mind the example of telephone directory or employee directory. In the key, we have the name of the person and the entry contains the address, telephone number and the remaining information. In our AVL tree, we will store this data in the nodes. Though, the search will be on the key, yet as we already noticed that the insert is proportional to *log n*. Being a balanced tree, it will not become degenerated balance tree. The objective of AVL tree is to make the binary trees balanced. Therefore the find will be *log n*. Similarly the time required for the removal of the node is

proportional to *log n*.



We have discussed all the five implementations. In some implementations, time required is proportional to some constant time. In case of a sorted list, we have to search before the insertion. However for an unsorted list, a programmer will insert the item in the start. Similarly we have seen the data structure where insertions, deletions and searches are proportional to *n*. In link list, insertions and deletions are proportional to *n* whereas search is *log n*. It seems that *log n* is the lower limit and we cannot reduce this number more.

Is it true that we cannot do better than *log n* in case of table? Think about it and send your proposals. So far we have find, remove and insert where time varies between constant and *log n*. It would be nice to have all the three as constant time operations. We do not want to traverse the tree or go into long loops. So it is advisable to find the element in first step. If we want to insert or delete, it should be done in one step. How can we do that? The answer is Hashing.

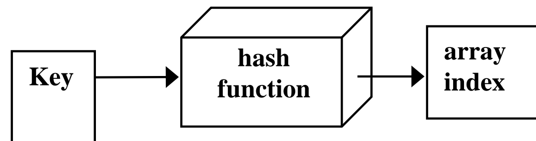
Hashing

The hashing is an algorithmic procedure and a methodology. It is not a new data structure. It is a way to use the existing data structure. The methods- *find*, *insert* and *remove* of table will get of constant time. You will see that we will be able to do this in a single step. What is its advantage? If we need table data structure in some program, it can be used easily due to being very efficient. Moreover, its operations are of constant time. In the recent lectures, we were talking about the algorithms and procedures rather than data structure. Now we will discuss about the strategies and methodologies. Hashing is also a part of this.

We will store the data in the array but *TableNodes* are not stored consecutively. We are storing the element's data in the *TableNodes*. You have seen the array implementation of the *Table* data structure. We have also seen how to make the data sorted. There will be no

gap in the array positions whether we use the sorted or unsorted data. This means that there is some data at the 1st and 2nd position of array and then the third element is stored at the 6th position and 4th and 5th positions are empty. We have not done like this before. In case of link list, it is non-consecutive data structure with respect to memory.

In Hashing, we will internally use array. It may be static or dynamic. But we will not store data in consecutive locations. Their place of storage is calculated using the key and a *hash function.* Hash function is a new thing for you. See the diagram below:



We have a key that may be a name, or roll number or login name etc. We will pass this key to a hash function. This is a mathematical function that will return an array index. In other words, an integer number will be returned. This number will be in some range but not in a sequence.

Keys and entries are scattered throughout the array. Suppose we want to insert the data of our employee. The key is the name of the employee. We will pass this key to the hash function which will return an integer. We will use this number as array index. We will insert the data of the employee at that index.

key entry

4

10

123

The insert will calculate place of storage and insert in *TableNode*. When we get a new data item, its key will be generated with the help of hash function to get the array index. Using this array index, we insert the data in the array. This is certainly a constant time

operation. If our hash function is fast, the insert operation will also rapid. It will take only one step to perform this.

Next we have *find* method. It will calculate the place of storage and retrieve the entry. We will get the key and pass it to the hash function and obtain the array index. We get the data element from that array position. If data is not present at that array position, it means data is not found. We do not need to find the data at some other place. In case of binary search tree, we traverse the tree to find the element. Similarly in list structure we continue our search. Therefore find is also a constant time operation with Hashing.

Finally, we have *remove* method. It will calculate the place of storage and set it to null. That means it will pass the key to the hash function and get the array index. Using this array index, it will remove the element.

Examples of Hashing

Let's see some examples of hashing and hash functions. With the help of these examples you will easily understand the working of *find*, *insert* and *remove* methods.

Suppose we want to store some data. We have a list of some fruits. The names of fruits are in string. The key is the name of the fruit. We will pass it to the hash function to get the hash key.

Suppose our hash function gave us the following values:

```
HashCode ("apple")        =  5
hashCode ("watermelon")   =  3
hashCode ("grapes")       =  8
hashCode ("cantaloupe")   =  7
hashCode ("kiwi")         =  0
hashCode ("strawberry")   =  9
hashCode ("mango")        =  6
hashCode ("banana")       =  2
```

Our hash function name is *hashCode*. We pass it to the string "apple". Resultantly, it returns a number 5. In case of "watermelon" we get the number 3. In case of "grapes" there is number 8 and so on. Neither we are sending the names of the fruits in some order to the function, nor is function returning the numbers in some order. It seems that some random numbers are returned. We have an array to store these strings. Our array will look like as:

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

We store the data depending on the indices got from the *hashCode*. The array size is 10. In case of apple, we get the index 5 from *hashCode* so "apple" is stored at array index 5. As we are dealing with strings, so the array will be an array of strings. The "watermelon" is at position 3 and so on every element is at its position. This array will be in the private part of our data structure and the user will not know about it. If our array is *table* then it will look like as under:

```
table[5]   =  "apple"
table[3]   =  "watermelon"
table[8]   =  "grapes"
table[7]   =  "cantaloupe"
table[0]   =  "kiwi"
table[9]   =  "strawberry"
table[6]   =  "mango"
table[2]   =  "banana"
```

We will store our data in the Table array using the string copy. The user is storing the data using the names of the fruits and wants to retrieve or delete the data using the names of fruits. We have used the array for storage purposes but did not store the data consecutively. We store the data using the hash function which provides us the array index. You can note that there are gaps in the array positions.

Similarly we will retrieve the data using the names of fruit and pass it to the *hashCode* to get the index. Then we will retrieve the data at that position. Consider the table array, it seems that we are using the names of fruits as indices.

```
table["apple"]
```

table["watermelon"]
table["grapes"]
table["cantaloupe"]
table["kiwi"]
table["strawberry"]
table["mango"]
table["banana"]

We are using the array as table [“apple”], table [“watermelon”] and so on. We are not using the numbers as indices here. Internally we are using the integer indices using the *hashCode*. Here we have used the fruit names as indices of the array. This is known as associative array. Now this is the internal details that we are thinking it as associative array or number array.

Let’s discuss about the *hashCode*. How does it work internally? We pass it to strings that may be persons name or name of fruits. How does it generate numbers from these? If the keys are strings, the hash function is some function of the characters in the strings. One possibility is to simply add the ASCII values of the characters. Suppose the mathematical notation of hash function is *h*. It adds all the ASCII values of the string characters. The characters in a string are from 0 to *length - 1*. Then it will take mod of this result with the size of the table. The size of the table is actually the size of our internal array. This formula can be written mathematically as:

$$h(str) = \left(\sum_{i=0}^{length-1} str[i] \right) \% TableSize$$

$$Example : h(ABC) = (65 + 66 + 67) \quad TableSize$$

Suppose we use the string “ABC” and try to find its hash value. The ASCII values of A, B and C are 65, 66 and 67 respectively. Suppose the tableSize is 55. We will add these three numbers and take mod with 55. The result (3.6) will be the hash value. To represent character data in the computer ASCII codes are used. For each character we have a different bit pattern. To memorize this, we use its base 10 values. All the characters on the keyboard like $, %, ‘ have ASCII values. You can find the ASCII table in you book or on the internet.

Let’s see the C++ code of *hashCode* function.

```
int hashCode( char* s )
{
        int i, sum;
        sum = 0;
        for(i=0; i < strlen(s); i++ )
```

```
            sum = sum + s[i]; // ascii value
        return sum % TABLESIZE;
    }
```

The return type of *hashCode* function is an integer and takes a pointer to character. It declares local variable *i* and *sum*, then initializes the *sum* with zero. We use the *strlen* function to calculate the length of the string. We run a loop from 0 to *length – 1*. In the loop, we start adding the ASCII values of the characters. In C++, characters are stored as ASCII values. So we directly add *s[i]*. Then in the end, we take mod of sum with *TABLESIZE*. The variable *TABLESIZE* is a constant representing the size of the table.

This is the one of the ways to implement the hash function. This is not the only way of implementing hash function. The hash function is a very important topic. Experts have researched a lot on hash functions. There may be other implementations of hash functions.

Another possibility is to convert the string into some number in some arbitrary base *b* (*b* also might be a prime number). The formula is as:

$$h(str) = \left(\sum_{i=0}^{length-1} str[i] \times b^i \right) \% T$$

$$Example: h(ABC) = (65b^0 + 66b^1 + 67b^2) \% T$$

We are taking the ASCII value and multiply it by *b* to the power of *i*. Then we accumulate these numbers. In the end, we take the mod of this summation with *tableSize* to get the result. The *b* may be some number. For example, we can take *b* as a prime number and take 7 or 11 etc. Let's take the value of *b* as 7. If we want to get the hash value of ABC using this formula:

H(ABC) = (65 * 7 ^0 + 66 * 7^1 + 67 * 7^2) mod 55 = 15

We are free to implement the hash function. The only condition is that it accepts a string and returns an integer.

If the keys are integers, *key%T* is generally a good hash function, unless the data has some undesirable features. For example, if *T = 10* and all keys end in zeros, then *key%T = 0* for all keys. Suppose we have employee ID i.e. an integer. The employee ID may be in hundreds of thousand. Here the table size is 10. In this case, we will take mod of the employee ID with the table size to get the hash value. Here, the entire employee IDs end in zero. What will be the remainder when we divide this number with 10? It will be 0 for all employees. So this hash function cannot work with this data. In general, to avoid situations like this, *T* should be a prime number.

We are given N items to build a heap, this can be done with _______________ successive inserts.
N-1
N
N+1
N^2
Which one of the following algorithms is most widely used due to its good average time,
Bubble Sort
Insertion Sort
Quick Sort
Merge Sort
There are three decision involved in designing a sample. Which of the following is NOT a part of sample designing?
Sampling unit
Sampling size
Sampling procedure
Sampling cost
Economic factors focus on which one of the following concepts?
Level of economic development
Bureaucracy
Behavioral pattern
Stability of government
Price is used to encourage buyers to try a new product or to purchase existing brands during periods when sales slow down (e.g., recessions). This illustrates the pricing objective of which one of the following options?
Gain market share
Achieve financial performance
Create product positioning
Stimulate demand
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
Union is a _______________ time operation.
Constant
Polynomial
Exponential
None of the given options
If we have 1000 sets each containing a single different person. Which of the following relation will be true on each set:
Reflexive

Symmetric
Transitive
Associative

To ensure the effectiveness of segmentation, the segments should be measurable, substantial, accessible and which one of the followings?

Precise
Stable
Economic
Reachable

All of the following options are the part of 4 Cs EXCEPT:

Care
Choice
Community
Culture

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

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;

To reduce inventory management costs, many companies use a system where they carry only small inventories of parts or merchandise, often only enough for a few days of operation refers to which of the following concepts?

Just-in-time logistics
Limited inventory logistics
Economic order quantity
Supply chain management

Which promotion mix ingredient costs considerably more than advertising to reach one person but can provide more immediate feedback?

Publicity
Sales promotion
Personal selling
Public relations

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.

Those factors that determine the size and means of payment exchanged for goods and services are part of which one of the following mix?

Price promotion mix

Price factor mix
Basic price mix
Production price mix
In complete binary tree the bottom level is filled from ________________
Left to right
Right to left
Not filled at all
None of the given options
Which one of the following is TRUE about recursion?
Recursion extensively uses stack memory.
Threaded Binary Trees use the concept of recursion.
Recursive function calls consume a lot of memory.
Iteration is more efficient than iteration.
Why is the headline of a print advertisement such a critical component of the copy?
It determines the final layout design.
It is often the only part of the advertisement that is read.
It takes up the most space.
It links the copy to the signature.
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
Consider a min heap, represented by the following array:
10,30,20,70,40,50,80,60
After inserting a node with value 31.Which of the following is the updated min heap?
10,30,20,31,40,50,80,60,70
10,30,20,70,40,50,80,60,31
10,31,20,30,40,50,80,60,31
31,10,30,20,70,40,50,80,60
Publics including workers, managers, volunteers and the board of directors show which of the following publics?
Citizen-action publics
Local publics
General publics
Internal Publics
Gillette was the first marketer of disposable razors to offer a product specifically designed for men. This is an example of segmentation using which of the following variables?
Demographic
Psychographic
Geographic
Product use
Suppose A is an array containing numbers in increasing order, but some numbers occur more than once when using a binary search for a value, the binary search always finds ________________

the first occurrence of a value.
the second occurrence of a value.
may find first or second occurrence of a value.
None of the given options.
The search for new-product ideas should be _______________ rather than haphazard.
Consistent
Systematic
Continual
Seldom
Which of the following statements is correct property of binary trees?
A binary tree with N internal nodes has N+1 internal links.
A binary tree with N external nodes has 2N internal nodes.
A binary tree with N internal nodes has N+1 external nodes.
None of above statement is a property of the binary tree.
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
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.
Use of binary tree in compression of data is known as _______________ .
Traversal
Heap
Union
Huffman encoding
A queue is a data structure where elements are,
inserted at the front and removed from the back.
inserted and removed from the top.
inserted at the back and removed from the front.
inserted and removed from both ends.
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
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
Marketing manager wants to improve the packaging of new products after reading customer responses to its customer opinion poll. Which one of the following is NOT a function of packaging?
It contains and protects the product
It contains the brand mark
It determines product quality
It may contain the brand symbol
If a complete binary tree has n number of nodes then its height will be,
Log2 (n+1) -1
2n
Log2 (n) - 1
2n - 1
Collecting, analyzing and interpretation of data refer to which of the following concepts?
Marketing research
Marketing intelligence
Marketing information
Marketing knowledge
In threaded binary tree the NULL pointers are replaced by the
preorder successor or predecessor
inorder successor or predecessor
inorder successor or predecessor
NULL pointers are not replaced
While building Huffman encoding tree the new node that is the result of joining two nodes has the frequency.
Equal to the small frequency
Equal to the greater
Equal to the sum of the two frequencies
Equal to the difference of the two frequencies
If the bottom level of a binary tree is NOT completely filled, depicts that the tree is NOT a
Expression tree
Threaded binary tree
complete Binary tree
Perfectly complete Binary tree

1) If there are 100 elements in a heap and 100 delete Min operation are

performed, will get

_____list

a. Sorted

b. Unsorted

c. Nonlinear

d. Noe

2) Sorting procedure normally takes ____times

a) NLogN

b) 2N

c) N*N*N

d) N

3) The expression if(!heap-> is empty ())

Checks

a) Heap is empty

b) Heap is full

c) Heap is not empty

4) 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

5) A binary relation R over S is called an equivalence relation if it has following property(S)?

a) Reflexivity

b) Symmetry

c) Transitivity

d) All of the given

6) If a tree has 20 edges/links, then the total number of nodes in the tree will be:

a) 19

b) 20

c) 21

d) Cannot be determined

7) For a perfect binary tree of height 4, what will be the sum of highest of node

a) 31

b) 30

c) 27

d) 26

8) 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

a) Reflexivity

b) Symmetry

c) Transitivity

d) All of the above

9) Which property of equivalence relation is satisfied if we say:

Ahmad is cousin of Ali and Ali is also cousin of Ahmed

AL-JUNAID INSTITUTE GROUP

a) Reflexivity

b) Symmetry

c) Transitivity

d) All of the given

10) Which one of the following is NOT the property of equivalence relation?

a) Reflexive

b) Symmetric

c) Transitive

d) Associative

11) The main reason of using heap in priority queue is

a) Improve performance

b) Code readable

c) Less code

d) Heap can' t be used in priority queues

12) The total number of nodes on 10th level of perfect binary tree are

a) 256

b) 512

c) 1024

d) Can’ t be determined

13) Suppose there are 100 elements in an equivalence class, so initially there will be 100 trees,

the collection of these trees is called ___________.

a) Cluster

b) Class

c) Forest

d) Bunch

14) The percolate Down procedure will move the smaller value ____ and bigger value ____.

a) Left, right

b) Right, left

c) Down, up

d) Up, down

15) 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) N - (h - 1)

16) Which of the following method is helpful in creating the heap at once?

AL-JUNAID INSTITUTE GROUP

a. Insert

b. Add

c. Update

d. percolateDown

10. If ahmad is boss of Ahsan 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 given

11. If we want to find 3rd minimum element from an array of element, then after applying build heap method. How many times deleteMin method will be called?

a. 1

b. 2

c. 3

d. 4

12. If we want to find median of 50 elements, then after applying builtHeap method, how many time deleteMin method will be called?

a. 5

b. 25

c. 35

d. 50

13. Which of the following properties are satisfied by equivalence relationship?

a. Reflexive, symmetric

b. Reflexive, transitive

c. Symmetric, transitive

d. Reflexive, symmetric and transitive

14. Sorting procedure normally takes ________ time.

a. NlogN

b. 2N

c. N*N*N

d. N

15. The Expression
if (! heap->isFull()
) Check
a. Heap is empty
b. Heap is full
c. Heap is not empty
d. Heap is not full
16. The Expression
if (! heap->isEmpty())
Check
a. Heap is empty
b. Heap is full
c. Heap is not empty
d. Not a valid expression

AL-JUNAID INSTITUTE GROUP

17. 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 _______.
a. 31
b. 34
c. 44
d. 56
18. Which of the following heap method increase the value if key at position ‘ p’ by
the amount ‘ delta’ ?
a. increaseKey(p, delta)
b. decreaseKey(p, delta)
c. percolateDown(p, delta)

d. remove(p, delta)

19. Which of the following heap method lowers the value if key at position ‘ p’ by
the amount ‘ delta’ ?
a. increaseKey(p, delta)
b. decreaseKey(p, delta)
c. percolateDown(p, delta)
d. remove(p, delta)

20. 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 Above

21. The total number of nodes on 5th level of perfect binary tree are:
a. 16
b. 15
c. 31
d. 32

22. Which property of equivalence relation is satisfied if we
say: Ahmad is cousin of Ali and Ali is also Cousin of
Ahmad
a. Reflexivity
b. Symmetry
c. Transitivity
d. All of the Above

23. If a tree has 50 nodes , then the total edges/links in the tree will be
a. 55
b. 51
c. 50
d. 49

24. If the height of perfect binary tree is 4, what will be the total number of nodes in
it?
a. 15
b. 16
c. 31
d. 32

AL-JUNAID INSTITUTE GROUP
AL-JUNAID INSTITUTE GROUP

25. Suppose there are set of fruits and the set of vegetables, both sets are _______ sets.
a. Disjoint
b. Subset
c. Whole
d. Equal

26. A binary relation R over S is called an equivalence relation if it has following property(s)
a. Reflexivity
b. Symmetry
c. Transitivity
d. All of Above

27. Heap can be used to implement
a. Stack
b. Linked list
c. Queue
d. Priority queue

28. If a tree has 20 edges/links, then the total number of nodes in the tree will be:

a. 19

b. 20

c. 21

d. Can' t be determined

29. If there are 100 elements in heap, and 100 deleteMin operation are performed, will get______ list.

a. Sorted

b. Unsorted

c. Nonlinear

d. None of given

30. If there are 100 elements in an equivalence class, then we will have _______ sets initially.

a. 50

b. 100

c. 1000

d. 80

31. 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?

a. 31

b. 34

c. 44

d. 56

AL-JUNAID INSTITUTE GROUP

Question No: 1

A solution is said to be efficient if it solves the problem within its resource constraints i.e. hardware and time.

► True (Page 4)
► False
Question No: 2
Which one of the following is known as "Last-In, First-Out" or LIFO Data Structure?
► Linked List
► Stack (Page 54)
► Queue
► Tree
Question No: 3
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: 4
For compiler a postfix expression is easier to evaluate than infix expression?
► True
► False
Question No: 5
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
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 (Page 304)

AL-JUNAID INSTITUTE GROUP

► N+1
► N+2
► N
Question No: 8
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
► N
► N +1 (Page 303)
► N +2
Question No: 9
If we have 1000 sets each containing a single different person. Which of the following relation will be true on
each set:
► Reflexive (page 387)
► Symmetric
► Transitive
► Associative
Question No: 10
Which one of the following is NOT the property of equivalence relation:
► Reflexive
► Symmetric

► Transitive
► Associative (page 387)
Question No: 11
A binary tree of N nodes has .
► Log_{10} N levels
► Log_2 N levels (Page 212)
► N / 2 levels
► N x 2 levels
Question No: 12
The easiest case of deleting a node from BST is the case in which the node to be deleted .
► Is a leaf node (Page 173)
► Has left subtree only
► Has right subtree only
► Has both left and right subtree
Question No: 13
If there are N elements in an array then the number of maximum steps needed to find an element using Binary
Search is .
► N
► N^2
► $Nlog_2N$
► log_2N (page 440)

AL-JUNAID INSTITUTE GROUP

Question No: 14
Merge sort and quick sort both fall into the same category of sorting algorithms. What is this category?
► O(nlogn) sorts
► Interchange sort (not sure)
► Average time is quadratic
► None of the given options. (Page 488)
Question No: 15

If one pointer of the node in a binary tree is NULL then it will be a/an .
► External node (Page 303)
► Root node
► Inner node
► Leaf node
Question No: 16
We convert the _ pointers of binary to threads in threaded binary tree.
► Left
► Right
► NULL (Page 312)
► None of the given options
Question No: 17
If the bottom level of a binary tree is NOT completely filled, depicts that the tree is NOT
► Expression tree
► Threaded binary tree
► complete Binary tree (Page 323)
► Perfectly complete Binary tree
Question No: 18
What is the best definition of a collision in a hash table?
► Two entries are identical except for their keys.
► Two entries with different data have the exact same key
► Two entries with different keys have the same exact hash value. (page 464)
► Two entries with the exact same key have different hash values.
Question No: 19
Suppose that a selection sort of 100 items has completed 42 iterations of the main loop. How many items are now guaranteed to be in their final spot (never to be moved again)
► 21
► 41
► 42
► 43

AL-JUNAID INSTITUTE GROUP

Question No: 20

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 (page 334)

Question No: 21

Do you see any problem in the code of nextInOrder below:

```
TreeNode * nextInorder(TreeNode * p)
{
if(p->RTH == thread)
return( p->R );
else {
p = p->R;
while(p->LTH == child)
p = p->R;
return p;
}
}
```

► The function has no problem and will fulfill the purpose successfully.

► The function cannot be compile as it has syntax error.

► The function has logical problem, therefore, it will not work properly.

► The function will be compiled but will throw runtime exception immediately after the control is transferred to this function.

Question No: 22

Which of the following statement is correct about find(x) operation:

► A find(x) on element x is performed by returning exactly the same node that is found.

► A find(x) on element x is performed by returning the root of the tree containing x.

► A find(x) on element x is performed by returning the whole tree itself containing x.

► A find(x) on element x is performed by returning TRUE.

Question No: 23

Which of the following statement is NOT correct about find operation:

► It is not a requirement that a find operation returns any specific name, just that finds on two elements

return the same answer if and only if they are in the same set.

► One idea might be to use a tree to represent each set, since each element in a tree has the same root,
thus the root can be used to name the set.

► Initially each set contains one element.

► Initially each set contains one element and it does not make sense to make a tree of one node only.

Question No: 24

In complete binary tree the bottom level is filled from

► Left to right (Page 323)

► Right to left

► Not filled at all

► None of the given options

Question No: 25

AL-JUNAID INSTITUTE GROUP

Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

The array after the FIRST iteration of the large loop in a selection sort (sorting from smallest to largest).

► 0 3 8 9 1 7 5 2 6 4 (Page 477)

► 2 6 4 0 3 8 9 1 7 5

► 2 6 4 9 1 7 0 3 8 5

► 0 3 8 2 6 4 9 1 7 5

Question No: 26

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

Question No: 27

Which one of the following operations returns top value of the stack?

► Push

► Pop

► Top (page 53)

► First

Question No: 28

Compiler uses which one of the following in Function calls,

► Stack (page 80)

► Queue

► Binary Search Tree

► AVL Tree

Question No: 29

Every AVL is

► Binary Tree

► Complete Binary Tree

► None of these

► Binary Search TreeQuestion No 30

If there are 56 internal nodes in a binary tree then how many external nodes this binary tree will have?

► 54

► 55

► 56

► 57 (page 303)

Question No: 31

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 (page 303)

AL-JUNAID INSTITUTE GROUP

Question No: 32

Which one of the following is not an example of equivalence relation?

► Electrical connectivity

► Set of people

► <= relation (page 388)

► Set of pixels

Question No: 33

Binary Search is an algorithm of searching, used with the data.

► Sorted (page 432)

► Unsorted

► Heterogeneous

► Random

Question No: 34

Which one of the following is NOT true regarding the skip list?

► Each list S_i contains the special keys + infinity and - infinity.

► List S_0 contains the keys of S in non-decreasing order.

► Each list is a subsequence of the previous one.

► List S_h contains only the n special keys. (page 446)

Question No: 35

A simple sorting algorithm like selection sort or bubble sort has a worst-case of

► O(1) time because all lists take the same amount of time to sort

► O(n) time because it has to perform n swaps to order the list.

► O(n_2) time because sorting 1 element takes O(n) time - After 1 pass through the list,either of
these algorithms can guarantee that 1 element is sorted. (page 487)

► O(n^3) time, because the worst case has really random input which takes longer to sort.Question No: 36

Which of the following is a property of binary tree?

► A binary tree of N external nodes has N internal node.
► A binary tree of N internal nodes has N+ 1 external node. (page 303)
► A binary tree of N external nodes has N+ 1 internal node.
► A binary tree of N internal nodes has N- 1 external node.

Question No: 37

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 (page 306)
► Heap data structure
► Huffman encoding

Question No: 38

Which of the following statement is true about dummy node of threaded binary tree?

► This dummy node never has a value.
► This dummy node has always some dummy value.

AL-JUNAID INSTITUTE GROUP

► This dummy node has either no value or some dummy value. (Page 321)
► This dummy node has always some integer value.

Question No: 39

For a perfect binary tree of height h, having N nodes, the sum of heights of nodes is

► N – (h – 1)
► N – (h + 1) (page 373)
► N – 1
► N – 1 + h

Question No: 40

What is the best definition of a collision in a hash table?

► Two entries are identical except for their keys.

► Two entries with different data have the exact same key

► Two entries with different keys have the same exact hash value. (page 464)

► Two entries with the exact same key have different hash values.

Question No: 41

Which formula is the best approximation for the depth of a heap with n nodes?

► log (base 2) of n (page 353)

► The number of digits in n (base 10), e.g., 145 has three digits

► The square root of n

► n

Question No: 42

Which of the following statement is NOT correct about find operation:

► It is not a requirement that a find operation returns any specific name, just that finds on two elements

return the same answer if and only if they are in the same set.

► One idea might be to use a tree to represent each set, since each element in a tree has the same
root, thus the root can be used to name the set.

► Initially each set contains one element.

► Initially each set contains one element and it does not make sense to make a tree of one node only.

Question No: 43

Which of the following is not true regarding the maze generation?

► Randomly remove walls until the entrance and exit cells are in the same set.

► Removing a wall is the same as doing a union operation.

► Remove a randomly chosen wall if the cells it separates are already in the same set. (page 424)

► Do not remove a randomly chosen wall if the cells it separates are already in the same set.

Question No: 44

In threaded binary tree the NULL pointers are replaced by ,

► preorder successor or predecessor

► inorder successor or predecessor (page 307)

► postorder successor or predecessor

► NULL pointers are not replaced

Question No: 45

Which of the given option is NOT a factor in Union by Size:

► Maintain sizes (number of nodes) of all trees, and during union.

AL-JUNAID INSTITUTE GROUP

► Make smaller tree, the subtree of the larger one.

► Make the larger tree, the subtree of the smaller one. (page 408)

► Implementation: for each root node i, instead of setting parent[i] to -1, set it to -k if tree rooted at i has k

nodes.

Question No: 46

Suppose we had a hash table whose hash function is " n % 12" , if the number 35 is already in the hash table,

which of the following numbers would cause a collision?

► 144

► 145

► 143

► 148

Question No: 47

What requirement is placed on an array, so that binary search may be used to locate an entry?

► The arror 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

Question No: 48

A binary tree with 24 internal nodes has external nodes.

- 22
- 23
- 48
- 25 (page 303)

Question No: 49

In case of deleting a node from AVL tree, rotation could be prolong to the root node.

► Yes (Page 267)

► No

Question No: 49

when we have declared the size of the array, it is not possible to increase or decrease it during the
of the program.

► Declaration

► Execution (page 17)

► Defining

► None of the above

Question No: 50

it will be efficient to place stack elements at the start of the list because insertion and removal take
time.

► Variable

► Constant (page 60)

► Inconsistent

► None of the above

AL-JUNAID INSTITUTE GROUP

Question No: 51

is the stack characteristic but was implemented because of the size limitation of the
array.

► isFull(),isEmpty()

► pop(), push()

► isEmpty() , isFull() (page 59)

► push(),pop()

AL-JUNAID INSTITUTE GROUP

Question No: 52

What kind of list is best to answer questions such as "What is the item at position n?"

► Lists implemented with an array.
► Doubly-linked lists.
► Singly-linked lists.
► Doubly-linked or singly-linked lists are equally best

Question No: 53

Each node in doubly link list has,

► 1 pointer
► 2 pointers (page 39)
► 3 pointers
► 4 pointers

Question No: 54

If there are 56 internal nodes in a binary tree then how many external nodes this binary tree will have?

► 54
► 55
► 56
► 57 (page 303)

Question No: 55

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
► N
► N +1 (page 303)
► N +2

Question No: 56

A binary tree with N internal nodes has links, links to internal nodes and links to external nodes

► N+1, 2N, N-1
► N+1, N-1, 2N
► 2N, N-1, N+1 (page 304)
► N-1, 2N, N+1

Question No: 57

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 (page 385)
- ► For all elements w, x, y and z, if x R y and w R z then x R z

AL-JUNAID INSTITUTE GROUP

Question No: 58

Which one of the following is not an example of equivalence relation:

- ► Electrical connectivity
- ► Set of people
- ► <= relation (page 388)
- ► Set of pixels

Question No: 59

Union is a time operation.

- ► Constant (page 405)
- ► Polynomial
- ► Exponential
- ► None of the given options

Question No: 60

Binary Search is an algorithm of searching, used with the data.

- ► Sorted (page 432)
- ► Unsorted
- ► Heterogeneous
- ► Random

Question No: 61

A simple sorting algorithm like selection sort or bubble sort has a worst-case of

- ► O(1) time because all lists take the same amount of time to sort
- ► O(n) time because it has to perform n swaps to order the list.
- ► $O(n_2)$ time because sorting 1 element takes O(n) time - After 1 pass through the list, either of these algorithms can guarantee that 1 element is sorted. (page 487)

► $O(n^3)$ time, because the worst case has really random input which takes longer to sort.

Question No: 62

Merge sort and quick sort both fall into the same category of sorting algorithms. What is this category?

► O(nlogn) sorts

► Interchange sort

► Average time is quadratic

► None of the given options. (Page 488)

Question No: 63

Huffman encoding uses tree to develop codes of varying lengths for the letters used in the original message.

► Linked list

► Stack

► Queue

► Binary tree (page 287)

AL-JUNAID INSTITUTE GROUP

Question No: 64

Which of the following statement is true about dummy node of threaded binary tree?

► The left pointer of dummy node points to the itself while the right pointer points to the root of tree.

► The left pointer of dummy node points to the root node of the tree while the right pointer points itself i.e. to dummy node (page 321)

► The left pointer of dummy node points to the root node of the tree while the right pointer is always NULL.

► The right pointer of dummy node points to the itself while the left pointer is always NULL.

Question No: 65

Consider a min heap, represented by the following array:

10,30,20,70,40,50,80,60

After inserting a node with value 31.Which of the following is the updated min heap?

► 10,30,20,31,40,50,80,60,70 (page 336)

► 10,30,20,70,40,50,80,60,31

► 10,31,20,30,40,50,80,60,31

► 31,10,30,20,70,40,50,80,60

Question No: 66

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 (page 336)

► 11,22,33,44,66,55

► 11,22,33,66,44,55

► 11,22,66,33,44,55

Question No: 67

Suppose that a selection sort of 100 items has completed 42 iterations of the main loop. How many items are

now guaranteed to be in their final spot (never to be moved again)?

► 21

► 41

► 42

► 43

Question No: 68

is a data structure that can grow easily dynamically at run time without having to copy existing

elements.

► Array ()

► List

► Both of these (page 10)

► None of these

AL-JUNAID INSTITUTE GROUP

Question No: 69

The maximum number of external nodes (leaves) for a binary tree of height H is

► 2^H

► $2^H +1$

► $2^H -1$

► $2^H +2$

Question No: 70

A complete binary tree of height has nodes between 16 to 31 .

► 2

► 3

► 4 (page 373)

► 5

Question No: 71

Which of the given option is NOT a factor in Union by Size:

► Maintain sizes (number of nodes) of all trees, and during union.

► Make smaller tree, the subtree of the larger one.

► Make the larger tree, the subtree of the smaller one. (page 408)

► Implementation: for each root node i, instead of setting parent[i] to -1, set it to -k if tree rooted at i has k

nodes.

Question No: 72

Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

The array after the FIRST iteration of the large loop in a selection sort (sorting from smallest to largest).

► 0 3 8 9 1 7 5 2 6 4 (Page 477)

► 2 6 4 0 3 8 9 1 7 5

► 2 6 4 9 1 7 0 3 8 5

► 0 3 8 2 6 4 9 1 7 5

Question No: 73

Suppose A is an array containing numbers in increasing order, but some numbers occur more than once when
using a binary search for a value, the binary search always finds

- ► the first occurrence of a value.
- ► the second occurrence of a value.
- ► may find first or second occurrence of a value.
- ► None of the given options.

Question No: 74
A binary tree with 24 internal nodes has external nodes.

AL-JUNAID INSTITUTE GROUP

- ► 22
- ► 23
- ► 48
- ► 25 (page 303)

Question No: 75
it will be efficient to place stack elements at the start of the list because insertion and removal take
time.

- ► Variable
- ► Constant (page 60)
- ► Inconsistent
- ► None of the above

Question No: 76
"+" is a operator.

- ► Unary
- ► Binary (page 64)
- ► Ternary
- ► None of the above

Question No: 77
A kind of expressions where the operator is present between two operands called expressions.

- ► Postfix
- ► Infix (page 64)

► Prefix

► None of the above.

Question No: 78

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?

AL-JUNAID INSTITUTE GROUP

► 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.

Question No: 79

A binary tree with N internal nodes has links, links to internal nodes and links to external nodes

► N+1, 2N, N-1

► N+1, N-1, 2N

► 2N, N-1, N+1 (page 304)

► N-1, 2N, N+1

Question No: 80

Each node in doubly link list has,

► 1 pointer

► 2 pointers (Page 39)

► 3 pointers

► 4 pointers

Question No: 81

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 (page 10)

► None of these

Question No:82

Which one of the following is not an example of equivalence relation:

► Electrical connectivity

► Set of people

► <= relation (Page 388)

► Set of pixels

Question No: 83

If a complete binary tree has height h then its no. of nodes will be,

► Log (h)

► 2^{h+1}- 1 (page 125)

► Log (h) - 1

► 2^{h} - 1

AL-JUNAID INSTITUTE GROUP

Question No: 84

If a max heap is implemented using a partially filled array called data, and the array contains n elements (n >
0), where is the entry with the greatest value? Data[0] is correct

► data[1]

► data[n-1]

► data[n]

► data[2*n+1]

Question No: 85

Which one is a self-referential data type?

► Stack

► Queue
► Link list
► All of these
Question No: 86
There is/are case/s for rotation in an AVL tree,
► 1
► 3
► 2
► 4 (page 229)
Question No: 87
Which of the following can be the inclusion criteria for pixels in image segmentation.
► Pixel intensity
► Texture
► Threshold of intensity
► All of the given options (page 421)
Question No: 88
Consider te following array
23 15 5 12 40 10 7
After the first pass of a particular algorithm, the array looks like
15 5 12 23 10 7 40
Name the algorithm used
► Heap sort
► Selection sort
► Insertion sort
► Bubble sort

AL-JUNAID INSTITUTE GROUP

Question No: 89
In a perfectly balanced tree the insertion of a node needs .
► One rotation (Page 225)
► Two rotations

► Rotations equal to number of levels

► No rotation at all

Question No: 89

If there are N elements in an array then the number of maximum steps needed to find an element using Binary

Search is .

► N

► N_2

► $Nlog_2N$

► log_2N (page 440)

Question No: 90

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. (page 408)

► 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.

Question No: 91

If both pointers of the node in a binary tree are NULL then it will be a/an .

► Inner node

► Leaf node (page 120)

► Root node

► None of the given options

Question No: 92

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.(page 506)

► 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.

AL-JUNAID INSTITUTE GROUP

Question No 93

What is the best definition of a collision in a hash table?

► Two entries are identical except for their keys.

► Two entries with different data have the exact same key

► Two entries with different keys have the same exact hash value. (page 464)

► Two entries with the exact same key have different hash values.

Question No: 94

For a perfect binary tree of height h, having N nodes, the sum of heights of nodes is

► N – (h – 1)

► N – (h + 1) (Page 373)

► N – 1

► N – 1 + h

Question No: 95

A binary tree with 33 internal nodes has links to internal nodes.

► 31

► 32 (Page 304)

► 33

► 66

Question No: 96

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 (see min heap property at page 337)

Question No: 97

Which of the following is not true regarding the maze generation?

► Randomly remove walls until the entrance and exit cells are in the same set.

► Removing a wall is the same as doing a union operation.
► Remove a randomly chosen wall if the cells it separates are already in the same set. (Page 424)
► Do not remove a randomly chosen wall if the cells it separates are already in the same set.

AL-JUNAID INSTITUTE GROUP

Question No: 98
Which formula is the best approximation for the depth of a heap with n nodes?
► log (base 2) of n (Page 353)
► The number of digits in n (base 10), e.g., 145 has three digits
► The square root of n
► n
Question No: 99
In threaded binary tree the NULL pointers are replaced by ,
► preorder successor or predecessor
► inorder successor or predecessor (Page 307)
► postorder successor or predecessor
► NULL pointers are not replaced
Question No: 100
The method of list will position the currentNode and lastCurrentNode at the start of the list.
► Remove
► Next
► Start (Page 38)
► Back
Question No: 101
Mergesort makes two recursive calls. Which statement is true after these recursive calls finish, but before the
merge step?
► Elements in the first half of the array are less than or equal to elements in the second half of the array.

► None of the given options.

► The array elements form a heap.

► Elements in the second half of the array are less than or equal to elements in the first half of the
array.

Question No: 102

Suppose we had a hash table whose hash function is " n % 12" , if the number 35 is already in the hash table,
which of the following numbers would cause a collision?

► 144

► 145

► 143

► 148

AL-JUNAID INSTITUTE GROUP

Question No: 103

The arguments passed to a function should match in number, type and order with the parameters in the function
definition.

► True

► False

Question No: 104

If numbers 5, 222, 4, 48 are inserted in a queue, which one will be removed first?

► 48

► 4

► 222

► 5

Question No: 105

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: 106

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

Question No: 107

Here is a small function definition:

```
void f(int i, int &k)
{
i = 1;
k = 2;
}
```

AL-JUNAID INSTITUTE GROUP

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.

Question No: 108

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: 109

Compiler uses which one of the following to evaluate a mathematical equation,

► Binary Tree

► Binary Search Tree

► Parse Tree (Page 274)

► AVL Tree

Question No: 110

If there are 56 internal nodes in a binary tree then how many external nodes this binary tree will have?

► 54

► 55

► 56

► 57 (Page 303)

Question No: 111

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 (n-1) (Page 304)

Question No: 112

Which of the following method is helpful in creating the heap at once?

► insert

► add

► update

► preculateDown (Page 358)

AL-JUNAID INSTITUTE GROUP

Question No: 113

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 (Page 385)

► For all elements w, x, y and z, if x R y and w R z then x R z

Question No: 114

A binary tree of N nodes has .

► Log_{10} N levels

► Log_2 N levels (Page 349)

► N / 2 levels

► N x 2 levels

Question No: 115

If there are N elements in an array then the number of maximum steps needed to find an element using Binary

Search is .

► N

► N^2

► $Nlog_2N$

► log_2N (page 440)

Question No: 116

Consider te following array

23 15 5 12 40 10 7

After the first pass of a particular algorithm, the array looks like

15 12 23 10 7 40

Name the algorithm used

► Heap sort

► Selection sort

► Insertion sort

► Bubble sort

Question No: 117

If both pointers of the node in a binary tree are NULL then it will be a/an .

► Inner node

► Leaf node (Page 313)

► Root node

► None of the given options

AL-JUNAID INSTITUTE GROUP

Question No: 118

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 (page 306)
► Heap data structure
► Huffman encoding

Question No: 119

A complete binary tree of height 3 has between _ nodes.

► 8 to 14
► 8 to 15 (Page 124)
► 8 to 16
► 8 to 17

2^(d+1) - 1 = 2^(3+1) -1 = 2^4 -1 = 16-1 = 15

Question No: 120

Consider a min heap, represented by the following array:

3,4,6,7,5,10

After inserting a node with value 1.Which of the following is the updated min heap?

► 3,4,6,7,5,10,1
► 3,4,6,7,5,1,10
► 3,4,1,5,7,10,6
► 1,4,3,5,7,10,6 close to correct but correct ans is 1,4,3,7,5,10,6 (page 337)

Question No: 121

Consider a min heap, represented by the following array:

10,30,20,70,40,50,80,60

After inserting a node with value 31.Which of the following is the updated min heap?

► 10,30,20,31,40,50,80,60,70 (page 337)
► 10,30,20,70,40,50,80,60,31
► 10,31,20,30,40,50,80,60,31
► 31,10,30,20,70,40,50,80,60

Question No: 122

Which one of the following algorithms is most widely used due to its good average time,

► Bubble Sort
► Insertion Sort
► Quick Sort
► Merge Sort

AL-JUNAID INSTITUTE GROUP

Question No: 123

Which of the following statement is correct about find(x) operation:

► A find(x) on element x is performed by returning exactly the same node that is found.
► A find(x) on element x is performed by returning the root of the tree containing x.
► A find(x) on element x is performed by returning the whole tree itself containing x. (Page 10)
► A find(x) on element x is performed by returning TRUE.

Question No 124

Which of the following statement is NOT correct about find operation:

► It is not a requirement that a find operation returns any specific name, just that finds on two elements

return the same answer if and only if they are in the same set.
► One idea might be to use a tree to represent each set, since each element in a tree has the same
root, thus the root can be used to name the set.
► Initially each set contains one element.

► Initially each set contains one element and it does not make sense to make a tree of one node only.

Question No: 125

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

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: 126

The maximum number of external nodes (leaves) for a binary tree of height H is

► 2^H

► $2^H +1$

► $2^H -1$

► $2^H +2$

Question No: 127

In complete binary tree the bottom level is filled from

► Left to right (Page 323)

► Right to left

► Not filled at all

► None of the given options

AL-JUNAID INSTITUTE GROUP

Question No: 128

We are given N items to build a heap, this can be done with successive inserts.

► N-1

► N (Page 353)

► N+1

► N^2

Question No: 129

Suppose we had a hash table whose hash function is " n % 12" , if the number 35 is already in the hash table,

which of the following numbers would cause a collision?

► 144

► 145

► 143

► 148

Question No: 130

Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

The array after the FIRST iteration of the large loop in a selection sort (sorting from smallest to largest).

► 0 3 8 9 1 7 5 2 6 4 (Page 477)

► 2 6 4 0 3 8 9 1 7 5

► 2 6 4 9 1 7 0 3 8 5

► 0 3 8 2 6 4 9 1 7 5

Question No: 131

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.

Question No: 132

In case of deleting a node from AVL tree, rotation could be prolong to the root node.

► Yes (Page 267)

► No

AL-JUNAID INSTITUTE GROUP

Question No 133

only removes items in reverse order as they were entered.

► Stack (Page 81)

► Queue

► Both of these

► None of these

Question No:134

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.

Question No:135

Select the one FALSE statement about binary trees:

► Every binary tree has at least one node. (Page 113)

► Every non-empty tree has exactly one root node.

► Every node has at most two children.

► Every non-root node has exactly one parent.

Question No: 136

Every AVL is

► Binary Tree

► Complete Binary Tree

► None of these

► Binary Search Tree

Question No: 137

Searching an element in an AVL tree take maximum time (where n is no. of nodes in AVL tree),

► $Log_2(n+1)$

► $Log_2(n+1) -1$

► $1.44 Log_2 n$ (Page 227)

► $1.66 Log_2 n$

AL-JUNAID INSTITUTE GROUP

Question No: 138

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.

Question No: 139

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

According to max heap property

Question NO 140

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 (N-1)

Question No 141

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 (Page 304)

► N+1

► N+2

► N

Question No: 142

Which one of the following is NOT the property of equivalence relation:

► Reflexive

► Symmetric

► Transitive

► Associative (Page 385)

AL-JUNAID INSTITUTE GROUP

Question No: 143

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 (Page 385)

► For all elements w, x, y and z, if x R y and w R z then x R z

Question No: 144

Union is a time operation.

► Constant (Page 120)

► Polynomial

► Exponential

► None of the given option

Question No: 145

Which of the following is NOT a correct statement about Table ADT.

► In a table, the type of information in columns may be different. yes

► A table consists of several columns, known as entities. (Page 408)

► 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.

Question No: 146

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 log2 N (Page 441)

Question No: 147

Binary Search is an algorithm of searching, used with the data.

► Sorted (Page 432)

► Unsorted

► Heterogeneous

► Random

Question No: 148

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 PREOREDR 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. (Page 307)

► 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.

AL-JUNAID INSTITUTE GROUP

Question No: 149

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 (page 306)

► Heap data structure

► Huffman encoding

Question No: 150

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. (page 321)

Question No: 151

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 (page 337)

► 11,22,33,44,66,55

► 11,22,33,66,44,55

► 11,22,66,33,44,55

Question No: 152

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 (page 349)
► 4,6,5,7
Question No: 153
We can build a heap in _ time.
► Linear (Page 353)
► Exponential
► Polynomial
► None of the given options

AL-JUNAID INSTITUTE GROUP

Question No: 154
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. (page 506)
► 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.
Question No: 155
Which formula is the best approximation for the depth of a heap with n nodes?
► log (base 2) of n (Page 353)
► The number of digits in n (base 10), e.g., 145 has three digits
► The square root of n
► n
Question No 156
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
Question No: 157
While joining nodes in the building of Huffman encoding tree if there are more nodes with same frequency, we
choose the nodes _.
► Randomly (Page 289)
► That occur first in the text message
► That are lexically smaller among others.
► That are lexically greater among others
Question No: 158
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 (Page 52)
► (i) tree (ii) insertion (iii) removals (iv) top

AL-JUNAID INSTITUTE GROUP

Question No: 159
A binary tree with 33 internal nodes has links to internal nodes.
► 31
► 32 (n-1 links to internal nodes) (Page 304)
► 33
► 66 .
Question No: 160
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
Question No: 161
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.
Question No: 162
Which of the following is a non linear data structure?
► Linked List
► Stack
► Queue
► Tree (Page 112)
Question No: 163
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: 164
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

AL-JUNAID INSTITUTE GROUP

Question No: 165
Which one of the following is a valid postfix expression?

- ► ab+c*d-
- ► abc*+d- (According to rule)
- ► abc+*d-
- ► (abc*)+d-

Question No: 166

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: 167

I have implemented the queue with a circular array. If data is a circular array of CAPACITY elements, and last

is an index into that array, what is the formula for the index after last?

- ► (last % 1) + CAPACITY
- ► last % (1 + CAPACITY)
- ► (last + 1) % CAPACITY
- ► last + (1 % CAPACITY)

This expression will point to field after last that will be the first field.

Question No: 168

Which one of the following is TRUE about recursion?

- ► Recursion extensively uses stack memory. (page 149)
- ► Threaded Binary Trees use the concept of recursion.
- ► Recursive function calls consume a lot of memory.
- ► Iteration is more efficient than iteration.

Question No: 169

Compiler uses which one of the following to evaluate a mathematical equation,

- ► Binary Tree
- ► Binary Search Tree
- ► Parse Tree (Page 274)
- ► AVL Tree

AL-JUNAID INSTITUTE GROUP

Question No: 170

Which one of the following is TRUE about iteration?

► Iteration extensively uses stack memory.

► Threaded Binary Trees use the concept of iteration.

► Iterative function calls consumes a lot of memory.

►Recursion is more efficient than iteration.

Question No: 171

If a max heap is implemented using a partially filled array called data, and the array contains n elements (n >

0), where is the entry with the greatest value? Data[0] is correct

► data[1]

► data[n-1]

► data[n]

► data[2*n+1]

Question No: 172

If there are 56 internal nodes in a binary tree then how many external nodes this binary tree will have?

► 54

► 55

► 56

► 57 (n+1)

Question No: 173

Which of the following heap method increase the value of key at position „ p" by the amount „ delta" ?

► increaseKey(p,delta) (Page 363)

► decreaseKey(p,delta)

► preculateDown(p,delta)

► remove(p,delta)

Question No: 174

If we have 1000 sets each containing a single different person. Which of the following relation will be true on
each set:

- ► Reflexive (page 387)
- ► Symmetric
- ► Transitive
- ► Associative

AL-JUNAID INSTITUTE GROUP

Question No: 175

Which one of the following is not an example of equivalence relation?

- ► Electrical connectivity
- ► Set of people
- ► <= relation (Page 388)
- ► Set of pixels

Question No: 176

A binary tree of N nodes has .

- ► Log$_{10}$ N levels
- ► Log$_{2}$ N levels (Page 212)
- ► N / 2 levels
- ► N x 2 levels

Question No: 177

Binary Search is an algorithm of searching, used with the data.

- ► Sorted (Page 432)
- ► Unsorted
- ► Heterogeneous
- ► Random

Question No: 178

Consider te following array

23 15 5 12 40 10 7

After the first pass of a particular algorithm, the array looks like

15 5 12 23 10 7 40

Name the algorithm used

- ► Heap sort
- ► Selection sort
- ► Insertion sort
- ► Bubble sort (According to rule)

Question No: 179

Which of the following statements is correct property of binary trees?

- ► A binary tree with N internal nodes has N+1 internal links.
- ► A binary tree with N external nodes has 2N internal nodes.
- ► A binary tree with N internal nodes has N+1 external nodes. (page 304)
- ► None of above statement is a property of the binary tree.

Question No: 180

Which of the following is a property of binary tree?

- ► A binary tree of N external nodes has N internal node.
- ► A binary tree of N internal nodes has N+ 1 external node. (Page 304)
- ► A binary tree of N external nodes has N+ 1 internal node.
- ► A binary tree of N internal nodes has N- 1 external node.

AL-JUNAID INSTITUTE GROUP

Question No: 181

Which of the following statement is true about dummy node of threaded binary tree?

► The left pointer of dummy node points to the itself while the right pointer points to the root of tree.

► The left pointer of dummy node points to the root node of the tree while the right pointer points
itself i.e. to dummy node (Page 321)

► The left pointer of dummy node points to the root node of the tree while the right pointer is always NULL.

► The right pointer of dummy node points to the itself while the left pointer is always NULL.

Question No: 182

If the bottom level of a binary tree is NOT completely filled, depicts that the tree is NOT a

- ► Expression tree
- ► Threaded binary tree
- ► complete Binary tree (Page 323)
- ► Perfectly complete Binary tree

Question No: 183

In a selection sort of n elements, how many times the swap function is called to complete the execution of the algorithm?

- ► n-1
- ► n log n
- ► n_2
- ► 1

Question No: 184

Which of the following statement is correct about find(x) operation:

- ► A find(x) on element x is performed by returning exactly the same node that is found.
- ► A find(x) on element x is performed by returning the root of the tree containing x.
- ► A find(x) on element x is performed by returning the whole tree itself containing x.
- ► A find(x) on element x is performed by returning TRUE.

Question No: 185

Which of the following statement is NOT correct about find operation:

- ► It is not a requirement that a find operation returns any specific name, just that finds on two elements return the same answer if and only if they are in the same set.
- ► One idea might be to use a tree to represent each set, since each element in a tree has the same root, thus the root can be used to name the set.
- ► Initially each set contains one element.
- ► Initially each set contains one element and it does not make sense to make a tree of one node only.

AL-JUNAID INSTITUTE GROUP

Question No: 186

Consider the following postfix expression S and the initial values of the variables.

S = A B - C + D E F - + ^

Assume that A=3, B=2, C=1, D=1, E=2, F=3

What would be the final output of the stack?

► 1

► 2

► 0

► -1

Question No: 187

The maximum number of external nodes (leaves) for a binary tree of height H is

► 2H

► 2H +1

► 2H -1

► 2H +2

Question No: 188

In threaded binary tree the NULL pointers are replaced by ,

► preorder successor or predecessor

► inorder successor or predecessor (Page 307)

► postorder successor or predecessor

► NULL pointers are not replaced

Question No: 189

In a min heap , preculateDown procedure will move smaller value and bigger value .

► left,right

► right,left

► up,down (Page 358)

► down,up

Question No: 190

Which of the following statement is correct about union:

► To perform Union of two sets, we merge the two trees by making the root of one tree point to the
root of the other. (Greedy algorithms , Page 7)

► To perform Union of two sets, we merge the two trees by making the leaf node of one tree point to the
root of the other.

► To perform Union of two sets, merging operation of trees in not required at all.

► None of the given options.

AL-JUNAID INSTITUTE GROUP

Question No: 191

Suppose A is an array containing numbers in increasing order, but some numbers occur
more than once when using a binary search for a value, the binary search always finds

► the first occurrence of a value.

► the second occurrence of a value.

► may find first or second occurrence of a value.

► None of the given options.

Question No: 192

Let heap stored in an array as H = [50, 40, 37, 32, 28, 22, 36, 13]. In other words, the root of
the heap contains the maximum element. What is the result of deleting 40 from this heap

► [50,32, 37,13, 28, 22, 36] according to max heap property.

► [37, 28, 32, 22, 36, 13]

► [37, 36, 32, 28, 13, 22]

► [37, 32, 36, 13, 28, 22]

Question No: 193

In an array we can store data elements of different types.

► True

► False

Question no 194

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 (page 18)
- In linked list each element also has the address of the element next to it
- In an array the elements are contiguous

Question no 195

Doubly Linked List always has one NULL pointer.

- True
- False(page 39)

Question No: 196

A queue is a data structure where elements are,

- inserted at the front and removed from the back. . (page #89 nd 90)
- inserted and removed from the top.
- inserted at the back and removed from the front.
- inserted and removed from both ends.

Question No: 197

Each node in doubly link list has,

- 1 pointer
- 2 pointers(page 39)
- 3 pointers
- 4 pointers

AL-JUNAID INSTITUTE GROUP

Question No: 198

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.

Since it is an empty queue the front and rear are initialize to -1, so on insertion both the
pointers will change and point to 0.

Question No: 199

Compiler uses which one of the following to evaluate a mathematical equation,

► Binary Tree

► Binary Search Tree

► Parse Tree(page 274)

► AVL Tree

Question No: 200

If a complete binary tree has n number of nodes then its height will be,

► $\text{Log}_2 (n+1) - 1$(page 139)

► 2^n

► $\text{Log}_2 (n) - 1$

► $2^n - 1$

Question No: 201

If a complete binary tree has height h then its no. of nodes will be,

► Log (h)

► $2^{h+1} - 1$(page 324)

► Log (h) - 1

► $2^h - 1$

Question No: 202

A binary relation R over S is called an equivalence relation if it has following property(s)

► Reflexivity

► Symmetry

► Transitivity

► All of the given options (page 387)

Question No: 203

Binary Search is an algorithm of searching, used with the data.

► Sorted (page 432)

► Unsorted

► Heterogeneous

AL-JUNAID INSTITUTE GROUP

► Random

Question No: 204

If there are N elements in an array then the number of maximum steps needed to find an

element using Binary Search is .

► N

► N^2

► $N\log_2 N$

► $\log_2 N$ (page 440)

Question No: 205

Use of binary tree in compression of data is known as .

► Traversal

► Heap

► Union

► Huffman encoding (page 287)

Question No: 206

While building Huffman encoding tree the new node that is the result of joining two nodes has

the frequency.

► Equal to the small frequency

► Equal to the greater

► Equal to the sum of the two frequencies (page 293)

► Equal to the difference of the two frequencies

Question No: 207

Which of the following statements is correct property of binary trees?

► A binary tree with N internal nodes has N+1 internal links.

► A binary tree with N external nodes has 2N internal nodes.

► A binary tree with N internal nodes has N+ 1 external node. (page 303)

► None of above statement is a property of the binary tree.

Question No 208

Which of the following is a property of binary tree?

► A binary tree of N external nodes has N internal node.
► A binary tree of N internal nodes has N+ 1 external node. (page 303)
► A binary tree of N external nodes has N+ 1 internal node.
► A binary tree of N internal nodes has N- 1 external node.

Question No: 209
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 PREOREDR successor.

AL-JUNAID INSTITUTE GROUP

► 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.
(Page 307)

► 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.

Question No: 210
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 PREOREDR successor.

► 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 predecessor.

► 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 predecessor.

Question No: 211

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 successor.

► levelorder
► Preorder
► Inorder
► Postorder

Question No: 212

Which of the following statement is true about dummy node of threaded binary tree?

► This dummy node never has a value.
► This dummy node has always some dummy value.
► This dummy node has either no value or some dummy value. .(page 321)
► This dummy node has always some integer value.

Question No: 213

A complete binary tree is a tree that is filled, with the possible exception of the bottom level.

► partially
► completely (page 323)
► incompletely

AL-JUNAID INSTITUTE GROUP

► partly

Question No: 214

A complete binary tree of height 3 has between _ nodes.

► 8 to 14
► 8 to 15 (page 124)
► 8 to 16

► 8 to 17

Question No: 215

We can build a heap in _ time.

► Linear (page 353)

► Exponential

► Polynomial

► None of the given options

Question No: 216

Suppose that a selection sort of 100 items has completed 42 iterations of the main loop.

How many items are now guaranteed to be in their final spot (never to be moved again)?

► 21

► 41

► 42

► 43

Question No: 217

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 It's not satisfy the min heap property

Question No: 218

Which of the following statement is NOT correct about find operation:

► It is not a requirement that a find operation returns any specific name, just that finds on two elements return the same answer if and only if they are in the same set.

► One idea might be to use a tree to represent each set, since each element in a tree has the same root, thus the root can be used to name the set.

► Initially each set contains one element.

► Initially each set contains one element and it does not make sense to make a tree of

one node only.

Question No: 219

(Consider the following infix
expression:

x – y a + b / c

AL-JUNAID INSTITUTE
GROUP

Which of the following is a correct equivalent expression(s) for the above?

► x y -a * b +c /

► x *y a - b c / +

► x y a * - b c / + Hint :- (x – y * a) +(b / c)

► x y a * - b/ + c

Question No: 220

A complete binary tree of height has nodes between 16 to 31 .

► 2

► 3

► 4 (page 124)

► 5

Question No: 221

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.