Programming style

Procedural programming is a paradigm based on the concept of using procedures. Procedure (sometimes also called subprogram, routine or method) is a sequence of commands to be executed. Any procedure can be called from any point within the general program, including other procedures or even itself

*re-usability of pieces code designed as procedures
*ease of following the logic of program;
*Maintainability of code.
*Emphasis is on doing things (algorithms).
*Most of the functions share global data.
*Data move openly around the system from function to function.
*Functions transform data from one form to another.

Modular Programming is progressively decomposed into smaller partition called modules. The program can easily be written in modular form, thus allowing an overall problem to be decomposed into a sequence of individual sub programs. Thus we can consider, a module decomposed into successively subordinate module. Conversely, a number of modules can combined together to form a superior module.

*Define the problem
*Outline the steps needed to achieve the goal
*Decompose the problem into subtasks
*Prototype a subprogram for each sub task
*Repeat step 3 and 4 for each subprogram until further decomposition seems counter productive

Advantages

*Reduce the complexity of the entire problem *Avoid the duplication of code
*debugging program is easier and reliable
*Improves the performance
*Modular program hides the use of data structure *Global data also hidden in module
*Reusability- modules can be used in other program without rewriting and retesting
*Modular program improves the portability of program
*It reduces the development work

Top- down modular programming

The principles of top-down design dictate that a program should be divided into a main module and its related modules. Each module should also be divided into sub modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary process that are intrinsically understood and cannot be further subdivided.

Bottom-up algorithm design is the opposite of top-down design. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written.

Structured Programming

It is a programming style; and this style of programming is known by several names: Procedural decomposition, Structured programming, etc. Structured programming is not programming with structures but by using following types of code structures to write programs:

*Sequence of sequentially executed statements
*Conditional execution of statements (ie., "if statements)
*Looping or iteration (ie., "for, do...while, and while" statements)
*Structured subroutine calls {ie., functions)

Advantages

1. clarity: structured programming has a clarity and logical pattern to their control

structure and due to this tremendous increase in programming productivity
2. another key to structured programming is that each block of code has a single entry point and single exit point.so we can break up long sequence of code into modules
3. Maintenance: the clarity and modularity inherent in structured programming is of great help in finding an error and redesigning the required section of code.

0bect Oriented Programming

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural or modular approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function.

FEATURES

•Emphasis is on data rather than procedure.
•Programs are divided into what are known as objects.
•Data structures are designed such that they characterize the objects.
•Functions that operate on the data of an object are ties together in the data structure.
•Data is hidden and cannot be accessed by external function.
•Objects may communicate with each other through function.
•New data and functions can be easily added whenever necessary.
•Follows bottom up approach in program design

Array

*Elements are always stored at continuous / contiguous memory locations.
*No need to store the address of the next element.
*Uses the concept of Static Memory Allocation.
*All the required memory needs to be allocated in advance.
*Has a fixed dimension. So cannot grow or shrink at run time.
*Operations like Insertion and Deletion are time-consuming.
*They might require major shifting of elements.

Linked List

*Elements might be stored at discontinuous memory locations.
*Need to store the address of the next element along with the value.
*Uses the concept of Dynamic Memory Allocation.
*Memory can be allocated on demand and not in advance.
*Can always grow or shrink at run time.
*Operations like Insertion and Deletion are simple and less time-consuming.
*Shifting of elements is not required. Only adjustment of links are required.

Frequency count method can be used to analyze a program .Here we assume that every statement takes the same constant amount of time for its execution.

Insert An Element Into The Beginning Of An Array

```
#include<stdio.h> ,#include<conio.h>
int main() ,{ ,int a[10],i,n,p;
printf(“Enter the number of elements\n”);
scanf(“%d”,&n);
printf(“Enter the Elements\n”);
for(i=0;i<n;i++) ,{
scanf(“%d”,&a[i]); ,} ,if(n==10)
{ ,printf(“Array is full”); ,} ,Else
{ ,printf(“Enter the element to be
inserted”)
scanf(“%d”,&p); ,for(i=n-1;i>=0;i--) ,{
a[i+1]=a[i]; ,} ,n=n+1 ,a[0]=p;
printf(“After insertion elements are:\n”);
for(i=0;i<n;i++) ,{ ,printf(“%d\t”,a[i]);
} ,} ,getch(); ,}
```

insert an element into the end of an array

```
#include<stdio .h> ,#include<conio. h>
int main() ,{ ,int a[10],i,n,p;
printf("Enter the number of elements\n");
scanf("%d",&n); ,printf("Enter
Elements\n"); ,for(i=0;i<n;i++) ,{
scanf("%d",&a[i]); ,} ,if(n==10) ,{
printf("Array is full"); ,} ,else ,{
printf("Enter the element to be inserted")
scanf("%d",&p); ,a[n+l]=p; ,n=n+l;
printf("After insertion elements are:\n");
for(i=0;i<n;i++) ,{ ,printf("%d\t",a[i]);
} ,} ,getchO; ,}
```

Insert An Element Into The Particular Position of an array

```
#include <stdio.h> ,int main()
{ ,int array[100], position, c, n, value;
printf("Enter number of elements in
array\n"); ,scanf("%d", &n);
printf("Enter %d elements\n", n);
for (c = 0; c < n; C++)
scanf("%d", &array[c]);
printf("Enter the location where you wish
to insert an element\n");
scanf("%d", &position); ,if(position>=100)
printf("Array is full") ,else ,{
printf("Enter the value to insert\n");
scanf("%d", &value);
for (c = n - 1; c >= position - 1; c~)
array[c+l] = array[c];
array[position-l] = value; ,n=n+l;
printf("Resultant array is\n");
for (c = 0; c <=n; C++)
printf("%d\n", array[c]); ,} ,getchO; ,}
```

Delete An Element From The Beginning Of An Array

```
#include<stdio.h> ,#include<conio.h>
int main() ,{ ,int a[10],i,n,p;
printf("enter the number of elements\n");
,scanf("%d",&n);
printf("enter the elements\n");
for(i=0;i<n;i++) ,{ ,scanf("%d",&a[i]);
} ,p=a[0]; ,for(i=l;i<n;i++) ,{
a[i-l]=a[i]; ,} ,n=n-l;
printf("deleted elements are %d",p);
if(n==0) ,printf("array is empty"); ,else ,{
printf("after deletion elements are:\n");
for(i=0;i<n;i++) ,{ ,printf("%d\t",a[i]);
} } ,getcho; ,}
```

Delete An Element From The End Of An Array

```
#include<stdio.h> ,#include<conio.h>
int main() ,{ ,int a[10],i,n;
printf("Enter the number of elements\n");
scanf("%d",&n); ,printf("Enter the
Elements\n"); ,for(i=0;i<n;i++) ,{
scanf("%d",&a[i]); ,} ,p=a[n];
printf("Deleted element is %d",p);
n=n-l; ,if(n==0) ,printf("Array is empty");
else ,{
printf("After deletion elements are:\n");
for(i=0;i<n;i++) ,{ ,printf("%d\t",a[i]);
} ,getchO; ,}
```

Delete An Element From The Particular Position Of An Array

```
#include <stdio.h> ,int main() ,{
int array[100], position, c, n, value;
printf("Enter number of elements in array\n");
scanf("%d", &n);
printf("Enter %d elements\n", n);
for (c = 0; c < n; C++) ,scanf("%d", &array[c]);
printf("Enter the location where you wish to
delete an element\n");scanf("%d", &position);
value=array[position-1 ];
for (c = position; c >n; c~) array[c-l] = array [c];
```

```
n=n-l;
printf("Deleted element is %d\n",value);
if(==0) ,printf("Array is empty"); ,else
{ ,printf("Resultant array is\n");
for (c = 0; c <=n; C++) ,printf("%d\n", array[c]);
} ,getchO; ,}
```

Vector APPLICATIONS

*Stores Elements of Same Data Type
*Used for Maintaining multiple variable names using single name
*Can be Used for Sorting Elements
*Can Perform Matrix Operation
*Can be Used in CPU Scheduling
*Can be Used in Recursive Function
*Can be used to implement abstract data structures like stack, queue etc

VECTOR is an array with a dynamic size.
*Instead of having a predefined size to the structure it increases, decreases its size as you add/remove elements from/ to it.
*Which together gives as some other advantages as adding elements at a specific index and removing from a specific index.
*Vector is not as fast as the array, but altogether efficient. Operations on a vector offer the same big O as their counterparts on an array.
*Like arrays, vector data is allocated in contiguous memory. This can be done either explicitly or by adding more data.
*In order to do this efficiently, the typical vector implementation grows by doubling its allocated space (rather than incrementing it) and often has more space allocated to it at any one time than it needs. This is because reallocating memory can sometimes be an expensive operation.

Analysis of Algorithm After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathe matical techniques to prove it correct

Space Complexity the amount of memory it needs to run to completion. Some of the reasons for studying space complexity are: 1. If the program is to run on multi user system, it may be required to specify the amount of memory to be allocated to the program. 2. We may be interested to know in advance that whether sufficient memory is available to run the program. 3. There may be several possible solutions with different space requirements. 4. Can be used to estimate the size of the largest problem that a program can solve.

The time complexity of an algorithm or a program is the amount of time it needs to run to completion.The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/ specifications and so on. it also depends on the amount of data input to an algorithm, but we can calculate the order of magnitude for the time required. That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used.

Binary Search Trees

Creation

void search(node *tree, int digit)
if(tree==NULL) then step 3 otherwise goto step 4
printf("The Number does not exists");
else if(digit==tree->num) then goto step 5 otherwise go to step 6
printf("%d is found",digit);
else if(digit<tree->num) then goto step 7 otherwise go to step 8 ,search(tree->left, digit) ,else search(tree->right,digit) ,End

Insertion

void insert(node *tree,mt digit)
if(tree==NULL) then step from 3 to 6 then goto step 6
tree=(node*)malloc(sizeof(node));
tree->left=tree->right=NULL;
tree->num=digit;
if(digit<tree->num) then goto step 7 otherwise step 8
tree->left=insert(tree->left,digit)
if(digit>tree->num) then goto step 9 otherwise goto step 10
tree->right=insert(tree->right,digit)
if(digit==tree->num)then goto step 11
print "Duplicate Nodes"
return(tree) ,End

Deletion

void deletenode(node *tree,mt digit)
{ ,struct node *r,*q; ,if(tree==NULL)
{ ,print "Tree is empty" ,exit(O); ,}
if(digit<tree->num)
deletenode(tree->left, digit);
else if(digit>tree->num)
deletenode(tree->right, digit); ,q=tree;
if((q->right==NULL)&&(q->left==NULL))
q=NULL; ,else if(q->right==NULL)
tree=q->left; ,else if (q->left==NULL)
tree=q->right; ,else ,delnum(q->left,digit);
free(q) ,}
delnum(int struct node *r,int digit) ,{
struct node * q; ,if(r->right!=NULL)
delnum(r->right,digit); ,Else
q->num=r->num; ,q=r; ,r=r->left; ,}

Algorithm for push and pop

Start ,Settop=-l
Read choice ch
If ch=l(PUSH), then goto 4.a else 4.b
If top<4 then
Read the item, set the top=top+l
Set stack[top]=item
Else Print stack overflow
If ch=2(P0P), then goto 5.a else 5.b
if top>0 then ,Set item=stack[top]
top=top-l
Print number deleted is item
Else Print stack overflow
If ch=3, then
Display the elements in the stack
If ch=4, then Exit ,Stop

PROGRAM

#inc lude<stdio. h> ,#include<conio.h>
void main() ,{
int stack[10],ch,i,item,top=-l;
clrscrO; ,while(l) ,{
printf("\nMENU:\nl.Push\n2.Pop\n3.Trav
erse\n4.Exit\nEnter your choice: ");
scanf("%d",&ch); ,switch(ch) ,{
case 1: ,if(top<=9) ,{
printf("\nStack overflow");
} ,else ,{ ,printf("\nEnter the item: ");
scanf("%d",&item); ,top=top+l;
stack[top]=item; ,} ,break; ,case 2:
if(top==-l) ,printf("\nStack underflow");
else ,{ ,item=stack[top]; ,top=top-l;
printf("\nDeleted item is %d\n",item);
} ,break; ,case 3: ,if(top>=0)
printf("\nNo elements in stack");
else ,{ ,printf("\nElements are: ");
for(i=top; i>=0; i~) ,printf(" %d",stack[i]);
} ,break; ,case 4: ,exit(O); ,} ,getchO; ,}}

First Fit; The simplest algorithm. The process manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible

Best Fit: it searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Worst Fit: Always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

Stack Using Linked LIST

Push & Pop

```
#include<stdio.h> ,#include<conio.h>
struct node ,{ ,int data; ,struct node *ptr;
}; ,void main() ,{
typedef struct node NODE;
NODE *top=NULL,*temp,*t;
int ch,item; ,clrscrO; ,while(l) ,{
printf("\nMENU:\nl.Push\n2.Pop\n3.Displ
ay\n4.Exit\nEnter your choice: ");
scanf("%d",&ch); ,switch(ch)
{ ,case 1:
temp=(NODE*)malloc(sizeof(NODE));
printf("\nEnter the item: ");
scanf("%d",&item); ,temp->data=item;
if(top==NULL) ,{ ,temp->ptr=NULL;
top=temp; ,} ,else ,{ ,temp->ptr=top;
top=temp; ,} ,break; ,case 2: if(top==NULL)
printf("\nDeletion is not possible");
else if(top->ptr==NULL) ,{ ,temp=top;
top=NULL; ,printf("\nPoped item is
%d",temp->data); ,free(temp); ,} ,else ,{
temp=top; ,top=top->ptr;
printf("\nPoped item is %d",temp->data);
free(temp); ,} ,break; ,case 3:
if(start==NULL) printf("\nStack is empty");
else ,{ ,printf("\nElements are:");
for(t=top;t! =NULL;t=t->ptr)
printf(" %d",t->data); ,} break; ,case 4:
exit(O); ,default: printf("\nWrongChoice");
break; } ,getchO; }}
```

Queue using array

```
#include<stdio.h> ,#include<conio.h>
#include<stdlib.h> ,int main()
{ ,int ch, front=-l, rear=-l, q[10], item, i;
while(l) ,{
printf("\n\t QUEUE\n\n 1. Insert \n 2.
Delete \n 3. Display \n 4. Exit\n Enter ur
choice: "); ,scanf("%d", &ch); ,switch(ch) ,{
case 1: ,if(rear==9) ,{ ,printf("Que is full");
} ,else ,{ ,if(front==-l && rear==-l)
{ ,front=0; rear=0; ,} ,else ,rear++;
printf("Enter the inserted item : ");
scanf("%d", &item); ,q[rear]=item; ,}
break; ,case 2:
if((front==-l && rear==-l) || front==rear+l)
{ ,printf("Queue is empty!"); ,} ,else
{ ,item=q[front]; ,front++;
printf("Deleted item is : %d", item); ,}
break; ,case 3:
if((front==-l && rear==-l) || front==rear+l)
{ ,printf("Queue is empty ! "); ,} ,else ,{
printf("\nElements are ,for(i=front;
i<=rear; i++) ,printf("%d \t", q[i]); ,} ,break;
case 4: ,exit(O); ,break; ,getchO;
```

Application of Stack

l.Stack Frames Programs compiled from C make use of a stack frame for the working memory of each procedure or flinction invocation. When any procedure or function is called, a number of words - the stack frame -is push onto a program stack. When the procedure or function returns, this frame of data is popped off the stack. For example, when a program sends parameters to a function, the parameters are placed ON THE STACK

2. Reversing a String As the characteristics of stack is reversing the order of execution. This fact can be exploited to reverse strings of line of characters. This can be simply thought of simply as pushing the individual characters, and when the complete line is pushed onto the stack, then individual characters of the line are popped off. Because the last inserted character pushed on stack would be the first character to be popped off, the line obviously be removed.

3. Converting INFIX to POSTFIX The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to right.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized.
3. The sub-expression which has been converted into postfix is to be treated as single operand
4. Once the expression is converted to postfix from remove the parenthesis.

Queue Using Linked List

```
#include<stdio.h> ,struct node ,{
int data; ,struct node *ptr; ,};
int main() ,{ ,typedef struct node NODE;
NODE *front=NULL,*rear=NULL, *temp,
*p; ,int i, ch, pos, item; ,while(l) ,{
printf("\n\n\tQueue Using Linked
List\nLInsert\n2.Delete\n3.Display\n4.Exit
\n\nEnter your choice: ");
scanf("%d", &ch); ,switch(ch) ,case 1:
temp=(NODE*)malloc((sizeof(NODE)));
printf("Enter the data to be inserted: ");
scanf("%d", &temp->data); if(rear==NULL)
{ ,temp->ptr=NULL; ,front=rear=temp;
} ,else ,{ ,p=front; ,while(p!=rear) ,{
p=p->ptr; ,} ,p->ptr=temp;
temp->ptr=NULL; ,rear=temp; ,} ,break;
case 2: ,if(front==NULL) ,{
printf("No elements to delete!");
} ,else ,{ ,if(front->ptr==NULL) ,{
temp=front; ,item=temp->data;
printf("Deleted element is %d", temp-
>data); ,free(temp); ,front=rear=NULL;
} ,else ,{ ,temp=front; ,item=temp->data;
front=front->ptr; ,free(temp);
printf("Deleted element: %d", item); ,} ,}
break; ,case 3: ,if(front==NULL) ,{
printf("No elements"); ,} ,else ,{
printf("\nElements are: ") ,p=front;
while(p!=rear) ,{ ,printf("%d" p->data);
p=p->ptr; ,} ,printf("%d", p->data); ,}
break; ,case 4: ,exit(O); ,} ,} ,getchO; ,}
```

Applications of Queue

1.Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life. Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive. First come first served.

Double ended Queue

Algorithm for Insertion at rear end

```
Step -1; [Check for overflow]
if(rear==MAX)
Print("Queue is Overflow"); ,return;
Step-2: [Insert element] ,else ,rear=rear+l;
q[rear]=no; ,[Set rear and front pointer]
if rear=-l ,rear=front=0; ,Step-3: return
```

Algorithm for Insertion at font end

```
Step-1 : [Check for the front position]
if(front==0)
Print ("Cannot add item at front end");
return; ,Step-2 : [Insert at front] ,else
front=front-l; ,q[front]=no; Step-3 : Return
```

Algorithm for Deletion from front end

```
Step-1 [ Check for front pointer]
if front=-l ,print(" Queue is Underflow");
return; ,Step-2 [Perform deletion] ,else
no=q[front];
print("Deleted element is",no);
[Set front and rear pointer] ,if front=rear
front=rear=-l; ,else ,front=front+l;
Step-3 : Return
```

Algorithm for Deletion from rear end

```
Step-1 : [Check for the rear pointer]
if rear=0
print("Cannot delete value at rear end");
return; ,Step-2: [ perform deletion] ,else
no=q[rear]; ,[Check for the front and rear
pointer] ,if front= rear ,front=rear=-l; ,else
rear=rear-l; ,print("Deleted element
is",no); ,Step-3 : Return
```

Multiple Stacks

```
#include<stdio.h> ,#include<stdlib.h>
void main() ,{
int n,topl,top2,ch=l,a,i,arr[100];
printf("Enter size of array you want to
use\n"); ,scanf("%d",&n); ,topl=-l;
top2=n/2; ,while(ch!=0) ,{
```

```
printf("What do u want to do?\n");
printf(" 1 .Push element in stack l\n");
printf("2.Push element in stack 2\n");
printf("3.Pop element from stack l\n");
printf("4.Pop element from stack 2\n");
printf("5.Display stack l\n");
printf("6.Display stack 2\n");
printf("O.EXIT\n"); ,scanf("%d",&ch);
switch(ch) ,{
case 1: ,{ ,printf("Enter the element\n");
scanf("%d",&a); ,if(topl==n/2-l)
printf("Stack 1 is full"); ,else ,{ ,topl++;
arr[topl]=a; ,} ,break; ,} ,case 2: ,{
printf("Enter the element\n");
scanf("%d",&a); ,if(top2==n)
printf("Stack 2 is MM"); ,else ,{ ,top2++;
arr[top2]=a; ,} ,break; ,} ,case 3: ,{
if(topl==-l) ,printf("Stack 1 is empty\n");
else ,{ ,a=arr[topl]; ,topl-s
printf("%d\n",a); ,} ,break; ,} ,case 4: ,{
if(top2==n/2) ,printf("Stack2 is empty\n");
else ,{ ,a=arr[top2]; ,top2~;
printf("%d\n",a); ,} ,break; ,} ,case 5: ,{
if(topl==-l) ,printf("Stack 1 is empty\n");
else ,{ ,printf(" Stack 1 is~»\n");
for(i=0;i<=top 1 ;i++) ,printf("%d ",arr[i]);
printf("\n"); ,} ,break; ,} ,case 6: ,{
if(top2==n/2) ,printf("Stack2 is empty\n");
else ,{ ,printf("Stack2 is~»\n");
for(i=(n/2+1); i<=top2; i++)
printf("%d ",arr[i]); ,printf("\n");
} ,break; ,} ,case 0: ,break; }}}
```

Multiple Queues

Insert into First Queue

```
if(rearl=n/2-l) ,{
printf("First Queue is full\n"); ,} ,else ,{
printf("Enter the element to be inserted");
scanf("%d",&item);
If((front 1 ==-1 )&&(rear 1 ==-1)) ,{
front l=rear 1=0; ,} ,else ,{ ,rearl++; ,}
Q[rearl]=item; ,}
```

Insert into Second Queue

```
if(rear2=n/-l) ,{
printf("Second Queue is full\n"); ,} else {
printf("Enter the element to be inserted");
scanf("%d",&item);
If((front2==n/2-1 )&&(rear 1 ==n/2-1)) ,{
front2=rear2=n/2; ,} ,else ,{ ,rear2++;
} ,Q[rear2]=item; ,}
```

Delete from First Queue

```
if(frontl==-l) ,{
printf("First Queue is empty\n"); } ,else
{ ,item=Q[frontl]; ,printf("The deleted
element in the first queue is %d",item);
If((front 1 ==rear 1) ,{ ,front l=rearl=-l;
} ,else ,{ ,front 1++; ,} ,}
```

Delete from Second Queue

```
if(front2==n/2-1) ,{
printf("Second Queue is empty\n"); ,} else
{ ,item=Q[front2];
printf("The deleted element in the first
queue is %d",item); ,If((front2==rear2)
{,front2=rear2=n/2-1;,} ,else ,{ ,front2++;}}
```

Abstract Data Structures

*Concrete data types or structures (CDT's) are direct implementations of a relatively simple concept

*Data types which are absolutely defined

*Array, records, linked lists, trees, graphs

Concrete Data Structures

*Abstract Data Types (ADT's) offer a high level view (and use) of a concept independent of its implementation

*constructed from known -or unknown data

*stacks, queues and heaps

Insert Element Into The Beginning Of An Array

```
#include<stdio.h> ,#include<conio.h>
int main() ,{ ,int a[10],i,n,p;
printf("Enter the number of elements\n");
scanf("%d",&n); ,printf("Enter the
Elements\n"); ,for(i=0;i<n;i++) ,{
scanf("%d",&a[i]); ,} ,if(n==10)
{ ,printf("Array is full"); ,} ,else ,{
printf("Enter the element to be inserted")
scanf("%d",&p); ,for(i=n-l;i>=0;i—) ,{
a[i+l]=a[i]; ,} ,n=n+l ,a[0]=p;
printf("After insertion elements are:\n");
for(i=0;i<n;i++) ,{ ,printf("%d\t",a[i]); ,}
} ,getchO; ,}
```

BUBBLE SORT

Start ,i=0 ,While i<n-l ,j=0
While j<n-l-I ,If a[j]>a[j+l]
temp=a[j] ,a[j]=a[j+l]
a[j+l]=temp ,end if ,j=j+l
end while ,i=i+l ,end while

SELECTION SORT

Start ,i=0 ,while i<n-l do ,j=i+l
small=I ,while j<n do ,if a[small]>a[j]
small=j ,end if ,j=j+l ,end while ,if i!=small
temp=a[i] ,a[i]=a[small] ,a[small]=temp
end if ,i=i+l ,end while ,stop

INSERTION SORT

Start ,i=l ,While i<n ,j=I ,While j>0
t=a[j] ,a[j]=a[j-l] ,a[j-l]=t ,j=j-l
end while ,i=i+l ,end while ,stop

mergesort(start,end)

start ,if(start!= end) ,mid=(start+end)/2
mergesort(start,mid) ,mergesort(mid+1
,end) ,merge(start,mid,end) ,end if ,stop

Merge(start,mid,end)

i=start ,j=mid+l ,k=start
while i<=mid and j<= end do ,ifa[i]<=a[j]
temp[k]=a[i] ,i=i+l ,k=k+l ,Else
Temp[k] =a[j] ,J=j+1 ,k=k+l ,end if
end while ,while i<=mid do ,temp[k]=a[i]
i=i+l ,k=k+l ,end while ,While j<=end
Temp[k]=a[j] ,j=j+l ,k=k+l ,end while
k=start ,while k<=end ,a[k]=temp[k]
k=k+l ,end while ,stop

Quicksort(lb,ub)

Start ,iflb<ub ,loc=partition(lb,ub)
quicksort(lb,loc-1) ,quicksort(loc+1 ,ub)
end if ,stop

Partition(lb,ubl)

pivot=a[lb] ,up=ub ,down=lb
while down< up
while pivot>=a[down] and down<=up
down=down+1 ,end while
while a[up]>pivot ,up=up-l ,end while
if down< = up ,swap(a[down], a[up])
end if ,end while ,swap(a[lb],a[up])
return up ,stop

LINEAR SEARCH

Start ,i=0 ,flag=0 ,While i<n and flag=0
If a[i]=key ,Flag=l ,Index=I ,end if
i=i+l ,end while ,if flag=l
print "the key is found at location index"
else ,print "key is not found" ,end if ,stop

BINARY SEARCH

Start ,Start=0,end=n-1
Middle=(start+end)/2
While key!=a[middle] and start<end
If key>a[middle]
Start=middle+1 ,else ,end= middle-1
end if ,middle=(start+end) / 2 ,end while
if key=a[middle]
print "the key is found at the position"
else ,print "the key is not found" ,end if
stop

Concatenate Two String With Using String Functions

#include <stdio.h> ,#include <string.h>

Algorithm Name	Best Case	Average Case	Worst Case
Quick	0(n log n)	0(n log n)	O(n')
Merge	0(n log n)	0(n log n)	0(n log n)
Heap	0(n log n)	0(n log n)	0(n log n)
Bubble	0(n)	O(n')	O(n')
Selection	O(n')	O(n')	O(n')
Insertion	0(n)	O(n')	O(n')
Binary	o(i)	0(log n)	0(log n)
Linear	o(i)	0(n)	0(n)

int main() ,{ ,chara[100],b[100];
printf("Enter the first string\n");
printf("Enter the second string\n");
gets(b); ,strcat(a,b);
printf("String obtained on concatenation
is %s\n",a);

Concatenate Two String Without Using String Functions

#include<stdio. h> ,#include<string.h>
int main() ,{ char si[50], s2[30];
int i=0,j=0; ,printf("\nEnter String 1
gets(sl); ,printf("\nEnter String 2
gets(s2); ,while(sl[i]!='\0') ,{ i++; }
while(s2|j]!='\0') ,{ sl[i]=s2D-];
i++; j++; } ,sl[i] = '\0';
printf("nConcated string is :%s", si); }

Substring searching

```
#include<stdio. h> ,#include<string.h>
int main() ,{ ,charsl[20],s2[20]; ,intl,j,m,l,ll;
printf("Enter the 1ST string"); ,gets(sl);
printf("Enter the 2ND string"); ,gets(s2);
l=strlen(sl); ll=strlen(s2); for(i=0;l<l;i++)
{ ,j=0; ,if(sl[i]==s2U]) ,{ ,m=i; ,sign=0;
while((s2[j]!='\0')&&(sign!=l)) ,{
if(sl[m]==s2|j]) ,{ ,m++; ,j++; ,} ,else
sign=l; ,}}} ,if(sign==0) ,{
printf("The given substring is present in
the location %d",i+l); ,} ,else ,{
printf("Substring not present"); } }
```

Substring Deletion

```
#include<stdio.h> ,#include<string.h>
int main() ,{ ,charsl[20],s2[20]; ,intl,j,m,l,ll;
printf("Enter the 1st string"); ,gets(sl);
printf("Enter the 2Nd string"); ,gets(s2);
l=strlen(sl); ,ll=strlen(s2); ,for(i=0;l<l;i++) ,{
j=0; ,if(sl[i]==s2U]) ,{ ,m=i; ,sign=0;
while((s2[j]!='\0')&&(sign!=l)) ,{
if(sl[m]==s2[j]) ,{ ,m++; ,j++; ,} ,Else
sign=l; },},} ,if(sign==0) ,{
printf("The given substring is present in
the location %d",i+l); ,m=i+ll;
,whiIe(sl[m]!-\0') { ,sl[i]=sl[m]; ,i++; ,m++;
} ,sl[i]='\0'; printf("Substring after deletion
is"); ,puts(sl); ,} ,else
{ ,printf("Substring is not present"); ,}}
```

Tree

collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

1.Root: the first node is called as Root Node. Every tree must have root node

2.Edge the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of'N-1' number of edges.

3.Parent node which is predecessor of any

4.Child node which is descendant of any

5.Siblings nodes which belong to same Parent

6.Leaf/Terminal Node the node which does not have a child . A leaf is a node with no child. The degree of a node is zero then it is a leaf

7.Internal Nodes/Non Terminal Nodes the node which has atleast one Child.

8.Degree the total number of children of a node. The Degree of a node is total number of children it has

9.Level the root node is said to be at Level 0 and the children of root node are at Level 1 and and so on...

10.Height the total number of egdes from leaf node to a particular node in the longest ,height of all leaf nodes is '0'.

11.Depth the total number of egdes from root node to a particular ,the total number of edges from root node to a leaf node in the longest path

12.Path sequence of Nodes and Edges from one node to another

13.Sub Tree each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node

Advantages of trees

*Trees are so useful and frequently used, because they have some very serious advantages:

*reflect structural relationships in the data

*are used to represent hierarchies

*provide an efficient insertion and searching

*are very flexible data, allowing to move subtrees around with minumum effort

Application of tree

*Manipulate hierarchical data.

*Make information easy to search

*Manipulate sorted lists of data.

*As a workflow for compositing digital images for visual effects.

*Router algorithms *Windows file system

Singly Linked List

1. Insert at beginning
2. Insert at particular position
3. Insert at end
4. Delete from beginning
5. Delete from particular position
6. Delete from end
7. Display

```
#include<stdio.h> ,#include<conio.h>
#include<stdlib.h> ,struct node ,{
int data; ,struct node*ptr; ,};
void main() ,{
typedef struct node NODE;
NODE *start=NULL,*temp,*p,*t;
int ch,item,pos,i; ,clrscr(); ,while(1)
{ ,printf("\nMENU: \n1.Insert at
beginning\n2.Insert at particular
position\n3.Insert at end\n4.Delete from
beginning\n5.Delete from particular
position\n6.Delete from
end\n7.Display\n8.Exit\nEnter your
choice: "); ,scanf("%d",&ch);
switch(ch) ,{
case 1: ,printf("\nEnter the number: ");
scanf("%d",&item);
temp=(NODE*)malloc(sizeof(NODE));
temp->data=item; ,if(start==NULL) ,{
temp->ptr=NULL; ,start=temp; ,} ,else
{ ,temp->ptr=start; ,start=temp;
} ,break; ,case 2:
printf("\nEnter the number: ");
scanf("%d",&item);
temp=(NODE*)malloc(sizeof(NODE));
temp->data=item;
printf("\nEnter the position: ");
scanf("%d",&pos); ,p=start;
for(i=1;i<pos-1;i++) ,p=p->ptr;
temp->ptr=p->ptr; ,p->ptr=temp;
break; ,case 3:
printf("\nEnter the number: ");
scanf("%d",&item);
temp=(NODE*)malloc(sizeof(NODE));
temp->data=item; ,temp->ptr=NULL;
if(start==NULL) ,start=temp; ,else
{ ,p=start; ,while(p->ptr!=NULL)
p=p->ptr; ,p->ptr=temp; ,} ,break;
case 4: ,if(start==NULL)
printf("\nDeletion is not possible");
else if(start->ptr==NULL) ,{
temp=start; ,start=NULL;
printf("\nDeleted item is %d",temp-
>data); ,free(temp); ,} ,else ,{
temp=start; ,start=start->ptr;
printf("\nDeleted item is %d",temp-
>data);
free(temp); ,} ,break; ,case 5:
printf("\nEnter the position: ");
scanf("%d",&pos); ,temp=start;
for(i=1;i<pos-1;i++) ,temp=temp->ptr;
t=temp->ptr; ,temp->ptr=t->ptr;
printf("\nDeleted item is %d",t->data);
free(t); ,break; ,case 6: ,if(start==NULL)
printf("\nDeletion is not possible");
else if(start->ptr==NULL) ,{ ,temp=start;
start=NULL; ,printf("\nDeleted item is
%d",temp->data); ,free(temp); ,} ,else
{ ,temp=start; ,t=start->ptr;
while(t->ptr!=NULL) ,{ ,t=t->ptr;
temp=temp->ptr; ,} ,temp->ptr=NULL;
printf("\nDeleted item is %d",t->data);
free(t); ,} ,break; ,case 7: ,if(start==NULL)
printf("\nList is empty"); ,else ,{
printf("\nElements are:");
for(p=start;p!=NULL;p=p->ptr)
printf(" %d",p->data); ,}
break; ,case 8: ,exit(0); ,defaut:
printf("\nWrong Choice"); ,break; ,}
getch(); ,} ,}
```

Big-Ω notation

f(n) = Ω(g(n)) iff there are two positive constants c and n0 such that |f(n)| ≥ c |g(n)| for all n ≥ n0 .If f(n) is nonnegative, we can simplify the last condition to 0 ≤ c g(n) ≤ f(n) for all n ≥ n0 • then we say that "f(n) is omega of g(n)." • As n increases, f(n) grows no slower than g(n). In other words, g(n) is an asymptotic lower bound on f(n)

Big-Θ notation

f(n) = Θ(g(n)) iff there are three positive constants c1, c2 and n0 such that c1|g(n)| ≤ |f(n)| ≤ c2|g(n)| for all n ≥ n0 . If f(n) is nonnegative, we can simplify the last condition to 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n) for all n ≥ n0 .then we say that "f(n) is theta of g(n)." . As n increases, f(n) grows at the same rate as g(n). In other words, g(n) is an asymptotically tight bound on f(n).

Big-O Notation

f(n) = O(g(n)) iff there are two positive constants c and n0 such that |f(n)| ≤ c |g(n)| for all n ≥ n0 . If f(n) is nonnegative, we can simplify the last condition to 0 ≤ f(n) ≤ c g(n) for all n ≥ n0 . then we say that "f(n) is big-O of g(n)." . As n increases, f(n) grows no faster than g(n). In other words, g(n) is an asymptotic upper bound on f(n).

Big-0 notation

f(n) = ©(g(n)) iff there are three positive constants Ci, C2 and no such that cl I g(n) I < I f(n) I < c2 | g(n) | for all n > no . If f(n) is nonnegative, we can simplify the last condition to 0 < cl g(n) < f(n) < c2 g(n) for all n > no .then we say that "fl(n) is theta of g(n)." . As n increases, f(n) grows at the same rate as g(n). In other words, g(n) is an asymptotically tight bound on f(n).

Dequeue

#include<stdio.h> ,#include<conio.h>
#include<stdlib.h> ,int main() ,{
int ch, front=-1, rear=-1, q[10], item, i;
while(1) ,{
printf("\n\t QUEUE\n\n 1. Insert \n 2. Delete \n 3. Display \n 4. Exit\n Enter ur choice: "); ,scanf("%d", &ch); ,switch(ch) { ,case 1: ,if(rear==9) ,{
printf("Queue is full ! "); ,} ,else ,{
if(front==-1 && rear==-1) ,{
front=0; rear=0; ,} ,else ,rear++;
printf("Enter the inserted item : ");
scanf("%d", &item); ,q[rear]=item; ,}
break; ,case 2:
if((front==-1 && rear==-1) ||
front==rear+1) ,{
printf("Queue is empty!"); ,} ,else ,{
item=q[front]; ,front++;
printf("Deleted item is : %d", item); ,} ,}
break; ,case 3:
if((front==-1 && rear==-1) ||
front==rear+1) ,{
printf("Queue is empty ! "); ,} ,else ,{
printf("\nElements are :");
for(i=front; i<=rear; i++)
printf("%d \t", q[i]); ,} ,break; ,case 4:
exit(0); ,break; ,} ,} ,getch(); ,}

SEQUENTIAL SEARCH

1. Start ,2. i=0 ,3. flag=0
4. While i<n and flag=0 ,1. If a[i]=key
1. Flag=1 ,2. Index=I ,2.end if ,3. i=i+1
5. end while ,6. if flag=1
1. print "the key is found at location index" ,7. Else ,1.print "key is not found"
7.end if ,8. stop

Binary Search

1. Start ,2. Start=0,end=n-1
3. Middle=(start+end)/2
4. While key!=a[middle] and start<end
1. If key>a[middle] ,1. Start=middle+1
2.else ,1. end= middle-1 ,3. end if
4. middle=(start+end)/2 ,5. end while
6. if key=a[middle]
1. print "the key is found at the position"
7. else ,1. print "the key is not found"
8. end if ,9. stop

Doubly Linked List.

1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Display

```
#include<stdio.h> ,#include<conio.h>
struct node ,{ ,int data;
struct node *prev,*next; ,}; ,void main()
{ ,int ch,no;
typedef struct node NODE;
NODE*head=NULL,tail=NULL,*temp,*p,*t;
clrscr(); ,while(1) ,{
printf("\nMENU\n1.Insert at
beginning\n2.Insert at end"
"\n3.Delete from beginning\n4.Delete
from end\n5.Display\n6.Exit");
printf("\nEnter your choice: ");
scanf("%d",&ch); ,switch(ch) ,{ ,case 1:
temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: "); ,scanf("%d",&no);
temp->data=no; ,if(start==NULL) ,{
temp->prev=NULL; ,temp->next=NULL;
head=tail=temp; ,} ,else ,{
temp->next=head; ,head->prev=temp;
temp->prev=NULL; ,head=temp; ,} ,break;
case 2:
temp=(NODE*)malloc(sizeof(NODE));
printf("Enter the no: "); ,scanf("%d",&no);
temp->data=no; ,if(start==NULL) ,{
temp->prev=NULL; ,temp->next=NULL;
head=tail=temp; ,} ,else ,{
tail->next=temp; ,temp->prev=tail;
temp->next=NULL; ,tail=temp; ,} ,break;
case 3: ,if(head==NULL) ,{
printf("Deletion is not possible"); ,}
else if(head->next==NULL) ,{ ,temp=start;
head=tail=NULL; ,printf("Deleted element
is: %d",temp->data); ,free(temp); ,} ,else
{ ,temp=head; ,head=temp->next;
head->prev=NULL; ,printf("Deleted
element is: %d",temp->data); ,free(temp);
} ,break; ,case 4: ,if(head==NULL) ,{
printf("Deletion is not possible"); ,}
else if(head->next==NULL) ,{ ,temp=start;
head=tail=NULL; ,printf("Deleted element
is: %d",temp->data); ,free(temp); ,} ,else
{ ,temp=tail; ,tail=tail->prev; ,free(temp);
} ,break; ,case 5: ,if(start==NULL) ,{
printf("No elements"); ,} ,else ,{
printf("\nElements are:");
for(p=head;p!=NULL;p=p->next) ,{
printf(" %d",p->data); ,} ,} ,break; ,case 6:
exit(0); ,} ,getch(); ,} ,}
```

CIRCULAR LINKED LIST

Inserting a node at the beginning

```
struct node ,{ ,int data; ,struct node *next;
}; ,typedef struct node NODE;
NODE *head=NULL; ,NODE *tail=NULL;
NODE *temp; ,temp=(struct
NODE*)malloc(sizeof(NODE));
printf("Enter the element to be inserted")
scanf("%d",&item); ,temp->data=item;
if(head==NULL) ,{ ,head=tail=temp;
tail->next=head; ,} ,else ,{
temp->next=head; ,head=temp;
tail->next=head; ,}
```

Inserting a node at the End

```
NODE *temp;
temp=(structNODE*)malloc(sizeof(NODE);
printf("Enter the element to be inserted")
scanf("%d",&item); ,temp->data=item;
if(head==NULL) ,{ ,head=tail=temp;
tail->next=head; ,} ,else ,{tail->next=temp;
tail=temp; ,tail->next=head; ,}
```

Delete a node from the beginning

```
if(head==NULL) //List is empty
printf(Deletion is not possible\n"); ,}
else if(head==tail)//List contains only 1
node ,{ ,free(head); ,head=tail=NULL;
} ,else ,{ ,temp=head; ,head=head->next;
tail->next=head; ,free(temp);
```

Delete a node from the End

```
if(head==NULL)//List is empty
{ ,printf(Deletion is not possible\n"); ,}
else if(head==tail)//List contains only one
node ,{ ,free(head); ,head=tail=NULL;
} ,else ,{ ,p=head; ,while(p->next! =tail)
{ ,p-p->next; ,} ,temp=p->next; ,tail=p;
tail->next=head; ,free(temp); ,}
```

Algorithm for Polynomial Addition

1.Read the number of terms in the first polynomial P 2.Read the coefficient and exponent of the first polynomial
3.Read the number of terms in the second polynomial Q 4.Read the coefficient and exponent of the second polynomial
5.Set the temporary pointers p and q to travers the two polynomials respectively
6.Compare the exponents of two polynomials starting from the first nodes
a)If both exponents are equal then add the coefficient and store it in the resultant linked list b)If the exponent of the current term in the first polynomial P is less than the exponent of the current term of the second polynomial then added the second term to the resultant linked list. And, move the pointer q to point to the next node in the second polynomial Q.
c)If the exponent of the current term in the first polynomial P is greater than the exponent of the current term in the second polynomial Q, then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.
d) Append the remaining nodes of either of the polynomials to the resultant linked list.

Algorithm for Polynomial Multiplication

1.Read the number of terms in the first polynomial 2.Read the coefficient and exponent of the first polynomial
3.Read the number of terms in the second polynomial 4.Read the coefficient and exponent of the second polynomial
5.if one of the list is empty then the nonempty linked list is added to the resultant linked list Otherwise goto step 6.
6.for each term of the first list
a)multiply each term of the second linked list with a term of the first linked list
b)add the new term to the resultant polynomial c)reposition the pointer to the starting of the second linked list
d)go to the next node
e)adds a term to the polynomial in the descending order of the exponent
7.Display the resultant linked list

Stepwise Refinement Techniques

1. In the first stage, modeling, we try to represent the problem using an appropriate mathematical model such as a graph, tree etc. At this stage, the solution to the problem is an algorithm expressed very informally.
2. At the next stage, the algorithm is written in pseudo-language (or formal algorithm) that is, a mixture of any programming language constructs and less formal English statements. The operations to be performed on the various types of data become fixed.
3. In the final stage we choose an implementation for each abstract data type and write the procedures for the various operations on that type. The remaining informal statements in the pseudo-language algorithm are replaced by (or any programming language) C/C++ code.

MID SQUARE METHOD

Another hash function which has been widely used in many applications is the mid square method. The hash function H is defined by H(k)=x, where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. example
k : 1234 2345 3456
k^2 :1522756 5499025 11943936
H(k) : 525 492 933
For a three digit index requirement, after finding the square of key values,the digits at 2nd, 4th and 6th position are chosen as their hash values.

FOLDING METHOD

Another fair method for a hash function is folding method. In this method, the key k is partitioned into a number of parts k1 , k2..kn where each part has equal no.of digits as the required address(index)

width. Then these parts are added together in the hash function.
H(k)=k1+k2+…+kn. Where the last carry, if any is ignored. There are mainly two variations of this method.

Fold Shifting Method

In this method, after the partition even parts like k2, k4 are reversed before addition.

Fold boundary method

In this method, after the partition the boundary parts are reversed before addition

DIGIT ANALYSIS METHOD

This method is particularly useful in the case of static files where the key values of all the records are known in advance. The basic idea of this hash function is to form hash address by extracting and/or shifting the extracted digits of the key. For any given set of keys, the position in the keys and the same rearrangement pattern must be used consistently. The decision for extraction and rearrangement is finalized after analysis of hash functions under different criteria.

Closed Hashing

Suppose there is a hash table of size h and the key value is mapped to location i, with a hash function. The closed hashing can then be stated as follows.
Start with the hash address where the collision has occurred,let it be i. Then carry out a sequential search in the order:- i, i+1,i+2..h,1,2…,i-1 The search will continue until any one of the following occurs
*The key value is found
*An unoccupied location is found
*The searches reaches the location where search had started

Open Hashing

Closed hashing method for collision resolution deals with arrays as hash tables and thus random positions can be quickly refer red. Two main disadvantages of closed hashing are
1) It is very difficult to handle the problem of overflow in a satisfactory manner
2) The key values are haphazardly intermixed and, on the average majority of the key values are from their hash locations increasing the number of probes which degrades the overall performance

DFS

```
#include<stdio.h> ,#include<stdlib.h>
#define SIZE 100 ,int TOP=-1; ,int A[SIZE];
void push(int); ,int pop(); ,main() ,{
int VISIT[20],adjmat[20][20];
int n,i,j,v=0;
printf("Enter the no of vertices");
scanf("%d",&n);
printf("enter the adj metrix");
for(i=0,i<n,i++) ,{ ,for(j=0,j<n,j++) ,{
scanf("%d",&adjmat[i][j]; ,} ,}
if(n<=0) ,{
printf("the graph is empty"); ,} ,else
{ ,printf("The DFS Traversal is");
for(i=0,i<n,i++) ,{ ,VISIT[i]=0; ,}
push(v); ,while(TOP!=-1) ,{ ,v=pop();
if(VISIT[v]==0) ,{ ,printf("\t%d \n",v);
VISIT[v]=1; ,for(i=0,i<n,i++) ,{
if(adjmat[v][i]==1 && VISIT[i]==0)
push(i); ,},},},},} ,void push(int term) ,{
if(TOP>SIZE-1) ,{ ,printf("Stack is full");
else ,{ ,TOP=TOP+1; ,A[TOP]=item; } ,}
int pop() ,{ ,int item; ,if(TOP==-1) ,{
printf("Stack is empty"); ,return 0;
} ,else ,item=A[TOP]; ,TOP=TOP-1
return item; ,} ,}
```