UNIT I LINEAR DATA STRUCTURES – LIST

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation —singly linked lists- circularly linked lists- doubly-linked lists– applications of lists –Polynomial Manipulation – All operation (Insertion, Deletion, Merge, Traversal)

Data:

A collection of facts, concepts, figures, observations, occurrences or instructions in a formalized manner.

Information:

The meaning that is currently assigned to data by means of the conventions applied to those data(i.e. processed data)

Record:

Collection of related fields.

Data type:

Set of elements that share common set of properties used to solve a program.

Data Structures:

Data Structure is the way of organizing, storing, and retrieving data and their relationship with each other.
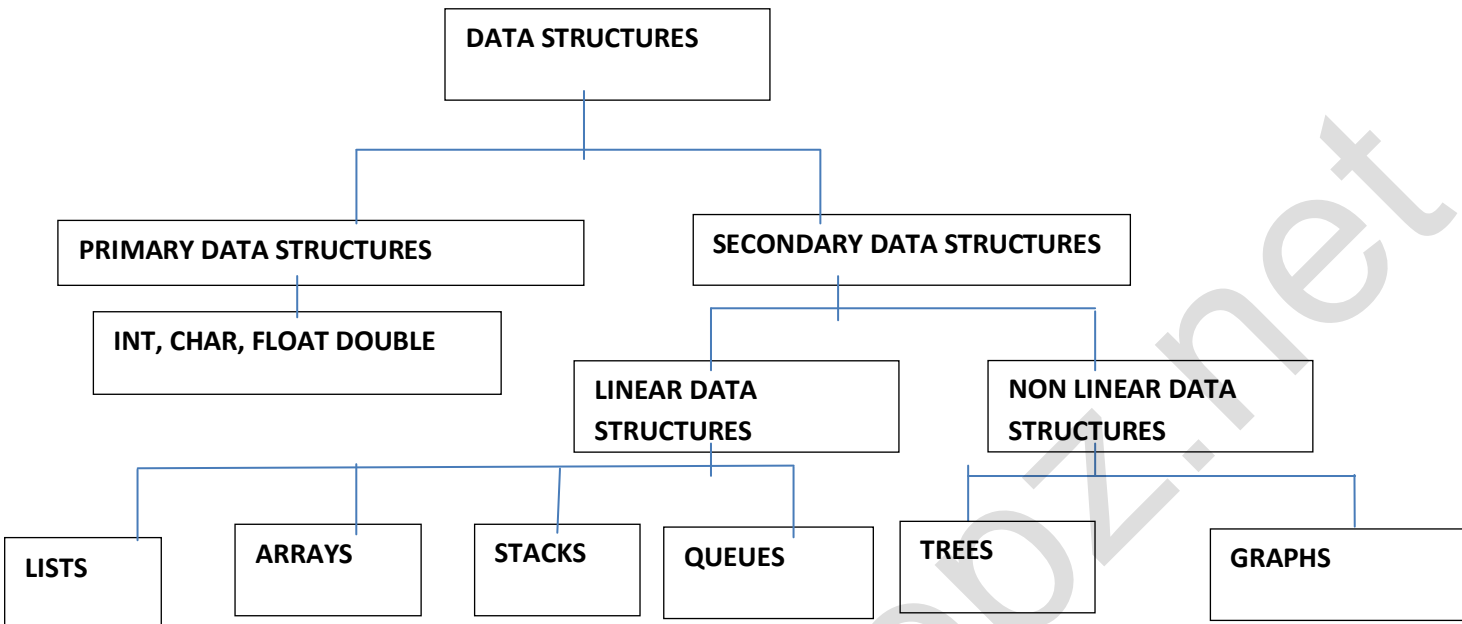
Characteristics of data structures:

1. It depicts the logical representation of data in computer memory.

2. It represents the logical relationship between the various data elements.

3. It helps in efficient manipulation of stored data elements.

4. It allows the programs to process the data in an efficient manner.

Operations on Data Structures:

1.Traversal

2.Search

3.Insertion

4.Deletion

CLASSIFICATION OF DATA STRUCTURES

```
                        ┌─────────────────────┐
                        │  DATA STRUCTURES    │
                        └─────────────────────┘
                                 │
              ┌──────────────────┴──────────────────┐
  ┌───────────────────────────┐         ┌───────────────────────────┐
  │ PRIMARY DATA STRUCTURES   │         │ SECONDARY DATA STRUCTURES │
  └───────────────────────────┘         └───────────────────────────┘
              │                                 │
  ┌───────────────────────────┐      ┌──────────┴──────────┐
  │ INT, CHAR, FLOAT DOUBLE    │  ┌──────────────┐   ┌──────────────┐
  └───────────────────────────┘  │ LINEAR DATA  │   │ NON LINEAR   │
                                  │ STRUCTURES   │   │ DATA         │
                                  └──────────────┘   │ STRUCTURES   │
                                                     └──────────────┘
```

| LISTS | ARRAYS | STACKS | QUEUES | TREES | GRAPHS |

Primary Data Strucutres/Primitive Data Structures:

Primitive data structures include all the fundamental data structures that can be directly

manipulated by machine-level instructions. Some of the common primitive data structures include integer, character, real, boolean etc

Secondary Data Structures/Non Primitive Data Structures:

Non primitive data structures, refer to all those data structures that are derived from one or more primitive data structures. The objective of creating non-primitive data structures is to form sets of homogeneous or heterogeneous data elements.

Linear Data Structures:

Linear data structures are data strucutres in which, all the data elements are arranged in i, linear or sequential fashion. Examples of data structures include arrays, stacks, queues, linked lists, etc.

Non Linear Structures:

In non-linear data strucutres, there is definite order or sequence in which data elements are arranged. For instance, a non-linear data structures could arrange data elements in a hierarchical fashion. Examples of non-linear data structures are trees and graphs.

Static and dynamic data structure:

Static Ds:

If a ds is created using static memory allocation, ie. ds formed when the number of data items are known in advance ,it is known as static data static ds or fixed size ds.

Dynamic Ds:

If the ds is created using dynamic memory allocation i.e ds formed when the number of data items are not known in advance is known as dynamic ds or variable size ds.

Application of data structures:

Data structures are widely applied in the following areas:

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Numerical analysis
- Simulation
- Artificial intelligence
- Graphics

ABSTRACT DATA TYPES (ADTS):

An abstract Data type (ADT) is defined as a mathematical model with a collection of operations defined on that model. Set of integers, together with the operations of union, intersection and set difference form a example of an ADT. An ADT consists of data together with functions that operate on that data.

Advantages/Benefits of ADT:

1.Modularity

2.Reuse

3.code is easier to understand

4.Implementation of ADTs can be changed without requiring changes to the program that uses the ADTs.

THE LIST AI)T:

List is an ordered set of elements.

The general form of the list is $A_1, A_2, \ldots, A_N$

A_1 - First element of the list

A_2- 1st element of the list

N –Size of the list

If the element at position i is A_i, then its successor is A_{i+1} and its predecessor is A_{i-1}

Various operations performed on List

1. Insert (X, 5)- Insert the element X after the position 5.

2. Delete (X) - The element X is deleted

3. Find (X) - Returns the position of X.

4. Next (i) - Returns the position of its successor element i+1.

5. Previous (i) Returns the position of its predecessor i-1.

6. Print list - Contents of the list is displayed.

7. Makeempty- Makes the list empty.

Implementation of list ADT:

1. Array based Implementation

2. Linked List based implementation

Array Implementation of list:

Array is a collection of specific number of same type of data stored in consecutive memory locations. Array is a static data structure i.e., the memory should be allocated in advance and the size is fixed. This will waste the memory space when used space is less than the allocated space.

Insertion and Deletion operation are expensive as it requires more data movements

Find and Print list operations takes constant time.

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

The basic operations performed on a list of elements are

  a. Creation of List.

  b. Insertion of data in the List

  c. Deletion of data from the List

  d. Display all data"s in the List

  e. Searching for a data in the list

Declaration of Array:

#define maxsize 10

int list[maxsize], n ;

Create Operation:

Create operation is used to create the list with „ n „ number of elements .If „ n „ exceeds the array"s maxsize, then elements cannot be inserted into the list. Otherwise the array elements are stored in the consecutive array locations (i.e.) list [0], list [1] and so on.

void Create ( )

```
    {
        int i;
        printf("\nEnter the number of elements to be added in the list:\t");
        scanf("%d",&n);
        printf("\nEnter the array elements:\t");
        for(i=0;i<n;i++)
          scanf("%d",&list[i]);
}
```

If n=6, the output of creation is as follows:

list[6]

20	10	30	40	50	60
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

Insert Operation:

Insert operation is used to insert an element at particular position in the existing list. Inserting the element in the last position of an array is easy. But inserting the element at a particular position in an array is quite difficult since it involves all the subsequent elements   to be shifted one position to the right.

Routine to insert an element in the array:

```
void Insert( )
{
        int i,data,pos;
        printf("\nEnter the data to be inserted:\t");
        scanf("%d",&data);
        printf("\nEnter the position at which element to be inserted:\t");
        scanf("%d",&pos);
        if (pos==n)
        printf ("Array overflow");
        for(i = n-1 ; i >= pos-1 ; i--)
           list[i+1] = list[i];
        list[pos-1] = data;
        n=n+1;
        Display();}
```

Consider an array with 5 elements [max elements = 10]

10	20	30	40	50					

If data 15 is to be inserted in the 2nd position then 50 has to be moved to next index position, 40 has to be moved to 50 position, 30 has to be moved to 40 position and 20 has to be moved to 30 position.

10	20	30	40	50					

10		20	30	40	50				

After this four data movement, 15 is inserted in the 2nd position of the array.

10	15	20	30	40	50				

Deletion Operation:

Deletion is the process of removing an element from the array at any position.

Deleting an element from the end is easy. If an element is to be deleted from any particular position ,it requires all subsequent element from that position is shifted one position towards left.

Routine to delete an element in the array:

```
void Delete( )
{
    int i, pos ;
    printf("\nEnter the position of the data to be deleted:\t");
        scanf("%d",&pos);
    printf("\nThe data deleted is:\t %d", list[pos-1]);
    for(i=pos-1;i<n-1;i++)
        list[i]=list[i+1];
    n=n-1;
    Display();
}
```

Consider an array with 5 elements [max elements = 10]

10	20	30	40	50					

If data 20 is to be deleted from the array, then 30 has to be moved to data 20 position, 40 has to be moved to data 30 position and 50 has to be moved to data 40 position.

10	20	30	40	50					

After this 3 data movements, data 20 is deleted from the 2nd position of the array.

10	30	40	50						

Display Operation/Traversing a list

Traversal is the process of visiting the elements in a array.

Display( ) operation is used to display all the elements stored in the list. The elements are stored from the index 0 to n - 1. Using a for loop, the elements in the list are viewed

Routine to traverse/display elements of the array:

```
 void display( )

 {
        int i;
        printf("\n**********Elements in the array**********\n");
        for(i=0;i<n;i++)
           printf("%d\t",list[i]);

}
```

Search Operation:

Search( ) operation is used to determine whether a particular element is present in the list or not. Input the search element to be checked in the list.

Routine to search an element in the array:

```
void Search( )

{
        int search,i,count = 0;
        printf("\nEnter the element to be searched:\t");
        scanf("%d",&search);
        for(i=0;i<n;i++)
          {
            if(search == list[i])
                count++;
          }
```

```
        if(count==0)
          printf("\nElement not present in the list");
        else
          printf("\nElement present in the list");
}
```

Program for array implementation of List

```
#include<stdio.h>
#include<conio.h>
#define  maxsize  10
int   list[maxsize],n;
void Create();
void Insert();
void Delete();
void Display();
void Search();
void main()
{
        int choice;
        clrscr();
        do
        {
        printf("\n Array Implementation of List\n");
        printf("\t1.create\n");
        printf("\t2.Insert\n");
        printf("\t3.Delete\n");
        printf("\t4.Display\n");
        printf("\t5.Search\n");
        printf("\t6.Exit\n");
        printf("\nEnter your choice:\t");
        scanf("%d",&choice);
        switch(choice)
        {
                case 1:  Create();
                        break;
                case 2:  Insert();
                        break;
                case 3:  Delete();
                        break;
                case 4:  Display();
                        break;
                case 5:  Search();
                        break;
                case 6:  exit(1);
```

```
                default:  printf("\nEnter option between 1 -  6\n");
                           break;
        }
        }while(choice<7);
}
void Create()
{
        int i;
        printf("\nEnter the number of elements to be added in the list:\t");
        scanf("%d",&n);
        printf("\nEnter the array elements:\t");
        for(i=0;i<n;i++)
        scanf("%d",&list[i]);
        Display();
}
void Insert()
{
        int i,data,pos;
        printf("\nEnter the data to be inserted:\t");
        scanf("%d",&data);
        printf("\nEnter the position at which element to be inserted:\t");
        scanf("%d",&pos);
        for(i = n-1 ; i >= pos-1 ; i--)
          list[i+1] = list[i];
        list[pos-1] = data;
        n+=1;
        Display();
}
void Delete( )
{
 int i,pos;
 printf("\nEnter the position of the data to be deleted:\t");
 scanf("%d",&pos);
 printf("\nThe data deleted is:\t %d", list[pos-1]);
 for(i=pos-1;i<n-1;i++)
 list[i]=list[i+1];
 n=n-1;
 Display();
}
void Display()
{
        int i;
        printf("\n**********Elements in the array**********\n");
        for(i=0;i<n;i++)
        printf("%d\t",list[i]);
}
```

```
void Search()
{
        int search,i,count = 0;
        printf("\nEnter the element to be searched:\t");
        scanf("%d",&search);
        for(i=0;i<n;i++)
        {
        if(search == list[i])
          {
            count++;
          }
        }
        if(count==0)
            printf("\nElement not present in the list");
        else
            printf("\nElement present in the list");
}
```

Output

 Array Implementation of List

1.create
2.Insert
3.Delete
4.Display
5.Search
6.Exit
Enter your choice: 1

Enter the number of elements to be added in the list: 5

Enter the array elements: 1 2 3 4 5

**********Elements in the array**********

1   2   3   4   5

 Array Implementation of List

1.create
2.Insert
3.Delete
4.Display
5.Search
6.Exit

Enter your choice: 2

Enter the data to be inserted: 3

Enter the position at which element to be inserted: 1

**********Elements in the array**********

3   1   2   3   4   5

 Array Implementation of List

1.create
2.Insert
3.Delete
4.Display
5.Search
6.Exit

Enter your choice: 3

Enter the position of the data to be deleted: 4

The data deleted is: 3

**********Elements in the array**********

3   1   2   4   5

 Array Implementation of List

1.create
2.Insert
3.Delete
4.Display
5.Search
6.Exit

Enter your choice: 5

Enter the element to be searched: 1

Element present in the list

 Array Implementation of List

1.create
2.Insert
3.Delete
4.Display
5.Search
6.Exit
   Enter your choice:6

Advantages of array implementation:
1.The elements are faster to access using random access
2.Searching an element is easier

Limitation of array implementation

- An array store its nodes in consecutive memory locations.
- The number of elements in the array is fixed and it is not possible to change the number of elements .
- Insertion and deletion operation in array are expensive. Since insertion is performed by pushing the entire array one position down and deletion is performed by shifting the entire array one position up.

Applications of arrays:
Arrays are particularly used in programs that require storing large collection of similar type data elements**.**

Differences between Array based and Linked based implementation

	Array	Linked List
Definition	Array is a collection of elements having same data type with common name	Linked list is an ordered collection of elements which are connected by links/pointers
Access	Elements can be accessed using index/subscript, random access	Sequential access
Memory structure	Elements are stored in contiguous memory locations	Elements are stored at available memory space
Insertion & Deletion	Insertion and deletion takes more time in array	Insertion and deletion are fast and easy
Memory Allocation	Memory is allocated at compile time i.e static memory allocation	Memory is allocated at run time i.e dynamic memory allocation
Types	1D,2D,multi-dimensional	SLL, DLL circular linked list
Dependency	Each elements is independent	Each node is dependent on each other as address part contains address of next node in the list

Linked list based implementation:

Linked Lists:

A Linked list is an ordered collection of elements. Each element in the list is referred as a node. Each node contains two fields namely,

Data field-The data field contains the actual data of the elements to be stored in the list

Next field- The next field contains the address of the next node in the list

DATA	NEXT

A Linked list node

L
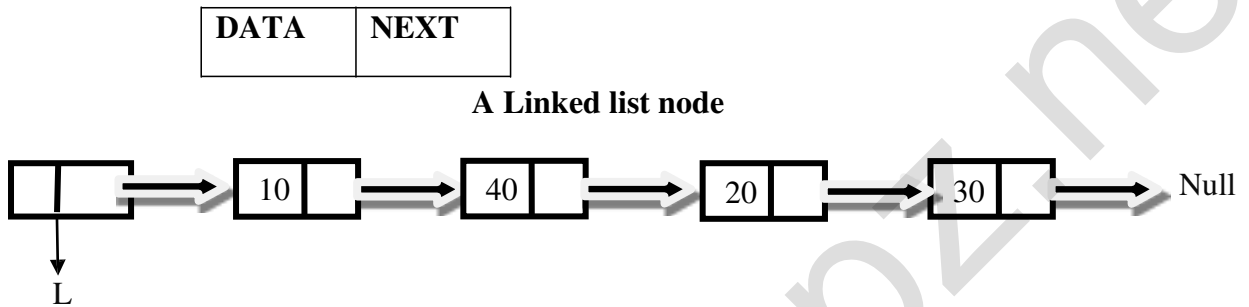
Advantages of Linked list:

1.Insertion and deletion of elements can be done efficiently

2.It uses dynamic memory allocation

3.Memory utilization is efficient compared to arrays

Disadvantages of linked list:

1.Linked list does not support random access

2.Memory is required to store next field

3.Searching takes time compared to arrays

Types of Linked List

1. Singly Linked List or One Way List

2. Doubly Linked List or Two-Way Linked List

3. Circular Linked List

Dynamic allocation

The process of allocating memory to the variables during execution of the program or at run time is known as dynamic memory allocation. C language has four library routines which allow this function.

Dynamic memory allocation gives best performance in situations in which we do not know memory requirements in advance. C provides four library routines to automatically allocate memory at the run time.

Memory allocation/de-allocation functions

Function	Task
malloc()	Allocates memory and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory
free()	Frees previously allocated memory
realloc()	Alters the size of previously allocated memory

To use dynamic memory allocation functions, you must include the header file stdlib.h.

malloc()

The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer.

The general syntax of malloc() is

ptr =(cast-type*)malloc(byte-size);

where ptr is a pointer of type cast-type. malloc() returns a pointer (of cast type) to an area of memory with size byte-size.

calloc():

calloc() function is another function that reserves memory at the run time. It is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. calloc() stands for contiguous memory allocation and is primarily used to allocate memory for arrays. The syntax of calloc() can be given as:

ptr=(cast-type*) calloc(n,elem-size);

The above statement allocates contiguous space for n blocks each of size elem-size bytes. The only difference between malloc() and calloc() is that when we use calloc(), all bytes are initialized to zero. calloc() returns a pointer to the first byte of the allocated region.

free():

The free() is used to release the block of memory.

Syntax:

The general syntax of the free()function is,

free(ptr);

where ptr is a pointer that has been created by using malloc() or calloc(). When memory is deallocated using free(), it is returned back to the free list within the heap.

realloc():

At times the memory allocated by using calloc() or malloc() might be insufficient or in excess. In both the situations we can always use realloc() to change the memory size already allocated by calloc() and malloc(). This process is called *reallocation of memory*. The general syntax for realloc() can be given as,

ptr = realloc(ptr,newsize);

The function realloc() allocates new memory space of size specified by newsize to the pointer

variable ptr. It returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region. Thus, we see that realloc() takes two arguments. The first is the pointer referencing the memory and the second is the total number of bytes you want to reallocate.

Singly Linked List

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list
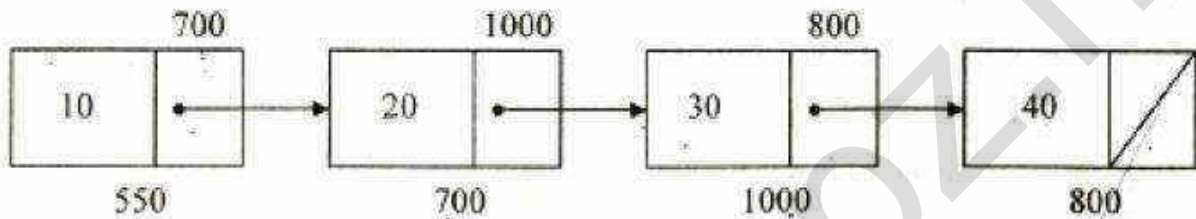
SLL
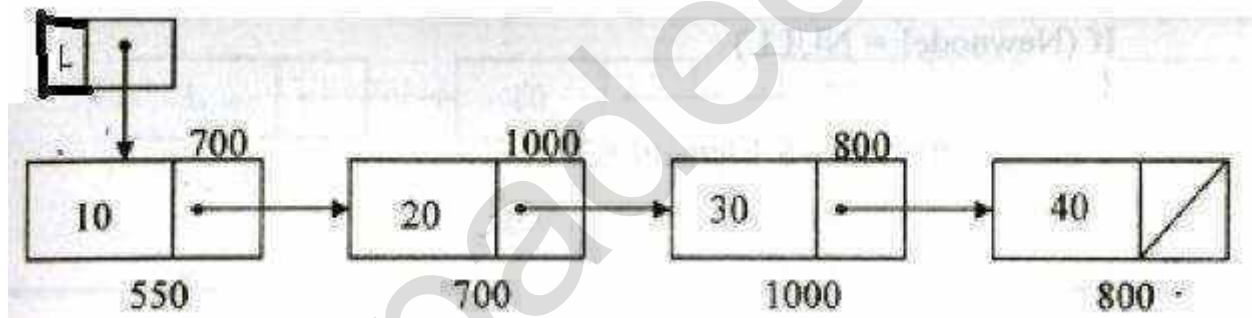
SLL with a Header

Basic operations on a singly-linked list are:

1. Insert – Inserts a new node in the list.
2. Delete – Deletes any node from the list.
3. Find – Finds the position(address) of any node in the list.
4. FindPrevious - Finds the position(address) of the previous node in the list.
5. FindNext- Finds the position(address) of the next node in the list.
6. Display-display the date in the list
7. Search-find whether a element is present in the list or not

Declaration of Linked List

```
void insert(int X,List L,position P);
void find(List L,int X); void
delete(int x , List L); typedef
struct  node  *position;
position  L,p,newnode;
struct node
{
        int data;
        position next;
};
```

Creation of the list:

This routine creates a list by getting the number of nodes from the user. Assume n=4 for this

example.

```
void create()
{
int i,n;
L=NULL;
newnode=(struct node*)malloc(sizeof(struct node));
printf("\n Enter the number of nodes to be inserted\n");
scanf("%d",&n);
printf("\n Enter the data\n");
scanf("%d",&newnode->data);
newnode->next=NULL;
L=newnode;
p=L;
for(i=2;i<=n;i++)
{
newnode=(struct node *)malloc(sizeof(struct node));
scanf("%d",&newnode->data);
newnode->next=NULL;
```

```
p->next=newnode;

p=newnode;

}

}
```

Initially the list is empty

List L

L
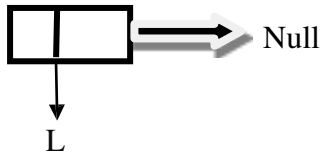
Insert(10,List L)- A new node with data 10 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.
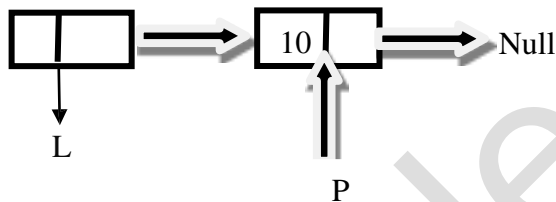
Insert(20,L) - A new node with data 20 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.
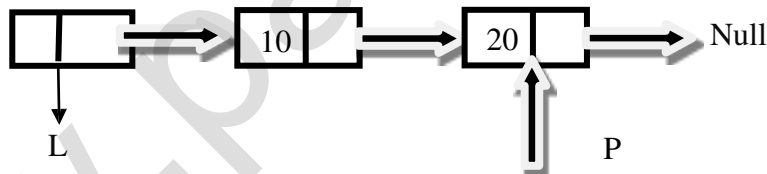
Insert(30,L) - A new node with data 30 is inserted and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.
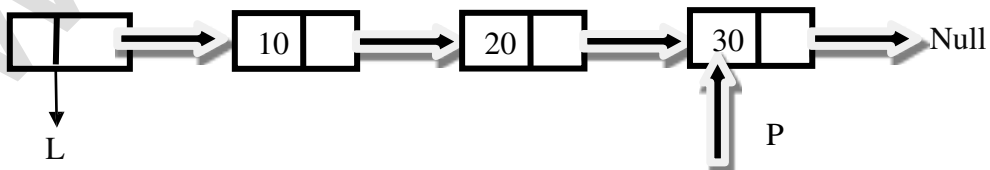
Case 1:Routine to insert an element in list at the beginning

void insert(int X, List L, position p)

{

                    p=L;

                    newnode=(struct node*)malloc(sizeof(struct node));

                    printf("\nEnter the data to be inserted\n");

          scanf("%d",&newnode->data);

          newnode->next=L;

          L=newnode;

}

Case 2:Routine to insert an element in list at Position

This routine inserts an element X after the position P.

    Void Insert(int X, List L, position p)

    {

        position newnode;

        newnode =(struct node*) malloc(sizeof(struct node));

        if(newnode = = NULL)

            Fatal error(" Out of Space ");

        else

         {

            Newnode -> data = x ;

            Newnode -> next = p ->next ;

            P -> next = newnode ;

         }

}

Insert(25,L, P) - A new node with data 25 is inserted after the position P and the next field is updated to NULL. The next field of previous node is updated to store the address of new node.
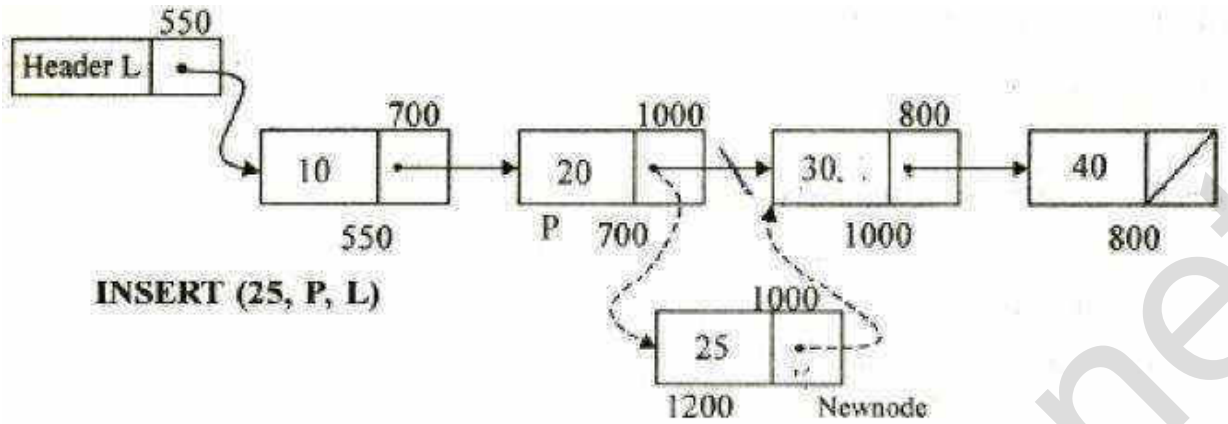
INSERT (25, P, L)

Case 3:Routine to insert an element in list at the end of the list

void insert(int X, List L, position p)

{

p=L;

 newnode=(struct node*)malloc(sizeof(struct node));

 printf("\nEnter the data to be inserted\n");

 scanf("%d",&newnode->data);

 while(p->next!=NULL)

 p=p->next;

 newnode->next=NULL;

 p->next=newnode;

 p=newnode;

}

Routine to check whether a list is Empty

This routine checks whether the list is empty .If the lis t is empty it returns 1

 int IsEmpty(List L)

 {

  if (L -> next = = NULL)

   return(1);

 }
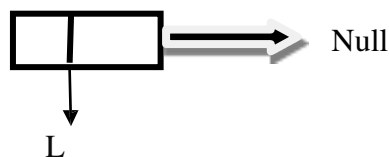
Routine to check whether the current position is last in the List

This routine checks whether the current position p is the last position in the list. It returns 1 if position p is the last position

```
int IsLast(List L , position p)
{
        if( p -> next= =NULL)
                return(1);
}
```


L
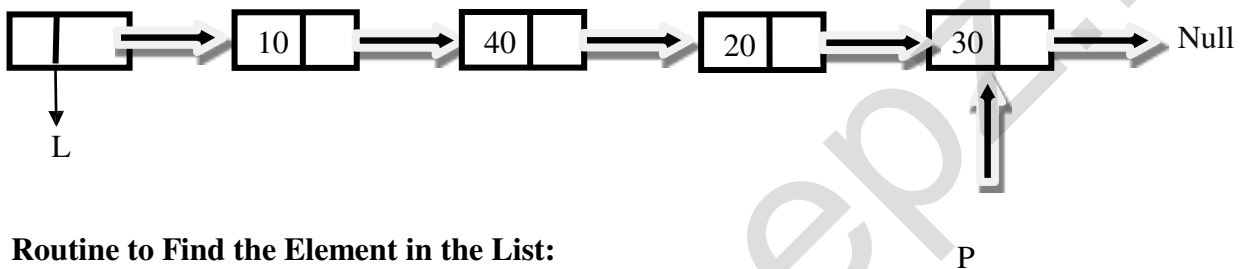
Routine to Find the Element in the List:

P

This routine returns the position of X in the list L

```
position find(List  L, int  X)
{
        position p;
        p=L->next;
        while(p!=NULL && p->data!=X)
                p=p->next;
        return(p);
}
```

Find(List L, 20) - To find an element X traverse from the first node of the list and move to the next with the help of the address stored in the next field until data is equal to X or till the end of the list

L

x   P
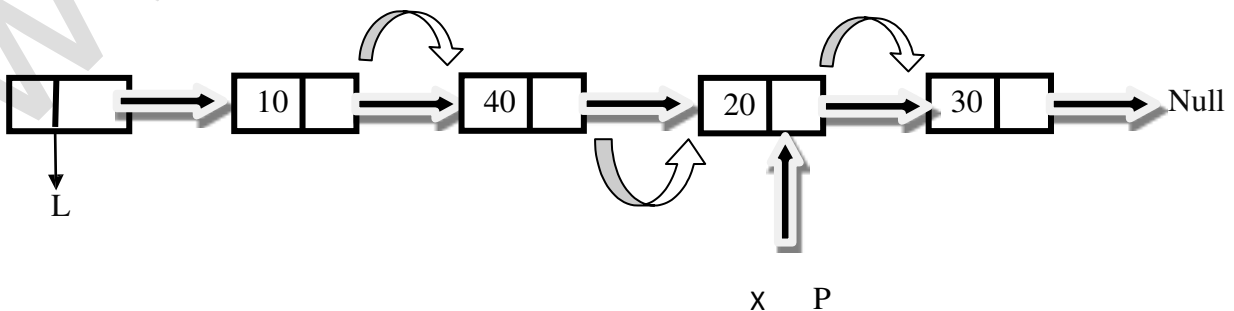
Find Previous

It returns the position of its predecessor.

position FindPrevious (int X, List L)

{

        position p;

        p = L;

        while(p -> next ! = NULL && p -> next -> data! = X)

                p = p -> next;

        return P;

}
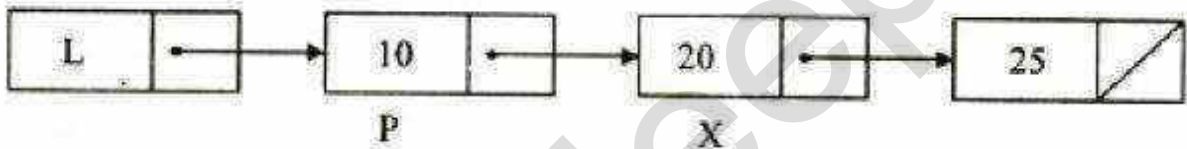
Routine to find next Element in the List

It returns the position of successor.

    void FindNext(int X, List L)

    {

        position P;

        P=L->next;

        while(P!=NULL && P->data!=X)

                P = P->next;

        return P->next;

}
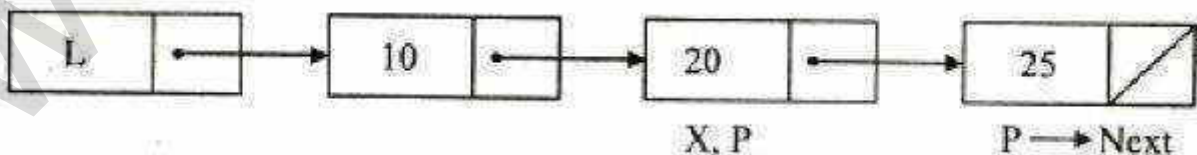
Routine to Count the Element in the List:

This routine counts the number of elements in the list

```
void count(List L)
{
        P = L -> next;
        while( p != NULL )
        {
                count++;
                p = p -> next;
        }
    print count;
}
```

Routine to Delete an Element in the List:

It delete the first occurrence of element X from the list L

```
void Delete( int x , List L){
        position p, Temp;
        p = FindPrevious( X, L);
        if( ! IsLast (p, L)){
                temp = p -> next;
                P -> next  =  temp -> next;
                free ( temp );
        }}
```


BEFORE DELETION

AFTER DELETION
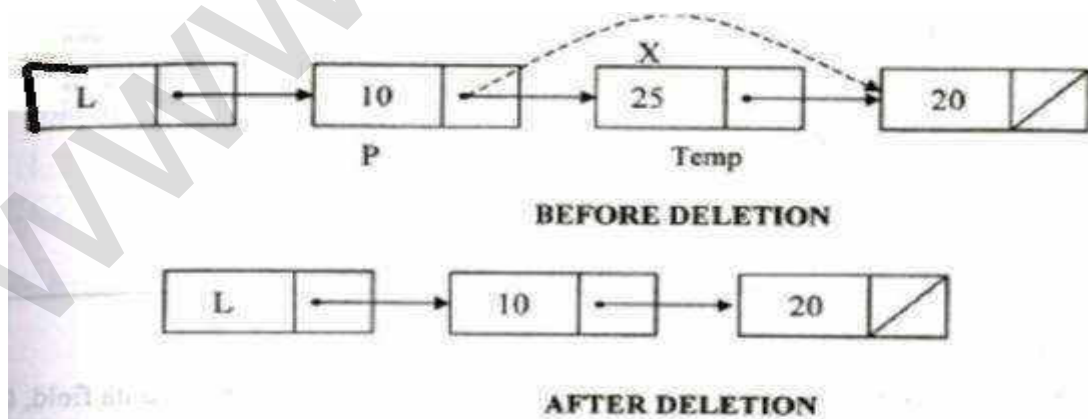
Routine to Delete the List

This routine deleted the entire list.

```
void Delete_list(List L)
    {
        position P,temp;
        P=L->next;
        L->next=NULL;
        while(P!=NULL)
        {
                temp=P->next;
                free(P);
                P=temp;
        }
    }
```
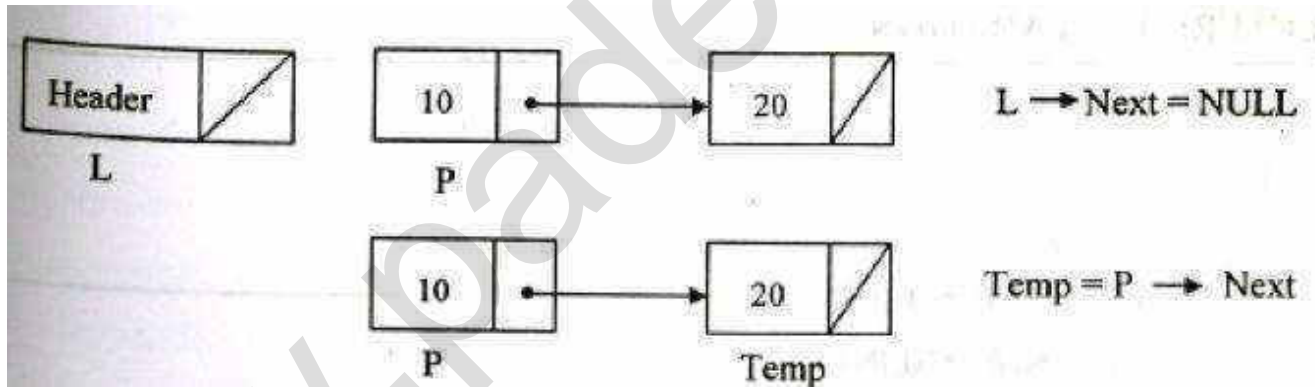

Program 1: Implementation of Singly linked List	Output

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void create();
void display();
void insert();
void find();
void delete();
typedef struct node *position;
position L,p,newnode;
struct node
{
	1.create
	
	2.display
	
	3.insert
	
	4.find
	
	5.delete
	
	Enter your choice

```
int data;
position next;
};
void main()
{
int choice;
clrscr();
do
{
printf("1.create\n2.display\n3.insert\n4.find\n5.delete\n\n\n");
printf("Enter your choice\n\n");
scanf("%d",&choice);
switch(choice)
{
case 1:
        create();
        break;
case  2:
        display();
        break;
case  3:
        insert();
        break;
case  4:
        find();
        break;
case 5:
        delete();
        break;
case 6:
        exit(0);
}
}
while(choice<7);
getch();
}
void create()
{
int i,n;
L=NULL;
newnode=(struct node*)malloc(sizeof(struct node));
printf("\n Enter the number of nodes to be inserted\n");
scanf("%d",&n);
printf("\n Enter the data\n");
scanf("%d",&newnode->data);
newnode->next=NULL;
```

1

Enter the number of nodes to be inserted

5

Enter the data

1

2

3

4

5

1.create

2.display

3.insert

4.find

5.delete

Enter your choice

2

1 -> 2 -> 3 -> 4 -> 5 ->

Null

1.create

2.display

3.insert

4.find

5.delete

Enter your choice

3

```
L=newnode; p=L;
for(i=2;i<=n;i++)
{
newnode=(struct  node *)malloc(sizeof(struct  node));
scanf("%d",&newnode->data);
newnode->next=NULL;
p->next=newnode;
p=newnode;
}
}
void display()
 { p=L;
 while(p!=NULL)
 {
 printf("%d -> ",p->data);
 p=p->next;
 }
 printf("Null\n");
 }
void insert()
{
int ch;

        printf("\nEnter ur choice\n");
printf("\n1.first\n2.middle\n3.end\n");
scanf("%d",&ch);
switch(ch)
{
case 2:
        {
        int pos,i=1;
        p=L;
                newnode=(struct  node*)malloc(sizeof(struct  node));
                printf("\nEnter  the data to be inserted\n");
        scanf("%d",&newnode->data);
        printf("\nEnter the position to be inserted\n");
        scanf("%d",&pos);
        newnode->next=NULL;
        while(i<pos-1)
         {
          p=p->next;
          i++;
         }
        newnode->next=p->next;
        p->next=newnode;
```

Enter ur choice

1.first

2.middle

3.end

1

Enter the data to be inserted

7

7 -> 1 -> 2 -> 3 -> 4 -> 5 -> Null

1.create

2.display

3.insert
4.find

5.delete

Enter your choice

```
        p=newnode;
        display();
        break;
        }
case 1:
        {
        p=L;
                newnode=(struct node*)malloc(sizeof(struct node));
                printf("\nEnter the data to be inserted\n");
        scanf("%d",&newnode->data);
        newnode->next=L;
        L=newnode;
        display();
        break;
        }
case 3:
        {
        p=L;
                newnode=(struct node*)malloc(sizeof(struct node));
                printf("\nEnter the data to be inserted\n");
        scanf("%d",&newnode->data);
        while(p->next!=NULL)
        p=p->next;
        newnode->next=NULL;
        p->next=newnode;
        p=newnode;
        display();
        break;
        }
}
}
void find()
{
int search,count=0;
printf("\n Enter the element to be found:\n");
scanf("%d",&search);
p=L;
while(p!=NULL)
{
if(p->data==search)
 {
  count++;
  break;
 }
p=p->next;
}
```

```
if(count==0)
printf("\n Element Not present\n");
else
printf("\n Element present in the list \n\n");
}
void delete()
{
position p,temp;
int x; p=L;
if(p==NULL)
{
printf("empty list\n");
}
else
{
printf("\nEnter the data to be deleted\n");
scanf("%d",&x);
if(x==p->data)
{ temp=p;
L=p->next;
free(temp);
display();
}
else
{
while(p->next!=NULL && p->next->data!=x)
{
p=p->next;
}
temp=p->next;
p->next=p->next->next;
free(temp);
display();
}
}
}
```

Advantages of SLL

1.The elements can be accessed using the next link

2.Occupies less memory than DLL as it has only one next field.

Disadvantages of SLL

1.Traversal in the backwards is not possible

2.Less efficient to for insertion and deletion.

Doubly-Linked List

A doubly linked list is a linked list in which each node has three fields namely Data, Next, Prev.

Data-This field stores the value of the element

Next-This field points to the successor node in the list

Prev-This field points to the predecessor node in the list

PREV	DATA	NEXT
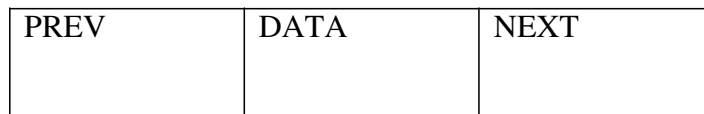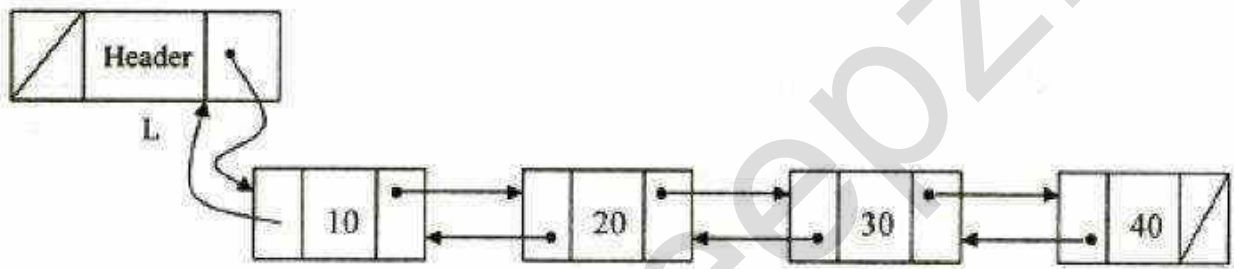
DLL NODE

DOUBLY LINKED LIST

Basic operations of a doubly -linked list are:

1. Insert – Inserts a new element at the end of the list.
2. Delete – Deletes any node from the list.
3. Find – Finds any node in the list.
4. Print – Prints the list

Declaration of DLL Node

typedef struct node *position ;

struct node

PREV	DATA	NEXT

{

int data;

    position prev;

    position next;

};
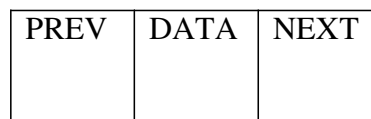
Creation of list in DLL

Initially the list is empty. Then assign the first node as head.

newnode->data=X;

newnode->next=NULL;

newnode->prev=NULL;

L=newnode;

list. If we add one more node in the list,then create a newnode and attach that node to the end of the

L->next=newnode;

newnode->prev=L;

Routine to insert an element in a DLL at the beginning

void Insert (int x, list L, position P){

struct node *Newnode;

if(pos==1)

P=L;

Newnode = (struc node*)malloc (sizeof(struct node));

if (Newnode! = NULL)

Newnode->data= X;

Newnode ->next= L ->next;

L->next ->prev=Newnode

L->next = Newnode;

Newnode ->prev = L;

}

Routine to insert an element in a DLL any position :

void Insert (int x, list L, position P)

{

struct node *Newnode;

Newnode = (struc node*)malloc (sizeof(struct node));

if (Newnode! = NULL)

Newnode->data= X;

Newnode ->next= P ->next;

P->next ->prev=Newnode

P ->next = Newnode;

Newnode ->prev = P:

}
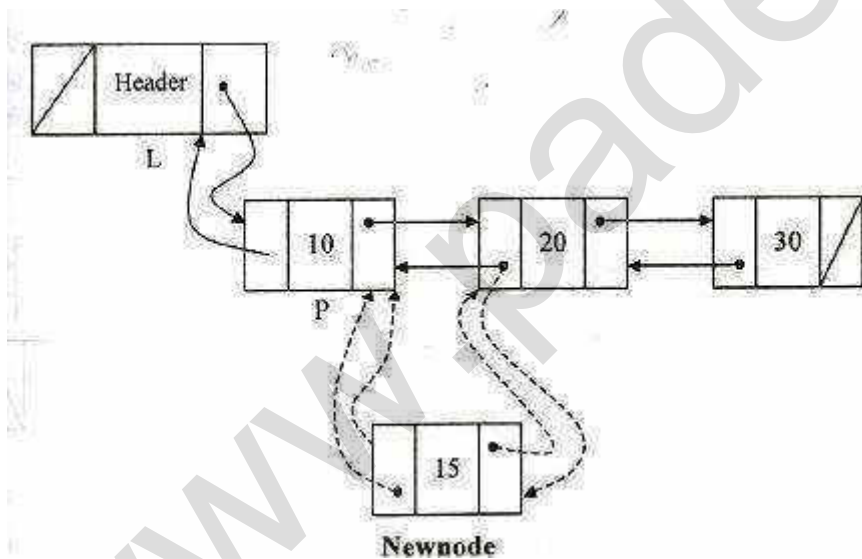
Newnode

Routine to insert an element in a DLL at the end:

void insert(int X, List L, position p)

{

        p=L;

        newnode=(struct node*)malloc(sizeof(struct node));

```
printf("\nEnter the data to be inserted\n");
scanf("%d",&newnode->data);
while(p->next!=NULL)
p=p->next;
newnode->next=NULL;
p->next=newnode;
newnode->prev=p;
}
```

Routine for deleting an element:

```
void Delete (int x ,List L)
{
  Position p , temp;
P = Find( x, L );
if(P==L->next)
temp=L;
L->next=temp->next;
temp->next->prev=L;
free(temp);
elseif( IsLast( p, L ) )
  {
    temp = p;
     p -> prev -> next = NULL;
      free(temp);
   }
  else
  {
  temp = p;
 p -> prev -> next = p -> next;
  p -> next -> prev = p -> prev;
  free(temp);
}
```
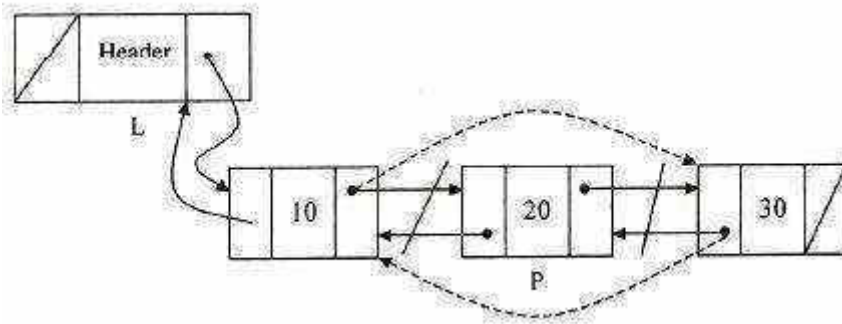
Routine to display the elements in the list:

```
void Display( List L )

{

 P = L -> next ;

while ( p != NULL)

{

   printf("%d", p -> data ;

   p = p -> next ;

}

printf(" NULL");

}
```

Routine to search whether an element is present in the list

```
void find()
{
int a,flag=0,count=0;
if(L==NULL)
printf("\nThe list is empty");
else
{
printf("\nEnter the elements to be searched");
scanf("%d",&a);
for(P=L;P!=NULL;P=P->next)
{
count++;
if(P->data==a)
{
flag=1;
printf("\nThe element is found");
printf("\nThe position is %d",count);
```

```
  break;
 }
 }
if(flag==0)
printf("\nThe element is not found");
 }
 }
```

Program 2 : Implementation of Doubly linked list	Output

`#include <stdio.h>`
`#include<conio.h>`
`void insert();`
`void deletion();`
`void display();`
`void find();`
`typedef struct node *position;`
`position newnode,temp,L=NULL,P;`
`struct node`
`{`
`int data;`
` position next;`
` position prev;`
`};`
`void main()`
`{`
`int choice;`
`clrscr();`
`do`
`{ printf("\n1.INSERT");`
`printf("\n2.DELETE");`
`printf("\n3.DISPLAY");`
`printf("\n4.FIND");`
`printf("\n5.EXIT");`
`printf("\nEnter ur option");`
`scanf("%d",&choice);`
`switch(choice)`
`{`
`case 1:`
`insert();`
`break;`
`case 2:`
`deletion();`
`break;`
`case 3:`
`display();`
	1.INSERT
	2.DELETE
	3.DISPLAY
	4.FIND
	5.EXIT
	Enter ur option1
	
	Enter the data to be inserted10
	
	1.INSERT
	2.DELETE
	3.DISPLAY
	4.FIND
	5.EXIT
	Enter ur option1
	
	Enter the data to be inserted 20
	
	Enter the position where the data is to be inserted 2

```
break;
case 4:
find();
break;
case 5:
exit(1);
}
}while(choice!=5);
getch();
}
void insert()
{
int pos,I;
newnode=(struct node*)malloc(sizeof(struct node));
printf("\nEnter the data to be inserted");
scanf("%d",&newnode->data);
if(L==NULL)
{
L=newnode;
L->next=NULL;
L->prev=NULL;
}
else
{
printf("\nEnter the position where the data is to be inserted");
scanf("%d",&pos);
if(pos==1)
{
newnode->next=L;
newnode->prev=NULL;
L->prev=newnode;
L=newnode;
}
else
{
P=L;
for(i=1;i<pos-1&&P->next!=NULL;i++)
{
P=P->next;
}
newnode->next=P->next;
P->next=newnode;
newnode->prev=P;
P->next->prev=newnode;
}
}
```

1.INSERT

2.DELETE

3.DISPLAY

4.FIND

5.EXIT

Enter ur option1

Enter the data to be inserted 30

Enter the position where the data is to be inserted3

1.INSERT

2.DELETE

3.DISPLAY

4.FIND

5.EXIT

Enter ur option 3

The elements in the list are

10 20 30

1.INSERT

2.DELETE

3.DISPLAY

4.FIND

5.EXIT

Enter ur option 2

```
}
void deletion()
{
int pos,I;
if(L==NULL)
printf("\nThe list is empty");
else
{
printf("\nEnter the position of the data to be deleted");
scanf("%d",&pos);

if(pos==1)
{ temp=L;
L=temp->next;
L->prev=NULL;
printf("\nThe deleted element is %d",temp->data);
free(temp);
}
else
{
P=L;
for(i=1;i<pos-1;i++)
P=P->next;
temp=P->next;
printf("\nThe deleted element is %d",temp->data);
P->next=temp->next;
temp->next->prev=P;
free(temp);
}
}
}
void display()
{
if(L==NULL)
printf("\nNo of elements in the list");
else
{
printf("\nThe elements in the listare\n");
for(P=L;P!=NULL;P=P->next)
printf("%d",P->data);
}
}
void find()
{
int a,flag=0,count=0;
```

Enter the position of the data to be deleted 2

The deleted element is 20

1.INSERT

2.DELETE

3.DISPLAY

4.FIND

5.EXIT

Enter ur option 3

The elements in the list are

10 30

1.INSERT

2.DELETE

3.DISPLAY

4.FIND

5.EXIT

Enter ur option4

Enter the elements to be searched 20

The element is not found

1.INSERT

2.DELETE

3.DISPLAY

4.FIND

5.EXIT

Enter ur option 4

if(L==NULL)
printf("\nThe list is empty");
else
{
printf("\nEnter the elements to be searched");
scanf("%d",&a);
for(P=L;P!=NULL;P=P->next)
{
count++;
if(P->data==a)
{
flag=1;
printf("\nThe element is found");
printf("\nThe position is %d",count);
break;
}
}
if(flag==0)
printf("\nThe element is not found");
}
}
	Enter the elements to be searched 30
	
	The element is found
	The position is 2
	1.INSERT
	2.DELETE
	3.DISPLAY
	4.FIND
	5.EXIT
	Enter ur option5
	Press any key to continue . . .

Advantages of DLL:

The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields we can access any node efficiently whereas in SLL only one link field is there which stores next node which makes accessing of any node difficult.

Disadvantages of DLL:

The DLL has two pointer fields. One field is prev link field and another is next link field. Because of these two pointer fields, more memory space is used by DLL compared to SLL

CIRCULAR LINKED LIST:

Circular Linked list is a linked list in which the pointer of the last node points to the first node.
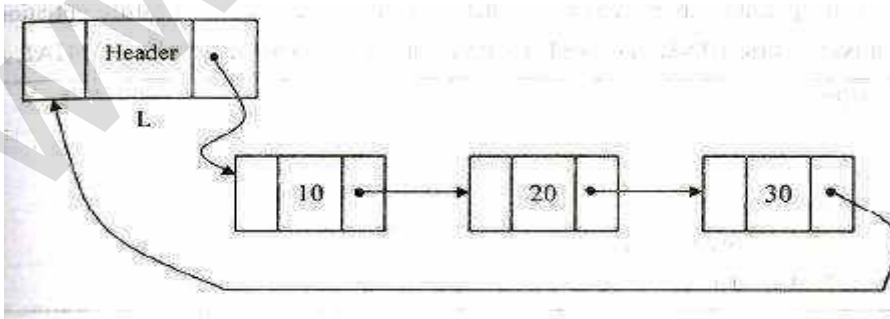
Types of CLL:

CLL can be implemented as circular singly linked list and circular doubly linked list.

Singly linked circular list:

A Singly linked circular list is a linked list in which the last node of the list points to the first node.

Declaration of node:

typedef struct node *position;
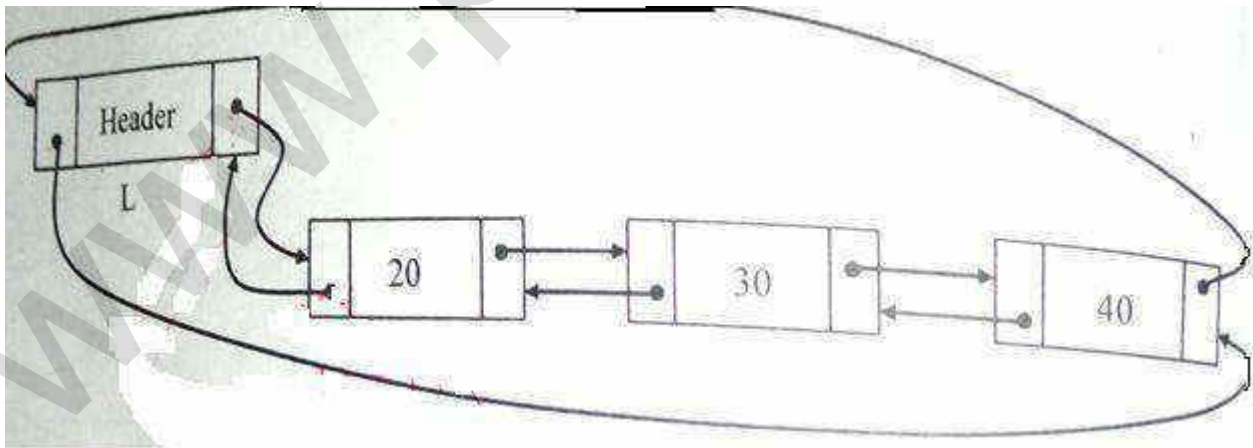
struct node
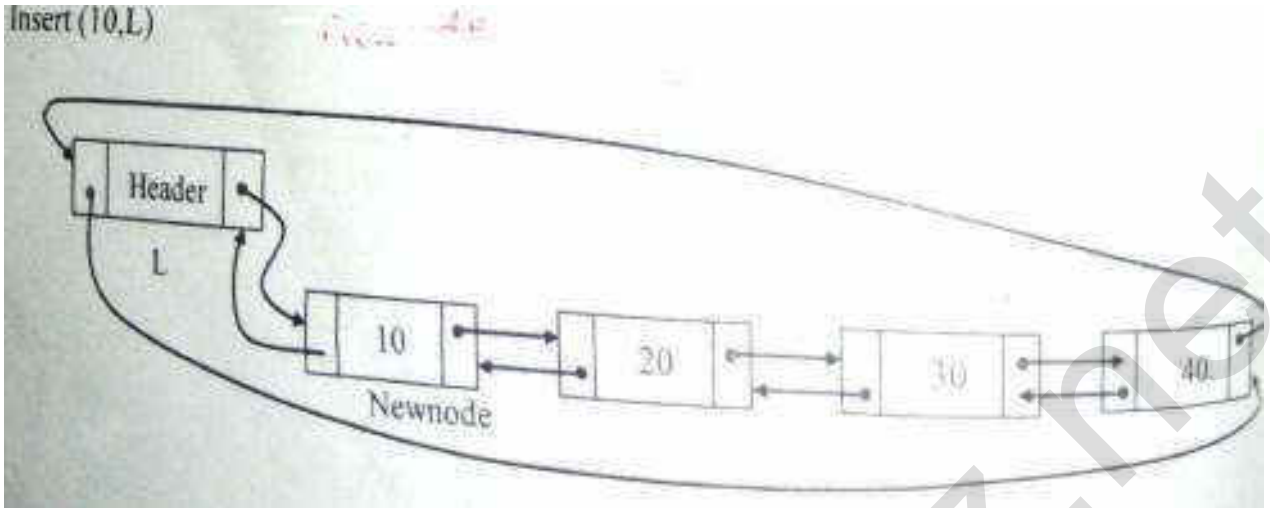
{
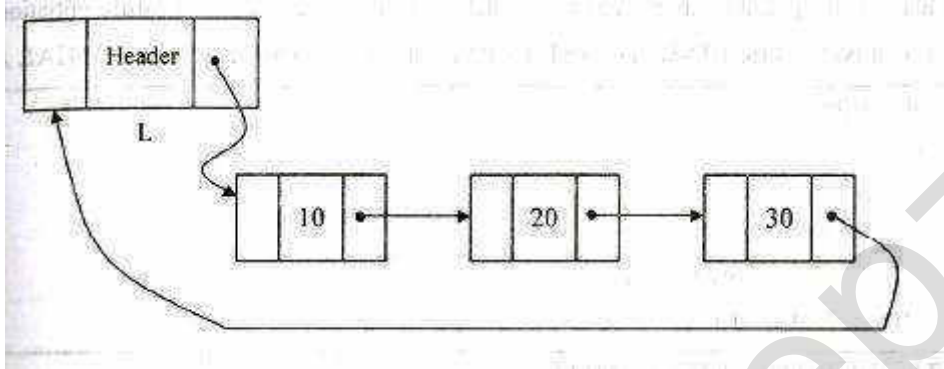
int data;

position next;

};

Routine to insert an element in the beginning

```
void insert_beg(int  X,List L)
 {
 position  Newnode;
 Newnode=(struct node*)malloc(sizeof(struct node));
 if(Newnode!=NULL)
 {
 Newnode->data=X;
 Newnode->next=L->next;
 L->next=Newnode;
 }
 }
```
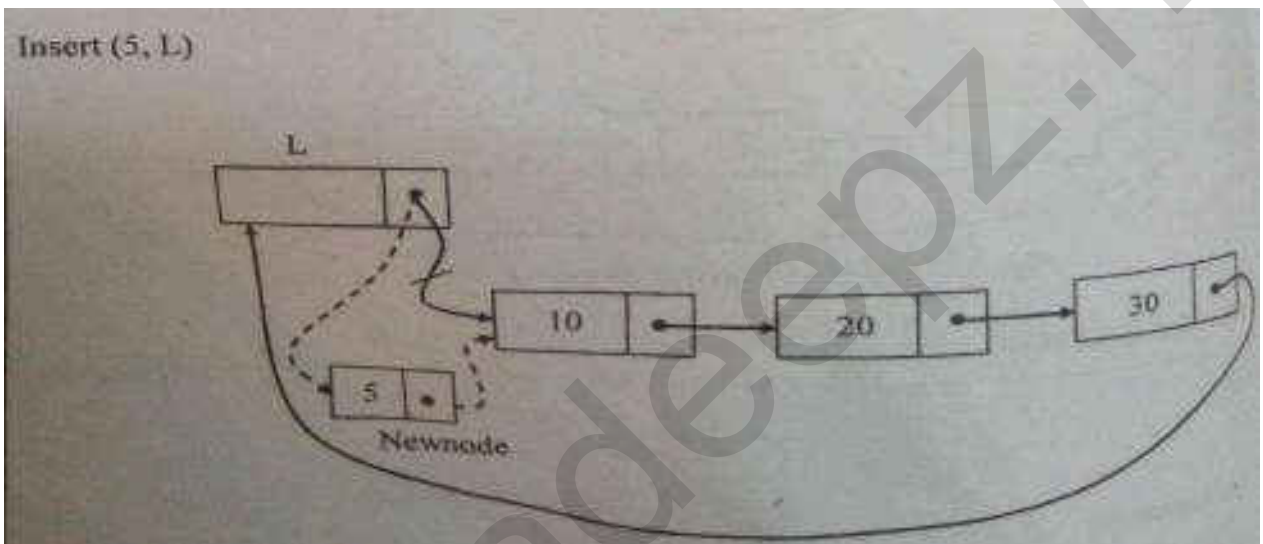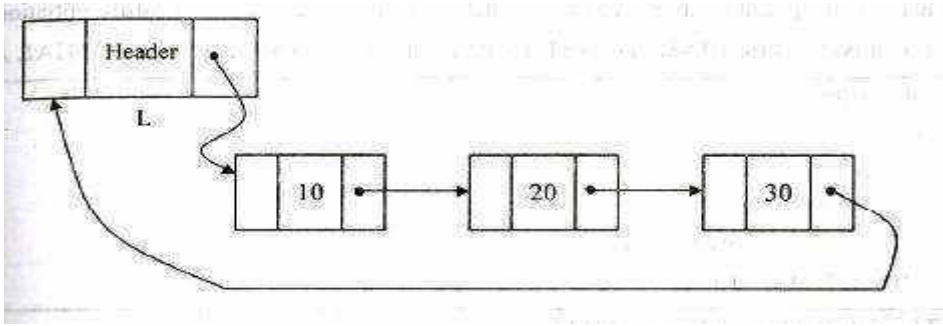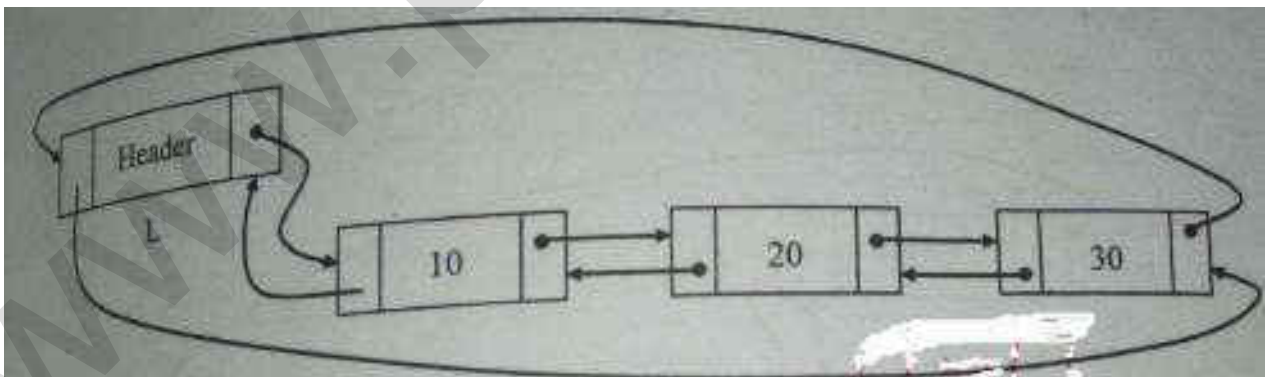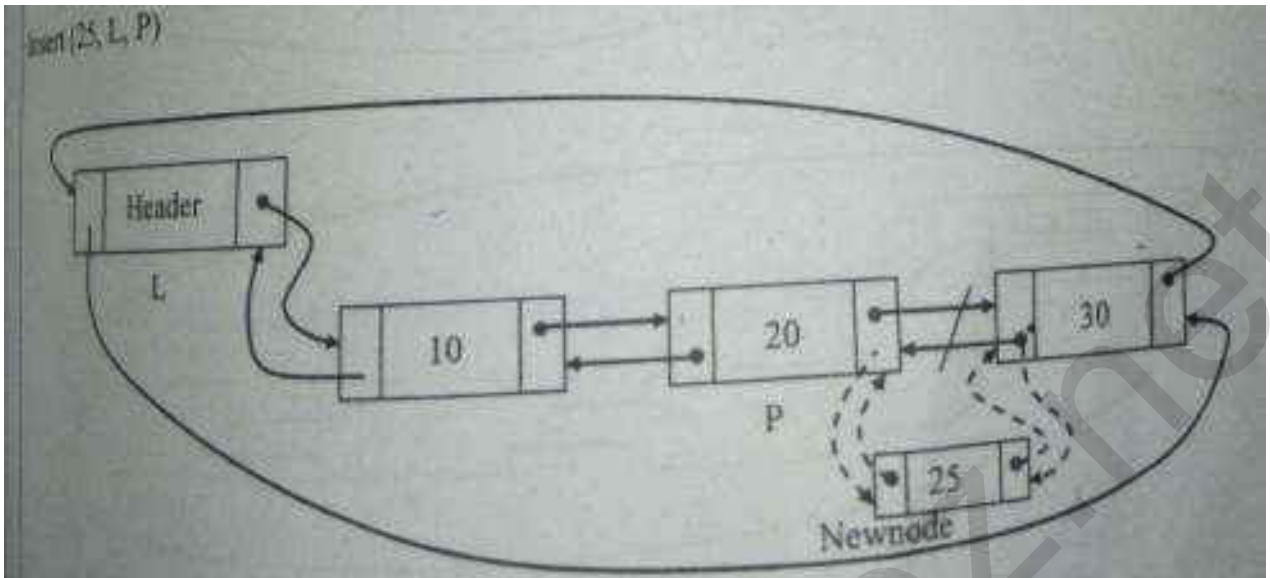
Insert (5, L)

Routine to insert an element in the middle

```
void insert_mid(int  X, List L, Position p)
{
position  Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
Newnode->data=X;
Newnode->next=p->next;
p->next=Newnode;
}
}
```

Routine to insert an element in the last

```
void insert_last(int  X, List L)
position  Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
P=L;
while(P->next!=L)
P=P->next;
Newnode->data=X;
P->next=Newnode;
Newnode->next=L;
}
}
```
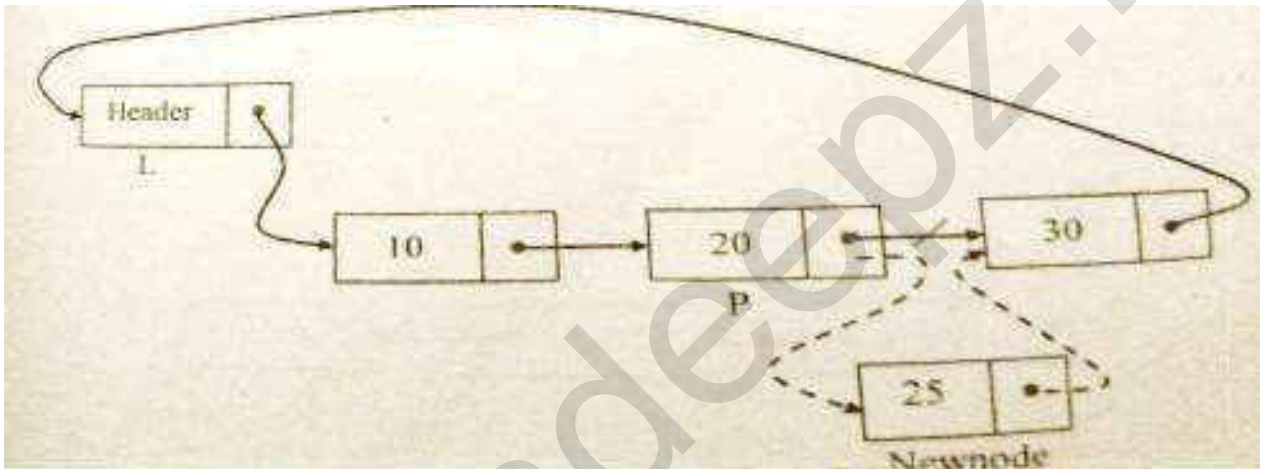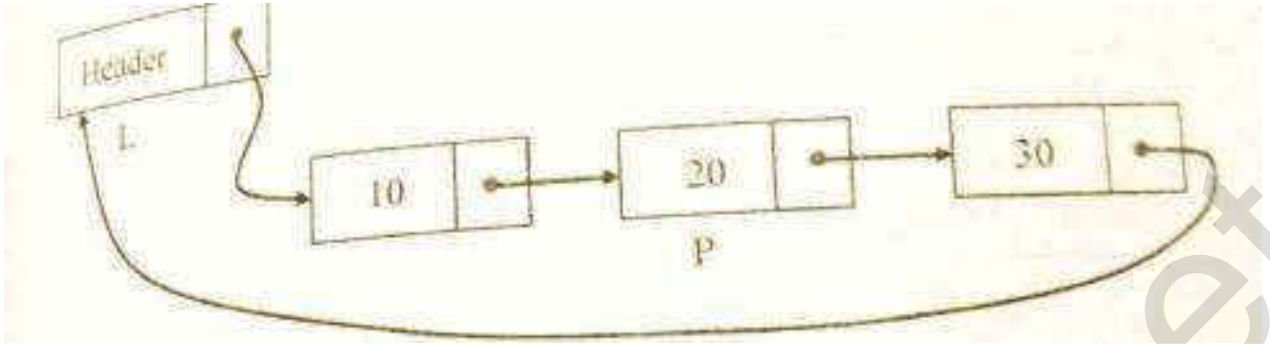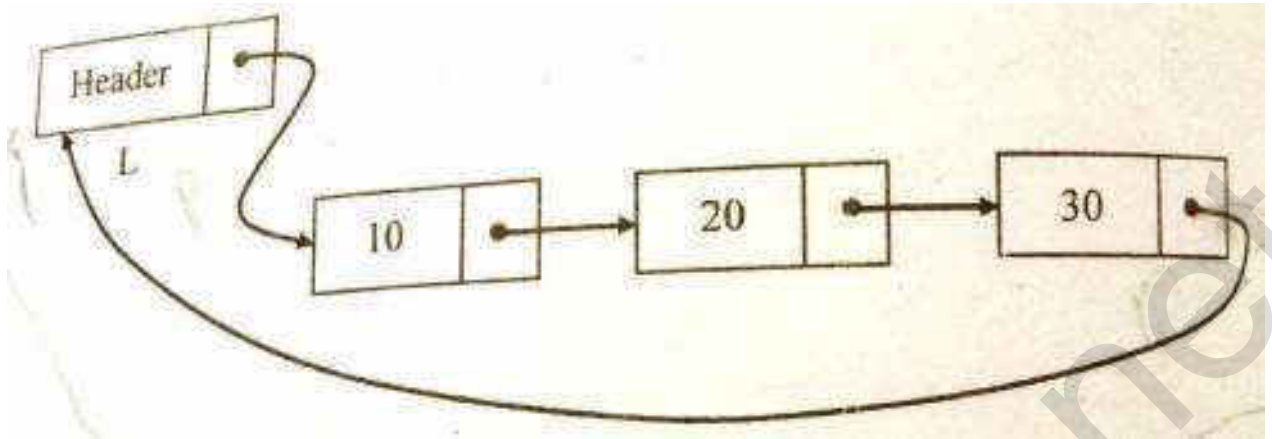
nsert (40, L)
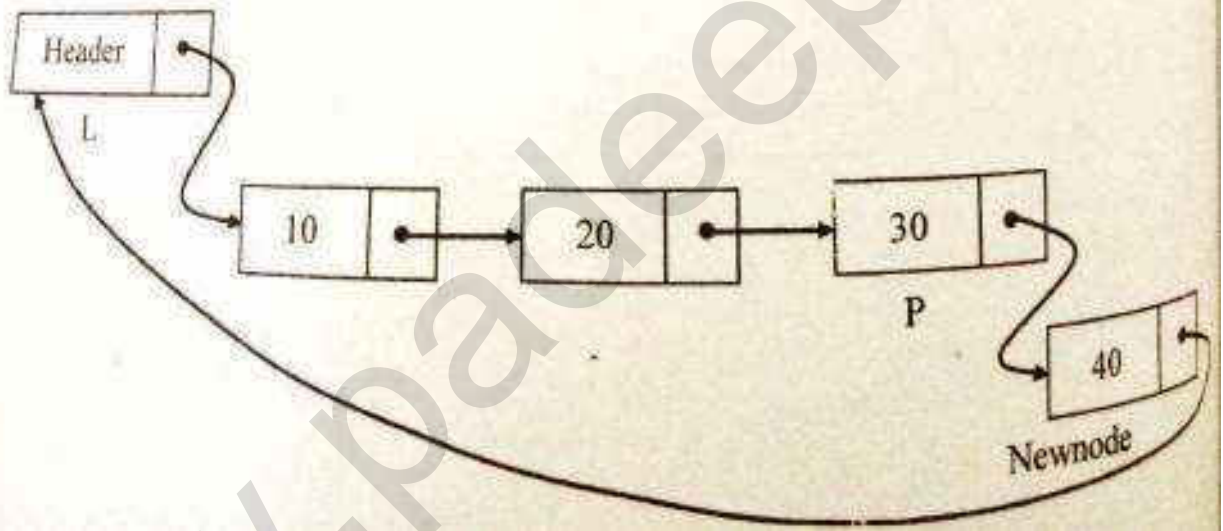
Routine to delete an element from the beginning

```
void del_first(List L)
{
position temp;
temp=L->next;
L->next=temp->next;
free(temp);
}
```
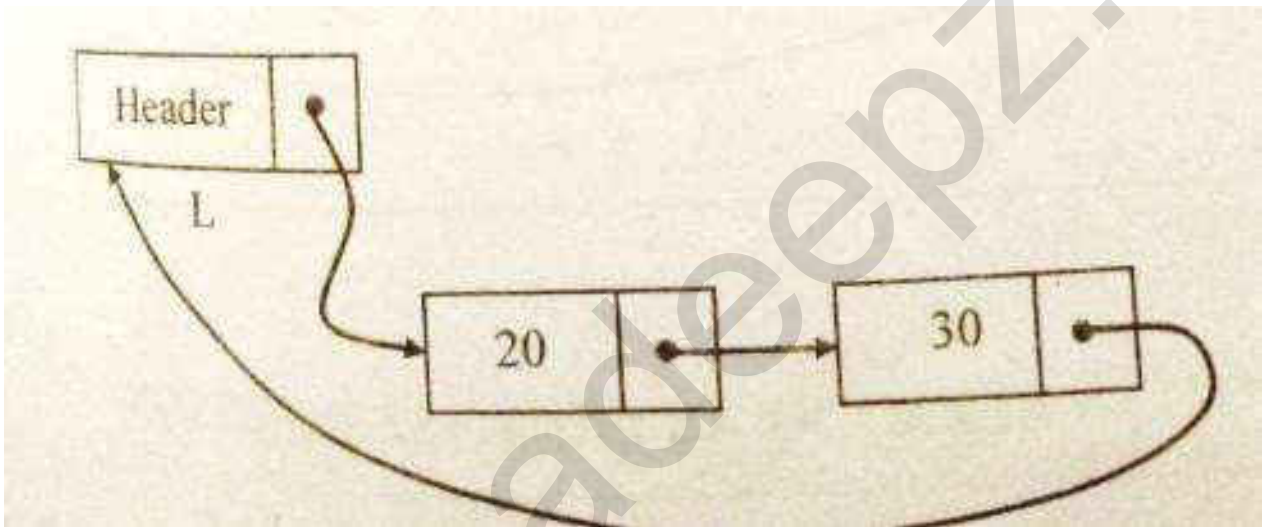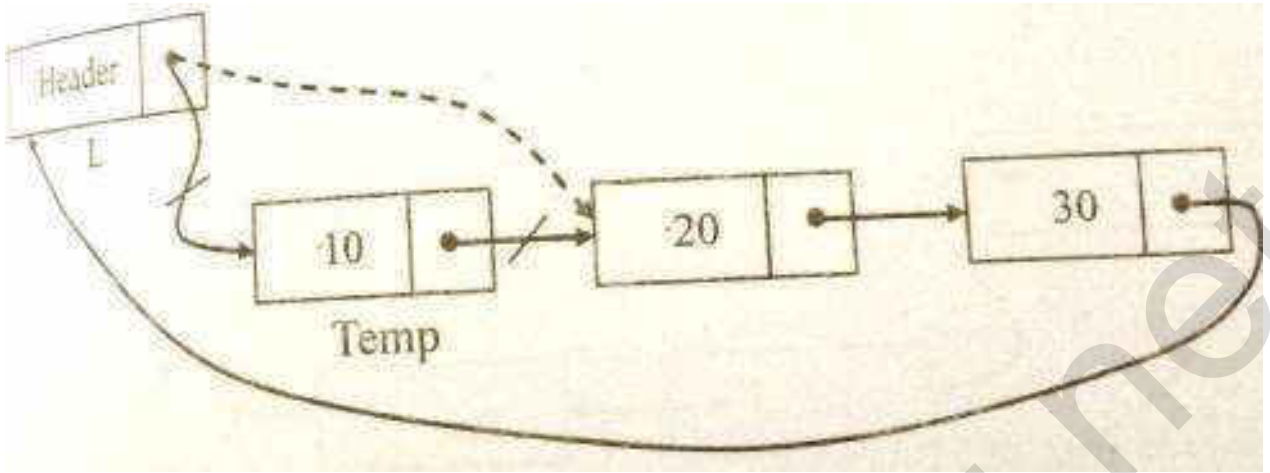
Routine to delete an element from the middle

```
void del_mid(int X,List L)
{
position p, temp;
p=findprevious(X,L);
if(!Islast(P,L))
{
temp=p->next;
p->next=temp->next;
free(temp);
}
}
```

Delete (30, L)
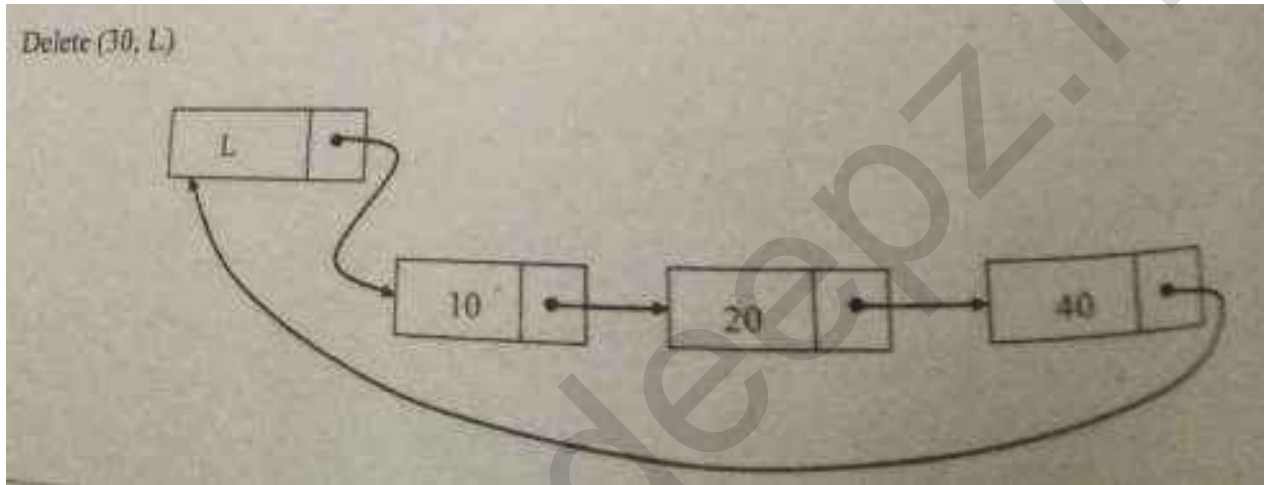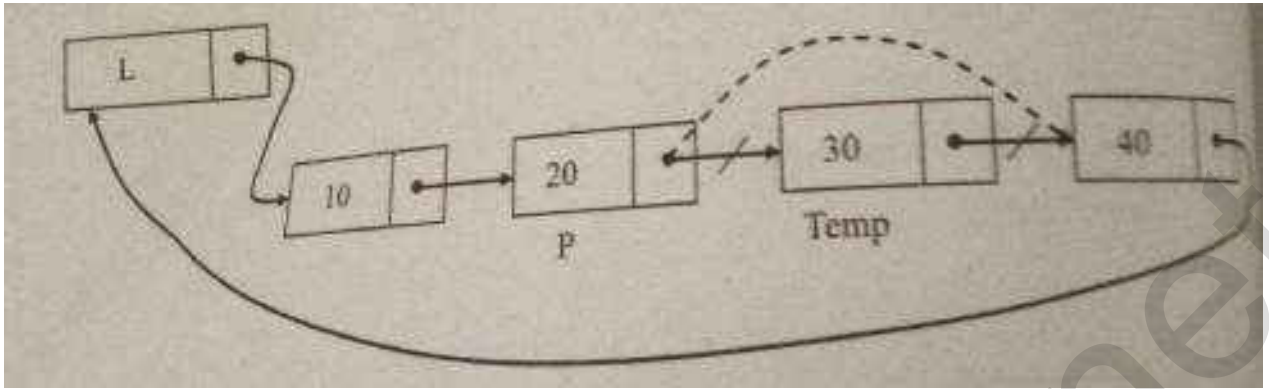
Routine to delete an element at the last position

```
void del_last(List L)
{
position p, temp;
p=L;
while(p->next->next!=L)
p=p->next;
temp=p->next;
p->next=L
free(temp);}
```
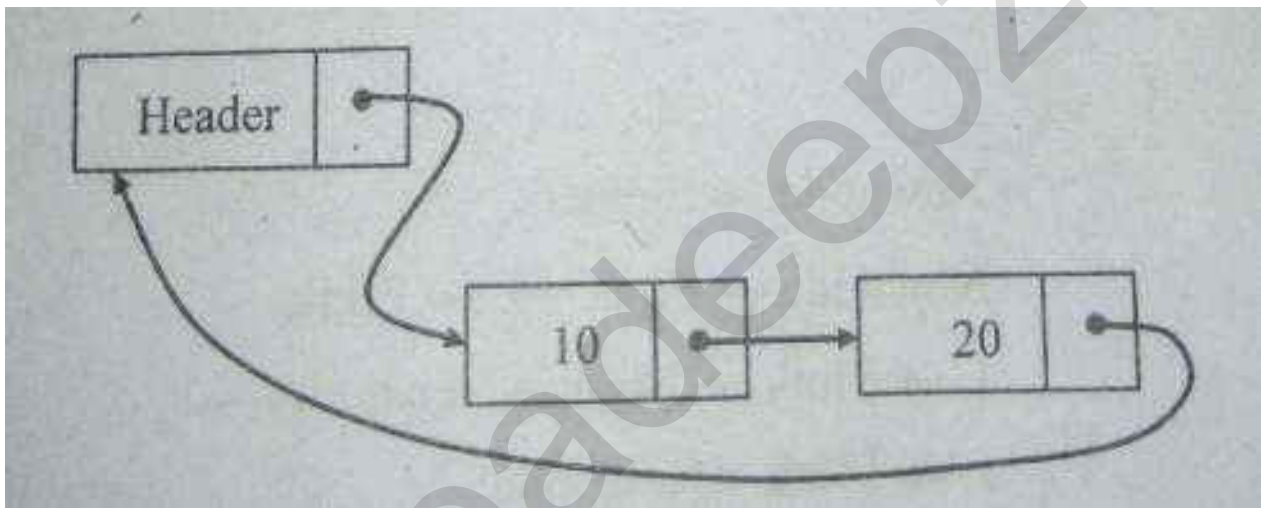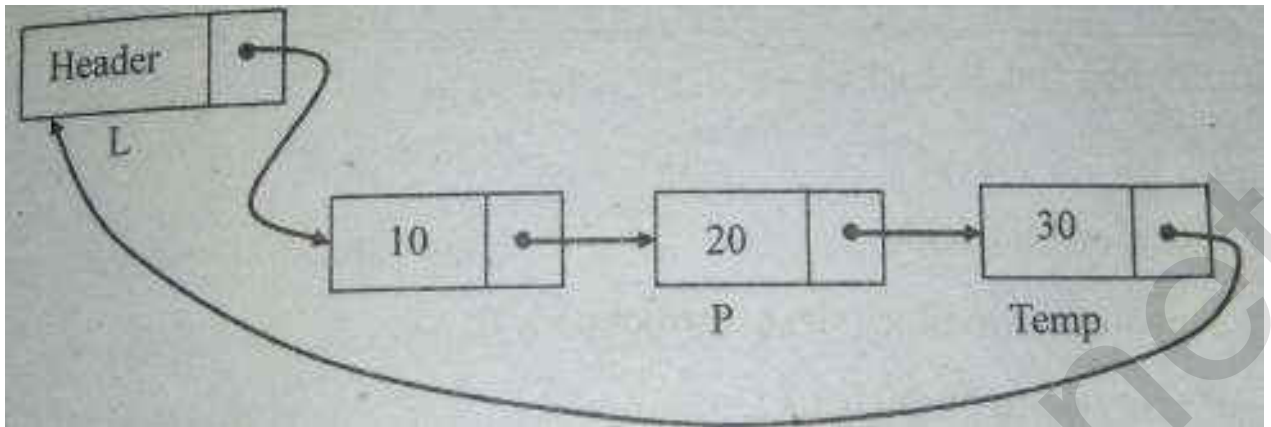
Doubly Linked circular list:

A doubly linked circular list is a doubly linked list in which the next link of the last node points to the first node and prev link of the first node points to the last node of the list.
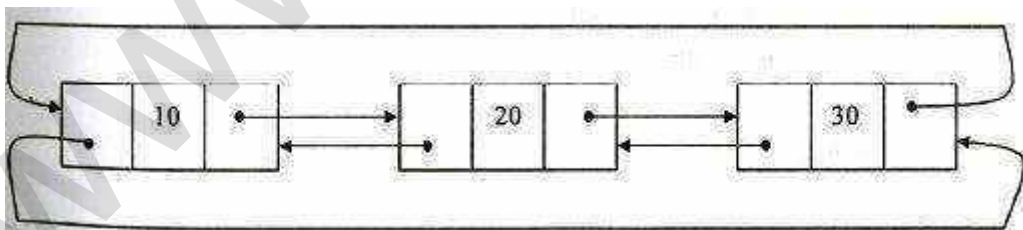
Declaration of node:

```
typedef struct node *position;
struct node
{
int data;
position next;
position prev;
};
```

Routine to insert an element in the beginning

```
void insert_beg(int  X,List L)
{
position  Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
Newnode->data=X;
Newnode->next=L->next;
L->next->prev=Newnode;
L->next=Newnode;
Newnode->prev=L;
}
}
```

Insert (10,L)

Routine to insert an element in the middle

```
void insert_mid(int  X, List L, Position p)
{
position  Newnode;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
Newnode->data=X;
Newnode->next=p->next;
p->next->prev=Newnode;
p->next=Newnode;
Newnode->prev=p;
}
}
```

insert (25, L, P)

Routine to insert an element in the last

```
void insert_last(int  X, List L)
{
position  Newnode,p;
Newnode=(struct node*)malloc(sizeof(struct node));
if(Newnode!=NULL)
{
p=L;
while(p->next!=L)
p=p->next;
Newnode->data=X;
p->next =Newnode;
Newnode->next=L;
Newnode->prev=p;
L->prev=newnode;
}
}
```
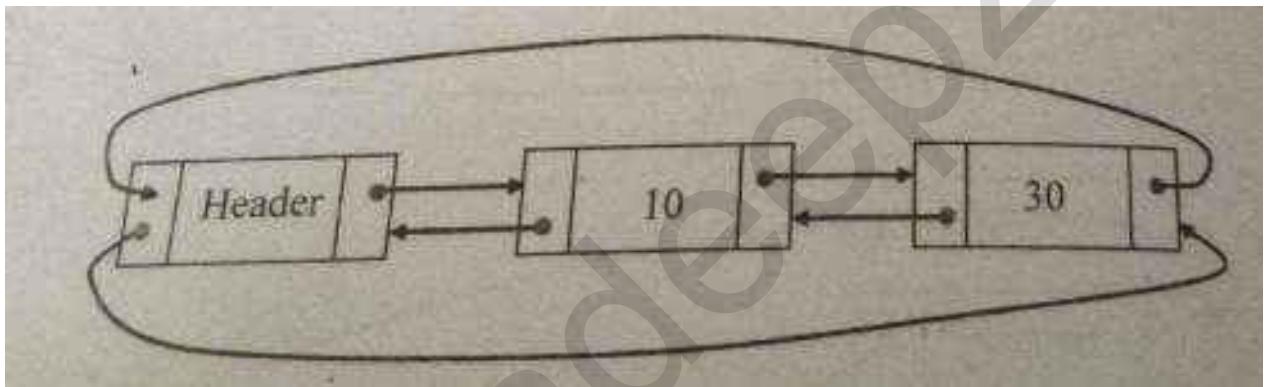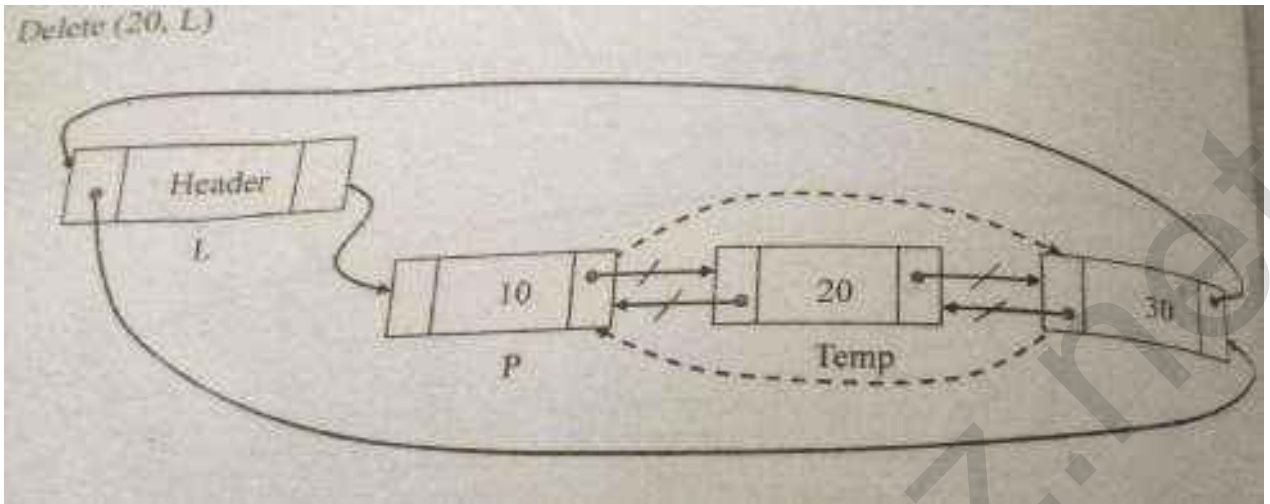
Insert (30, L)
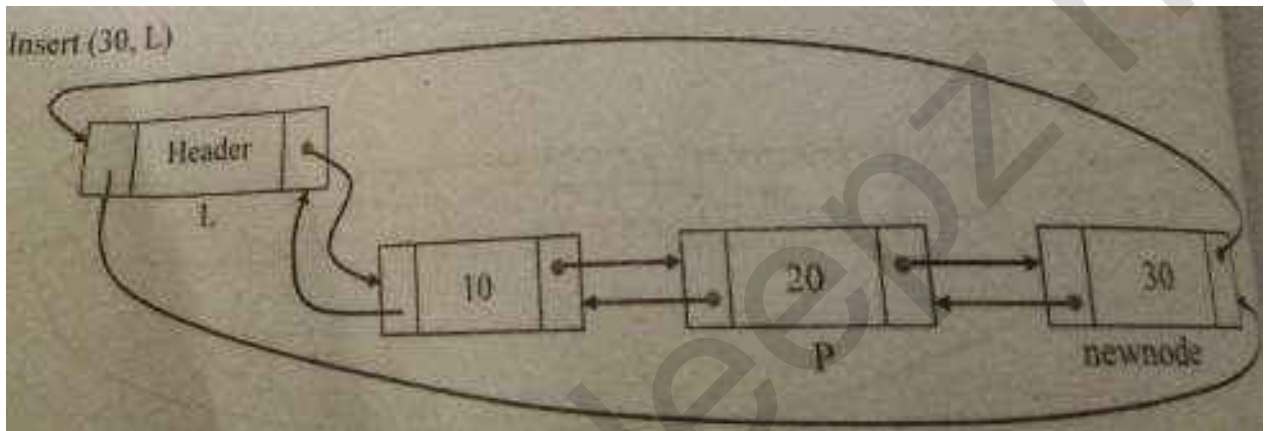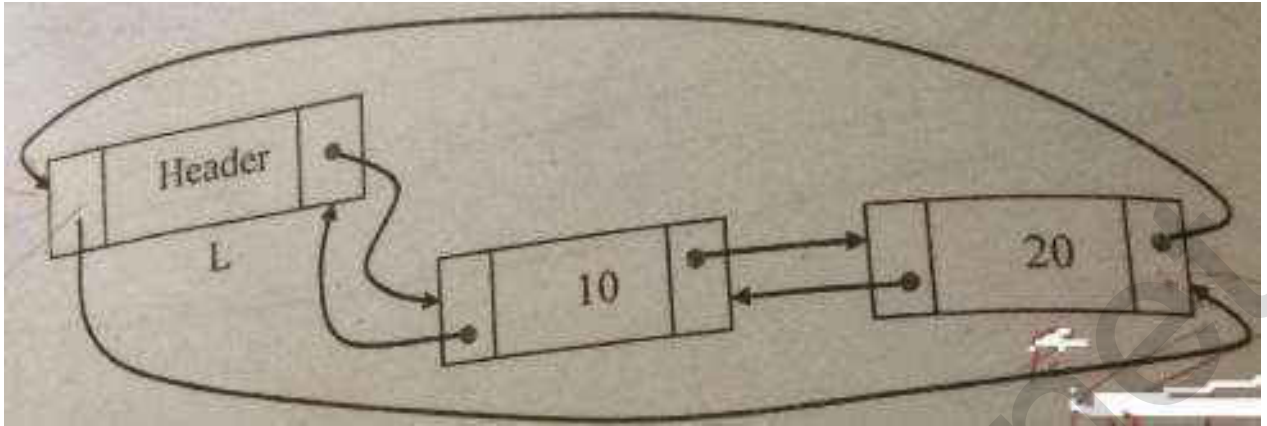
Routine to delete an element from the beginning

```
void del_first(List L)
{
position temp;
if(L->next!=NULL)
{
temp=L->next;
L->next=temp->next;
temp->next->prev=L;
free(temp);
}
```
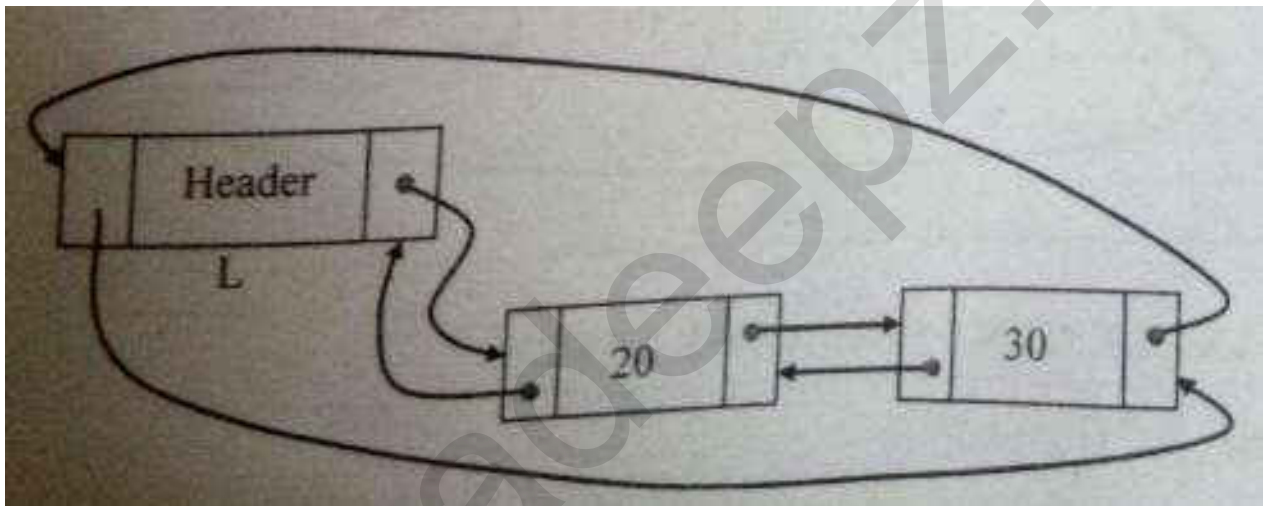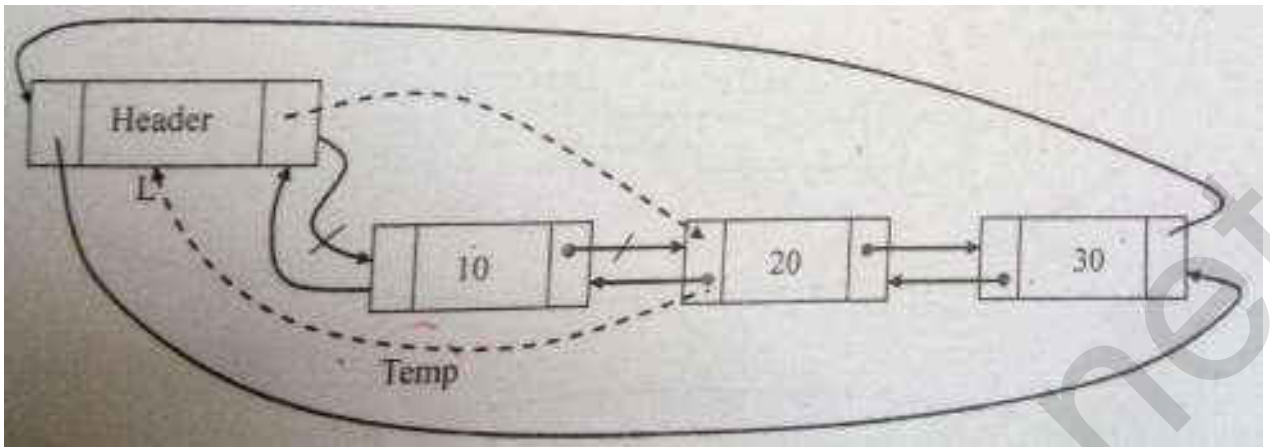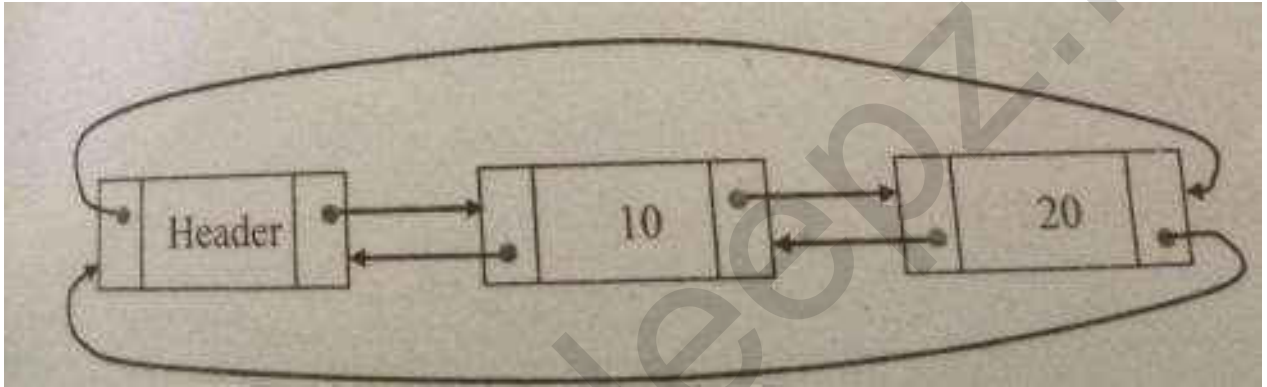
Routine to delete an element from the middle

```
void del_mid(int X,List L)
{
Position p, temp;
p=FindPrevious(X);
if(!IsLast(p,L))
{
temp=p->next;
p->next=temp-next;
temp->next->prev=p;
free(temp);
}
}
```
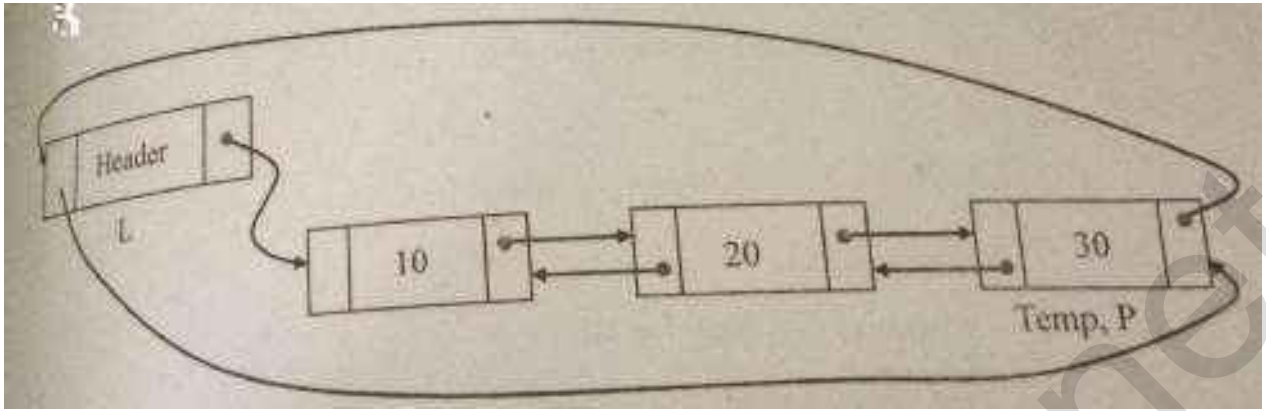
Delete (20, L)

Routine to delete an element at the last position

```
void del_last(List L)
{
position p, temp;
p=L;
while(p->next!=L)
p=p->next;
temp=p;
p->next->prev=L;
L->prev=p->prev;
free(temp);
}
```

Advantages of Circular linked List

- It allows to traverse the list starting at any point.
- It allows quick access to the first and last records.
- Circularly doubly linked list allows to traverse the list in either direction.

Applications of List:

1.Polynomial ADT

2.Radix sort

3.Multilist

Polynomial Manipulation

Polynomial manipulations such as addition, subtraction & differentiation etc.. can be performed using linked list

Declaration for Linked list implementation of Polynomial ADT

```
struct poly
{
int coeff;
int power;
struct poly *next;
}*list1,*list2,*list3;
```

Creation of the Polynomial

```
poly create(poly*head1,poly*newnode1)
{
    poly *ptr;
    if(head1==NULL)
    {
        head1=newnode1;
        return (head1);
    }
    else
    {
        ptr=head1;
        while(ptr->next!=NULL)
        ptr=ptr->next;
        ptr->next=newnode1;
    }
    return(head1);
}
```

Addition of two polynomials

```
void add()
{
    poly *ptr1, *ptr2, *newnode ;
    ptr1= list1;
    ptr2 = list2;
    while( ptr1 != NULL && ptr2 != NULL )
    {
        newnode = (struct poly*)malloc( sizeof ( struct poly ));
if( ptr1 -> power = = ptr2 -> power )
        {
            newnode -> coeff = ptr1 -> coeff + ptr2 -> coeff;
```

```
                              newnode -> power = ptr1 -> power ;
newnode -> next = NULL;
list3 = create( list3, newnode );
ptr1 = ptr1 -> next;
ptr2 = ptr2 -> next;
            }
        else if(ptr1 -> power > ptr2 -> power )
        {
                newnode -> coeff = ptr1 -> coeff;
                newnode -> power = ptr1 -> power;
                newnode -> next = NULL;
                list3 = create( list3, newnode );
                ptr1 = ptr1 -> next;
        }
        else
        {
                newnode -> coeff = ptr2 -> coeff;
                newnode -> power = ptr2 -> power;
                newnode -> next = NULL;
                list3 = create( list3, newnode );
ptr2 = ptr2 -> next;
        }}}
```

Subtraction of two polynomial
```
        {
void sub()   poly *ptr1, *ptr2, *newnode;
        ptr1 = list1;
        ptr2 = list2;
        while( ptr1 != NULL && ptr2 != NULL )
        {
                newnode = (struct poly*)malloc( sizeof (struct poly)) ;
      if(ptr1->power==ptr2->power)
                {
```

```
                    newnode->coeff=(ptr1-coeff)-(ptr2->coeff);
                    newnode->power=ptr1->power;
newnode->next=NULL;
list3=create(list3,newnode);
ptr1=ptr1->next;


ptr2=ptr2->next;
                }
            else
            {
                    if(ptr1-power>ptr2-power)
                    {
                            newnode->coeff=ptr1->coeff;
                            newnode->power=ptr1->power;
                            newnode->next=NULL;
                            list3=create(list3,newnode);
                            ptr1=ptr1->next;
                    }
            else
            {
                    newnode->coeff=-(ptr2->coeff);
                    newnode->power=ptr2->power;
                    newnode->next=NULL;
                    list3=create(list3,newnode);
                    ptr2=ptr2->next;
            }
        }
    }
}
```

Polynomial Differentiation:

```
void diff()

{

    poly *ptr1, *newnode;

        ptr1 = list1;

    while( ptr1 != NULL)

    {

    newnode = (struct poly*)malloc( sizeof (struct poly));

    newnode->coeff=(ptr1-coeff)*(ptr1->power);

    newnode->power=ptr1->power-1;
```

```
newnode->next=NULL;

list3=create(list3,newnode);

ptr1=ptr1->next;

}

}
```

Polynomial Multiplication

```
void mul()

{

    poly *ptr1, *ptr2, *newnode ;

    ptr1= list1;

    ptr2 = list2;

    while( ptr1 != NULL && ptr2 != NULL )

    {

            newnode = (struct poly*)malloc( sizeof ( struct poly ));

if( ptr1 -> power = = ptr2 -> power )

            {

                    newnode -> coeff = ptr1 -> coeff * ptr2 -> coeff;

                    newnode -> power = ptr1 -> power+ptr2->power; ;

                    newnode -> next = NULL;

                    list3 = create( list3, newnode );

                    ptr1 = ptr1 -> next;

                    ptr2 = ptr2 -> next;

            }}

}
```

gate
questions
withanswers
Linked Lists

1. In a circular linked list organization, insertion of a record involves modification of

    A. One pointer
    B. Two pointers
    C. Three pointers
    D. No pointer

    [B] Suppose we want to insert node A to which we have pointer p , after pointer q then we will

    Have following pointer operations

    1.p->next=q->next;

    2.q->next = p;

    So we have to do two pointer modifications

2. Consider a singly linked list having n nodes. The data items d_1, d_2, …., d_n are stored in the n nodes. Let Y be a pointer to the jth node ($1 \le j \le n$) in which d_j is stored. A new data item d stored in a node with address Y is to be inserted. Give an algorithm to insert d into the list to obtain a list having items d_1, d_2, ……, d_{j-1}, d, d_j, …..d_n in that order without using the header.

    Algorithm 1 insert_mid( )

    1: create newnode

    2: newnode->next=y->next

    3: newnode->data=y->data   /*which is dj*/

    4: y->next=newnode

    5: y->data=d

    Explanation:

    Since we didn't have the address of node which is previous to Y.

    So insert a new node after Y.

    And copy the data of Y to new node and modify data field of Y to d.

    Hence we get required sequence of data as d1,d2,…,dj-1,d,dj,…dn

3. Linked lists are not suitable for data structures for which one of the following problems?

   (A) Insertion sort
   (B) Binary search
   (C) Radix sort
   (D) Polynomial manipulation

[B] For binary search, if we are using array, then we can go to middle of array by just dividing index of array by 2. Since array is stored in contiguous memory. But that is not true in case of linked list. If you want to access middle of list then each time you have to traverse from its head. Hence use of linked list is not good idea for binary search.

4. The concatenation of two lists is to be performed in O(1) time. Which of the following implementations of a list should be used?

   (A) singly linked list   (B) doubly linked list

   (C) circular doubly linked list   (D) array implementation of list

[C} For merging of list you have to point next pointer of last node of first list to first node of 2nd list. To do this in O(1) time circular double list is useful. You can go to last node 1st list by head1->next->previous. And modify this field pointing to head2->next. And also modify head2->next->previous to head1->next.

5. (a)   Let p be a pointer as shown in the figure in a singly linked list.

P   Cell i   Cell (i +1)   Cell (i +2)   Cell (i +3)
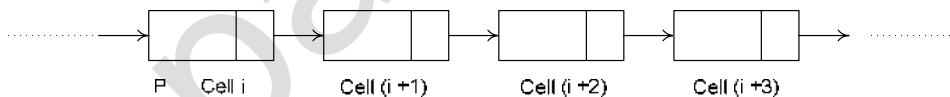
What do the following assignment statements achieve?

q: = p→ next

p → next: = q→ next

q → next: = (q → next) → next

(p → next) →next: = q

5a)

Que   Ans

q: = p→ next   cell i->cell (i+1) ->cell(i+2)->cell(i+3)   p->cell i,q->cell(i+1)

p → next: = q→ next   cell i→cell(i+2)->cell(i+3) & cell(i+1) ->cell(i+2)->cell(i+3)

q → next: = (q → next) → next   cell i->cell(i+2)->cell(i+3) & cell(i+1)->cell(i+3)

(p → next) →next: = q   cell->i->cell(i+2)->cell(i+1)->cell(i+3)

Write a constant time algorithm to insert a node with data D just before the node with address p of a singly linked list.

Constant time algorithm is

Insert before(p)

{

   n =newnode();

   n->next = p->next;

   p->next =n

   n->data=p->data;

   p->data = D;

}

As we can't actually insert before any node to with we have a pointer in singly linked list.

So idea is to insert a node after p , copy the data of p in new node and copy new data in node p.

6.   In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is

          A. $\log_2 n$
          B.   n/2
          C. $\log_2 n - 1$
          D.   n

[D] In worst case the element is in last node. So we require n comparisons in worst case.

7.   In a circular linked list organization, insertion of a record involves modification of

          A.   One pointer
          B.   Two pointers
          C.   Three pointers
          D.   No pointer

[B] Suppose we want to insert node A to which we have pointer p , after pointer q then we will

Have following pointer operations

   1.p->next=q->next;

   2.q->next = p;

So we have to do two pointer modifications

8.   Consider a singly linked list having n nodes. The data items d_1, d_2, …., d_n are stored in the n nodes. Let Y be a pointer to the jth node ($1 \leq j \leq n$) in which d_j is stored. A new data item d stored in a node with address Y is to be inserted. Give an algorithm to insert d into the list to obtain a list having items d_1, d_2, ……, d_{j-1}, d, d_j, …..d_n in that order without using the header.

Algorithm 1 insert_mid( )

 1: create newnode

 2: newnode->next=y->next

 3: newnode->data=y->data   /*which is dj*/

 4: y->next=newnode

 5: y->data=d

Explanation:

Since we didn't have the address of node which is previous to Y.

So insert a new node after Y.

And copy the data of Y to new node and modify data field of Y to d.

Hence we get required sequence of data as d1,d2,…,dj-1,d,dj,…dn

9.   The following C function takes a singly-linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1,2,3,4,5,6,7 in the given order. What will be the contents of the list after the function completes execution?

```
struct node  {

    int value;

    struct node *next;
```

```
};

void rearrange (struct node *list) {

    struct node *p, *q;

    int temp;


    if (!list || !list -> next) return;

    p = list; q = list -> next;

    while (q)  {

        temp = p -> value; p -> value =q -> value;

        q -> value = temp; p = q -> next;

        q = p ? -> next : 0;

    }

}
```

    (A) 1,2,3,4,5,6,7
    (B) 2,1,4,3,6,5,7
    (C) 1,3,2,5,4,7,6
    (D) 2,3,4,5,6,7,1

[B] The q pointer always point to next node of p. And here p is modified first. So swapping is done

10.   The data blocks of a very large file in the Unix file system are allocated using

    (A) contiguous allocation
    (B) linked allocation
    (C) indexed allocation
    (D) an extension of indexed allocation

[D] The file's inode contains pointer to first 10 data blocks.The 11th pointer in inode points at an indirect block that contain 128 data blocks. And so on…

11.   The following C function takes a singly linked list of integers as a parameter and rearranges the elements of the list. The list is represented as pointer to structure. The function is called with the list containing integers 1, 2, 3, 4, 5, 6, 7 in the given order. What will be the contents of the list after the function completes?

```
struct node {int value; struct node *next;};


void rearrange(struct node *list) {

    struct node *p, *q;
```

```
int temp;

if(!list || !list → next) return;

p = list; q = list → next;

while(q) {

        temp = p → value;

        p → value = q → value;

        q → value = temp;

        p = q → next;

        q = p? p → next : 0;

    }

}
```

(A) 1, 2, 3, 4, 5, 6, 7   (B) 2, 1, 4, 3, 6, 5, 7

(C) 1, 3, 2, 5, 4, 7, 6   (D) 2, 3, 4, 5, 6, 7, 1

[B] The q pointer always point to next node of p. And here p is modified first. So swapping is done only once for each

12. Let P be a singly linked list. Let Q be the pointer to an intermediate node x in the list. What is the worst case time complexity of the best-known algorithm to delete the node x from the list?

        (A) O(n)
        (B) O($\log^2 n$)
        (C) O(log n)
        (D) O(1)

[A] As Q is pointing to node X.

So the following algorithm will delete the node in O(1)

Algorithm-:

Delete()

Step1. Q->data =: (Q->next)->data

Step2. temp =: Q->next   (temp is temporary pointer variable of type list)

Step3. Q->next =: (Q->next)->next

Step4. delete temp

13. Suppose each set is represented as a linked list with elements in arbitrary order. Which of the operations among union, intersection, membership, and cardinality will be the slowest?

  (A) Union only
  (B) intersection, membership
  (C) membership, cardinality
  (D) union, intersection

[D] For intersection n*n comparisons are required.

For union, just merging of two list will not work. We have to find out common elements which require again n*n comparisons.

For membership only n comparisons are needed.

For Cardinality n comparisons. (ie. For each node check next !=NULL and increment)

14. Circularly linked list is used to represent a Queue. A single variable p is used to access the Queue. To which node should p point such that both the operations enQueue and deQueue can be performed in constant time?
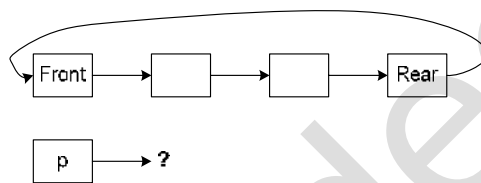
  (A) Rear node
  (B) Front node
  (C) Not possible with a single pointer
  (D) Node next to front

[A] p points to rear node

For enQueue

1: create newnode

2: newnode->next=p->next /*which is front node*/

3: p->next=newnode

4: /*rear=newnode;*/

5: p=rear

For deQueue

1:temp=p->next /*temp is pointing to front node

2: p->next=p->next->next

3:/* front=p->next*/

4:delete(temp)

Stack ADT – Evaluating arithmetic expressions- other applications- Queue ADT – circular queue implementation – Double ended Queues – applications of queues

STACK

- Stack is a Linear Data Structure that follows Last In First Out(LIFO) principle.
- Insertion and deletion can be done at only one end of the stack called TOP of the stack.
- Example: - Pile of coins, stack of trays

STACK ADT:
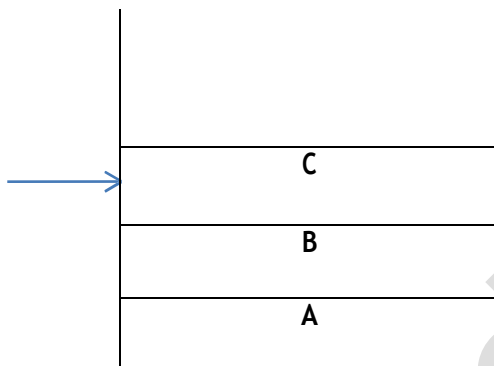
STACK MODEL

TOP pointer

It will always point to the last element inserted in the stack.

For empty stack, top will be pointing to -1. (TOP = -1)

Operations on Stack (Stack ADT)

Two fundamental operations performed on the stack are PUSH and POP.

    (a) PUSH:

        It is the process of inserting a new element at the Top of the stack.

For every push operation:

1. Check for Full stack (overflow).
2. Increment Top by 1. (Top = Top + 1)

3. Insert the element X in the Top of the stack.

(b) POP:

It is the process of deleting the Top element of the stack.

For every pop operation:

1. Check for Empty stack (underflow).
2. Delete (pop) the Top element X from the stack
3. Decrement the Top by 1. (Top = Top - 1)

Exceptional Conditions of stack

1. **Stack Overflow**

   - An Attempt to insert an element X when the stack is Full, is said to be stack overflow.
   - For every Push operation, we need to check this condition.

2. **Stack Underflow:**

   - An Attempt to delete an element when the stack is empty, is said to be stack underflow.
   - For every Pop operation, we need to check this condition.

12.4 Implementation of Stack

Stack can be implemented in 2 ways.

1. Static Implementation (Array implementation of Stack)
2. Dynamic Implementation (Linked List Implementation of Stack)

12.4.1 Array Implementation of Stack

- Each stack is associated with a Top pointer.
- For Empty stack, Top = -1.
- Stack is declared with its maximum size.

Array Declaration of Stack:

#define ArraySize 5
int S [Array Size];
   or
int S [5];

(i) Stack Empty Operation:

- Initially Stack is Empty.

- With Empty stack Top pointer points to – 1.

- It is necessary to check for Empty Stack before deleting (pop) an element from the stack.

Routine to check whether stack is empty

```
int IsEmpty ( Stack S )
{
   if( Top = = - 1 )
   return(1);
}
```

EMPTY Stack

TOP = -1

(ii) Stack Full Operation:

- As we keep inserting the elements, the Stack gets filled with the elements.

- Hence it is necessary to check whether the stack is full or not before inserting a new element into the stack.

Routine to check whether a stack is full

```
int IsFull ( Stack S )

   {    if( Top = = Arraysize – 1 )

      return(1);

   }
```

TOP → 4   50
3   40
2   30
1   20
0   10
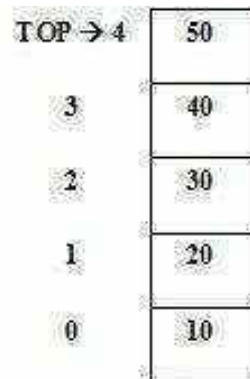
FULL Stack

(ii) Push Operation

- It is the process of inserting a new element at the Top of the stack.

- It takes two parameters. Push(X, S) the element X to be inserted at the Top of the Stack S.

- Before inserting an Element into the stack, check for Full Stack.

- If the Stack is already Full, Insertion is not possible.

- Otherwise, Increment the Top pointer by 1 and then insert the element X at the Top of the Stack.

Routine to push an element into the stack

```
void Push ( int X , Stack S )
{
if ( Top = = Arraysize - 1)
        Error("Stack is full!!Insertion is not possible");
 else
    {       Top = Top + 1;
          S [ Top ] =X;
    }
}
```
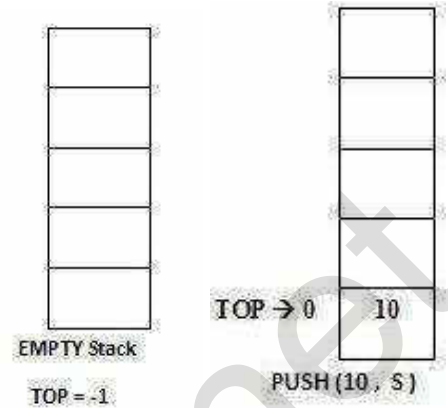
(iv)   Pop Operation

- It is the process of deleting the Top element of the stack.
- It takes only one parameter. Pop(X).The element X to be deleted from the Top of the Stack.
- Before deleting the Top element of the stack, check for Empty Stack.
- If the Stack is Empty, deletion is not possible.
- Otherwise, delete the Top element from the Stack and then decrement the Top pointer by 1.

Routine to Pop the Top element of the stack

```
void Pop ( Stack S )
{
 if ( Top = = - 1)
     Error ( "Empty stack! Deletion not possible");
else
{     X = S [ Top ] ;
      Top = Top – 1 ;
}
 }
```
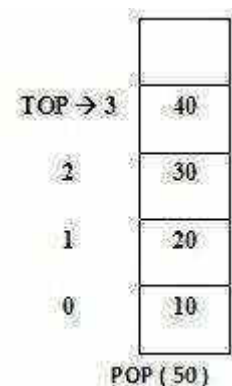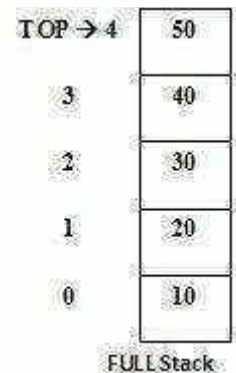
(v) Return Top Element

- Pop routine deletes the Top element in the stack.
- If the user needs to know the last element inserted into the stack, then the user can return the Top element of the stack.
- To do this, first check for Empty Stack.
- If the stack is empty, then there is no element in the stack.
- Otherwise, return the element which is pointed by the Top pointer in the Stack.

Routine to return top Element of the stack

```
int TopElement(Stack S)
{
  if(Top==-1)
  {
     Error("Empty stack!!No elements");
     return 0;
  }
 else
     return S[Top];
}
```


EMPTY Stack

TOP = -1

FULL Stack

Implementation of stack using Array
/* static implementation of stack*/

```
#include<stdio.h>
#include<conio.h>
#define size 5
int stack [ size ];
int top;
void push( )
{
      int n ;
      printf( "\n Enter item in stack" ) ;
      scanf( " %d " , &n ) ;
      if( top = = size - 1)
      {
              printf( "\nStack is Full" ) ;
      }
      else
      {
              top = top + 1 ;
```

```
                        stack [ top ] = n ;
          }
}
void pop( )
{
          int item;
          if( top = = - 1)
          {
                        printf( "\n Stack is empty" );
          }
          else
          {
                        item = stack[ top ] ;
                        printf( "\n item popped is = %d" , item );
                        top - -;
          }
}
void display( )
{
          int i;
          printf("\n item in stack are");
          for(i = top; i > = 0; i --)
          printf("\n %d", stack[ i ] );
}
void main( )
{
          char ch,ch1;
          ch = 'y';
          ch1 = 'y';
          top = -1;
          clrscr( );
          while(ch !='n')
          {
                        push( );
                        printf("\n Do you want to push any item in stack y/n");
                        ch=getch( );
          }
          display( );
          while( ch1!='n' )
          {
                        printf("\n Do you want to delete any item in stack y/n");
                        ch1=getch( );
                        pop( );
          }
          display( );
          getch( );}
```

OUTPUT:

Enter item in stack20
Do you want to push any item in stack y/n
Enter item in stack25
Do you want to push any item in stack y/n
Enter item in stack30
Stack is Full
Do you want to push any item in stack y/n
item in stack are
25
20
15
10
5
Do you want to delete any item in stack y/n
item popped is = 25
Do you want to delete any item in stack y/n
item popped is = 20
Do you want to delete any item in stack y/n
item popped is = 15
item in stack are
10
5

Linked list implementation of Stack

- Stack elements are implemented using SLL (Singly Linked List) concept.

- Dynamically, memory is allocated to each element of the stack as a node.

Type Declarations for Stack using SLL

struct node;

typedef struct node *stack;

typedef struct node *position;

stack S;

struct node{ int

data; position

next;};

int IsEmpty(Stack S);

void Push(int x, Stack S);

void Pop(Stack S);

int TopElement(Stack S);

(i) Stack Empty Operation:

- Initially Stack is Empty.
- With Linked List implementation, Empty stack is represented as S -> next = NULL.
- It is necessary to check for Empty Stack before deleting (pop) an element from the stack.

Routine to check whether the stack is empty

```
int IsEmpty( Stack S)
{
if ( S -> next = = NULL)
  return ( 1 );
}
```

S

| HEADER | NULL |

EMPTY STACK
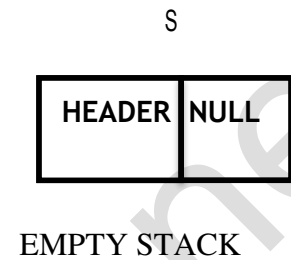
(ii) Push Operation

- It is the process of inserting a new element at the Top of the stack.
- With Linked List implementation, a new element is always inserted at the Front of the List.(i.e.) S -> next.
- It takes two parameters. Push(X, S) the element X to be inserted at the Top of the StackS.
- Allocate the memory for the newnode to be inserted.
- Insert the element in the data field of the newnode.
- Update the next field of the newnode with the address of the next node which is stored in the S -> next.
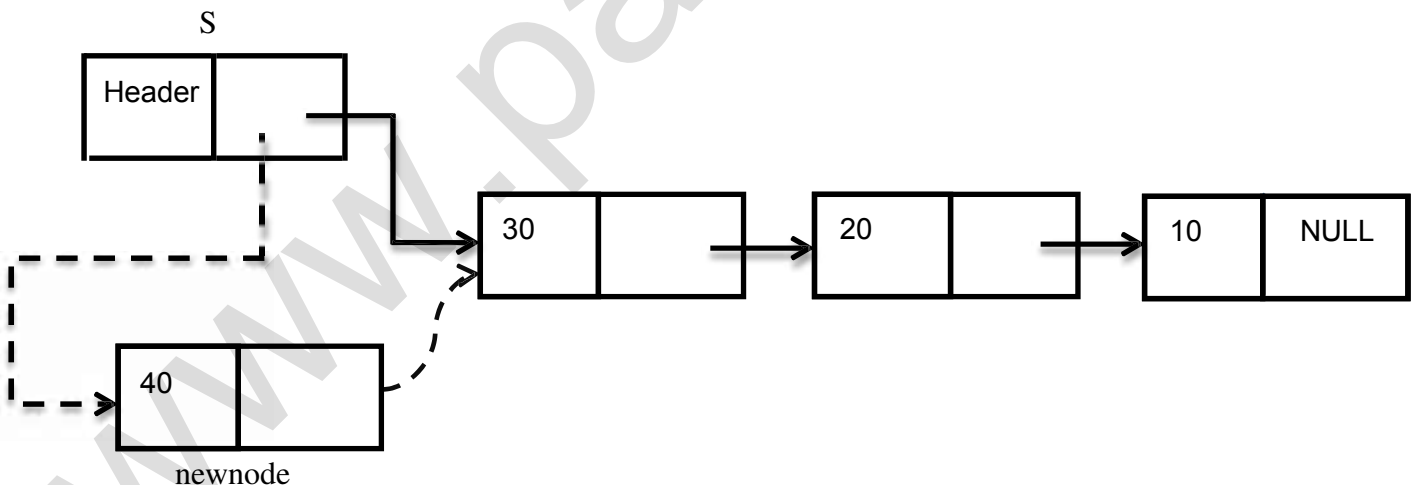
Before Insertion

Push routine   **/*Inserts element at front of the list**

```
void push(int X, Stack S)
{
     Position newnode, Top;
        newnode = malloc (sizeof( struct node ) );
        newnode -> data = X;
        newnode -> next = S -> next;
        S -> next = newnode;
        Top = newnode;
}
```

S

Header	

| 40 | | — | 30 | | — | 20 | | — | 10 | NULL |

After Insertion

TOP
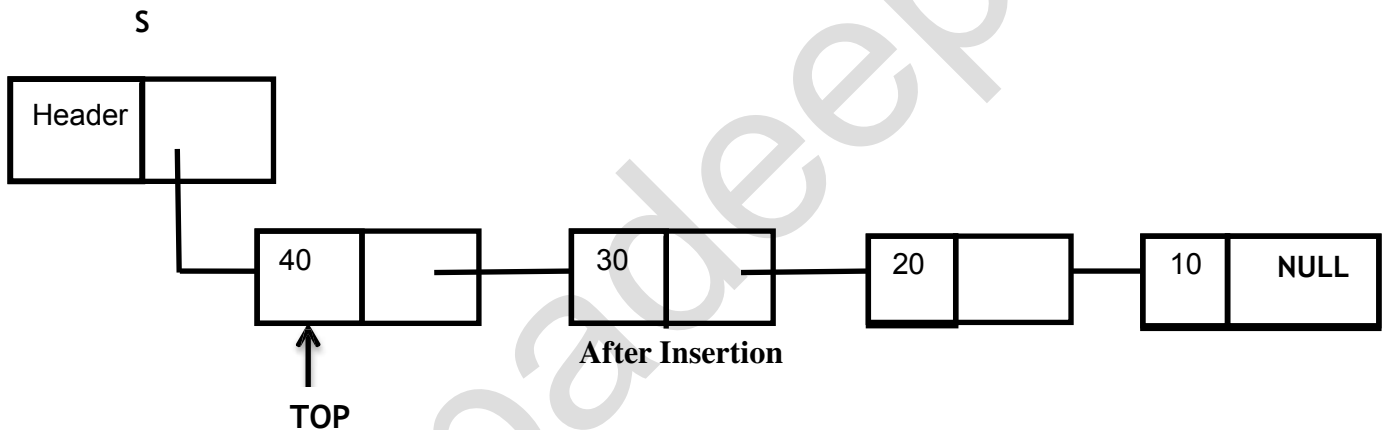
(iii)   Pop Operation

- It is the process of deleting the Top element of the stack.
- With Linked List implementations, the element at the Front of the List
  (i.e.) S -> next is always deleted.
- It takes only one parameter. Pop(X).The element X to be deleted from the Front of the List.
- Before deleting the front element in the list, check for Empty Stack.
- If the Stack is Empty, deletion is not possible.
- Otherwise, make the front element in the list as "temp".
- Update the next field of header.
- Using free ( ) function, Deallocate the memory allocated for temp node.

S

| Header | |

| 40 | | → | 30 | | → | 20 | | → | 10 | NULL |

TOP
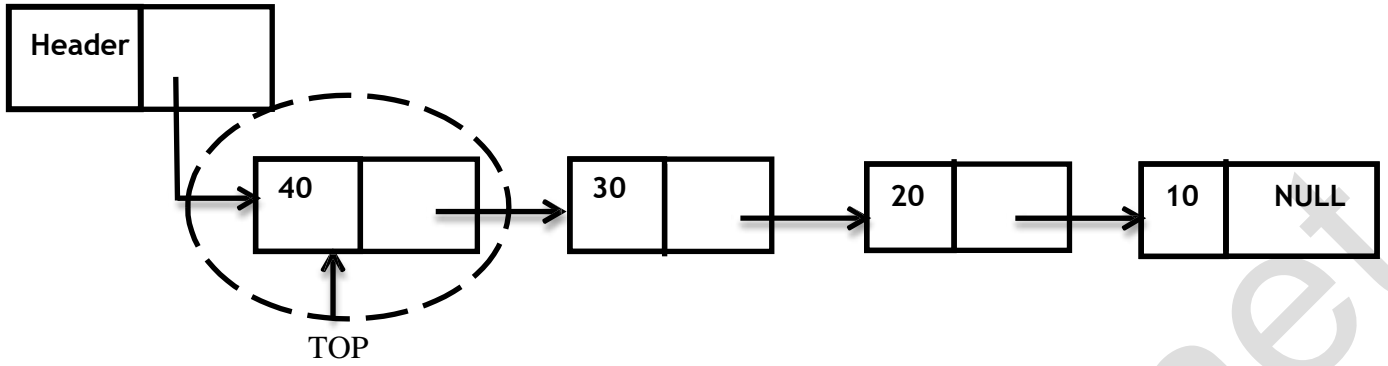
Before Deletion

Pop routine   **/*Deletes the element at front of list**

```
void Pop( Stack S )
{
        Position temp, Top;
         Top = S -> next;
         if( S -> next = = NULL)
                Error("empty stack! Pop not possible");
         else
        {
                Temp = S -> next;
                S -> next = temp -> next;
                free(temp);
                Top = S -> next;
        } }
```

| Header | |

| 40 | | → | 30 | | → | 20 | | → | 10 | NULL |

S

| HEADER | |

| 30 | | → | 20 | | → | 10 | NULL |

TOP
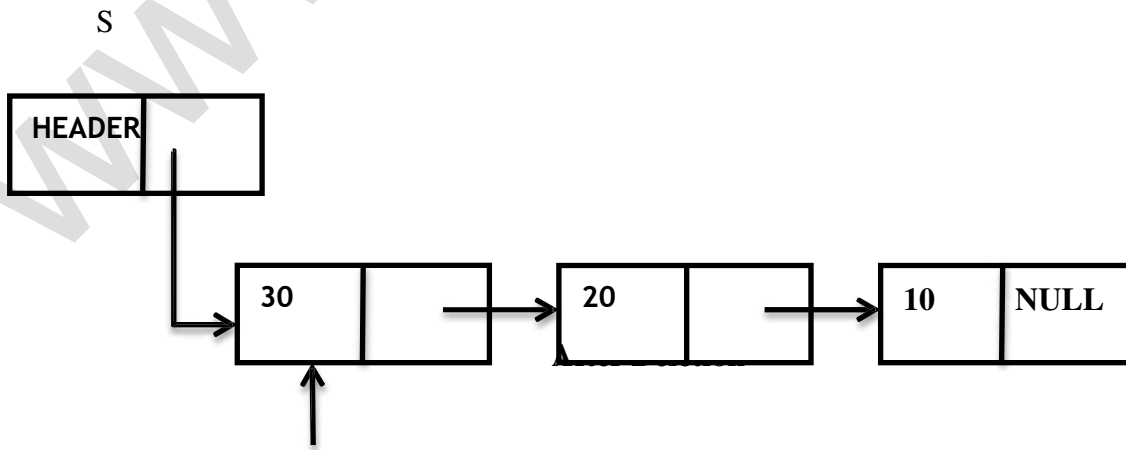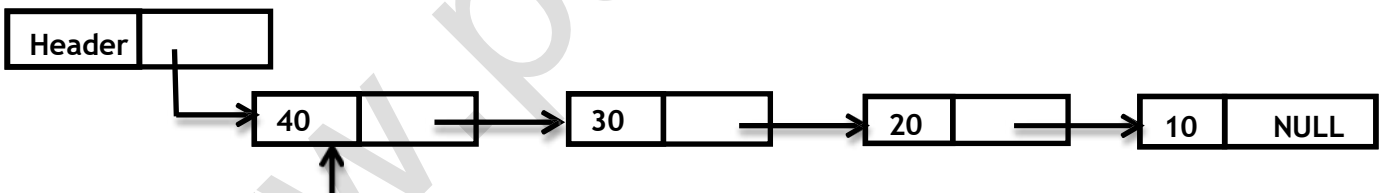
(iv)   Return Top Element

- Pop routine deletes the Front element in the List.

- If the user needs to know the last element inserted into the stack, then the user can return the Top element of the stack.

- To do this, first check for Empty Stack.

- If the stack is empty, then there is no element in the stack.

- Otherwise, return the element present in the S -> next -> data in the List.

Routine to Return Top Element

```
int TopElement(Stack S)

{

   if(S->next==NULL)

    {

       error("Stack is empty");

       return 0;

   else

       return  S->next->data;

}
```

S

TOP
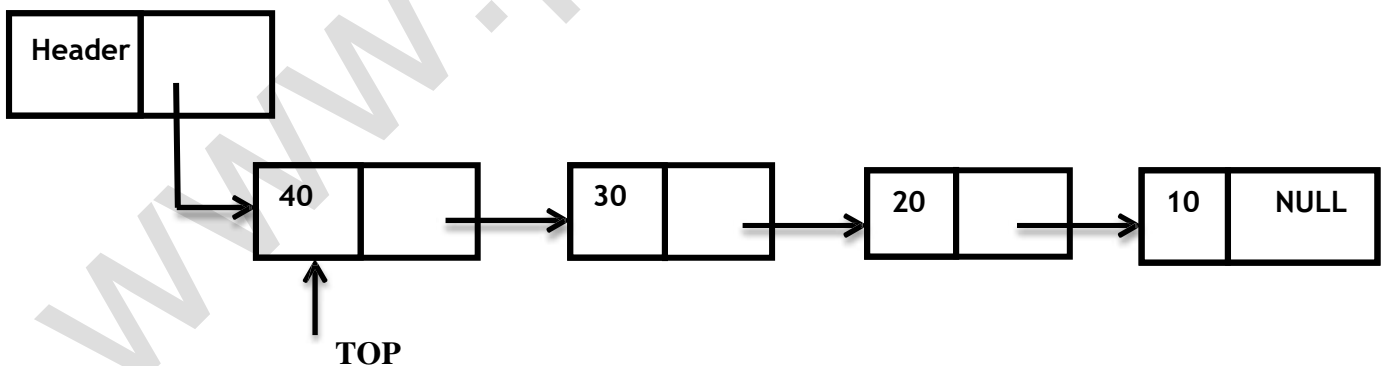
Implementation of stack using 'Linked List'

```c
/* Dynamic implementation of stack*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct node * position ;
struct node
{
 int data ;
 position next ;
 } ;
 void create( ) ;
void push( ) ;
void pop( ) ;
void display( ) ;
position s, newnode, temp, top ; /* Global Declarations */
void main() {
 /* Main Program */
 int op ;
 clrscr( ) ;
 do {
  printf( "\n ### Linked List Implementation of STACK Operations ### \n\n" ) ;
  printf( "\n Press 1-create\n 2-Push\n 3-Pop\n 4-Display\n5-Exit\n" ) ;
  printf( "\n Your option ? " ) ;
  scanf( " % d ", & op) ;
  switch (op) {
  case 1:
        create( ) ;
        break ;
  case 2:
        push();
        break;
  case 3:
        pop();
        break;

  case 4:
      display();
        break;
 case 5:
 exit(0);

 }
 }while(op<5);
```

```
getch();
}
 void create()
 {
   int n,i;
 s=NULL;
 printf("Enter the no of nodes to be created\n");
 scanf("%d",&n);
 newnode=(struct node*)malloc(sizeof(struct node));
 printf("Enter the data\t");
 scanf("%d",&newnode->data);
 newnode->next=NULL;
 top=newnode;
 s=newnode;
 for(i=2;i<=n;i++)
 {
  newnode=(struct node*)malloc(sizeof(struct node));
 printf("Enter the data\t");
 scanf("%d",&newnode->data);
 newnode->next=top;
 s=newnode;
 top=newnode;
 } }
 void display()
 { top=s;
 while(top!=NULL)
 {
 printf("%d->",top->data);
 top=top->next;
 }
 printf("NULL\n");
 }
 void push()
 {    top=s;
   newnode=(struct node*)malloc(sizeof(struct node));
 printf("Enter the data\t");
 scanf("%d",&newnode->data);
 newnode->next=top;
 top=newnode;
 s=newnode;
 display();
 }
 void pop()
 {
 top=s;
```

```
if(top==NULL)
printf("Empty stack\n\n");
else
{
temp=top;
printf("Deleted element is \t %d\n\n",top->data);
s=top->next;
free(temp);
display();
} }
```

Output

```
### Linked List Implementation of STACK Operations ###
 Press 1-create
 2-Push
 3-Pop
 4-Display
5-Exit
 Your option ? 1
Enter the no of nodes to be created5
Enter the data 10
Enter the data20
Enter the data30
Enter the data40
Enter the data50
 ### Linked List Implementation of STACK Operations ###
 Press 1-create
 2-Push
 3-Pop
 4-Display
5-Exit
 Your option ? 4
50->40->30->20->10->NULL
 ### Linked List Implementation of STACK Operations ###
 Press 1-create
 2-Push
 3-Pop
 4-Display
5-Exit
 Your option ?2
Enter the data60
Your option ? 4
60->50->40->30->20->10->NULL
Your option ?2
Enter the data70
Your option ? 4
```

70->60->50->40->30->20->10->NULL
Your option ?3
Deleted element is70
Your option ? 4
50->40->30->20->10->NULL

Applications of Stack

The following are some of the applications of stack:

1. Evaluating the arithmetic expressions

    o Conversion of Infix to Postfix Expression

    o Evaluating the Postfix Expression

2. Balancing the Symbols

3. Function Call

4. Tower of Hanoi

5. 8 Queen Problem

Evaluating the Arithmetic Expression

There are 3 types of Expressions

- Infix Expression

- Postfix Expression

- Prefix Expression

INFIX:

The arithmetic operator appears between the two operands to which it is being applied.

$$A / B + C$$

POSTFIX:

The arithmetic operator appears directly after the two operands to which it applies. Also called reverse polish notation.

$$((A / B) + C)$$

$$AB/C+$$

PREFIX:

The arithmetic operator is placed before the two operands to which it applies. Also called polish notation.

$$((A/B) + C)$$

$$+/ABC$$

Evaluating Arithmetic Expressions

1. Convert the given infix expression to Postfix expression
2. Evaluate the postfix expression using stack.

Algorithm to convert Infix Expression to Postfix Expression:

Read the infix expression one character at a time until it encounters the delimiter "#"
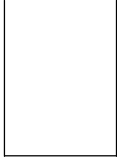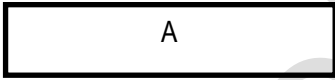
Step 1: If the character is an operand, place it on the output.

Step 2: If the character is an operator, push it onto the stack. If the stack operator has a higher or equal priority than input operator then pop that operator from the stack and place it onto the output.

Step 3: If the character is left parenthesis, push it onto the stack
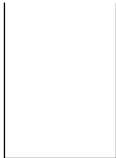
Step 4: If the character is a right parenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

E.g. Consider the following Infix expression: - A*B+(C-D/E)#

Read char	Stack	Output
A		A
*	*	A
B	+ *	AB
+	+	AB*
(	(+	AB*

Read char	Stack	Output
C	(+	AB*C
-	- (+	AB*C
D	- (+	AB*CD
/	/ - (+	AB*CD
E	/ - (+	AB*CDE
)	/ - (+	AB*CDE/-

Read char	Stack	Output
#		AB*CDE/-+

Output: Postfix expression:- AB*CDE/-+

Evaluating the Postfix Expression

Algorithm to evaluate the obtained Postfix Expression

Read the postfix expression one character at a time until it encounters the delimiter „#‟

Step 1: If the character is an operand, push its associated value onto the stack.

Step 2: If the character is an operator, POP two values from the stack, apply the operator to them and push the result onto the stack.

E.g consider the obtained **Postfix expression:- AB*CDE/-+**

Operand	Value
A	2
B	3
C	4
D	4
E	2

Char Read	Stack
A	2
B	3 2

Char Read	Stack
*	6
C	4 6
D	4 4 6
E	
/	2 4 6
-	2 6
+	8

OUTPUT = 8

Example 2: Infix expression:- (a+b)*c/d+e/f#

Read char	Stack	Output
(	(	
a	(	a

+	+ (	a
b	+ (	ab
)		ab+
*	*	ab+
c	*	ab+c
/	/	ab+c*
d	/	ab+c*d
+	+	ab+c*d/

e	 +	ab+c*d/e
/	 / +	ab+c*d/e
f	 / +	ab+c*d/ef
#	 	ab+c*d/ef/+

Postfix expression:- ab+c*d/ef/+

Evaluating the Postfix Expression

Operand	Value
a	1
b	2
c	4
d	2
e	6
f	3

Char Read	Stack
a	1
b	2 1
+	3
c	4 3
*	12
d	2 12
/	6
e	6 6
F	3 6 6
/	2 6
+	8

Output = 8

Infix to Postfix Conversion	Output
| ```c
#define SIZE 50        /* Size of Stack */
#include <ctype.h>
char s[SIZE];
int top=-1;      /* Global declarations */

void push(char elem)
{
   s[++top]=elem;
}

char pop()
{
   return(s[top--]);
}

int pr(char elem)
{            /* Function for precedence */
   switch(elem)
    {
    case '#': return 0;
    case '(': return 1;
    case '+':
    case '-': return 2;
    case '*':
    case '/': return 3;
    }
    return 0;
}

Void main()
{              /* Main Program */
   char infx[50],pofx[50],ch,elem;
   int i=0,k=0;
   printf("\nRead the Infix Expression ? ");
   scanf("%s",infx);
   push('#');
   while( (ch=infx[i++]) != '\0')
   {
     if( ch == '(') push(ch);
     else
        if(isalnum(ch)) pofx[k++]=ch;
        else
           if( ch == ')')
           {
              while( s[top] != '(')
``` | **Read the Infix Expression ?**<br><br><br><br>**(a+b)-(c-d)**<br><br><br><br>**Given Infix Expn: (a+b)\*(c-d)**<br><br>**Postfix Expn: ab+cd-\*** |

```
                pofx[k++]=pop();
            elem=pop();   /* Remove */
        }
        else
        {       /* Operator */
          while( pr(s[top]) >= pr(ch) )
            pofx[k++]=pop();
          push(ch);
        }
    }
    while( s[top] != '#')    /* Pop from stack
till empty
      pofx[k++]=pop();
    pofx[k]='\0';         /* Make pofx as valid
string */
    printf("\n\nGiven Infix Expn: %s Postfix
Expn: %s\n",infx,pofx);
}
```

Towers of Hanoi

Towers of Hanoi can be easily implemented using recursion. Objective of the problem is moving a collection of N disks of decreasing size from one pillar to another pillar. The movement of the disk is restricted by the following rules.

Rule 1 : Only one disk could be moved at a time.

Rule 2 : No larger disk could ever reside on a pillar on top of a smaller disk.

Rule 3 : A 3rd pillar could be used as an intermediate to store one or more disks, while they

were being moved from source to destination.

Initial Setup of Tower of Hanoi

Recursive Solution

N - represents the number of disks.

Step 1. If N = 1, move the disk from A to C.

Step 2. If N = 2, move the 1st disk from A to B. Then move the 2nd disk from A to C, The move the 1st disk from B to C.

Step 3. If N = 3, Repeat the step (2) to more the first 2 disks from A to B using C   as intermediate. Then the 3rd disk is moved from A to C. Then repeat the step (2) to move 2 disks from B to C using A as intermediate.

In general, to move N disks. Apply the recursive technique to move N - 1 disks from A to B using C as an intermediate. Then move the Nth disk from A to C. Then again apply the recursive technique to move N - 1 disks from B to C using A as an intermediate

Recursive Routine for Towers of Hanoi

```
void hanoi (int n, char s, char d, char i)
{
/* n    no. of disks, s    source, d    destination i    intermediate
*/ if (n = = 1)
{
print (s, d);
return;
}
else
{
hanoi (n - 1, s, i, d);
print (s, d)
hanoi (n-1, i, d, s);
return;
}
}
```

Source Pillar Intermediate Pillar Destination Pillar

Tower 1   Tower 2   Tower 3

1. Move Tower1 to Tower3

Tower 1   Tower 2   Tower 3

2. Move Tower1 to Tower2

Tower 1   Tower 2   Tower 3

3. Move Tower 3 to Tower 2

Tower 1   Tower 2   Tower 3

4. Move Tower 1 to Tower 3

Tower 1   Tower 2   Tower 3

5. Move Tower 2 to Tower 1

6. Move Tower 2 to Tower 3

7. Move Tower 1 to Tower 3

Since disks are moved from each tower in a LIFO manner, each tower may be considered as a Stack. Least Number of moves required solving the problem according to our algorithm is given by,

$$O(N)=O(N-1)+1+O(N-1) =2^N-1$$

Function Calls

When a call is made to a new function all the variables local to the calling routine need to be saved, otherwise the new function will overwrite the calling routine variables. Similarly the current location address in the routine must be saved so that the new function knows where to go after it is completed.

Recursive Function to Find Factorial

```
int fact(int n)
{
int S;
if(n==1)
    return(1);
else
        S = n * fact( n – 1 );
        return(S)
}
```

Balancing the Symbols

- Compilers check the programs for errors, a lack of one symbol will cause an error.
- A Program that checks whether everything is balanced.
- Every right parenthesis should have its left parenthesis.
- Check for balancing the parenthesis brackets braces and ignore any other character.

Algorithm for balancing the symbols

Read one character at a time until it encounters the delimiter `#'.

Step 1 : - If the character is an opening symbol, push it onto the stack.

Step 2 : - If the character is a closing symbol, and if the stack is empty report an error as missing opening symbol.

Step 3 : - If it is a closing symbol and if it has corresponding opening symbol in the stack, POP it from the stack. Otherwise, report an error as mismatched symbols.

Step 4 : - At the end of file, if the stack is not empty, report an error as Missing closing symbol. Otherwise, report as balanced symbols.

E.g. Let us consider the expression ((B*B)-{4*A*C}/[2*A]) #

((B*B)-{4*A*C}/[2*A]) #	
Read Character	**Stack**
(	(
(	((
)	▢ (
{	{ (
}	▢ (

[	[(
]	☐ (
)	☐

Empty stack, hence the symbols the balanced in the given expression.

Example for unbalanced symbols:

Consider the expression ((a + b) # : -

Read Character	Stack	Read Character	Stack
(	(	(	((
a	((	+	((
b	((	)	(
#	(	← Stack is not Empty	

Report an error message

Illustration For Unbalanced Symbols

QUEUES:

- Queue is a Linear Data Structure that follows First in First out (FIFO**)** principle.

- Insertion of element is done at one end of the Queue called "**Rear "**end of the Queue.

- Deletion of element is done at other end of the Queue called "**Front** "end of the Queue.

- Example: - Waiting line in the ticket counter.

Front End   RearEnd

←──────────── ┌─────────────────┐ ←──────────

Deletion   │   QUEUE Q   │

                 └─────────────────┘   Insertion

Queue Model

Front Pointer:-

It always points to the first element inserted in the Queue.

Rear Pointer:-

It always points to the last element inserted in the Queue.

For Empty Queue:-

┌──────────────────────┐
│ Front (F) = - 1   │
│   │
│ Rear (R) = - 1   │
└──────────────────────┘

Operations on Queue

Fundamental operations performed on the queue are

1. EnQueue
2. DeQueue

(i)   EnQueue operation:-

- It is the process of inserting a new element at the rear end of the Queue.
- For every EnQueue operation
  - Check for Full Queue
  - If the Queue is full, Insertion is not possible.
  - Otherwise, increment the rear end by 1 and then insert the element in the rear end of the Queue.

(ii)   DeQueue Operation:-

- It is the process of deleting the element from the front end of the queue.
- For every DeQueue operation
  - Check for Empty queue
  - If the Queue is Empty, Deletion is not possible.
  - Otherwise, delete the first element inserted into the queue and then increment the front by 1.

Exceptional Conditions of Queue

- Queue Overflow
- Queue Underflow

(i)   Queue Overflow:

- An Attempt to insert an element X at the Rear end of the Queue when the Queue is full is said to be Queue overflow.
- For every Enqueue operation, we need to check this condition.

(ii)   Queue Underflow:

- An Attempt to delete an element from the Front end of the Queue when the Queue is empty is said to be Queue underflow.
- For every DeQueue operation, we need to check this condition.

Implementation of Queue

Queue can be implemented in two ways.

1. Implementation using Array (**Static Queue**)
2. Implementation using Linked List (**Dynamic Queue**)

Array Declaration of Queue:

#define ArraySize 5

int Q [ArraySize];

  or

int Q [5];

Initial Configuration of Queue:

(i) Queue Empty Operation:

- Initially Queue is Empty.
- With Empty Queue, Front (F) and Rear (R) points to – 1.
- It is necessary to check for Empty Queue before deleting (DeQueue) an element from the Queue (Q).

Routine to check for Empty Queue

```
int IsEmpty ( Queue Q )
{
if( ( Front = = - 1) && ( Rear = = - 1 ) )
return ( 1 );
}
```

```
int IsEmpty ( Queue Q )
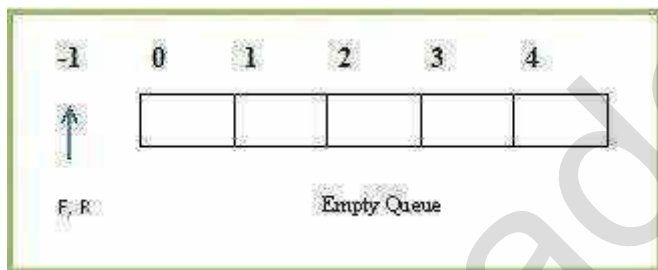
{

if( ( Front = = - 1) && ( Rear = = - 1 ) )

    return ( 1 );
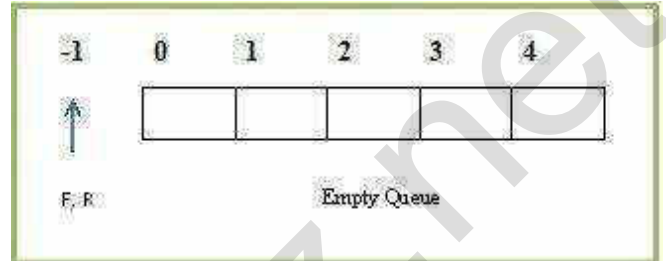
}
```


(ii) Queue Full Operation

- □ As we keep inserting the new elements at the Rear end of the Queue, the Queue becomes full.
- □ When the Queue is Full, Rear reaches its maximum Arraysize.
- □ For every Enqueue Operation, we need to check for full Queue condition.

Routine to check for Full Queue

```
int IsFull( Queue Q )

{

if ( Rear = = ArraySize - 1 )

    return ( 1 );
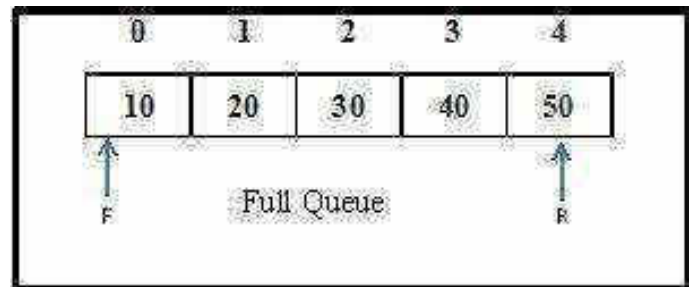
}
```

(iii) Enqueue Operation

☐ It is the process of inserting a new element at the Rear end of the Queue.

☐ It takes two parameters, Enqueue(X, Q). The elements X to be inserted at the Rear end of the Queue Q.

☐ Before inserting a new Element into the Queue, check for Full Queue.

☐ If the Queue is already Full, Insertion is not possible.

☐ Otherwise, Increment the Rear pointer by 1 and then insert the element X at the Rear end of the Queue.

☐ If the Queue is Empty, Increment both Front and Rear pointer by 1 and then insert the element X at the Rear end of the Queue.

Routine to Insert an Element in a Queue

```
void EnQueue (int X , Queue Q)
{
   if ( Rear = = Arraysize - 1)
      print (" Full Queue !!!!. Insertion not
                  possible");
    else if (Rear = = - 1)
      {
        Front = Front + 1;
        Rear = Rear + 1;
        Q [Rear] = X;
      }
        else
          {
            Rear = Rear + 1;
            Q [Rear] = X;
          }
}
```


Full Queue

Empty Queue

Queue with One element

(iv) DeQueue Operation

- ☐ It is the process of deleting a element from the Front end of the Queue.

- ☐ It takes one parameter, DeQueue (Q). Always front element in the Queue will be deleted.

- ☐ Before deleting an Element from the Queue, check for Empty Queue.

- ☐ If the Queue is empty, deletion is not possible.

- ☐ If the Queue has only one element, then delete the element and represent the empty queue by updating Front = - 1 and Rear = - 1.

- ☐ If the Queue has many Elements, then delete the element in the Front and move the Front pointer to next element in the queue by incrementing Front pointer by 1.

ROUTINE FOR DEQUEUE

```
void DeQueue ( Queue Q )
{
  if ( Front = = - 1)
      print (" Empty Queue !. Deletion not possible " );
        else if( Front = = Rear )
          {
            X = Q [ Front ];
            Front = - 1;
            Rear = - 1;
          }
              else
            {
            X = Q [ Front ];
            Front = Front + 1 ;
            }
}
```


Empty Queue

Queue with One element

After DeQueue

Array implementation of Queue

```c
#include<stdio.h>
#include<conio.h>
#define SIZE 5
int front = - 1;
int rear = - 1;
int q[SIZE];
void insert( );
void del( );
void display( );
void main( )
{
        int choice;
        clrscr( );
        do
        {
                printf("\t Menu");
                printf("\n 1. Insert");
                printf("\n 2. Delete");
                printf("\n 3. Display ");
                printf("\n 4. Exit");

                printf("\n Enter Your Choice:");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1:
                                insert( );
                                display( );
                                break;
                        case 2:
                                del( );
                                display( );
                                break;
                        case 3:
                                display( );
                                break;
                        case 4:
                                printf("End of Program....!!!!");
                                exit(0);
                }
        }while(choice != 4);}
void insert( )
  {
        int no;
```
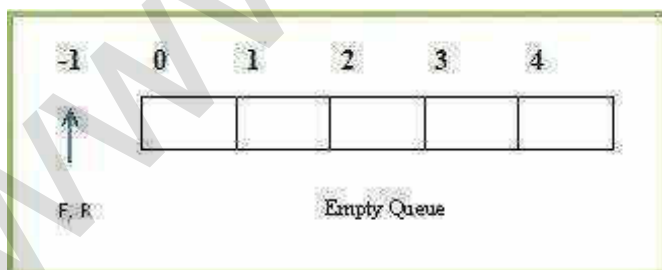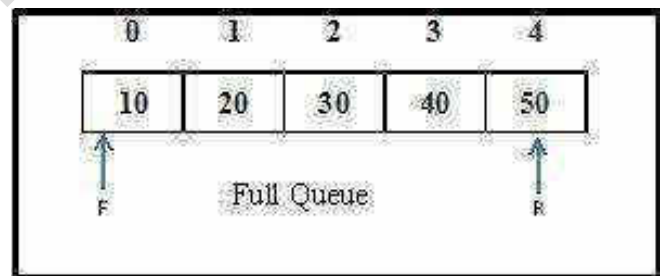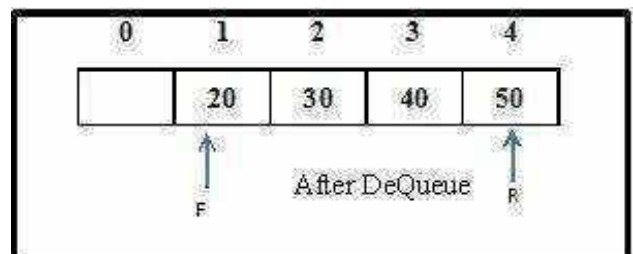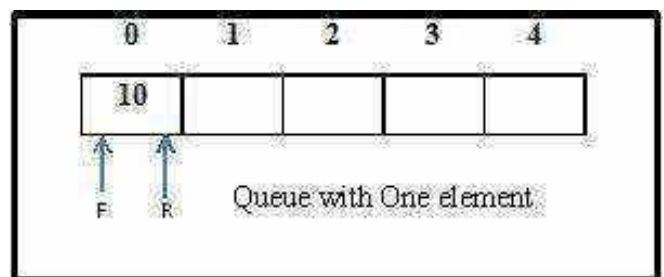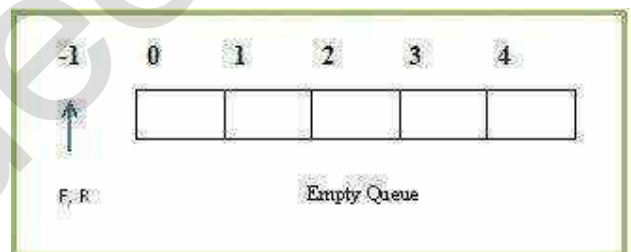
```
        printf("\n Enter No.:");
        scanf("%d", &no);

        if(rear < SIZE - 1)
        {
                q[++rear]=no;
                if(front = = - 1)
                front=0;// front=front+1;
        }
        else
        {
                printf("\n Queue overflow");
        }}

void del( )
{
        if(front = = - 1)
        {
                printf("\n Queue Underflow");
                return;
        }
        else
        {
            printf("\n Deleted Item:-->%d\n", q[front]);
        }
        if(front = = rear)
      {
                Front = - 1;
                Rear = - 1;
        }
        else
        {
                Front = front + 1;
        }}

void display( )
{
        int i;
        if( front = = - 1)
        {
                printf("\nQueue is empty....");
                return;
        }
        for(i = front; i<=rear; i++)
                printf("\t%d",q[i]);}
```
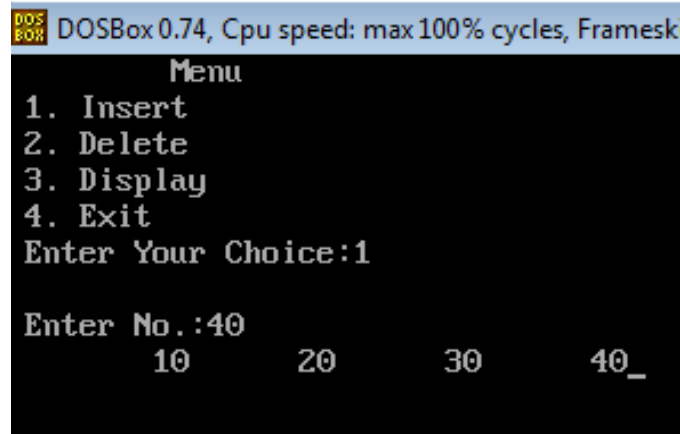
output

Linked List Implementation of Queue

- Queue is implemented using SLL (Singly Linked List) node.
- Enqueue operation is performed at the end of the Linked list and DeQueue operation is performed at the front of the Linked list.
- With Linked List implementation, for Empty queue

Front = NULL   &   Rear = NULL

Linked List representation of Queue with 4 elements

Q

Front   **Rear**
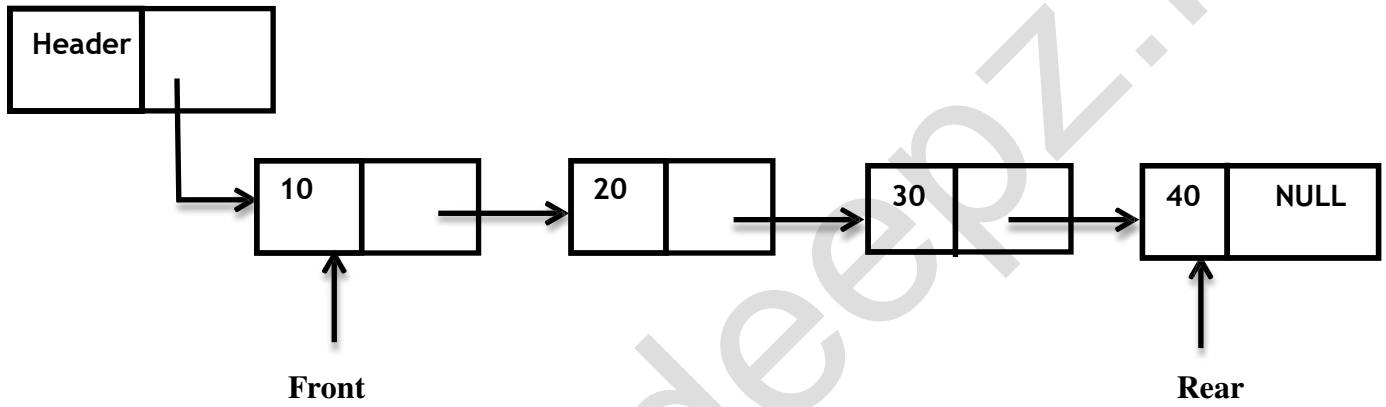
Declaration for Linked List Implementation of Queue ADT

```
struct node;
typedef struct node * Queue;
typedef struct node * position;
int IsEmpty (Queue Q);
Queue CreateQueue (void);
void MakeEmpty (Queue Q);
void Enqueue (int X, Queue Q);
void Dequeue (Queue  Q);
struct node
{
int data ;
position next;
}* Front = NULL, *Rear = NULL;
```

(i) Queue Empty Operation:

- Initially Queue is Empty.
- With Linked List implementation, Empty Queue is represented as S -> next = NULL.
- It is necessary to check for Empty Queue before deleting the front element in the Queue.

ROUTINE TO CHECK WHETHER THE QUEUE IS EMPTY

```
int IsEmpty (Queue Q
{
  if ( Q □Next = = NULL)
     return (1);
}
```

Q

Header	NULL
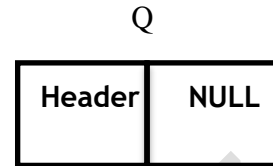
Empty Queue

(ii) EnQueue Operation

- It is the process of inserting a new element at the Rear end of the Queue.
- It takes two parameters, EnQueue (int X , Queue Q). The elements X to be inserted into the Queue Q.
- Using malloc ( ) function allocate memory for the newnode to be inserted into the Queue.
- If the Queue is Empty, the newnode to be inserted will become first and last node in the list. Hence Front and Rear points to the newnode.
- Otherwise insert the newnode in the Rear -> next and update the Rear pointer.

Routine to EnQueue an Element in Queue

```
void EnQueue ( int X, Queue Q )
{
   struct node *newnode;
   newnode = malloc (sizeof (struct  node));
   if (Rear = = NULL)
   {
    newnode □data = X;
    newnode□ next = NULL;
    Q -> next = newnode;
    Front = newnode;
    Rear = newnode;
   }
  else
   {
    newnode □data = X;
    newnode □next = NULL;
    Rear □next = newnode;
```

```
        Rear = newnode;
    }
}
```

Q

Header	NULL

Empty Queue

Before Insertion

Q

Header	

Front   Rear

After Insertion
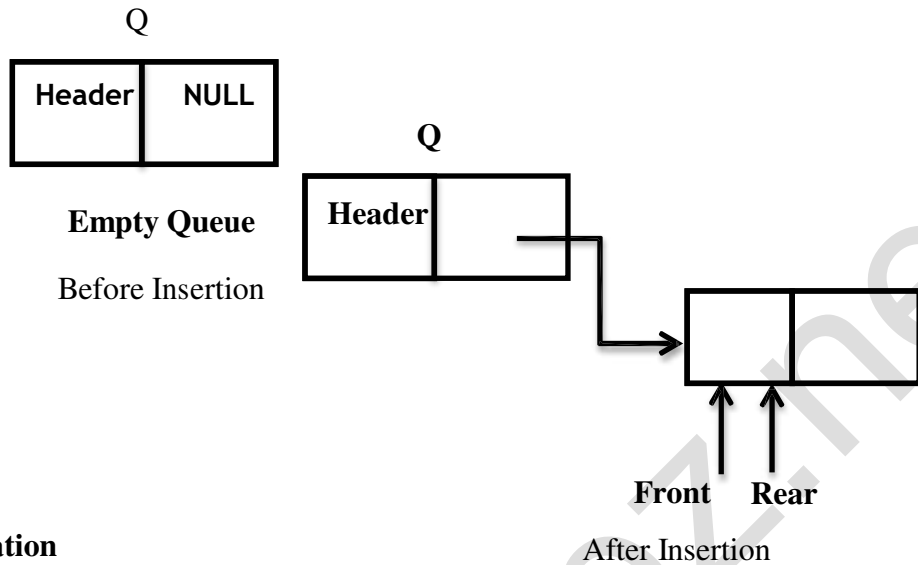
(iii) DeQueue Operation

It is the process of deleting the front element from the Queue.
It takes one parameter, Dequeue (Queue Q). Always element in the front (i.e) element pointed
by Q -> next is deleted always.
Element to be deleted is made "temp".
If the Queue is Empty, then deletion is not possible.
If the Queue has only one element, then the element is deleted and Front and Rear pointer is
made NULL to represent Empty Queue.
Otherwise, Front element is deleted and the Front pointer is made to point to next node in the list.
The free ( ) function informs the compiler that the address that temp is pointing to, is unchanged
but the data present in that address is now undefined.

Q

Header	

| 20 | | → | 30 | | → | 40 | NULL |
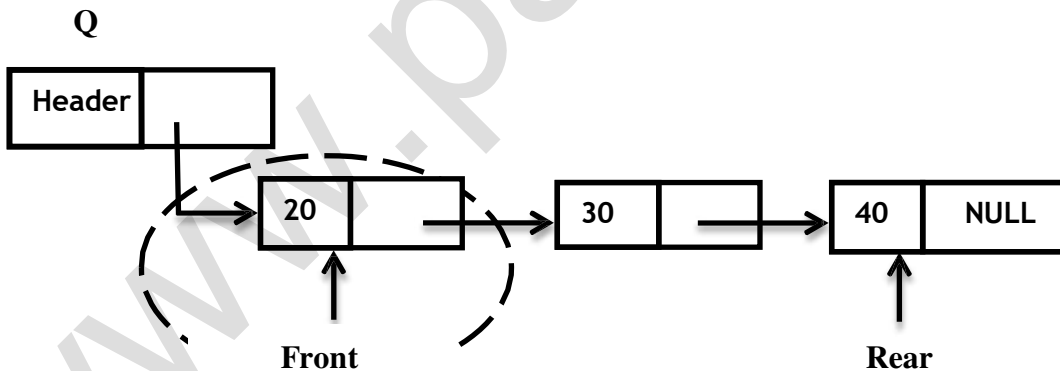
Front   **Rear**

Routine to DeQueue an Element from the Queue

```
void DeQueue ( Queue Q )
{
   struct node *temp;
    if ( Front = = NULL )
        Error ("Empty Queue!!! Deletion not possible." );
     else if (Front = = Rear)
        {
           temp = Front;
           Q -> next = NULL;
           Front = NULL;
           Rear = NULL;
            free ( temp );
        }
         else
            {
                temp = Front;
                Q -> next = temp -> next;
                Front = Front →Next;
                free (temp);
             }

}
```

Linked list implementation of Queue

```
#include<stdio.h>
#include<conio.h>
void enqueue();
void dequeue();
void display();
typedef struct node *position;
position front=NULL,rear=NULL,newnode,temp,p;

   struct node
    {
    int data;
    position next;
    };
   void main()
   {
   int choice;
   clrscr();
   do
   {
   printf("1.Enqueue\n2.Dequeue\n3.display\n4.exit\n");
   printf("Enter your choice\n\n");
```

```
        scanf("%d",&choice);
        switch(choice)
        {
        case 1:
                enqueue();
                break;
        case 2:
                dequeue();
                break;
        case 3:
                display();
                break;
        case 4:
                exit(0);
        }
        }
        while(choice<5);
        }

    void enqueue()
    {
            newnode=(struct node*)malloc(sizeof(struct node));
            printf("\n Enter the data to be enqueued\n");
            scanf("%d",&newnode->data);
            newnode->next=NULL;
            if(rear==NULL)
               front=rear=newnode;
            else {
               rear->next=newnode;
               rear=newnode;
             }
      display();
    }
     void dequeue()
     {
            if(front==NULL)
              printf("\nEmpty queue!!!!! Deletion not possible\n");
            else if(front==rear)
              {
              printf("\nFront element %d is deleted from queue!!!! now queue is
                        empty!!!! no more deletion possible!!!!\n",front->data);
              front=rear=NULL;
              }
            else
              {
               temp=front;
               front=front->next;
               printf("\nFront element %d is deleted from queue!!!!\n",temp->data);
               free(temp);
              }
        display();
```

```
   }
  void display()
  {
   p=front;
   while(p!=NULL)
   {
   printf("%d -> ",p->data);
   p=p->next;
   }
   printf("Null\n");
  }
```

Output

 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.
4. Batch processing in operating system.
5. Job scheduling Algorithms like Round Robin Algorithm uses Queue.

Drawbacks of Queue (Linear Queue)

- With the array implementation of Queue, the element can be deleted logically only by moving Front = Front + 1.
- Here the Queue space is not utilized fully.

To overcome the drawback of this linear Queue, we use Circular Queue.

CIRCULAR QUEUE

In Circular Queue, the insertion of a new element is performed at the very first location of the queue if the last location of the queue is full, in which the first element comes just after the last element.

- A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue.
- Circular queues have a fixed size.
- Circular queue follows FIFO principle.
- Queue items are added at the rear end and the items are deleted at front end of the circular queue
- Here the Queue space is utilized fully by inserting the element at the Front end if the rear end is full.
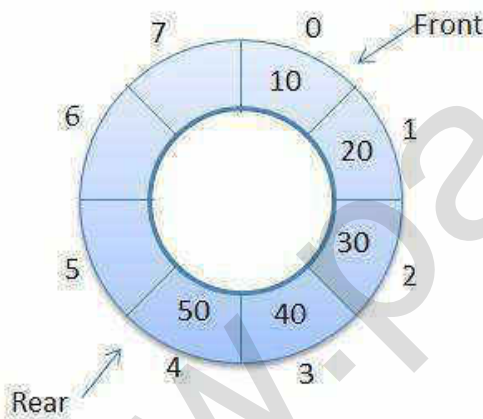
Operations on Circular Queue

Fundamental operations performed on the Circular Queue are

- Circular Queue Enqueue
- Circular Queue Dequeue

Formula to be used in Circular Queue

For Enqueue   Rear = (Rear + 1) % ArraySize

For Dequeue   Front = (Front + 1) % ArraySize

(i) Circular Queue Enqueue Operation

- ☐ It is same as Linear Queue EnQueue Operation (i.e) Inserting the element at the Rear end.
- ☐ First check for full Queue.
- ☐ If the circular queue is full, then insertion is not possible.
- ☐ Otherwise check for the rear end.
- ☐ If the Rear end is full, the elements start getting inserted from the Front end.

Routine to Enqueue an element in circular queue

```
void Enqueue ( int X, CircularQueue CQ )
{
    if( Front = = ( Rear + 1 ) % ArraySize)
            Error( "Queue is full!!Insertion not possible" );
    else if( Rear = = -1 )
      {
            Front = Front + 1;
            Rear = Rear + 1;
            CQ[ Rear ] = X;
      }
     else
      {
         Rear = ( Rear + 1 ) % Arraysize;
         CQ[ Rear ] = X;
      }
}
```


FULL QUEUE   FULL QUEUE
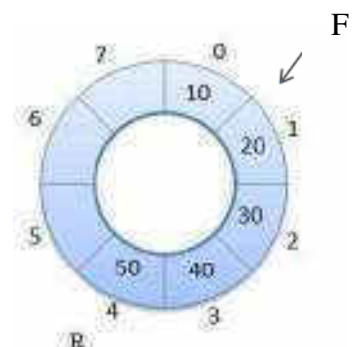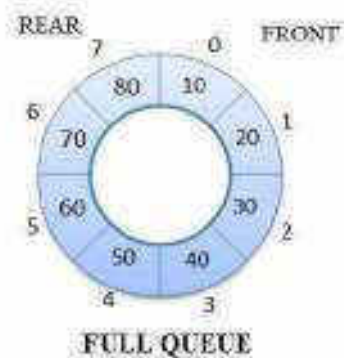
Circular Queue DeQueue Operation

□ It is same as Linear Queue DeQueue operation (i.e) deleting the front element.

□ First check for Empty Queue.

□ If the Circular Queue is empty, then deletion is not possible.

□ If the Circular Queue has only one element, then the element is deleted and Front and Rear pointer is initialized to - 1 to represent Empty Queue.

□ Otherwise, Front element is deleted and the Front pointer is made to point to next element in the Circular Queue.

F, R
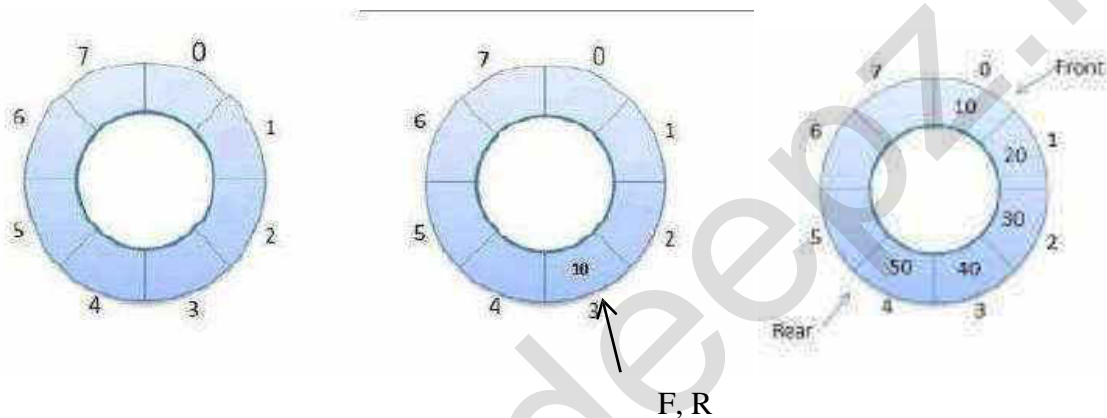
F= -1,R= --1

Routine To DeQueue An Element In Circular Queue

```
void DeQueue (CircularQueue CQ)
{
   if(Front== - 1)
       Empty("Empty Queue!");
   else if(Front==rear)
      {
        X=CQ[Front];
        Front=-1;
        Rear=-1;
      }
     else
      {
         X=CQ[Front];
         Front=(Front+1)%Arraysize;
      }}
```

Implementation of Circular Queue

```c
#include<stdio.h>
#include<conio.h>
#define max 3
void insert();   void delet();    void display();
int q[10],front=0,rear=-1;
void main()
{ int ch;
 clrscr();
 printf("\nCircular Queue operations\n"); printf("1.insert\n2.delete\n3.display\n4.exit\n");
 while(1)
 {
 printf("Enter your choice:");
 scanf("%d",&ch);
 switch(ch)
 {
 case 1:
        insert();          break;
 case 2:
        delet();           break;
 case 3:
        display();         break;
 case 4:
        exit();
 default:
        printf("Invalid option\n");
 }}}

void insert()
{
 int x;
 if((front==0&&rear==max-1)||(front>0&&rear==front-1))
 printf("Queue is overflow\n");
 else
  {
  printf("Enter element to be insert:");
  scanf("%d",&x);
  if(rear==max-1&&front>0)
   { rear=0;
    q[rear]=x;
   }
   else
   {
   if((front==0&&rear==-1)||(rear!=front-1))
   q[++rear]=x;
```

```
  } }}
void delet()
{
 int a;
 if((front==0)&&(rear==-1))
   printf("Queue is underflow\n");
 if(front==rear)
  {
   a=q[front];
   rear=-1;
   front=0;
  }
 else if(front==max-1)
  {
   a=q[front];
   front=0;
  }
 else
   a=q[front++];
   printf("Deleted element is:%d\n",a);
}
void display()
{
 int i,j; if(front==0&&rear==-1)
 printf("Queue is underflow\n");
 if(front>rear) {
 for(i=0;i<=rear;i++)
 printf("\t%d",q[i]);
 for(j=front;j<=max-1;j++)
 printf("\t%d",q[j]);
   printf("\nrear is at %d\n",q[rear]);
 printf("\nfront is at %d\n",q[front]); }
 else
  {
   for(i=front;i<=rear;i++)
     printf("\t%d",q[i]);  printf("\nrear
   is at %d\n",q[rear]); printf("\nfront
   is at %d\n",q[front]);
  }
   printf("\n");
}
```

OUTPUT


```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program:    TC
1.insert ( max 3 element)
2.delete
3.display
4.exit
Enter your choice:1
Enter element to be insert:10
Enter your choice:1
Enter element to be insert:20
Enter your choice:1
Enter element to be insert:30
Enter your choice:1
Queue is overflow
Enter your choice:2
Deleted element is:10
Enter your choice:1
Enter element to be insert:40
Enter your choice:1
Queue is overflow
Enter your choice:3
        40      20      30
rear is at 40

front is at 20

Enter your choice:
```

DOUBLE-ENDED QUEUE (DEQUE)

In DEQUE, insertion and deletion operations are performed at both ends of the Queue.
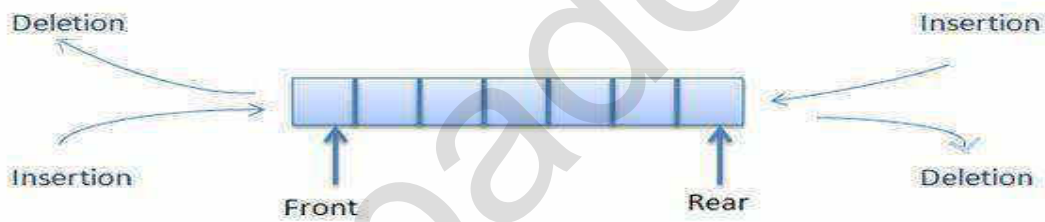
Exceptional Condition of DEQUE

(i) Input Restricted DEQUE

Here insertion is allowed at **one end** and deletion is allowed at **both ends.**
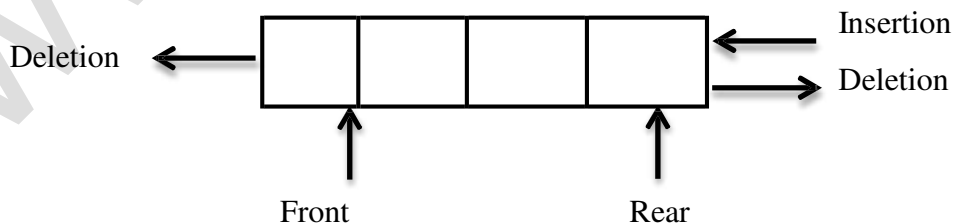
(ii) Output Restricted DEQUE

Here insertion is allowed at **both ends** and deletion is allowed at **one end.**

Insertion →

Insertion

Deletion ←
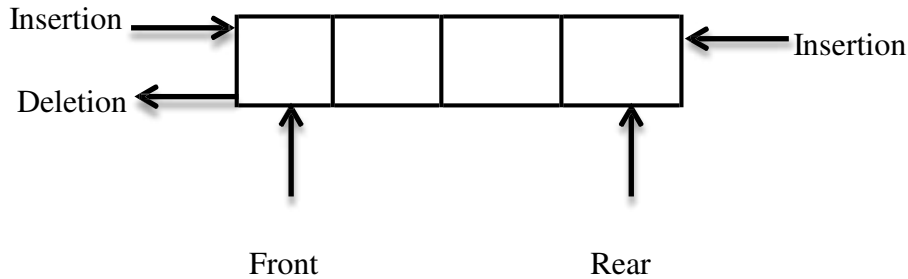
Front   Rear

Operations on DEQUE

Four cases for inserting and deleting the elements in DEQUE are

1. Insertion At Rear End [same as Linear Queue]
2. Insertion At Front End
3. Deletion At Front End [same as Linear Queue]
4. Deletion At Rear End

Insertion at the rear end

Step 1 : Check for the overflow condition.

Step 2 : If it is true, display that the queue is full

Step 3 : Otherwise, If the rear and front pointers are at the initial values (-1), Increment both the pointers. Goto step 5.

Step 4 : Increment the rear pointer

Step 5 : Assign the value to Q[rear]

Case 1: Routine to insert an element at Rear end

```
void Insert_Rear (int X, DEQUE DQ)
{
    if( Rear = = Arraysize - 1)
        Error("Full Queue!!!! Insertion not possible");
    else if( Rear = = -1)
        {
Front = Front + 1;
Rear = Rear + 1;
DQ[ Rear ] = X;
        }
```

```
    else
        {
Rear = Rear + 1;
DQ[ Rear ] = X;
        }
}
```

Insertion at the rear end
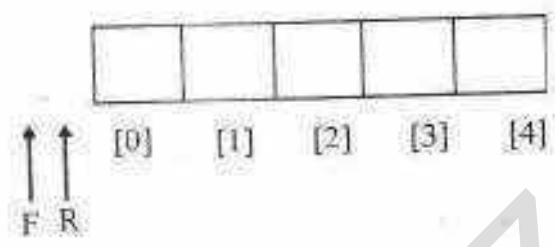
Insert_rear (10)
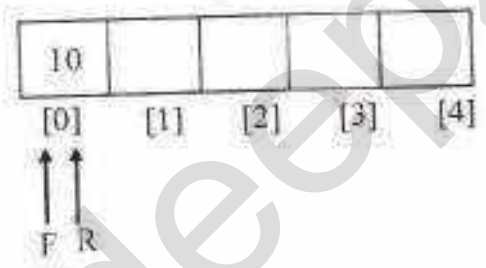
Insert_rear (20)
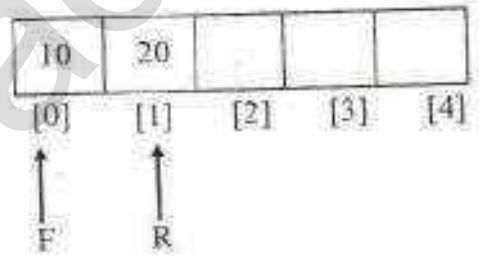
Insert_rear (30)
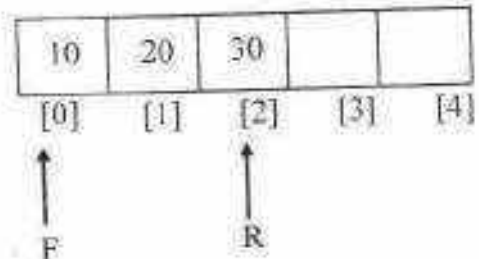
Insertion at front end

Step 1 : Check the front pointer, if it is in the first position (0) then display an error message that the value cannot be inserted at the front end.

Step 2 : Otherwise, decrement the front pointer

Step 3 : Assign the value to Q[front]

Case 2: Routine to insert an element at Front end

```
void Insert_Front ( int X, DEQUE DQ )
{
  if( Front = = 0 )
   Error("Element present in Front!!!!! Insertion not possible");
  else if(Front = = -1)
   {
     Front = Front + 1;
     Rear = Rear + 1;
     DQ[Front] = X;
   }
  else
   {
     Front = Front - 1;
     DQ[Front] = X;
   }
}
```

Insertion at the front position:

Case 1

Insert_front (30)

Insert_front (5)

10				
[0]	[1]	[2]	[3]	[4]

F R
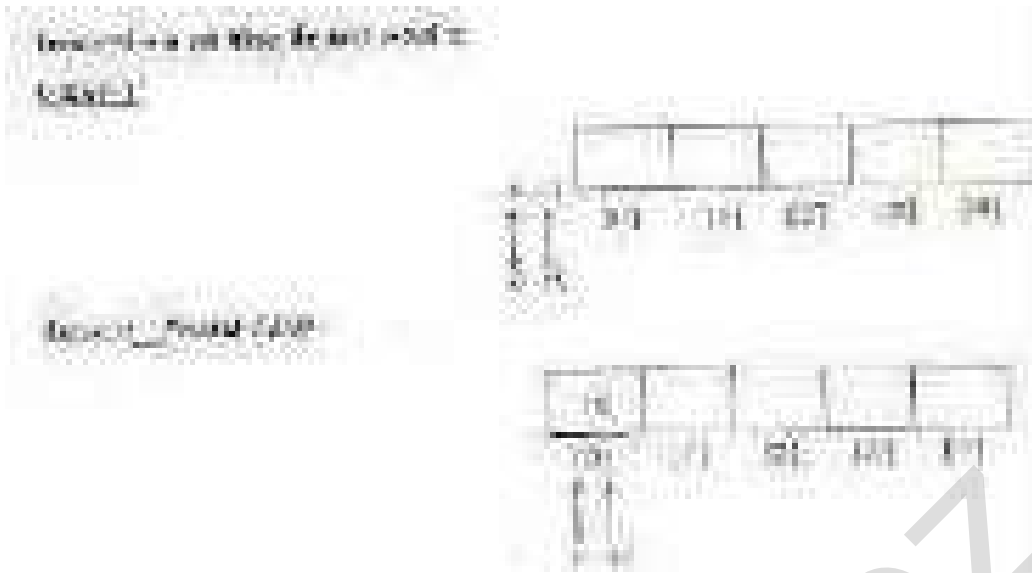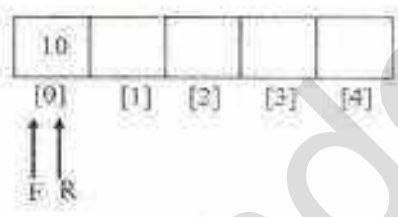
Here decrement of the front pointer may point to the initial (-1) position. So the value cannot be inserted even there is a space in the queue.

Case 2

			40	50
[0]	[1]	[2]	[3]	[4]

F R

Insert_front (30)

		30	40	50
[0]	[1]	[2]	[3]	[4]

F R

Insert_front (30)

	20	30	40	50
[0]	[1]	[2]	[3]	[4]

F R
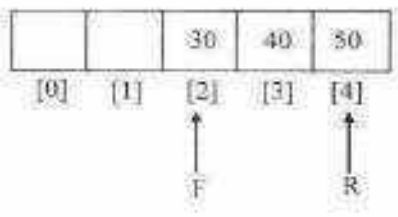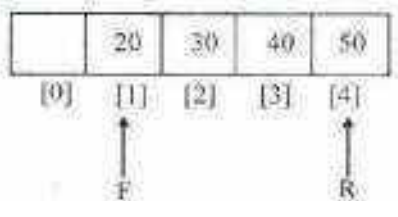
Case 3: Routine to delete an element from Front end

```
void Delete_Front(DEQUE  DQ)
{
   if(Front = = - 1)
      Error("Empty queue!!!! Deletion not possible");
   else if( Front = = Rear )
    {
       X = DQ[ Front];
       Front = - 1;
       Rear = - 1;
    }
  else
   {
     X = DQ [ Front ];
     Front = Front + 1;
   }
}
```

Deletion from Front End :

Step 1   :   Check for the underflow condition. If it is true display that the queue is empty.

Step 2   :   Otherwise, delete the element at the front position, by asigning X as Q[front]

Step 3   :   If the rear and front pointer points to the same position (ie) only one value is present, then reinitialize both the pointers.

Step 4   :   Otherwise, Increment the front pointer

Deletion from front end

Dequeue_ front ()

```
        ┌──────┬──────┬──────┬──────┬──────┐
        │      │      │      │      │      │
        └──────┴──────┴──────┴──────┴──────┘
  -1 -1   [0]    [1]    [2]    [3]    [4]
   ↑  ↑
   F  R
```

Q - is empty

```
  ┌──────┬──────┬──────┬──────┬──────┐
  │  10  │  20  │  30  │  40  │  50  │
  └──────┴──────┴──────┴──────┴──────┘
    [0]    [1]    [2]    [3]    [4]
     ↑                          ↑
     F                          R
```

Dequeue_ front ()

```
  ┌──────┬──────┬──────┬──────┬──────┐
  │      │  20  │  30  │  40  │  50  │
  └──────┴──────┴──────┴──────┴──────┘
    [0]    [1]    [2]    [3]    [4]
            ↑                   ↑
            F                   R
```
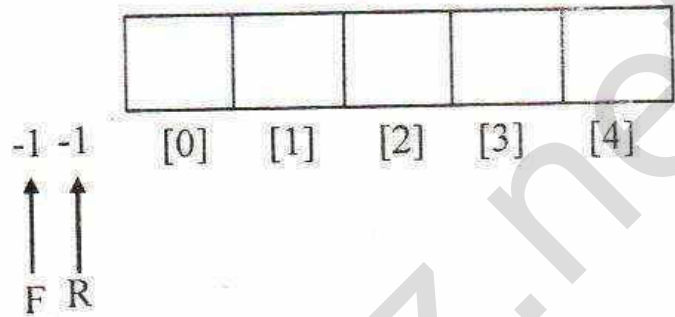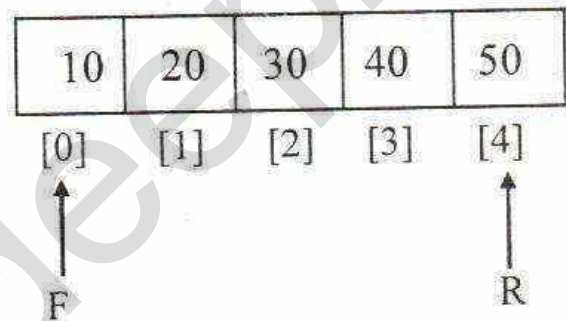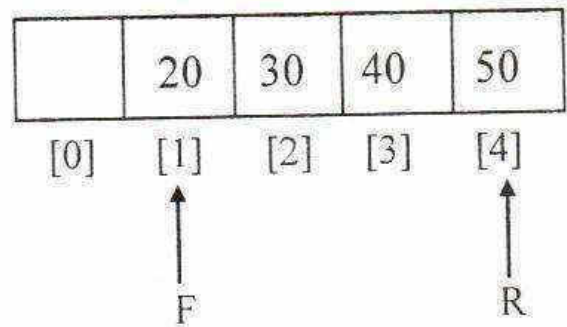
Case 4: Routine to delete an element from Rear end

```
void Delete_Rear(DEQUE DQ)
{
   if( Rear = = - 1)
       Error("Empty queue!!!! Deletion not possible");
  else if( Front = = Rear )
   {
    X = DQ[ Rear ];
    Front = - 1;
    Rear = - 1;
   }
   else
    {
    X = DQ[ Rear ];
    Rear = Rear - 1;
    }

}
```

Dequeue_front ( )

		30	40	50
[0]	[1]	[2]	[3]	[4]

F   R
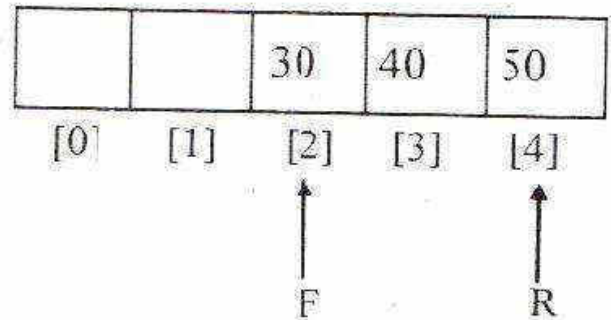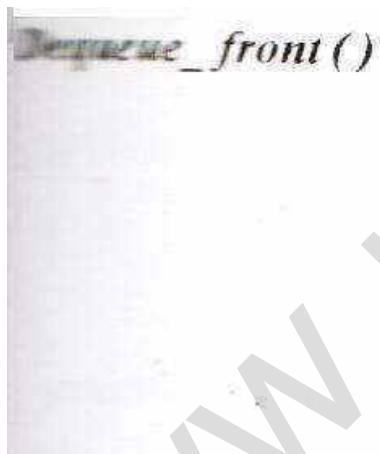
Deletion from Rear End

Step 1 : Check the rear pointer. If it is in the initial value then display that the value cannot be deleted.

Step 2 : Otherwise, delete element at the rear position.

Step 3 : If the rear and front pointers are at the same position, reinitialize both the pointers.

Step 4 : Otherwise, decrement the rear pointer.

Dequeue_rear ()
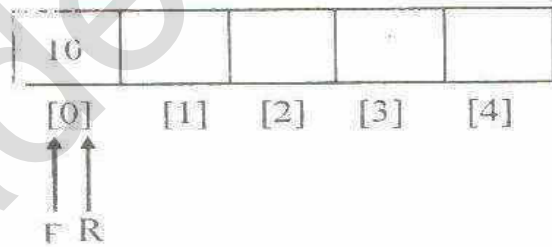
Dequeue_rear ()
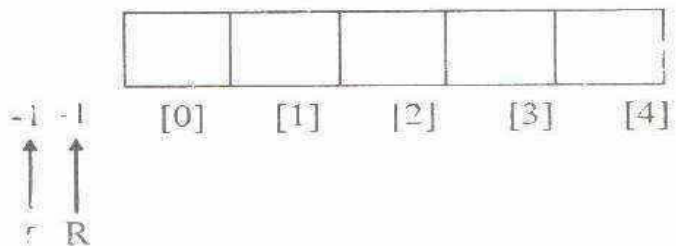
Gate questions and

answers

Stacks and Queues

1. Compute the postfix equivalent of the following infix arithmetic expression where a+b*c+d*e↑f; where ↑ represents exponentiation. Assume normal operator precedence.

Infix expression	postfix operation	stack
1 a+b*c+d*e↑f		
2 +b*c+d*e↑f	a	
3 b*c+d*e↑f	a	+
4 *c+d*e↑f	ab	+
5 c+d*e↑f	ab	+*
6 +d*e↑f	abc	+*
7 d*e↑f	abc*+	+
8 *e↑f	abc*+d	+
9 e↑f	abc*+d	+*
10 ↑f	abc*+de	+*
11 f	abc*+de	+* ↑
12	abc*+def	+* ↑

So postfix expression will be (after emptying the stack)

abc*+def↑*+

2. Suppose one character at a time comes as an input from a string of letters. There is an option either to (i) print the incoming letter or to (ii) put the incoming letter on to a stack. Also a letter from top of the stack can be popped out at any time and printed. The total number of total distinct words that can be formed out of a string of three letters in this fashion, is

    (A)
    (B)
    (C)

Total no of distict word can be 5.

because total no words starting with a would be 2. (abc ,acb)

Total no of words starting with 2 (bca,bac)

Total no of words starting with c will be 1 (cba) because to print c as first letter we have to push a and b in stack

So total no of words formed = 2+2+1

3.   The following sequence of operations is performed on a stack :

PUSH (10), PUSH (20), POP, PUSH (10), PUSH (20), POP, POP, POP, PUSH (20), POP.

The sequence of values popped out is:

   A.   20, 10, 20, 10, 20
   B.   20, 20, 10, 10, 20
   C.   10, 20,20,10,20
   D.   20,20,10,20,10
   E.   None of the above

[B]

String	Status of stack	Status of array(output)
Push(10)	10	
Push(20)	10 20	
Pop	10	20
Push(10)	10 10	20
Push(20)	10 10 20	20
Pop	10 10	20 20
Pop	10	20 20 10
Pop		20 20 10 10
Push(20)	20	20 20 10 10
Pop		20 20 10 10 20

4. A stack is use to pass parameters to procedures in a procedure call.

    (a) If a procedure P has two parameters as described in procedure definition:
             procedure P(var x: integer; y: integer);

         and if P is called by:

                          P(a,b)

         State precisely in a sentence what is pushed onto stack for parameters a and b.

         [QUESTION IS HIGHLY AMBIGUOUS]

    In the generated code for the body of procedure P, how will the addressing of formal parameters y and y differ?

5. Which of the following permutation can be obtained in the output (in the same order) using a stack assuming that the input is the sequence1, 2, 3, 4, 5 in that order?

         (A) 3, 4, 5, 1, 2
         (B) 3, 4, 5, 2, 1
         (C) 1, 5, 2, 3, 4
         (D) 5, 4, 3, 1, 2

    [B] As it can be verified as- push(1) push(2) push(3) pop push(4) pop push(5) pop pop pop

6.

     The postfix expression for the infix expression

         A + B*(C+D)/F + D*E is

         (a)   AB+CD+*F/D+E*
         (b)   ABCD+*F/DE*++
         (c)   A*B+CD/F*DE++
         (d)   A+*BCD/F*DE++

    Possible Answer:

Infix expression	Post fix operation	Operater stack
A+B*(C+D)/F+D*E		
+B*(C+D)/F+D*E	A	
B*(C+D)/F+D*E	A	+
*(C+D)/F+D*E	AB	+
(C+D)/F+D*E	AB	+
C+D)/F+D*E	AB	+*(
+D)/F+D*E	ABC	

D)/F+D*E	ABCD	
)/F+D*E	ABCD	+*(+
/F+D*E	ABCD+	+*
F+D*E	ABCD+*	+/
+D*E	ABCD+*F	+/
D*E	ABCD+*F/+	+
*E	ABCD+*F/+	+*
E	ABCD+*F/+DE	+*
	ABCD+*F/+DE*+	

7.   Consider the following statements.

   i   First-in-first-out types of computations are efficiently supported by STACKS.
   ii   Implementing LISTS on linked lists is more efficient than implementing LISTS on an array for almost all the basic LIST operations.
   iii   Implementing QUEUES on a circular is more efficient than implementing QUEUES
   iv   Last-in-first-out QUEUES type of computations are efficiently supported by QUEUES.
   Which of the following is correct?

   (A) (ii) and (iii) are true   (B) (i) and (ii) are true

   (C) (iii) and (iv) are true   (D) (ii) and (iv) are true

[A]

List perform almost all basic operation in O(1) accept merging

But array will take O(n) for all operations.

Here basic operation means insert delete at given position and

Queue using linked list are efficient in using memory. They do not waste space like in array implementation .

8.   Compute the postfix equivalent of the following infix expression.

   $3 * \log (x + 1) - a/2$

3X1+log*a2/- (assuming log as a operator not as a function and precedence of log is greatest)

9. A queue Q containing n items and an empty stack S are given. It is required to transfer all the items from the queue to the stack, so that the item at the front of the queue is on the top of the stack, and the order of all the other items is preserved. Show this how this can be done in O(n) time using only a constant amount of additional storage. Note that the only operations which can be performed on the queue and stack are Delete, Insert, Push and Pop. Do not assume any implementation of the queue or stack.

It can be done as –

Step1. Delete all the nodes of queue and push all the elements in stack sequentially.

Step2. Now queue is empty then pop every element form stack and inset it into queue

Step3. Now again repeat the step 1.

10. Which of the following is essential for converting an infix expression to the postfix form efficiently?

(A) An operator stack   (B) An operand stack

(C) An operand stack and an operator stack   (D) A parse tree

[A] Operater stack will be used to store operaters as the appear based on three rules

1. if operater 'x' is on top and operater 'y'comes which has a higher precedence than x the it will also be pushed to stack..
2. if x and y have same precedence then x will be poped out before y is pushed
3. same as rule 2 if x has a higher precedence than y.

11. A priority queue Q is used to implement a stack S that stores characters. PUSH(C) is implemented as INSERT(Q, C, K) where K is an appropriate integer key chosen by the implementation. POP is implemented as DELETEMIN(Q). For a sequence of operations, the keys chosen are in

(A) non-increasing order   (B) non-decreasing order

(C) strictly increasing order   (D) strictly decreasing order

[D] Implementing stack using priority queue require first element inserted in stack will be deleted at last, and to implement it using deletemin() operation of queue will require first element inserted must have highest priority so the key must be in decreasing order.

12. Suppose a stack implementation supports, in addition to PUSH and POP, an operation REVERSE, which reverses the order of the elements on the stack.

(a) To implement a queue using the above stack implementation, show how to implement ENQUEUE using a single operation and DEQUEUE using a sequence of 3 operations   (2)

(b) The following postfix expression, containing single digit operands and arithmetic operators + and *, is evaluated using a stack.
      5 2 * 3 4 + 5 2 * * +

a)

i)Implementation of enqueue is as-

Step1: push the element in stack

ii) implementation of dequeue is as-

Step1: reverse

Step2: pop

Step3:reverse

Show the contents of the stack

i After evaluating 5 2 * 3 4 +
ii After evaluating 5 2 * 3 4 + 5 2
iii At the end of evaluation

b)

Input in postfix	Stack<- top	evaluation
5	5	
2	52	
*	10	5*2=10
3	10 3	
4	10 3 4	
+	10 7	3+4=7
5	10 7 5	
2	10 7 5 2	
*	10 7 10	5*2=10
*	10 70	7*10=70

+	80	
Null		

i)contents of stack are 10 7

ii)contents of stack are10 7 5 2

iii)contents of stack null and the final answer is 80

13.   What is the minimum number of stacks of size n required to implement a queue of size n?
    A.   One
    B.   Two
    C.   Three
   D. Four

[B] First stack will be used to store the queue with rear at top..

We need to push element in to stack 1 in order to insert it in queue.

For deletion we will empty stack 1 in stack 2.that will put front on top so we can pop from stack 2 in order to effect deletion.

14.   Let S be a stack of size $n \geq 1$. Starting with the empty stack, suppose we Push the first n natural numbers in sequence, and then perform n Pop operations. Assume that Push and POP operations take X seconds each, and Y seconds elapse between the end of one such stack operation and the start of the next operation. For $m \geq 1$, define the stack-life of m as the time elapsed from the end of Push(m) to the start of the Pop operation that removes m from S. The average stack-life of an element of this stack is

    (A) n(X + Y)
    (B) 3Y + 2X
    (C) N(X + Y) – X
    (D) Y + 2X

[A]

Push1() and pop1() are push and pop operations in 1st stack

Push2() and pop2() are push and pop operations in 2nd stack

Using 1st stack we can insert value same as queue (push1())

And for deletion we have to pop all the elements of 1st stack into push it into 2nd stack and then pop one element form 2nd stack.(pop1()push2()…..until 1st stack is emptied then pop2()…for desired number of deletion)

15.   The following sequence of operations is performed on a stack :

PUSH (10), PUSH (20), POP, PUSH (10), PUSH (20), POP, POP, POP, PUSH (20), POP.

The sequence of values popped out is:

F.   20, 10, 20, 10, 20
G.   20, 20, 10, 10, 20
H.   10, 20,20,10,20
I.   20,20,10,20,10
J.   None of the above

[G]

String	Status of stack	Status of array(output)
Push(10)	10	
Push(20)	10 20	
Pop	10	20
Push(10)	10 10	20
Push(20)	10 10 20	20
Pop	10 10	20 20
Pop	10	20 20 10
Pop		20 20 10 10
Push(20)	20	20 20 10 10
Pop		20 20 10 10 20

16.   Consider three pegs A, B, C and four disks of different sizes. Initially, the four disks are stacked on peg A, in order of decreasing size. The task is to move all the disks from peg A to peg C with the help of peg B. the moves are to be made under the following constraints:

[i]   In each step, exactly one disk is moved from one peg to another.
[ii]   A disk cannot be placed on another disk of smaller size. If we denote the movement of a disk from one peg to another by y → y, where y, y are A, B or C, then represent the sequence of the minimum number

of moves to accomplish this as a binary tree with node labels of the form (y → y) such that the in-order traversal of the tree gives the correct sequence of the moves..

If there are n disks, derive the formula for the total number of moves required in terms of n
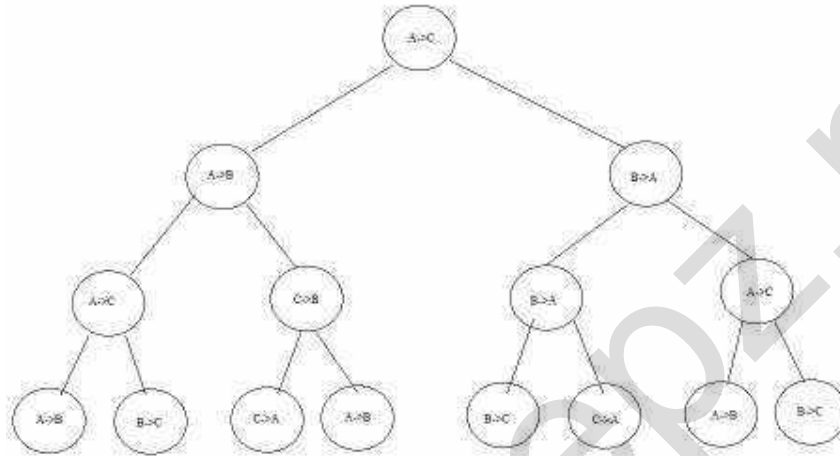
For one disk the steps involved is A->C ie 1 step

 For two disks the steps involved are A->B,A->C,B->C ie 3steps

For three disks the steps involved are A->C,A->B,C->B,A->C,B->A,B->C,A->C ie 7steps

For 4 disks the steps involved are A->B,A->C,B->C,A->B,C->A,C->B,A->B,A->C,B->C,B->A,C->A,B->A,A->B,A->C,B->C ie 15 steps

So for 1 disk $2^1-1=1$

2 disks $2^2-1=3$

3 disks $3^2-1=7$

4 disks $4^2-1=15$

……. ……….

……. ……….

n disks 2^n-1 (by induction)

17. A stack is use to pass parameters to procedures in a procedure call.

(b) If a procedure P has two parameters as described in procedure definition:
procedure P(var x: integer; y: integer);

and if P is called by:

P(a,b)

State precisely in a sentence what is pushed onto stack for parameters a and b.

In the generated code for the body of procedure P, how will the addressing of formal parameters y and y differ?

[QUESTION IS HIGHLY AMBIGUOUS]

18.   Which of the following permutation can be obtained in the output (in the same order) using a stack assuming that the input is the sequence1, 2, 3, 4, 5 in that order?

    (E)   3, 4, 5, 1, 2
    (F)   3, 4, 5, 2, 1
    (G)   1, 5, 2, 3, 4
    (H)   5, 4, 3, 1, 2

[F] As it can be verified as- push(1) push(2) push(3) pop push(4) pop push(5) pop pop pop

19.   Consider the following C program:

```
#include<stdio.h>

#define EOF -1

void push(int);              /* Push the argument on the stack */

int pop(void);               /* pop the top of the stack */

void flagError();

int main( )
{          int c, m, n, r;


           while ((c = getchar( )) != EOF)
           {          if (isdigit(c))

                           push (c)

                  else if (c = = '+') || (c = = '*'))

                  {          m = pop( );

                             n = pop( );

                             are = (c = = '+') ? n + m :  n*m;

                             push(r);

                  }

                  else if (c != ' ')
```

```
                                        flagError( );

                        }

                        printf("%c", pop( ));

                }
```

What is the output of the program for the following input?

                5 2 * 3 3 2 + * +

  (A) 15   (B) 25   (C) 30 (D) 150

[B] Sequence of operations will be

1. push(5)
2. push(2)
3. m = 2;n=5 ;r=5*2
4. push(r) /*push(10)*/
5. push(3)
6. push(3)
7. push(2)
8. m=2; n= 3;r=3+2
9. push(r) /* push(5)*/
10. r=5*3
11. push(r) /*push(15)*/
12. m=15;n = 10;r=15+10
13. push(r)
14. printf("%d",pop());

Which will print 25

20.   Suppose you are given an implementation of a queue of integers. The operations that can be performed on the queue are:

                isEmpty(Q)   –   returns true if the queue is empty, false otherwise.

                delete(Q)   – deletes the element at the front of the queue and   returns its value.

                 insert(Q, i)   – inserts the integer i at the rear of the queue.

        Consider the following function:

        void f(queue Q)

        {

                int i;

```
if(!isEmpty (Q)) {

        i = delete(Q);

        f(Q)

        insert(Q, i);

    }

}
```

What operation is performed by the above function f?

      (A) Leaves the queue Q unchanged
      (B) Reverse the order of elements in the queue Q
      (C) Deletes the element at the front of the queue Q and inserts it at the rear keeping the other elements in the same order
      (D) Empties the queue Q.

[B] Due to recursion the last element deleted will be the first to be inserted so element that was at rear will be at front in new queue and the second last element will be at second position hence the queue will be reversed.

21.   Let *w* be the minimum weight among all edge weights in an undirected connected graph. Let *e* be a specific edge of weight *w*. Which of the following is **FALSE**?

      (A) There is a minimum spanning tree containing *e*.
      (B) If *e* is not in a minimum spanning tree *T,* then in the cycle formed by adding *e* to T, all edges have the same weight.
      (C) Every minimum spanning tree has an edge of weight *w*.
      (D) *e* is present in every minimum spanning tree.

[D]

22.   Consider the following C function:

```
int  f(int n)

    {

     static int are = 0;

     If (n < = 0) return 1;

     If (n > 3)

            { r = n;

              return f (n – 2) + 2;

            }

     return f(n – 1) + r;
```

}

What is the value of f(5)?

(A) 3   (B) 7   (C) 9   (D) 18

[D]

f(5) will return f(3)+2

f(3) will return f(2)+5

f(2) will return f(1)+5

f(1) will return f(0)+5

f(0) will return 1

f(1) will return 6

f(2) will return 11

f(3) will return 16

f(5) will return 18

23.   The following postfix expression with single digit operands is evaluated using a stack:

$$8 \ 2 \ 3 \ \wedge \ / \ 2 \ 3 \ * \ + \ 5 \ 1 \ * \ -$$

Note that ^ is the exponentiation operator. The top two elements of the stack after the first *is evaluated are:

(A) 6, 1   (B) 5, 7   (C) 3, 2   (D) 1, 5

[A]

Symbols read so far	Operand stack
8	8
8 2	8 2
8 2 3	8 2 3
8 2 3^	8 8
8 2 3^/	1
8 2 3^/2	1 2

| 8 2 3^/2 3 | 1 2 3 |
| 8 2 3^/2 3 * | 1 6 |

24. An implementation of queue Q, using stacks S1 and S2 is given below:

```
void insert (Q, x) {

        push (S1, x);

}
void delete (Q) {

        if (stack-empty (S2)) then

                if (stack-empty (S1)) then {

                        print ( " Q is empty" );

                        return;

                }

                else while (! (stack-empty (S1))) then {

                        x = pop (S1);

                        push (S2, x);

                }
        x = pop (S2);

}
```

Let *n* insert and *m (≤ n)* delete operations be performed in an arbitrary order on an empty queue Q. let *x* and *y* be the number of push and pop operations performed respectively in the process. Which one of the following is true for all *m* and *n*?

(A) $n + m \leq x < 2n$ and $2m \leq y \leq n + m$ (B) $n + m \leq x < 2n$ and $2m \leq y \leq 2n$

(C) $2m \leq x < 2n$ and $2m \leq y \leq n + m$ (D) $2m \leq x < 2n$ and $2m \leq y \leq 2n$

[A]

Extreme case will be when all the elements are inserted and then deleted

For each insert we have a push operation

And for first delete we have n push operations if n elements are in stack, so n+m<=x in worst case

And no of pushes is also 2n : n for n insert and n for first delete

So (n+m)<= x < 2n

Also n pops for first delete and m-1 one pops for m-1 deletes so y= n+m-1 in extreme case y< n+m-1

25. A function f defined on stacks of integers satisfies the following properties.

   f(ϕ) = 0 and

   f(push(**S**, i)) = max(f(**S**), 0) + i for all stacks **S** and integers i.

If a stack **S** contains the integers 2, -3, 2, -1, 2 in order from bottom to top, what is f(**S**)?

(A) 6   (B) 4   (C) 3   (D) 2

[C] Answer is 3

Step1:top s=2

Step2: tops= -1

Step3: tops= 2

Step4: top s=1

   Step5: top s=3

26. Assume that the operators +, -, × are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, ×, +, -. The postfix expression corresponding to the infix expression a+b×c-d^e^f is

(A) abc×+def ^ ^ -
(B) abc×+de ^ f ^ -
(C) ab+c×d-e ^f ^
(D) -+a×bc ^ ^ def

[A]

Infix string	Stack	Postfix array
a		a
+	+	a
b	+	ab
*	+*	ab

c	+*	abc
-	-	abc*+
d	-	abc*+d
^	-^	abc*+d
e	-^	abc*+de
^	-^^ (not poped because of right associative)	abc*+de
f	-^^	abc*+def
	-^	abc*+def^
	-	abc*+def^^
		abc*+def^^-

27. The best data structure to check whether an arithmetic expression has balanced parenthesis is a

    (A) Queue
    (B) Stack
    (C) Tree
    (D) List

[B] We can put every opening parenthesis in stack and pop every time we get a closed one. In the ened if we get stack empty then we had balanced parenthesis else not.

Binary Tree using Array Representation

Each node contains **info, left, right** and **father** fields. The left, right and father fields of a node point to the node's left son, right son and father respectively.

Using the array implementation, we may declare,
```
#define NUMNODES 100
struct nodetype
        {
        int info;
        int left;
        int right;
        int father;
        };
struct nodetype node[NUMNODES];
```

This representation is called linked array representation.

Example: -
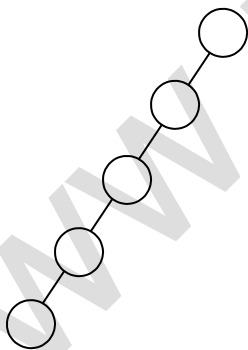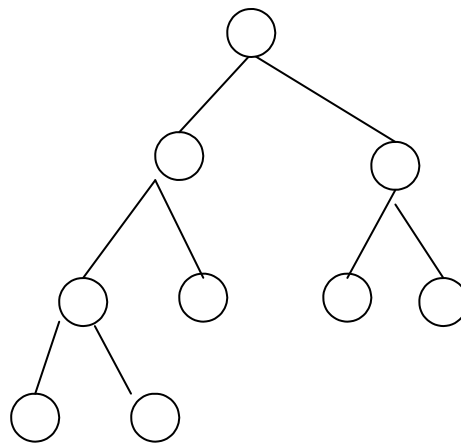
Fig (a)   Fig (b)

The above trees can be represented in memory sequentially as follows

| A |
B
C
D
E
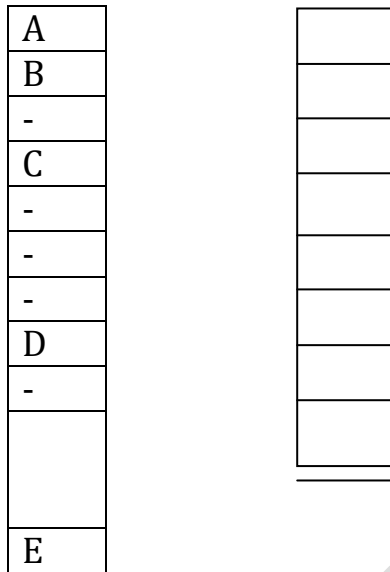
The above representation appears to be good for complete binary trees and wasteful for many other binary trees. In addition, the insertion or deletion of nodes from the middle of a tree requires the insertion of many nodes to reflect the change in level number of these nodes.
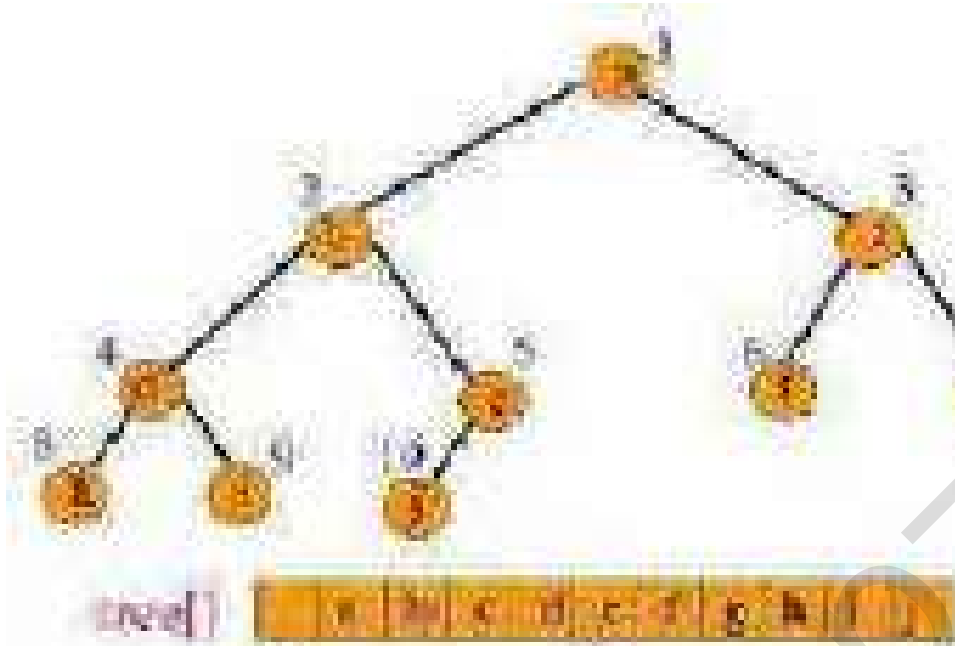
Figure 2.5
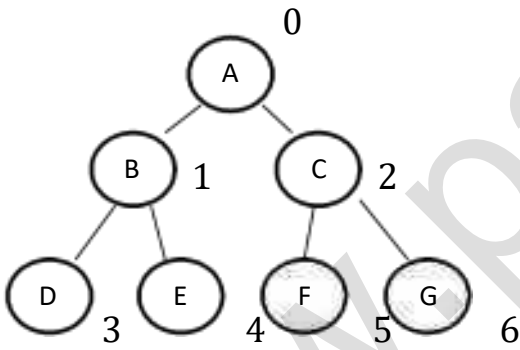
Figure 2.6

For Figure 2.5

Root = **i**

leftchild=2i+1
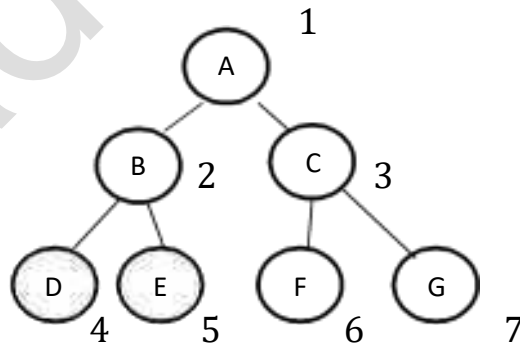
For Figure 2.6

Root = **i**

leftchild=2i

rightchild=2i+2

leftchild's parent position = i/2

$2^{n+1} - 1$ => array size

n => no of levels of a tree

rightchild's position= i-1/2

rightchild=2i+1

parent position= i/2

$2^{n+1} - 1$ => size of array

n => number of levels of a tree

_ _

Binary Tree using Link Representation

The problems of sequential representation can be easily overcome through the use of a linked representation.
Each node will have three fields LCHILD, DATA and RCHILD as represented below
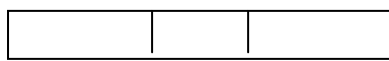
LCHILD   DATA   RCHILD
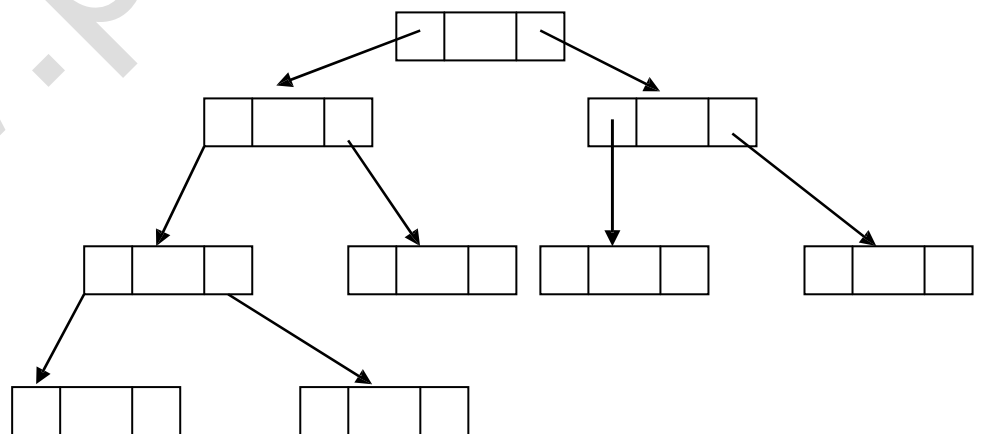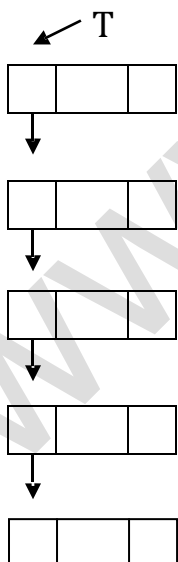
Fig (a)   Fig (b)

 In most applications it is adequate. But this structure make it difficult to determine the parent of a node since this leads only to the forward movement of the links.

Using the linked implementation we may declare,

Struct treenode
** {**
int data;
structtreenode *leftchild;
structtreenode *rightchild;
** }*T;**
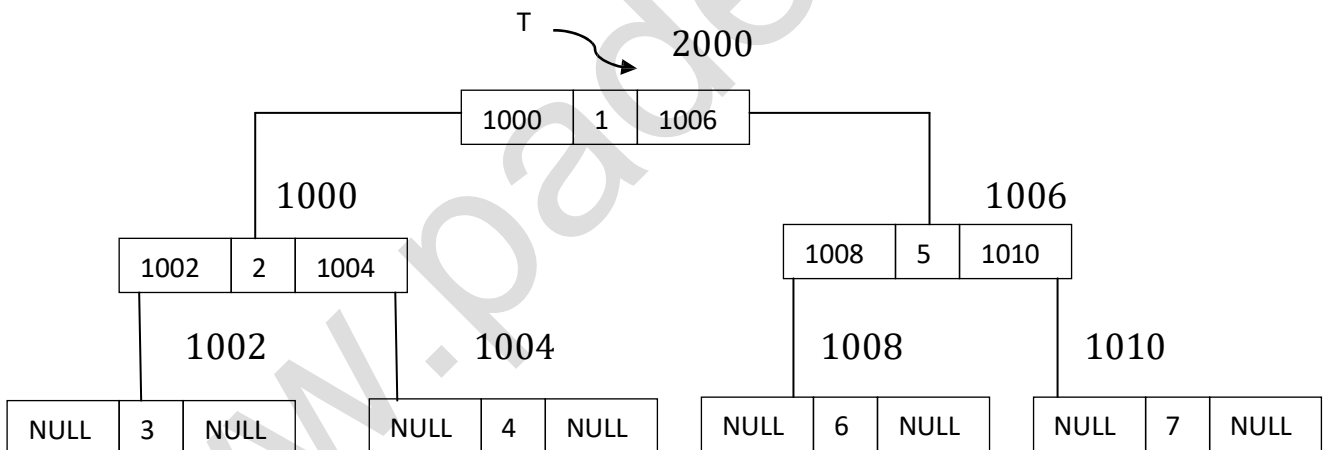
TYPES OF BINARY TREES

Left Skewed Binary tree :

A binary tree which has only left child is called left skewed binary tree.
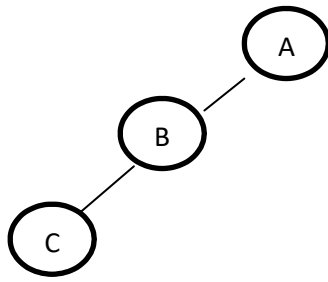
Right Skewed Binary tree :

A Binary tree which has only right child is called right skewed binary tree.
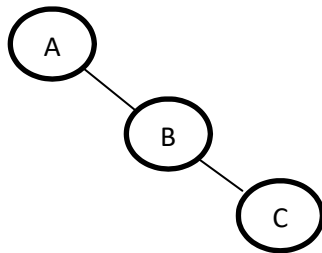
Full Binary Tree :

It is the one which has exactly two children for each node at each level and all the leaf nodes should be at the same level.

Complete Binary Tree :

It is the one tree where all the leaf nodes need not be at the same level and at the bottom level of the complete binary tree, the nodes should be filled from the left to the right.
All full binary trees are complete binary tree. But all complete binary trees need not be full binary tree.
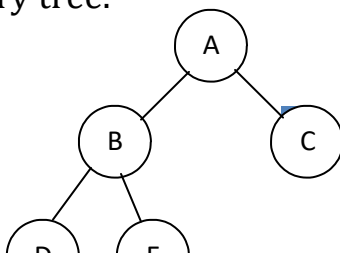
CONVERSION OF A GENERAL TREE TO BINARY TREE

General Tree:

A General Tree is a tree in which each node can have an unlimited out degree. Each node may have as many children as is necessary to satisfy its requirements. *Example: Directory Structure*
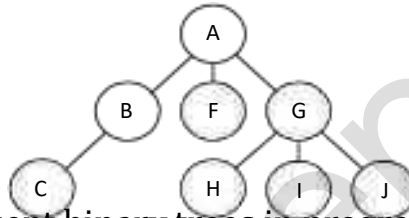
It is considered easy to represent binary trees in programs than it is to represent general trees. So, the general trees can be represented in binary tree format.

Changing general tree to Binary tree:

The binary tree format can be adopted by changing the meaning of the left and right pointers. There are two relationships in binary tree,
Parent to child
Sibling to sibling
Using these relationships, the general tree can be implemented as binary tree.

Algorithm

Identify the branch from the parent to its first or leftmost child. These branches from each parent become left pointers in the binary tree
Connect siblings, starting with the leftmost child, using a branch for each sibling to its right sibling.
Remove all unconnected branches from the parent to its children
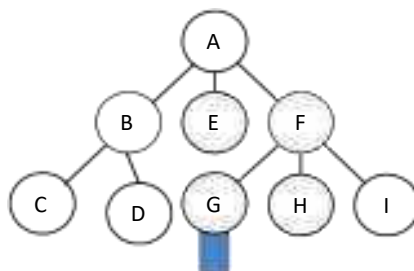
(a) General Tree
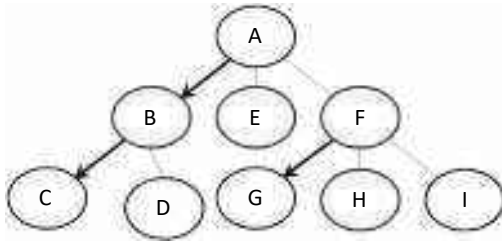
Step 1: Identify all leftmost children
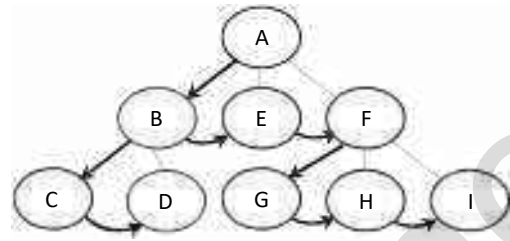
Step 2: Connect Siblings
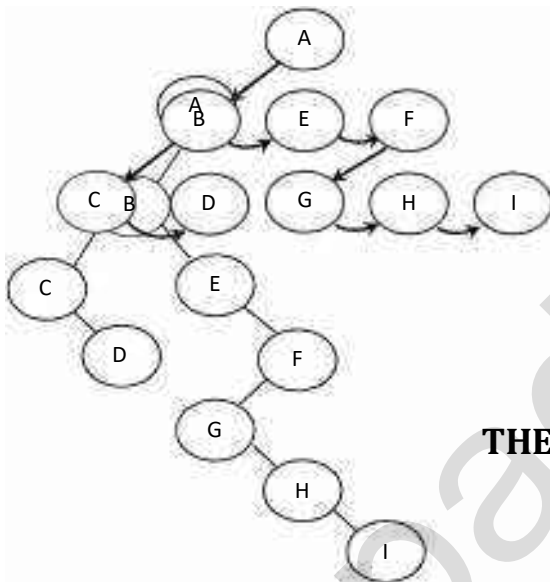
Step 3: Delete unneeded branches

THE RESULTING BINARY TREE

BINARY TREE TRAVERSALS

Compared to linear data structures like linked lists and one dimensional array, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node),

traversing to the left child node, and traversing to the right child node. Thus the process is most easily described through recursion.

A binary tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence.
The two general approaches to the traversal sequence are,

Depth first traversal
Breadth first traversal

Breadth-First Traversal

In a breadth-first traversal, the processing proceeds horizontally form the root to all its children, then to its children's children, and so forth until all nodes have been processed. In other words, in breadth traversal, each level is completely processed before the next level is started.
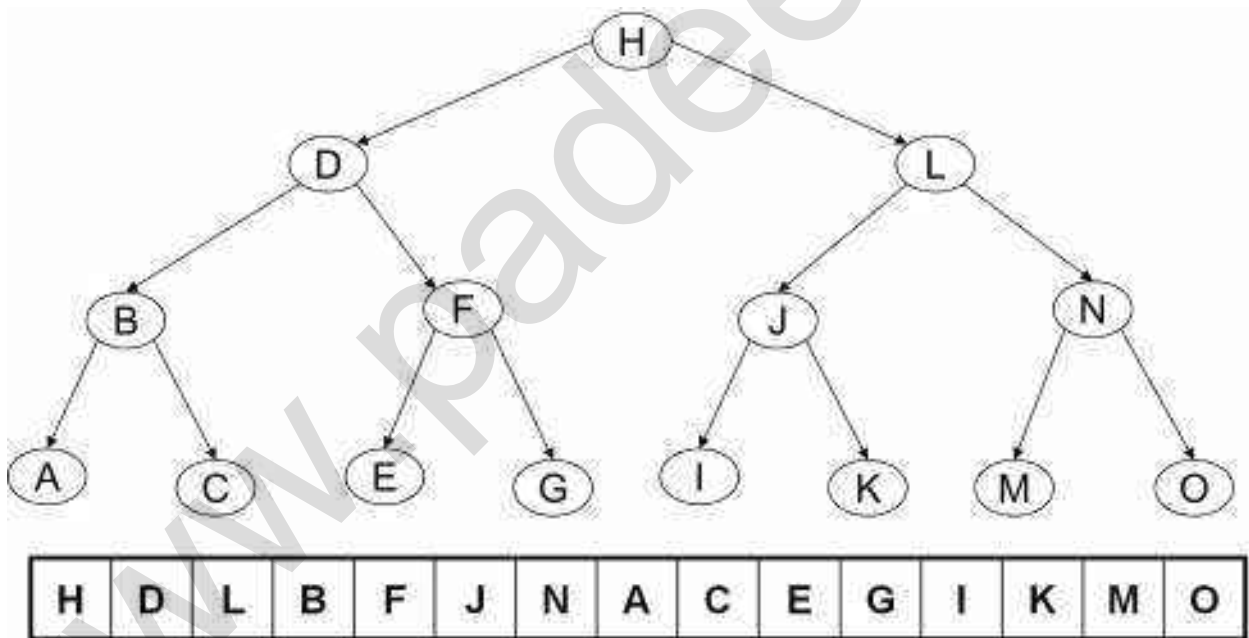
Depth-First Traversal

In depth first traversal, the processing proceeds along a path from the root through one child to the most distant descendent of that first child before

processing a second child. In other words, in the depth first traversal, all the descendants of a child are processed before going to the next child.

| H | D | B | A | C | F | E | G | L | J | I | K | N | M | O |
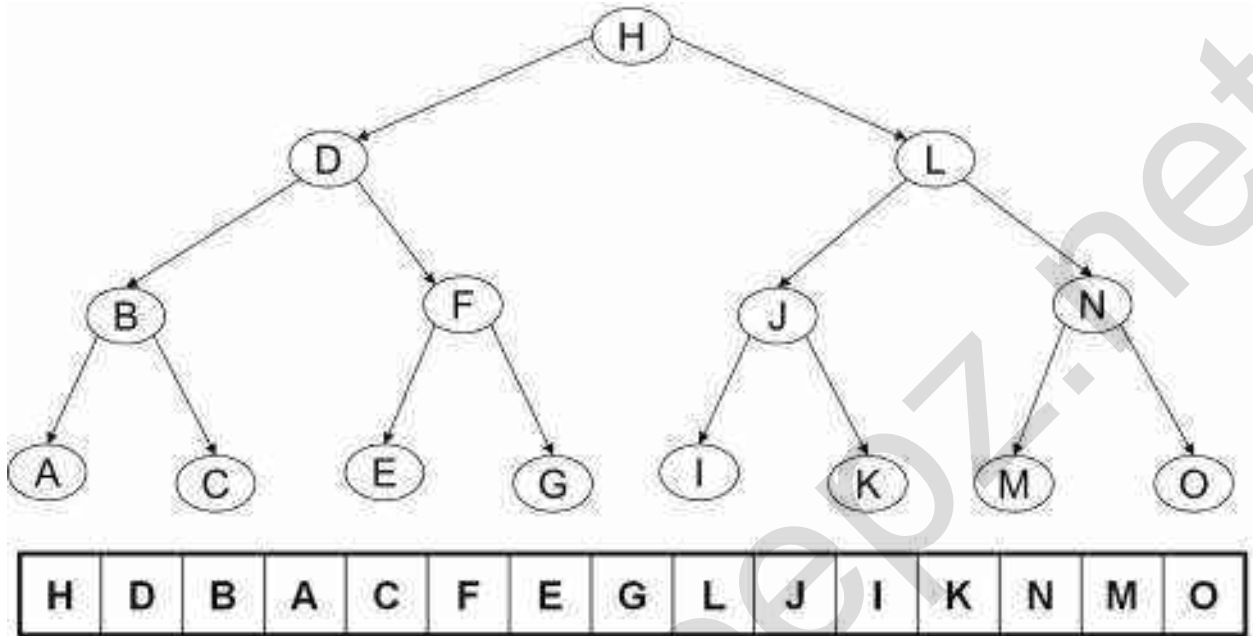
There are basically three ways of binary tree traversals.
1. Inorder --- (left child,root,right child)
2. Preorder --- (root,left child,right child)
3. Postorder --- (left child,right child,root)
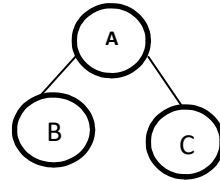
Inorder--- B A C

Preorder --- A B C

Postorder --- B C A

In C, each node is defined as a structure of the following form:
        struct node
                {
                int info;
                struct node *lchild;
                struct node *rchild;
                }
        typedef struct node NODE;

Inorder Traversal

Steps :
Traverse left subtree in inorder
Process root node
Traverse right subtree in inorder
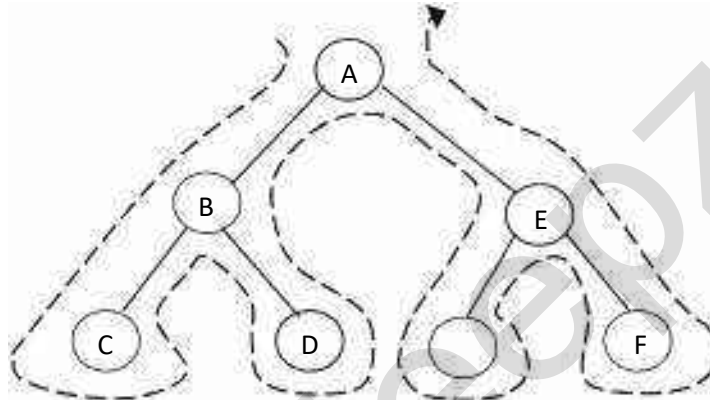
The Output is : C → B → D → A → E → F

Algorithm
Algorithm inoder traversal (BinTree T)
Begin
If (not empty (T)) then
Begin
Inorder_traversal (left subtree (T))
Print (info (T))   / * process node */
Inorder_traversal (right subtree (T))
End
End

Routines
void inorder_traversal (NODE * T)
{
if(T ! = NULL)
{

```
inorder_traversal(T->lchild);
printf("%d \t ", T->info);
inorder_traversal(T->rchild);
}
}
```

Preorder Traversal

Steps :
Process root node
Traverse left subtree in preorder
Traverse right subtree in preorder

Algorithm
Algorithm inoder traversal (BinTree T)
Begin
If (not empty (T)) then
Begin
Print (info (T))   / * process node */
Preorder_traversal (left subtree (T))
Preorder_traversal (right subtree (T))
End
End

Routines
```
void inorder_traversal ( NODE * T)
{
if( T ! = NULL)
{
printf("%d \t ", T->info);
preorder_traversal(T->lchild);
preorder_traversal(T->rchild);
}
}
```
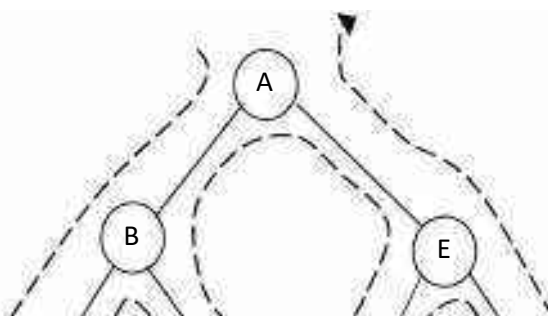
Output is : A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ F

Postorder Traversal

Steps :
Traverse left subtree in postorder
Traverse right subtree in postorder
process root node

Algorithm
Algorithm postorder traversal (BinTree T)
Begin
If (not empty (T)) then
Begin
Postorder_traversal (left subtree (T))
Postorder_traversal (right subtree(T))
Print (Info (T)) / * process node */
End
End

Routines
void postorder_traversal (NODE * T)
```
 {
if( T ! = NULL)
{
postorder_traversal(T->lchild);
postorder_traversal(T->rchild);
printf("%d \t", T->info);
```

```
}
}
```
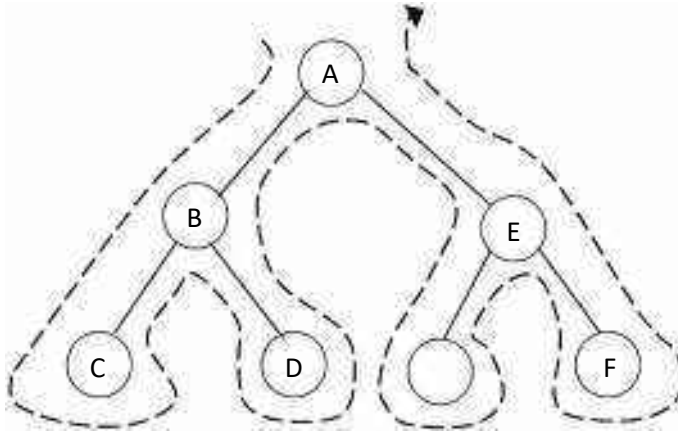

Output is : C → D → B → F → E → A

Examples :

1.FIND THE TRAVERSAL OF THE FOLLOWING TREE
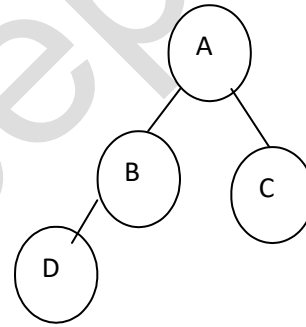
ANSWER : POSTORDER: DBCA   INORDER: DBAC   PREORDER:ABCD

2.FIND THE TRAVERSAL OF THE FOLLOWING TREE
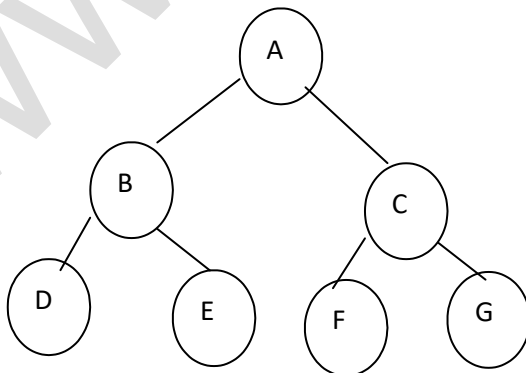
ANSWER :   POSTORDER: DEBFGCA   INORDER: DBEAFCG

PREORDER:ABDECFG

3.A BINARY TREE HAS 8 NODES. THE INORDER AND POSTORDER TRAVERSAL OF THE TREE ARE GIVEN BELOW. DRAW THE TREE AND FIND PREORDER.

POSTORDER: F E C H G D B A

INORDER: F C E A B H D G

Answer:
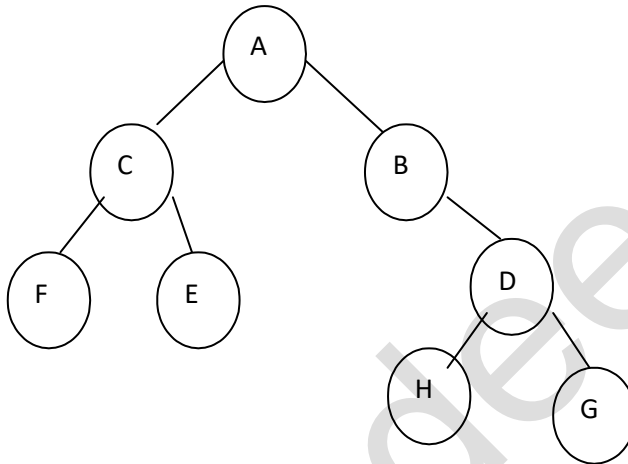
PREORDER: ACFEBDHG

Example 4
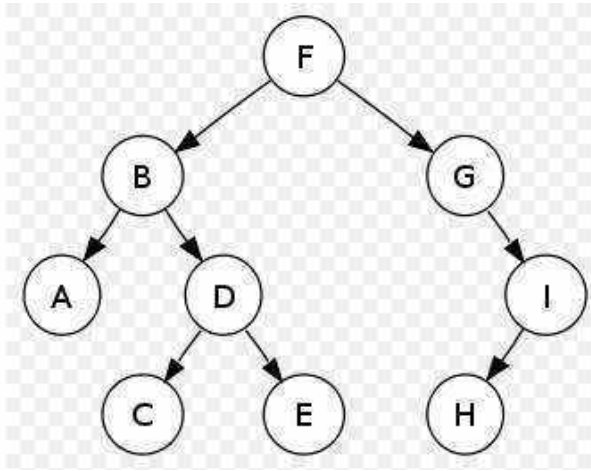
Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right)
Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)

APPLICATIONS

1.Some applications of preorder traversal are the evaluation of expressions in prefix notation and the processing of abstract syntax trees by compilers.
2.Binary search trees (a special type of BT) use inorder traversal to print all of their data in alphanumeric order.
3.A popular application for the use of postorder traversal is the evaluating of expressions in postfix notation.

EXPRESSION TREES

Algebraic expressions such as

a/b+(c-d)e
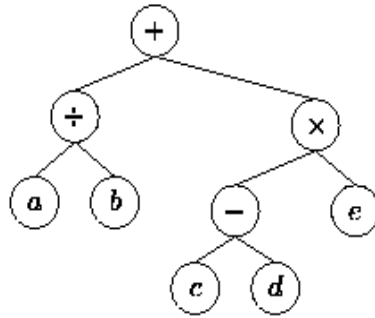
Tree representing the expression $a/b+(c-d)e$.

Converting Expression from Infix to Postfix using STACK

To convert an expression from infix to postfix, we are going to use a stack.

Algorithm
1) Examine the next element in the input.
2) If it is an operand, output it.
3) If it is opening parenthesis, push it on stack.
4) If it is an operator, then
i) If stack is empty, push operator on stack.
ii) If the top of the stack is opening parenthesis, push operator on stack.
iii) If it has higher priority than the top of stack, push operator on stack.
iv) Else pop the operator from the stack and output it, repeat step 4.
5) If it is a closing parenthesis, pop operators from the stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
6) If there is more input go to step 1
7) If there is no more input, unstack the remaining operators to output.

Example
Suppose we want to convert 2*3/(2-1)+5*(4-1) into Prefix form: Reversed Expression:)1-4(*5+)1-2(/3*2

Char Scanned	Stack Contents(Top on right)	Postfix Expression
2	Empty	2
*	*	2

wwww.padeepz.net

3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/5
(	+*(	23*21-/5
4	+*(	23*21-/54
-	+*(-	23*21-/54
1	+*(-	23*21-/541
)	+*	23*21-/541-
	Empty	23*21-/541-*+

So, the Postfix Expression is 23*21-/541-*+

Converting Expression from Infix to Prefix using STACK

It is a bit trickier algorithm, in this algorithm we first reverse the input
expression so that a+b*c will become c*b+a and then we do the conversion
and then again the output string is reversed. Doing this has an advantage that
except for some minor modifications the algorithm for Infix->Prefix remains
almost same as the one for Infix->Postfix.

Algorithm
1) Reverse the input string.
2) Examine the next element in the input.
3) If it is operand, add it to output string.
4) If it is Closing parenthesis, push it on stack.
5) If it is an operator, then

i) If stack is empty, push operator on stack.

ii) If the top of stack is closing parenthesis, push operator on stack.

iii) If it has same or higher priority than the top of stack, push operator on stack.

iv) Else pop the operator from the stack and add it to output string, repeat step 5.

6) If it is a opening parenthesis, pop operators from stack and add them to output string until a closing parenthesis is encountered. Pop and discard the closing parenthesis.

7) If there is more input go to step 2

8) If there is no more input, unstack the remaining operators and add them to output string.

9) Reverse the output string.

Example

Suppose we want to convert 2*3/(2-1)+5*(4-1) into Prefix form: Reversed Expression:)1-4(*5+)1-2(/3*2

Char Scanned	Stack Contents(Top on right)	Prefix Expression(right to left)
)	)	
1	)	1
-	)-	1
4	)-	14
(	Empty	14-
*	*	14-
5	*	14-5
+	+	14-5*
)	+)	14-5*
1	+)	14-5*1
-	+)-	14-5*1
2	+)-	14-5*12
(	+	14-5*12-

/	+/	14-5*12-
3	+/	14-5*12-3
*	+/*	14-5*12-3
2	+/*	14-5*12-32
	Empty	14-5*12-32*/+

Reverse the output string : +/*23-21*5-41 So, the final Prefix Expression is +/*23-21*5-41

EVALUATION OF EXPRESSIONS
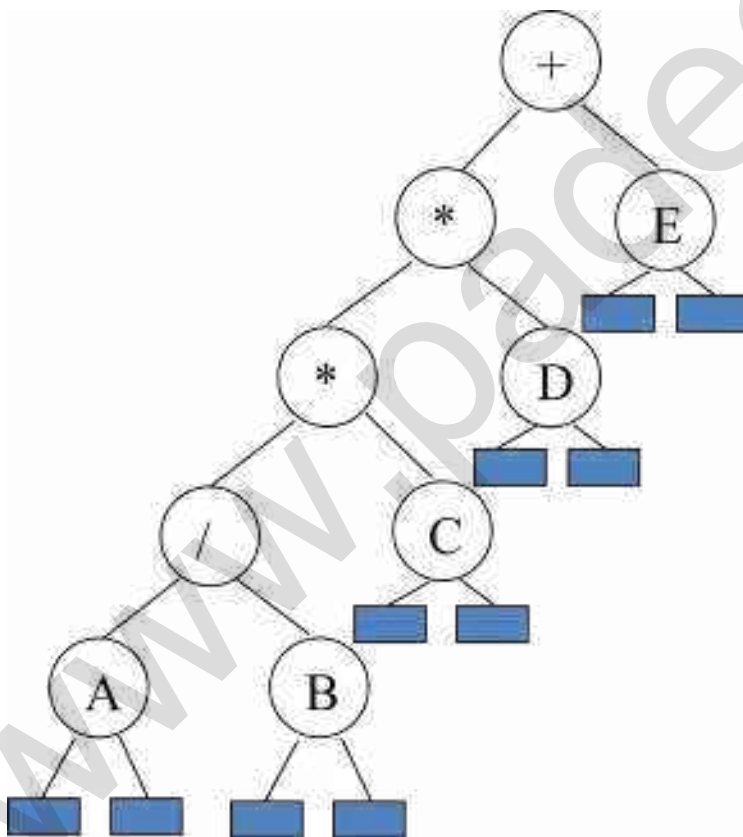
inorder traversal
A / B * C * D + E
infix expression

preorder traversal
+ * * / A B C D E
prefix expression

postorder traversal
A B / C * D * E +
postfix expression

CONSTRUCTING AN EXPRESSION TREE

Let us consider the postfix expression given as the input, for constructing an expression tree by performing the following steps :

1. Read one symbol at a time from the postfix expression.
2. Check whether the symbol is an operand or operator.

     i. If the symbol is an operand, create a one node tree and push a pointer on to the stack.

     ii. If the symbol is an operator, pop two pointers from the stack namely, T1 and T2 and form a new tree with root as the operator, and T2 as the left child and T1 as the right child.

     iii. A pointer to this new tree is then pushed on to the stack.

We now give an algorithm to convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix to postfix, we can generate expression trees from the two common types of input. The method we describe strongly resembles the postfix evaluation algorithm of Section 3.2.3. We read our expression one symbol at a time. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack.

As an example, suppose the input is

a b + c d e + * *

The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.*

*For convenience, we will have the stack grow from left to right in the diagrams.
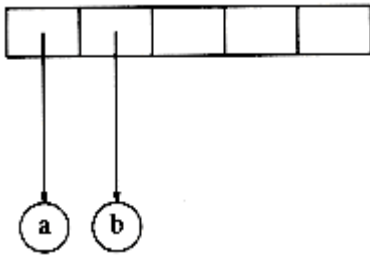
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.*
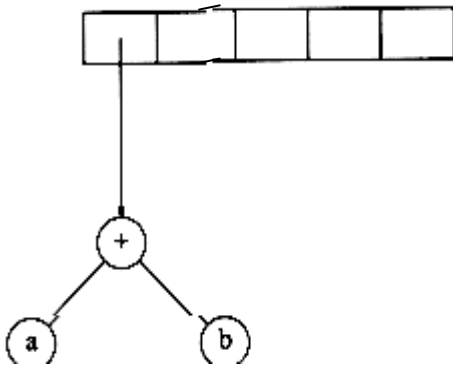
Next, c, d, and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.
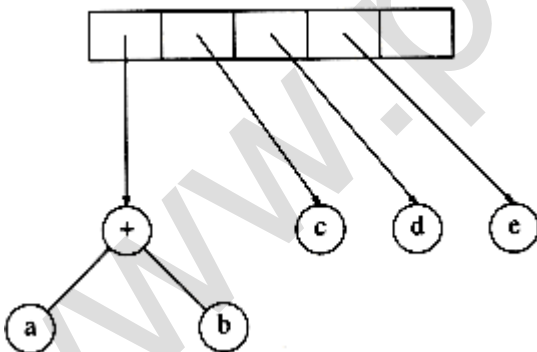
Now a '+' is read, so two trees are merged.
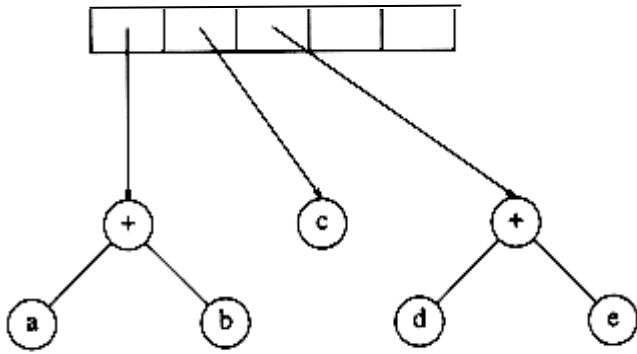
Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.
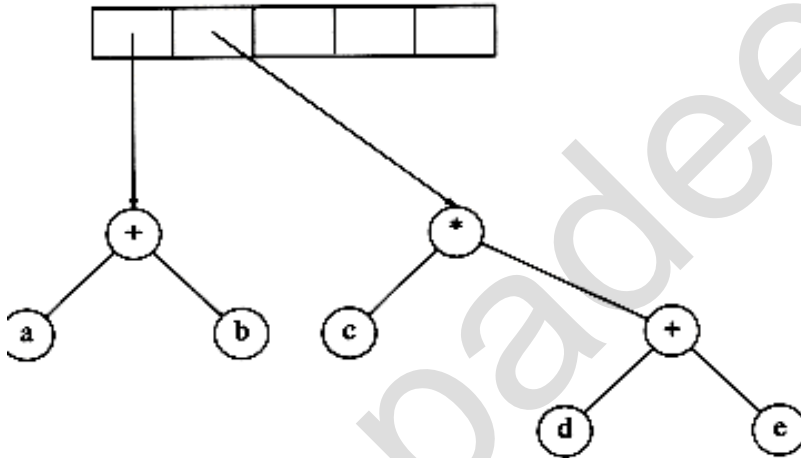
Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.
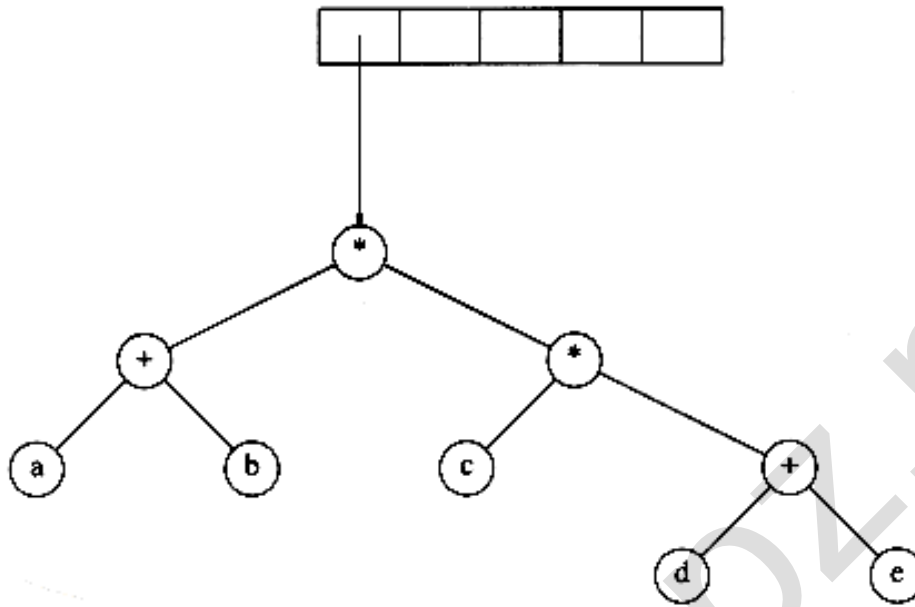
BINARY SEARCH TREE

Binary search tree (**BST**) is a node-based binary tree data structure which
has the following properties:

- The left sub-tree of a node contains only nodes with keys less than the
  node's key.
- The right sub-tree of a node contains only nodes with keys greater than
  the node's key.
- Both the left and right sub-trees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.
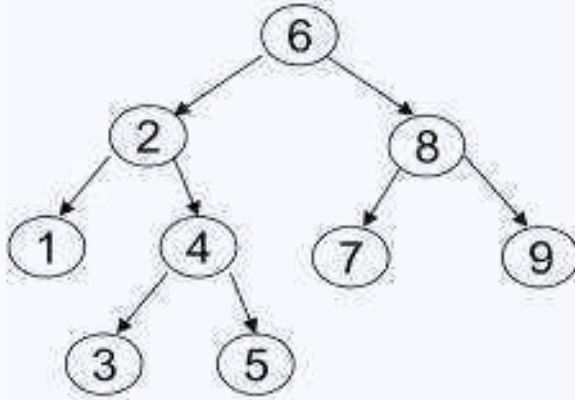
Program: Creating a Binary Search Tree

We assume that every node of a binary search tree is capable of holding an integer data item and that the links can be made to point to the root of the left subtree and the right subtree, respectively. Therefore, the structure of the node can be defined using the following declaration:

```
struct tnode
{
int data;
struct tnode *lchild,*rchild;
};
```

A complete C program to create a binary search tree follows:

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
{
int data;
struct tnode *lchild, *rchild;
};

struct tnode *insert(struct tnode *p,int val)
{
struct tnode *temp1,*temp2;
if(p == NULL)
{
```

```
p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as
root node*/
if(p == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
p->data = val;
p->lchild=p->rchild=NULL;
}
else
{
temp1 = p;
/* traverse the tree to get a pointer to that node whose child will be the newly
created node*/
while(temp1 != NULL)
{
temp2 = temp1;
if( temp1 ->data > val)
temp1 = temp1->lchild;
else
temp1 = temp1->rchild;
}
if( temp2->data > val)
{
temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/*inserts the
newly created node as left child*/
temp2 = temp2->lchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
else
{
```

```
temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the
newly created node
as left child*/
temp2 = temp2->rchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
}
}
return(p);
}
/* a function to binary tree in inorder */
void inorder(struct tnode *p)
{
if(p != NULL)
{
inorder(p->lchild);
printf("%d\t",p->data);
inorder(p->rchild);
}
}
void main()
{
struct tnode *root = NULL;
int n,x;
printf("Enter the number of nodes\n");
scanf("%d",&n);
while( n - > 0)
{
printf("Enter the data value\n");
scanf("%d",&x);
root = insert(root,x);
}
inorder(root);
```

}

EXAMPLE Construct a BST with nodes 2,4,5,7,1

Normal Tree
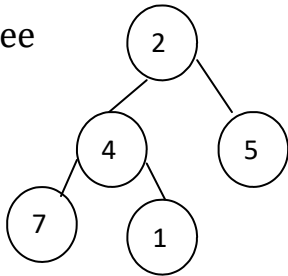
Binary Search Tree

- The Values in the left subtree must be smaller than the keyvalue to be inserted.
- The Values in the right subtree must be larger than the keyvalue to be inserted.

Take the 1st element 2 and compare with 4.   2<4

So
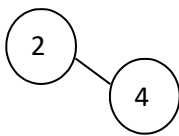
Similarly   2<5,5>4 and 7>2,7>4,7>5
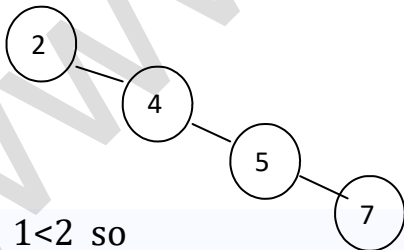
and 1<2 so
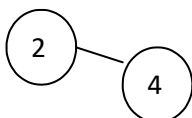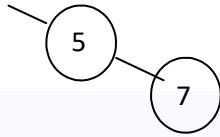
is the final BST.

OPERATIONS

- Operations on a binary tree require comparisons between nodes. These
- comparisons are made with calls to a comparator, which is a subroutine
  that computes the total order (linear order) on any two values. This
- comparator can be explicitly or implicitly defined, depending on the
  language in which the BST is implemented.
- The following are the operations that are being done in Binary Tree
  - ➢ Searching.
  - ➢ Sorting.
  - ➢ Deletion.
  - ➢ Insertion.

Binary search tree declaration routine

Struct treenode;
Typedef struct treenode *position;
Typedef struct treenode *searchtree;
Typedef int elementtype;
Structtreenode
{
Elementtype element;
Searchtree left;
Searchtree right;
};
Struct treenode
{
int element;
struct treenode *left;
struct treenode *right;

};

Make_null

This operation is mainly for initialization. Some programmers prefer to initialize the first element as a one-node tree, but our implementation follows the recursive definition of trees more closely. It is also a simple routine.

Routine to make an empty tree

```
SEARCH_TREE
make_null ( void )
{
return NULL;
}
```

Find

This operation generally requires returning a pointer to the node in tree *T* that has key *x*, or *NULL* if there is no such node. The structure of the tree makes this simple. If *T* is *, then we can just return* . Otherwise, if the key stored at *T* is *x*, we can return *T*. Otherwise, we make a recursive call on a subtree of *T*, either left or right, depending on the relationship of *x* to the key stored in *T*. The code in Figure 4.18 is an implementation of this strategy.

Find operation for binary search trees

```
Position find(structtreenode T, intnum)
{
While(T!=NULL)
{
if(num>T-->data)
{
T=T-->right;
if(num<T-->data)
```

```
T=T-->left;
}
else if(num< T-->data)
{
T=T-->left;
if(num>T-->data)
T=T-->right;
}
if(T-->data==num)
break;
}
return T;
}
// To find a Number
Position find(elementtype X, searchtree T)
{
If(T==NULL)
return NULL;
if(x< T-->element)
return find(x,T-->left);
else if(X> T-->element)
return find(X,T-->right);
else
return T;
}
```

Find_min and Find_max

Recursive implementation of find_min & find_max for binary search trees

```
// Finding Minimum
Position findmin(searchtree T)
{
if(T==NULL)
```

```
return NULL;
else if(T-->left==NULL)
return T;
else return findmin(T-->left);
}
// Finding Maximum
Position findmax(searchtree T)
{
if(T==NULL)
return NULL;
else if(T-->right==NULL)
return T;
else return findmin(T-->right);
}
```

Nonrecursive implementation of find_min & find_max for binary search trees

```
// Finding Maximum
Position findmax(searchtree T)
{
if( T!=NULL)
while(T-->Right!=NULL)
T=T-->right;
Return T;
}
// Finding Minimum
Position findmin(searchtree T)
{
if( T!=NULL)
while(T-->left!=NULL)
T=T-->left;
Return T;
}
```

Insert

The insertion routine is conceptually simple. To insert *x* into tree *T*, proceed down the tree as you would with a *find*. If *x* is found, do nothing (or "update" something). Otherwise, insert *x* at the last spot on the path traversed. Figure below shows what happens. To insert 5, we traverse the tree as though a *find* were occurring. At the node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct spot.

Duplicates can be handled by keeping an extra field in the node record indicating the frequency of occurrence. This adds some extra space to the entire tree, but is better than putting duplicates in the tree (which tends to make the tree very deep). Of course this strategy does not work if the key is only part of a larger record. If that is the case, then we can keep all of the records that have the same key in an auxiliary data structure, such as a list or another search tree.
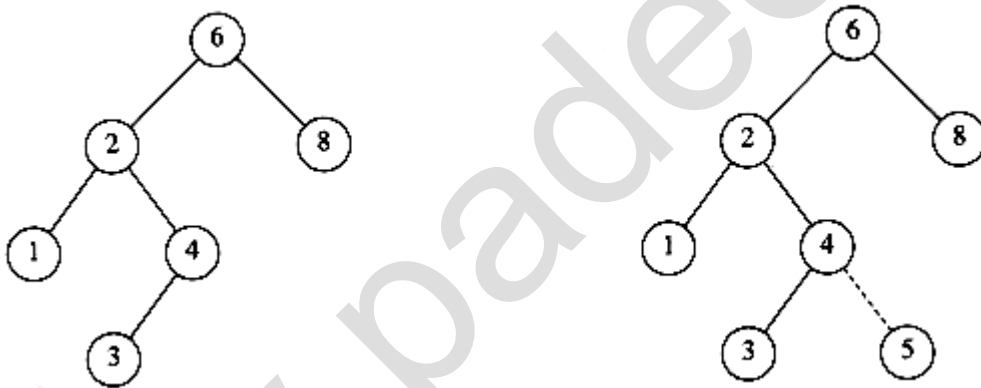
Figure shows the code for the insertion routine. Since *T* points to the root of the tree, and the root changes on the first insertion, *insert* is written as a function that returns a pointer to the root of the new tree. Lines 8 and 10 recursively insert and attach *x* into the appropriate subtree.

Insertion into a binary search tree

Searchtree insert(elementtype X, Searchtree T)

```
{
If(T== NULL)
{
/* create and return a one node tree*/
T=malloc(sizeof(structtreenode));
If(T==NULL)
Fatalerror("Out of Space");
Else
{
T-->element=X;
T-->left=T-->right=NULL;
}
}
Else if(x<T-->element)
T-->left=insert(X,T-->left);
Else if(X>=T-->left)
T-->right=insert(X,T-->right);
Return T;
}
```
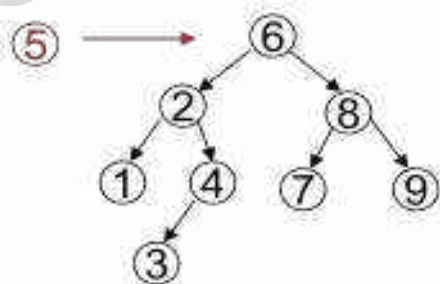

EXAMPLE   Insert node 5 in given tree

STEP 1: Now 5<6 and 5>2 and 5<4 so
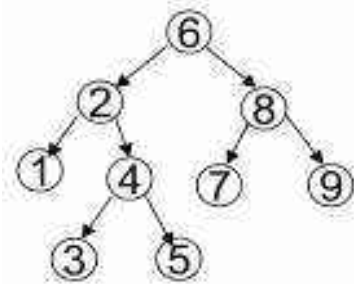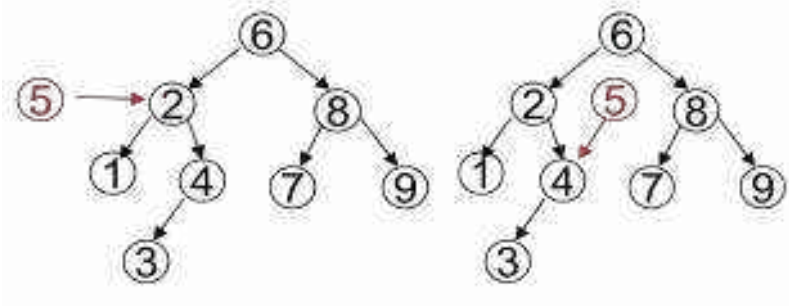
Thus 5 is inserted.

Delete

As is common with many data structures, the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node (we will draw the pointer directions explicitly for clarity).. Notice that the deleted node is now unreferenced and can be disposed of only if a pointer to it has been saved. The complicated case deals with a node with two children. The general strategy is to replace the key of this node with the smallest key of the right subtree (which is easily found) and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second *delete* is an easy one.

To delete an element, consider the following three possibilities :

Case 1: Node to be deleted is a leaf node.
Case 2: Node with only one child.
Case 3: Node with two children.

Case 1: Node with no children | Leaf node :
1. Search the parent of the leaf node and make the link to the leaf node as NULL.
2. Release the memory of the deleted node.

Case 2: Node with only one child :
1. Search the parent of the node to be deleted.
2. Assign the link of the parent node to the child of the node to be deleted.
3. Release the memory for the deleted node.

Case 3: Node with two children :
 It is difficult to delete a node which has two children.
 So, a general strategy has to be followed.
1. Replace the data of the node to be deleted with either the largest element from the left subtree or the smallest element from the right subtree.

Case 1:
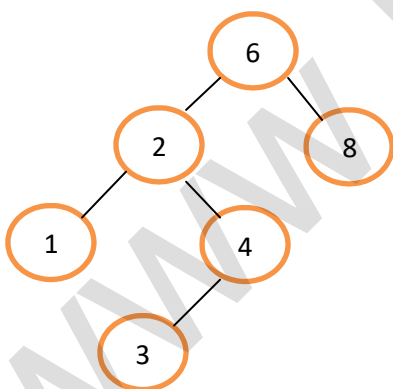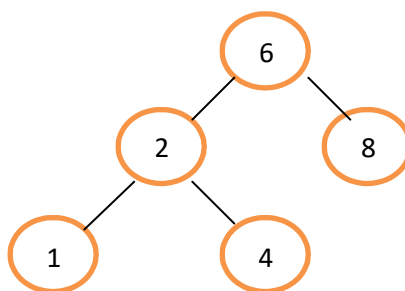
Before Deletion   After Deletion of 3

EXAMPLE   :
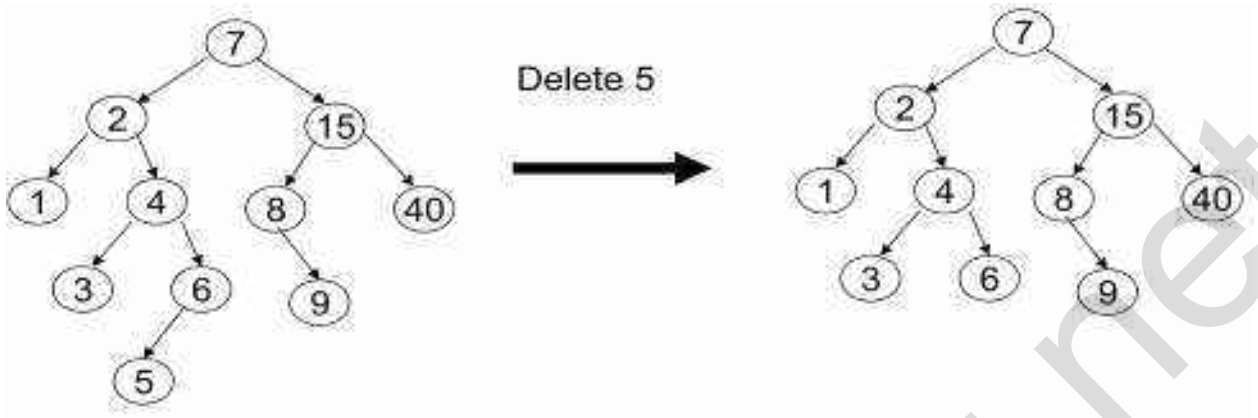
Delete 5

Case 2 :
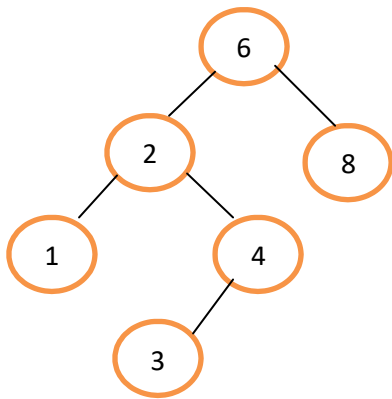
Before deletion of 4   After deletion of 4
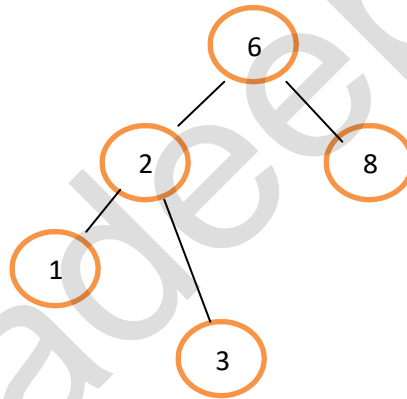
EXAMPLE   The right subtree of the node x to be deleted is empty.
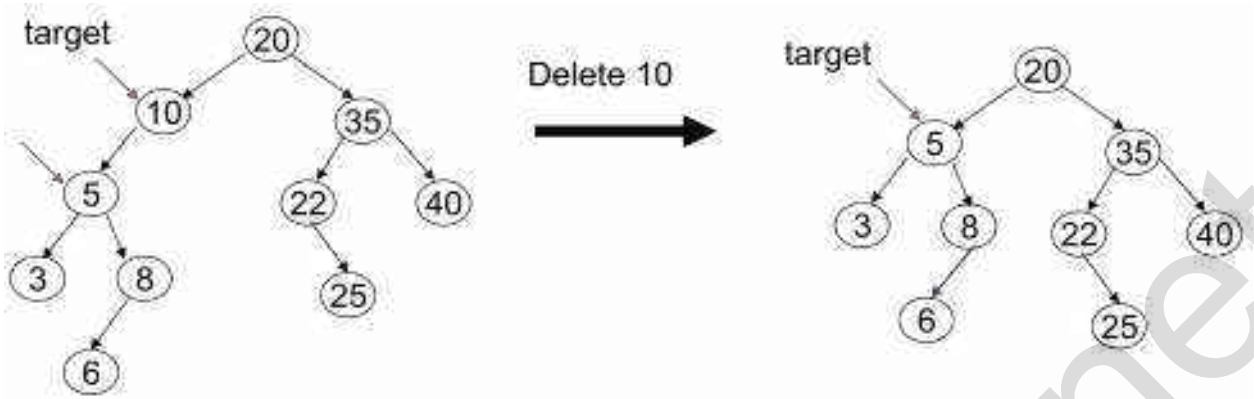
EXAMPLE <u>The left subtree of the node x to be deleted is empty.</u>
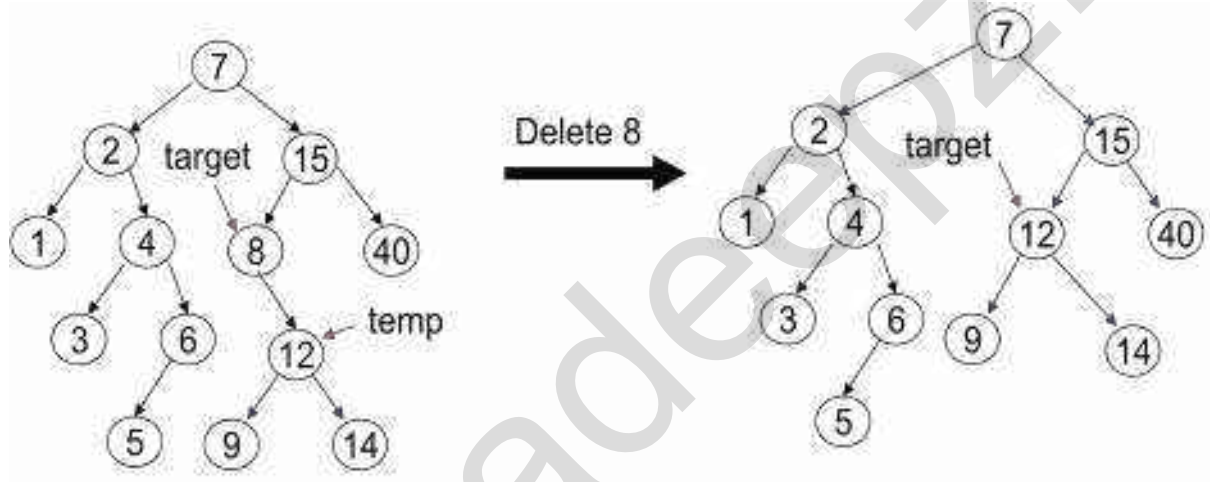
Case 3 :
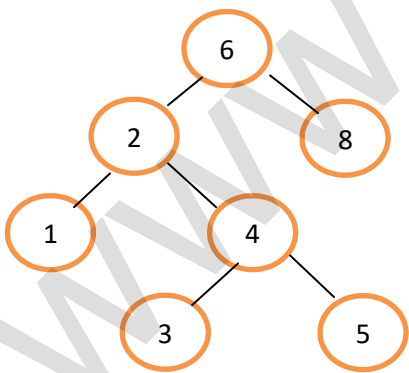
Before deletion of 2   After deletion
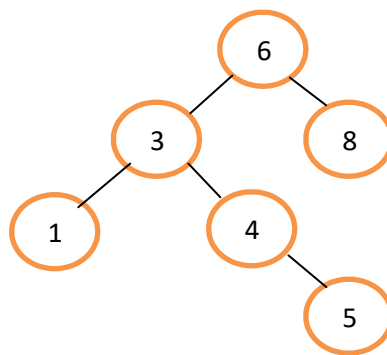
EXAMPLE

DELETION BY COPYING: METHOD#1
   **Copy the minimum key in the right subtree of x to the node x, then
delete the one-child or leaf-node with this minimum key.**
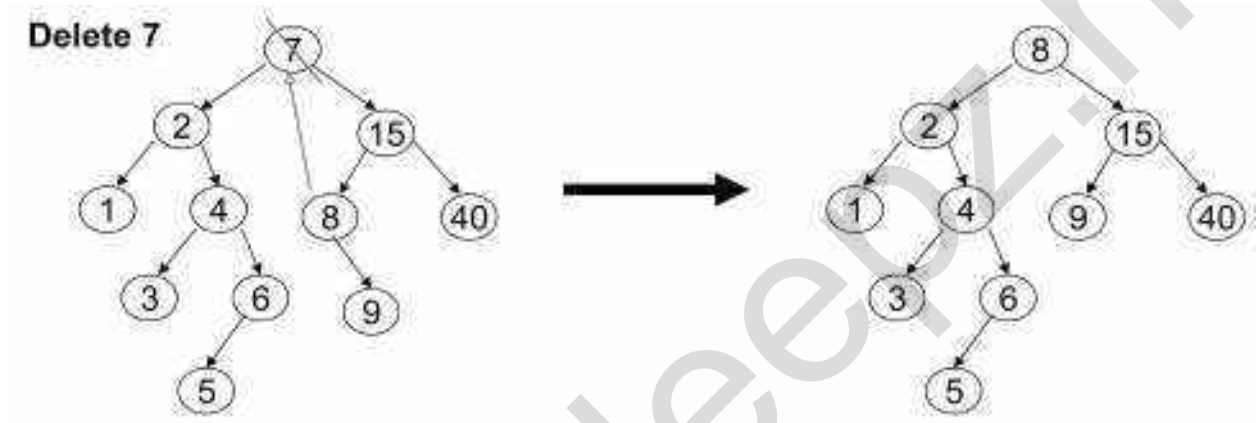
DELETION BY COPYING: METHOD#2
   Copy the **maximum** key in the **left** subtree of x to the node x, then delete
the one-child or leaf-node with this **maximum** key.
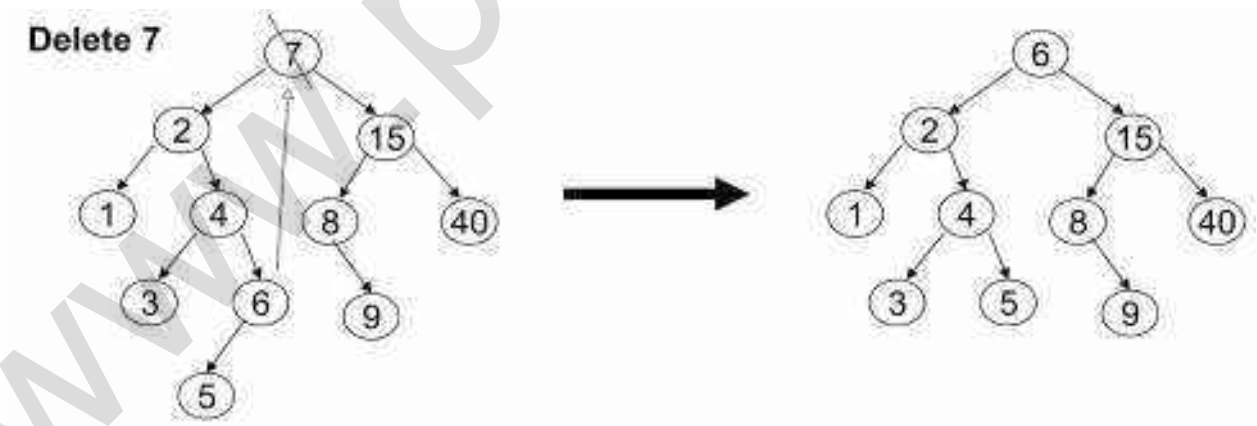
The code in performs deletion. It is inefficient, because it makes two passes
down the tree to find and delete the smallest node in the right subtree when

this is appropriate. It is easy to remove this inefficiency, by writing a special *delete_min* function, and we have left it in only for simplicity.

If the number of deletions is expected to be small, then a popular strategy to use is *lazy deletion:* When an element is to be deleted, it is left in the tree and merely *marked* as being deleted. This is especially popular if duplicate keys are present, because then the field that keeps count of the frequency of appearance can be decremented. If the number of real nodes in the tree is the same as the number of "deleted" nodes, then the depth of the tree is only expected to go up by a small constant (why?), so there is a very small time penalty associated with lazy deletion. Also, if a deleted key is reinserted, the overhead of allocating a new cell is avoided.

Deletion routine for binary search trees

```
Searchtree delete(elementtype X, searchtree T)
{
positiontmpcell;
if(T==NULL)
error("element not found");
else if(X<T-->element)
T-->left=delete(X,T-->left);
Else if(X>T-->element)
T-->right=delete(X,T-->right);
Else if(T-->left != NULL  && T-->right!=NULL)
{
/* Replace with smallest in right subtree*/
Tmpcell=findmin(T-->right);
T-->element=tmpcell-->element;
T-->right=delete(T-->element,T-->right);
}
Else
{
/* One or Zero children*/
tmpcell=T;
if(T-->left==NULL)
```

T=T-->right;
Else if(T-->right==NULL)
T=T-->left;
Free(tmpcell);
}
Return T;
}

COUNTING THE NUMBER OF NODES IN A BINARY SEARCH TREE

Introduction

To count the number of nodes in a given binary tree, the tree is required to be traversed recursively until a leaf node is encountered. When a leaf node is encountered, a count of 1 is returned to its previous activation (which is an activation for its parent), which takes the count returned from both the children's activation, adds 1 to it, and returns this value to the activation of its parent. This way, when the activation for the root of the tree returns, it returns the count of the total number of the nodes in the tree.

Program

A complete C program to count the number of nodes is as follows:

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
{
int data;
struct tnode *lchild, *rchild;
};
int count(struct tnode *p)
{
if( p == NULL)
return(0);
else
```

```
if( p->lchild == NULL && p->rchild == NULL)
return(1);
else
return(1 + (count(p->lchild) + count(p->rchild)));


}

struct tnode *insert(struct tnode *p,int val)
{
struct tnode *temp1,*temp2;
if(p == NULL)
{
p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as root
node*/
if(p == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
p->data = val;
p->lchild=p->rchild=NULL;
}
else
{
temp1 = p;
/* traverse the tree to get a pointer to that node whose child will be the newly
created node*/
while(temp1 != NULL)
{
temp2 = temp1;
if( temp1 ->data > val)
temp1 = temp1->lchild;
else
temp1 = temp1->rchild;
}
if( temp2->data > val)
{
temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode)); /
```

```
*inserts the newly created node
as left child*/
temp2 = temp2->lchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
else
{
temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the
newly created node
as left child*/
temp2 = temp2->rchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
}
return(p);
}
/* a function to binary tree in inorder */
void inorder(struct tnode *p)
{
if(p != NULL)
{
inorder(p->lchild);
printf("%d\t",p->data);
inorder(p->rchild);
}
}
```

```
void main()
{
struct tnode *root = NULL;
int n,x;
printf("Enter the number of nodes\n");
scanf("%d",&n);
while( n --- > 0)
{
printf("Enter the data value\n");
scanf("%d",&x);
root = insert(root,x);
}
inorder(root);
printf("\nThe number of nodes in tree are :%d\n",count(root));
}
```

Explanation

- Input:
  - o 1.The number of nodes that the tree to be created should have
    2. The data values of each node in the tree to be created
- Output:
  - o The data value of the nodes of the tree in inorder
    2. The count of number of node in a tree.

Example

- Input:
  - o 1.The number of nodes the created tree should have = 5
    2. The data values of nodes in the tree to be created are: 10, 20, 5, 9, 8
- Output: 1. 5 8 9 10 20
    2. The number of nodes in the tree is 5

 SWAPPING OF LEFT & RIGHT SUBTREES OF A GIVEN BINARY TREE

Introduction

An elegant method of swapping the left and right subtrees of a given binary tree makes use of a recursive algorithm, which recursively swaps the left and right subtrees, starting from the root.

Program

```c
#include <stdio.h>
#include <stdlib.h>
struct tnode
{
int data;
struct tnode *lchild, *rchild;
};

struct tnode *insert(struct tnode *p,int val)
{
struct tnode *temp1,*temp2;
if(p == NULL)
{
p = (struct tnode *) malloc(sizeof(struct tnode)); /* insert the new node as
root node*/
if(p == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
p->data = val;
p->lchild=p->rchild=NULL;
}
else
{
temp1 = p;
/* traverse the tree to get a pointer to that node whose child will be the newly
created node*/
while(temp1 != NULL)
{
temp2 = temp1;
if( temp1 ->data > val)
temp1 = temp1->lchild;
```

```
else
temp1 = temp1->rchild;
}
if( temp2->data > val)
{
temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the
newly created node
as left child*/
temp2 = temp2->lchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
else
{
temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/ *inserts the
newly created node
as left child*/
temp2 = temp2->rchild;
if(temp2 == NULL)
{
printf("Cannot allocate\n");
exit(0);
}
temp2->data = val;
temp2->lchild=temp2->rchild = NULL;
}
}
return(p);
}
/* a function to binary tree in inorder */
void inorder(struct tnode *p)
{
if(p != NULL)
```

```
{
inorder(p->lchild);
printf("%d\t",p->data);
inorder(p->rchild);
}
}
struct tnode *swaptree(struct tnode *p)
{
struct tnode *temp1=NULL, *temp2=NULL;
if( p != NULL)
{ temp1= swaptree(p->lchild);
temp2  =  swaptree(p->rchild);
p->rchild = temp1;
p->lchild = temp2;
}
return(p);
}
void main()
{
struct tnode *root = NULL;
int n,x;
printf("Enter the number of nodes\n");
scanf("%d",&n);
while( n - > 0)
{
printf("Enter the data value\n");
scanf("%d",&x);
root = insert(root,x);
}
printf("The created tree is :\n");
inorder(root);
printf("The tree after swapping is :\n");
root = swaptree(root);
inorder(root);
printf("\nThe original tree is \n");
root = swaptree(root);
inorder(root);
}
```

Explanation

- Input:
    - 1.The number of nodes that the tree to be created should have
      2. The data values of each node in the tree to be created
- Output:
    - 1.The data value of the nodes of the tree in inorder before interchanging the left and right subtrees
      2. The data value of the nodes of the tree in inorder after interchanging the left and right subtrees

Example

- Input:
    - 1.The number of nodes that the created tree should have = 5
      2. The data values of the nodes in tree to be created are: 10, 20, 5, 9, 8
- Output:
    - 1. 5 8 9 10 20
      2. 20 10 9 8 5

Applications of Binary Search Trees

One of the applications of a binary search tree is the implementation of a dynamic dictionary. This application is appropriate because a dictionary is an ordered list that is required to be searched frequently, and is also required to be updated (insertion and deletion mode) frequently. So it can be implemented by making the entries in a dictionary into the nodes of a binary search tree. A more efficient implementation of a dynamic dictionary involves considering a key to be a sequence of characters, and instead of searching by comparison of entire keys, we use these characters to determine a multi-way branch at each step. This will allow us to make a 26-way branch according to the first letter, followed by another branch according to the second letter and so on.

Applications of Trees

1. Compiler Design.
2. Unix / Linux.
3. Database Management.
4. Trees are very important data structures in computing.
5. They are suitable for:
   a. Hierarchical structure representation, e.g.,
      i. File directory.
      ii. Organizational structure of an institution.
      iii. Class inheritance tree.
   b. Problem representation, e.g.,
      i. Expression tree.
      ii. Decision tree.
   c. Efficient algorithmic solutions, e.g.,
      i. Search trees.
      ii. Efficient priority queues via heaps.

AVL TREE

The AVL tree is named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information."
Avl tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced.

The balance factor of a node is the height of its right subtree minus the height of its left subtree and a node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. This can be done by avl tree rotations

Need for AVL tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least Q(log N)
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

Thus we go for AVL tree.
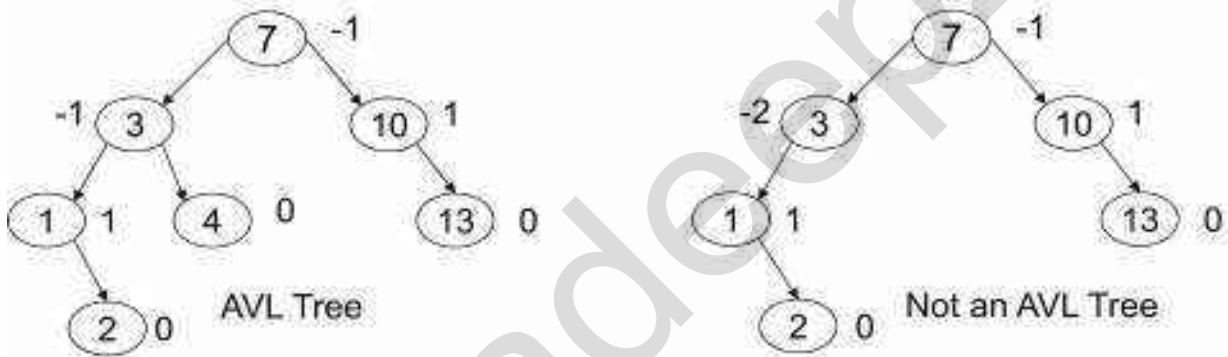
HEIGHTS OF AVL TREE

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" which is the difference between the heights of subtrees of a root in the tree. The "height" of tree is the "number of levels" in the tree. The height of a tree is defined as follows:

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.
4. The height of a leaf is 1. The height of a null pointer is zero.

The height of an internal node is the maximum height of its children plus 1.

FINDING THE HEIGHT OF AVL TREE

AVL trees are identical to standard binary search trees except that for every node in an AVL tree, the height of the left and right subtrees can differ by at most 1 . AVL trees are HB-k trees (height balanced trees of order k) of order HB-1. The following is the height differential formula:
|Height (Tl)-Height(Tr)|<=k

When storing an AVL tree, a field must be added to each node with one of three values: 1, 0, or -1. A value of 1 in this field means that the left subtree has a height one more than the right subtree. A value of -1 denotes the opposite. A value of 0 indicates that the heights of both subtrees are the same.
EXAMPLE FOR HEIGHT OF AVL TREE
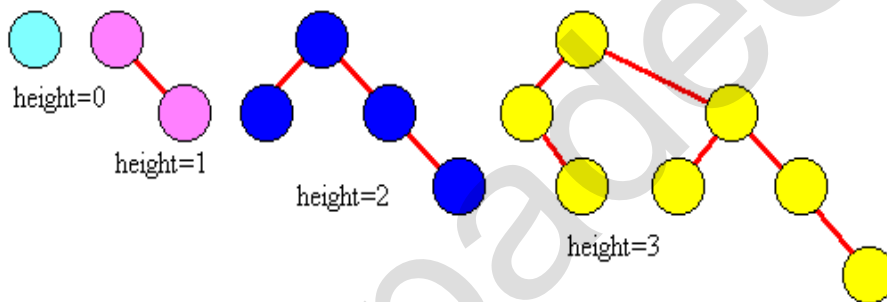
An AVL tree is a binary search tree with a balanced condition.
Balance Factor(BF) = Hl --- Hr. Hl
=> Height of the left subtree. Hr
=> Height of the right subtree.

 If BF={ --1,0,1} is satisfied, only then the tree is balanced.
AVL tree is a Height Balanced Tree.
If the calculated value of BF goes out of the range, then balancing has to be done.

Rotation :

Modification to the tree. i.e. , If the AVL tree is Imbalanced, proper rotations has to be done.

A rotation is a process of switching children and parents among two or three adjacent nodes to restore balance to a tree.

- There are two kinds of single rotation:

Right Rotation   Left Rotation

An insertion or deletion may cause an imbalance in an AVL tree.

The deepest node, which is an ancestor of a deleted or an inserted node, and whose balance factor has changed to -2 or +2 requires rotation to rebalance the tree.

Balance Factor :
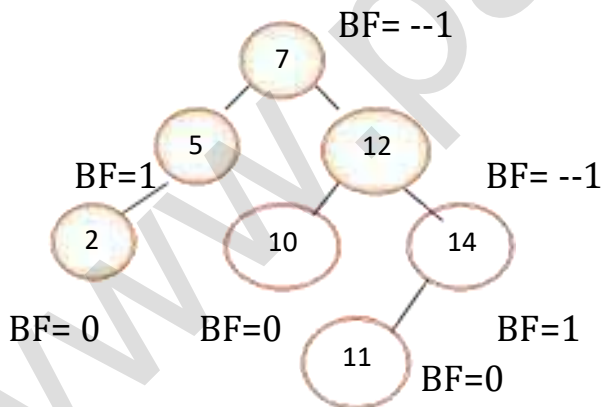
This Tree is an AVL Tree and a height balanced tree.

An AVL tree causes imbalance when any of following condition occurs:

   i. An insertion into Right child's right subtree.

  ii. An insertion into Left child's left subtree.

 iii. An insertion into Right child's left subtree.

iv. An insertion into Left child's right subtree.

These imbalances can be overcome by,

1. **Single Rotation** – (If insertion occurs on the outside,i.e.,LL or RR)

-> LL (Left -- Left rotation) --- Do single Right.
-> RR (Right -- Right rotation) – Do single Left.

2. **Double Rotation** - (If insertion occurs on the inside,i.e.,LR or RL)

-> RL (Right -- Left rotation) --- Do single Right, then single Left.
-> LR (Left -- Right rotation) --- Do single Left, then single Right.

General Representation of Single Rotation

1. LL Rotation :

- The right child y of a node x becomes x's parent.
- x becomes the left child of y.
- The left child T_2 of y, if any, becomes the right child of x.
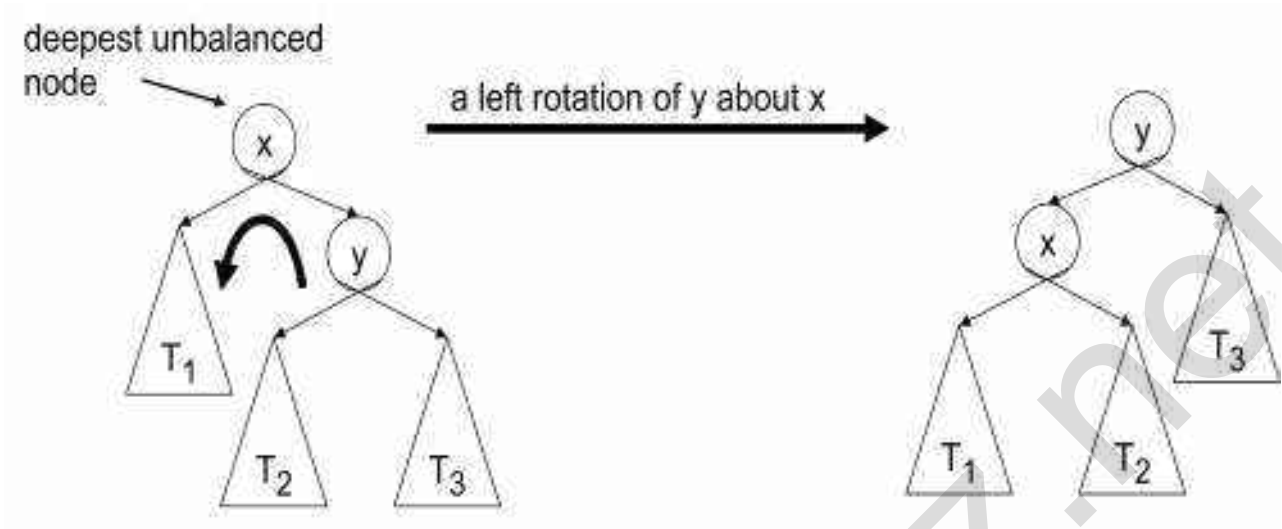
Note: The pivot of the rotation is the deepest unbalanced node

2. RR Rotation :

- The left child x of a node y becomes y's parent.
- y becomes the right child of x.
- The right child T_2 of x, if any, becomes the left child of y.
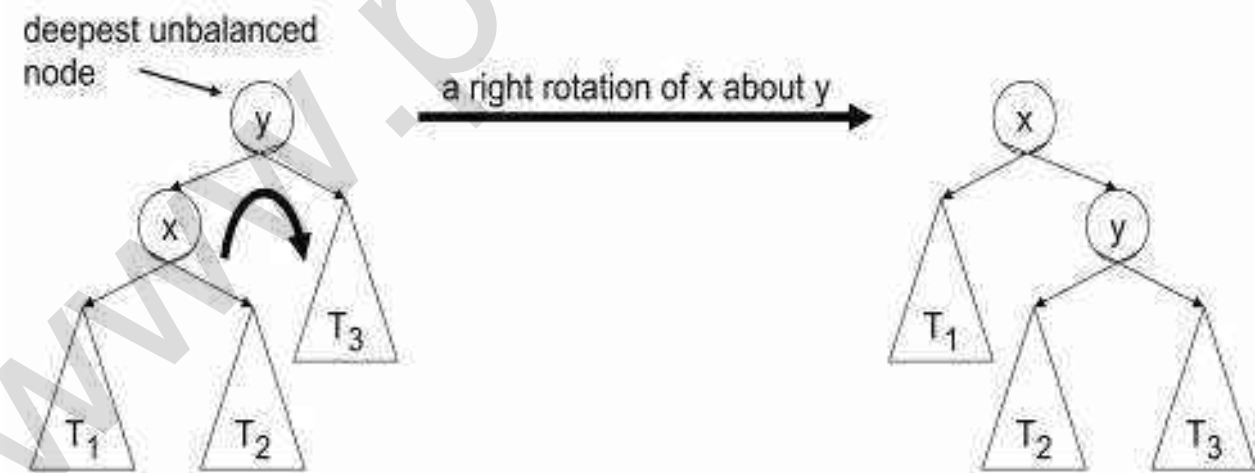
Note: The pivot of the rotation is the deepest unbalanced node

Example:

Consider the following tree which is balanced.
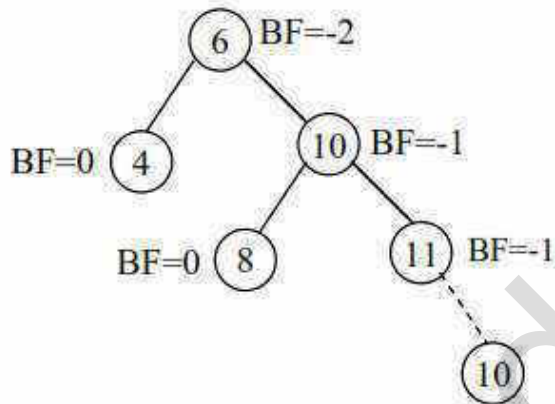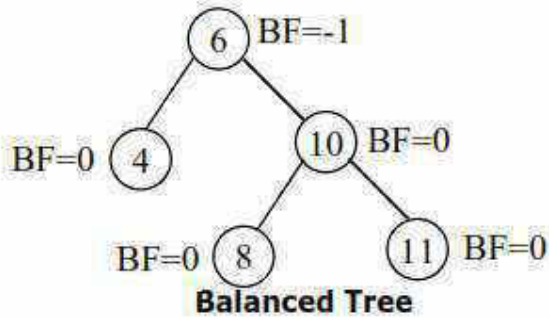
Balanced Tree

Now insert the value '13' it becomes unbalanced due to the insertion of new node in the Right Subtree of the Right Subtree. So we have to make single Right rotation in the root node.

After Right Rotation
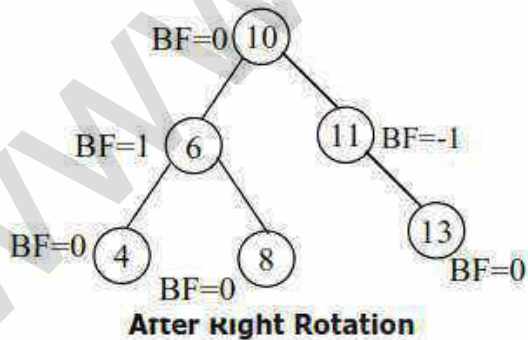
General Representation of Double Rotation

1. LR(Left -- Right rotation):

left rotation of w about v
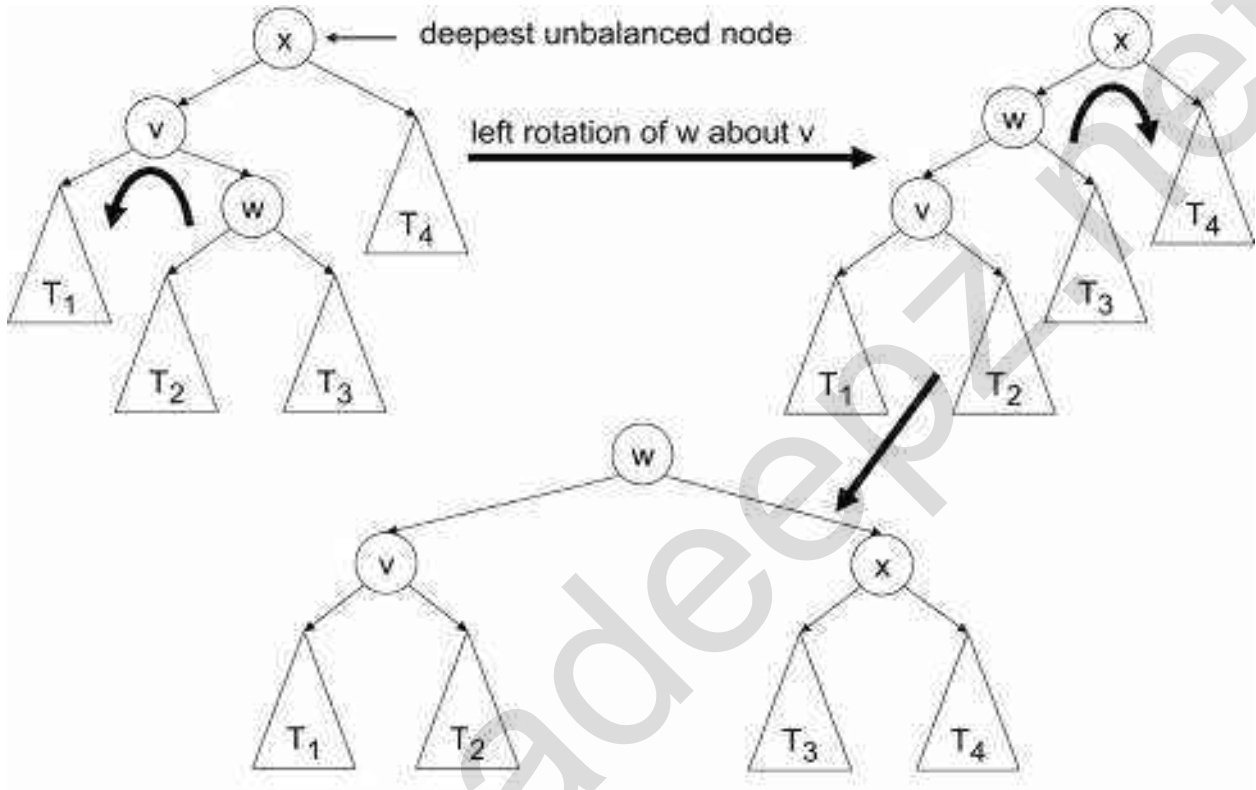
Note: First pivot is the left child of the deepest unbalanced node; second pivot is the deepest unbalanced node

2. RL(Right -- Left rotation) :

Note: First pivot is the right child of the deepest unbalanced node; second pivot is the deepest unbalanced node

deepest unbalanced node

right rotation of y about z

left rotation of Y about X
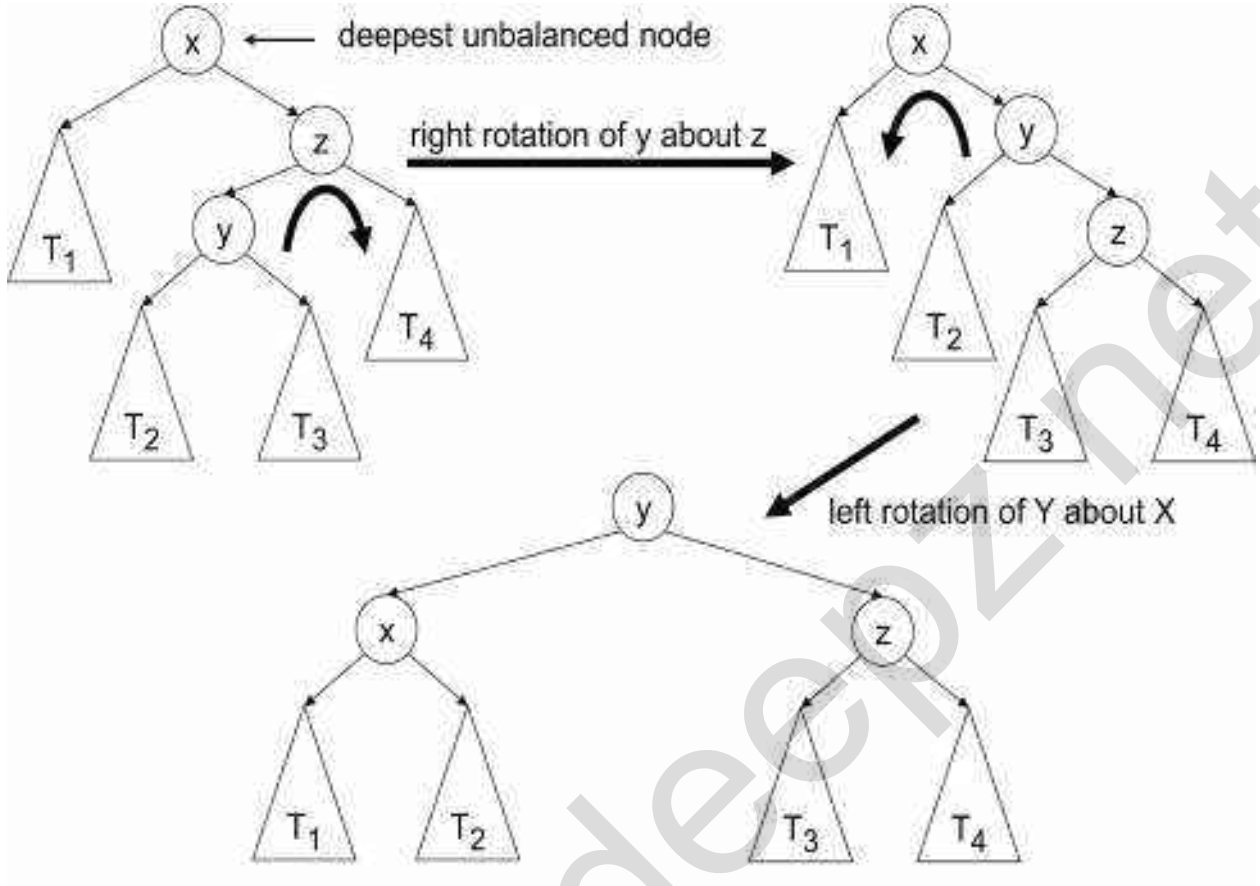
EXAMPLE:

LET US CONSIDER INSERTING OF NODES 20,10,40,50,90,30,60,70 in an AVL TREE

Step 1:(Insert the value 20)

(20) BF = 0

Step 2: (Insert the value 10)

(20) BF = 1

(10) BF = 0
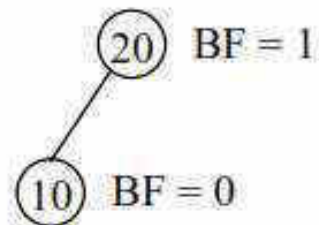
Step 3: (Insert the value 40)

BF = 1 at node 20

BF = 0 (10) (40) BF = 0
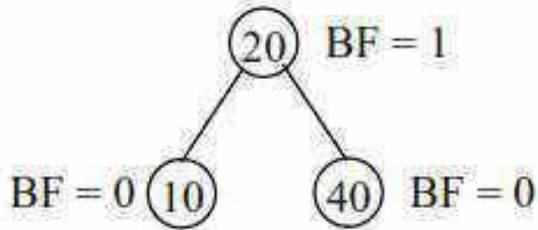
Step 4:(Insert the value 50)

BF = -1 at node 20

BF = 0 (10) (40) BF = -1

(50) BF = 0
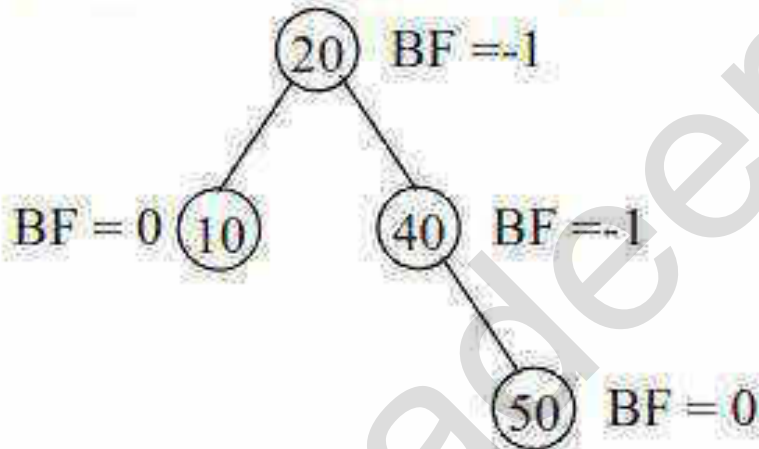
Step 5: (Insert the value 90)

BF = -2 at node 20

BF = 0 (10) (40) BF = -2

(50) BF = -1

(90) BF = 0
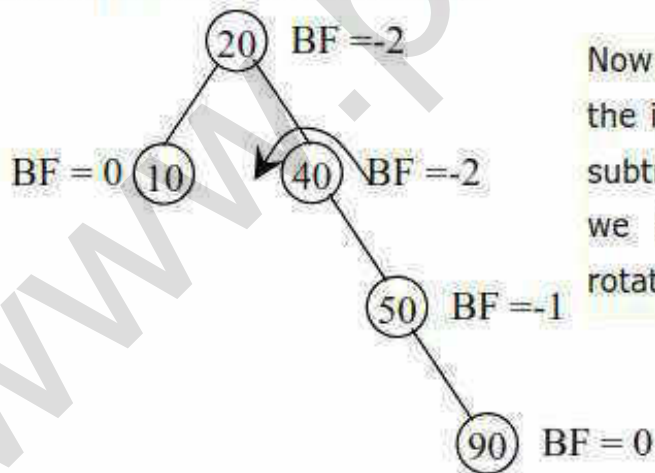
Now the tree is unbalanced due to the insertion of node in the Right subtree of the Right subtree. So we have to make single Right rotation in the node '40'.

After the Right Rotation

(20) BF = -1
BF = 0 (10)   (50) BF = 0
BF = 0 (40)   (90) BF = 0
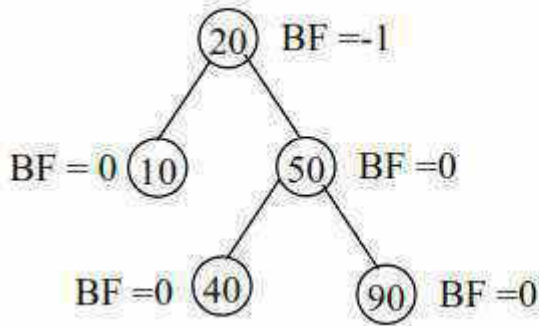
Now the tree is Balanced

Step 6: (Insert the value 30)

Now the tree is unbalance due to the insertion of value "30" in the left subtree of the right subtree. So we have to make Double rotation first with Left on the node "50" and then with Right on the node "20".

After Left Rotation:

(20) BF = -2
BF = 0 (10)   (40) BF = -1
BF = 0 (30)   (50) BF = -1
                    (90) BF = 0

After Right Rotation:

(40) BF = 0
BF = 0 (20)   (50) BF = -1
(10) (30)   (90)
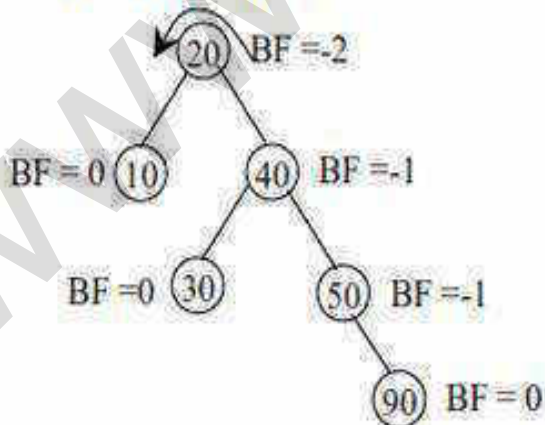
Now the tree is Balanced
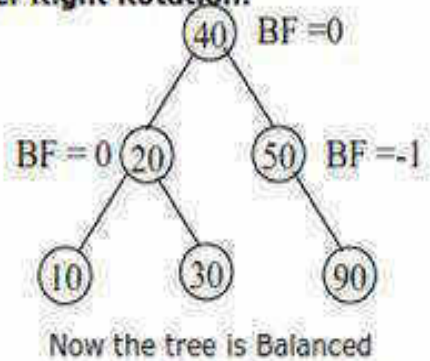
Step 7: (Insert the value 60)

40 BF =-1

BF = 0 20   50 BF =-2

10   30   90 BF =1

60 BF = 0

After Left Rotation:

40 BF =-1

BF = 0 20   50 BF =-2

10   30   60 BF =-1

90 BF = 0
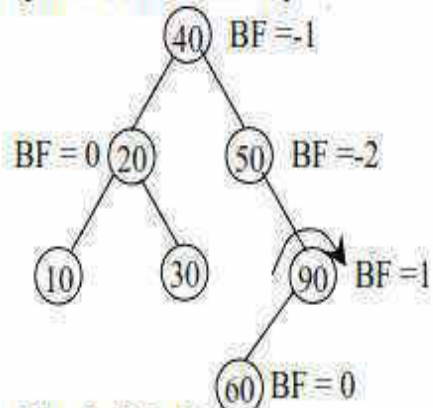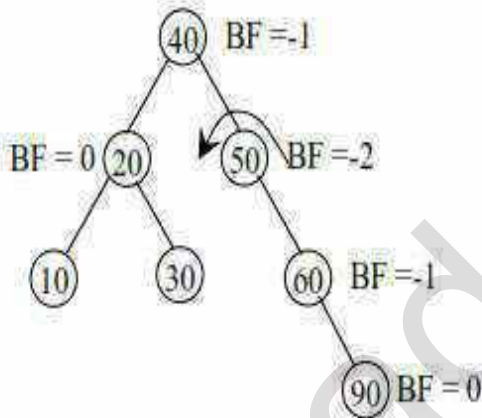
Now the tree at the node '50' is unbalanced due to the insertion of node '60' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '90' and then with Right on the node '50'.

After Right Rotation:

40 BF =0

BF = 0 20   60 BF =0

10   30   50   90
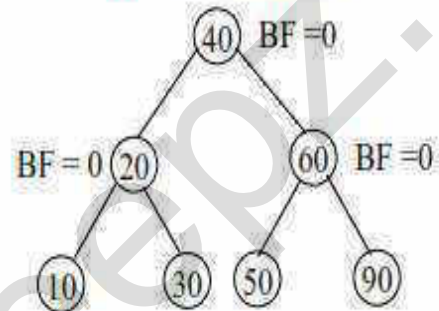
Now the tree is Balanced

Step 8: (Insert the value 70)

40 BF =0

BF = 0 20   60 BF =0

10   30 50   90

70
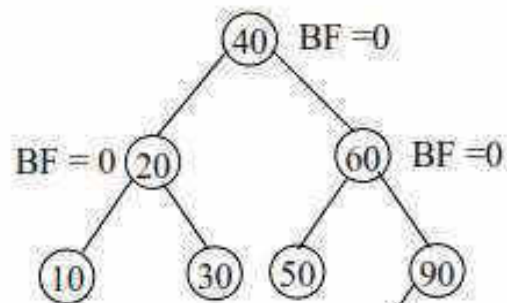
The tree is Balanced

AVL TREE ROUTINES

Creation of AVL Tree and Insertion

```
Struct avlnode
Typedef struct avlnode  *position;
Typedef structavlnode  *avltree;
Typedef int elementtype;
Struct avlnode
{
Elementtype element;
Avltree left;
Avltree right;
Int height;
};
Static int height(position P)
{
If(P==NULL)
return -1;
else
return P-->height;
}
Avltree insert(elementtype X, avltree T)
{
If(T==NULL)
{   / * Create and return a one node tree*/
T= malloc(sizeof(structavlnode));
If(T==NULL)
Fatalerror("Out of Space");
Else
{
T-->element=X;
```

```
T-->height=0;
T-->left=T-->right=NULL;
}
}
Else if(X<T-->element)
{
T-->left=Insert(X,T-->left);
If(height(T-->left) - height(T-->right)==2)
If(X<T-->left-->element)
T=singlerotatewithleft(T);
Else
T=doublerotatewithleft(T);
}
Else if(X>T-->element)
{
T-->right=insert(X,T-->right);
If(height(T-->left) - height(T-->right)==2)
If(X>T-->right-->element)
T= singlerotatewithright(T);
Else
T= doublerotatewithright(T);
}
T-->height=max(height(T-->left),height(T-->right)) + 1;
Return T;
}
```

Routine to perform Single Left :

. This function can be called only if k2 has a left child.
. Perform a rotate between a node k2 and its left child.
. Update height, then return the new root.

Static position singlerotatewithleft(position k2)

```
{
Position k1;
k1=k2-->left;
k2-->left=k1-->right;
k1-->right=k2;
k2-->height= max(height(k2-->left),height(k2-->right)) + 1;
k1-->height= max(height(k1-->left),height(k1-->right)) + 1;
return k1;    / * New Root * /
}
```

Routine to perform Single Right :

```
Static position singlerotationwithright(position k1)
{
position k2;
k2=k1-->right;
k1-->right=k2-->left;
k2-->left=k1;
k2-->height=max(height(k2-->left),height(k2-->right)) + 1;
k1-->height=max(height(k1-->left),height(k1-->right)) + 1;
return k1; / * New Root * /
}
```

Double rotation with Left :

```
Static position  doublerotationwithleft(position k3)
{
/ * Rotate between k1 & k2 * /
k3-->left=singlerotatewithright(k3-->left);
/ * Rotate between k3 & k2 * /
returnsinglerotatewithleft(k3);
}
```

Double rotation with Right :

```
Static position  doublerotatewithright(position k1)
{
/ * Rotation between k2& k3 * /
k1-->right=singlerotatewithleft(k1-->right);
/ * Rotation between k1 &k2 * /
returnsinglerotatewithright(k1);
}
```

PROBLEMS

APPLICATIONS

AVL trees play an important role in most computer related applications. The need and use of avl trees are increasing day by day. their efficiency and less complexity add value to their reputation. Some of the applications are

   - Contour extraction algorithm
   - Parallel dictionaries
   - Compression of computer files
   - Translation from source language to target language
   - Spell checker

 ADVANTAGES OF AVL TREE

   - AVL trees guarantee that the difference in height of any two subtrees rooted at the same node will be at most one. This guarantees an asymptotic running time of O(log(n)) as opposed to O(n) in the case of a standard bst.
   - Height of an AVL tree with n nodes is always very close to the theoretical minimum.

   - Since the avl tree is height balabced the operation like insertion and deletion have low time complexity.

- Since tree is always height balanced.Recursive implementation is possible.
- The height of left and the right sub-trees should differ by atmost 1.Rotations are possible.

DISADVANTAGES OF AVL TREE

- one limitation is that the tree might be spread across memory
- as you need to travel down the tree, you take a performance hit at every level down
- one solution: store more information on the path
- Difficult to program & debug ; more space for balance factor.
- asymptotically faster but rebalancing costs time.
- most larger searches are done in database systems on disk and use other structures

BINARY HEAPS

A **heap** is a specialized complete tree structure that satisfies the *heap property:*

- it is empty *or*
- the key in the root is larger than that in either child and both subtrees have the heap property.

- In general heap is a group of things placed or thrown, one on top of the other.
- In data structures a heap is a binary tree storing keys at its nodes.
- Heaps are based on the concepts of a complete tree

Structure Property :

COMPLETE TREE

A binary tree is completely full if it is of height, *h*, and has 2*h*+1-1 nodes.

- it is empty *or*
- its left subtree is complete of height *h*-1 and its right subtree is completely full of height *h*-2 *or*

- its left subtree is completely full of height h-1 and its right subtree is complete of height h-1.
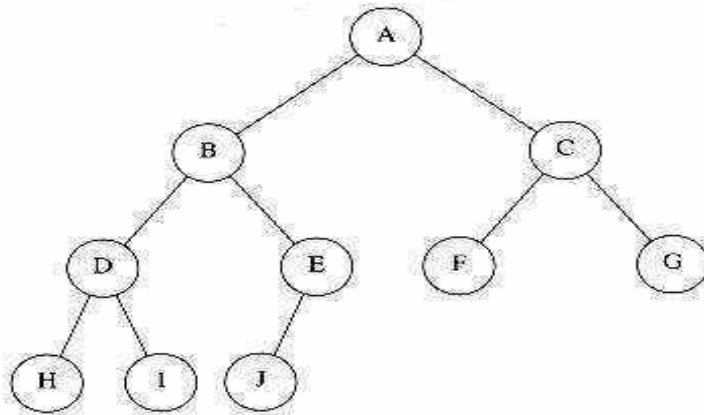
A complete binary tree
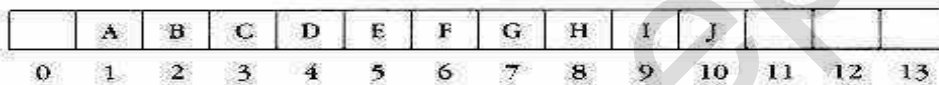
A complete tree is filled from the left:

- all the leaves are on
  - the same level *or*
  - two adjacent ones *and*
- all nodes at the lowest level are as far to the left as possible.

PROCEDURE

INSERTION:

Let us consider the element X is to be inserted.

- First the element X is added as the last node.
- It is verified with its parent and adjacent node for its heap property.
- The verification process is carried upwards until the heap property is satisfied.
- If any verification is not satisfied then swapping takes place.
- Then finally we have the heap.

DELETION:

- The deletion takes place by removing the root node.
- The root node is then replaced by the last leaf node in the tree to obtain the complete binary tree.
- It is verified with its children and adjacent node for its heap property.
- The verification process is carried downwards until the heap property is satisfied.
- If any verification is not satisfied then swapping takes place.
- Then finally we have the heap.

PRIORITY QUEUE

It is a data structure which determines the priority of jobs.
The Minimum the value of Priority, Higher is the priority of the job.
The best way to implement Priority Queue is **Binary Heap**.
A Priority Queue is a special kind of queue datastructure. It has zero or more collection of elements, each element has a priority value.

- Priority queues are often used in resource management, simulations, and in the implementation of some algorithms (e.g., some graph algorithms, some backtracking algorithms).
- Several data structures can be used to implement priority queues. Below is a comparison of some:

Basic Model of a Priority Queue

Deletion(h) ←——— PRIORITY QUEUE ———→ Insertion(h)

Implementation of Priority Queue
1. Linked List.
2. Binary Search Tree.
3. Binary Heap.

Linked List :

A simple linked list implementation of priority queue requires o(1) time to perform the insertion at the front and o(n) to delete at minimum element.

Binary Search tree :

This gives an average running time of o(**log** n) for both insertion and deletion.(deletemin).

The efficient way of implementing priority queue is Binary Heap (or) Heap.

Heap has two properties :
1. Structure Property.
2. Heap Order Preoperty.

1.Structure Property :

The Heap should be a complete binary tree, which is a completely filled tree, which is a completely filled binary tree with the possible exception of the bottom level, which is filled from left to right.
A Complete Binary tree of height H, has between 2^h and $(2^{h+1} - 1)$ nodes.

Sentinel Value :

The zeroth element is called the sentinel value. It is not a node of the tree. This value is required because while addition of new node, certain operations are performed in a loop and to terminate the loop, sentinel value is used.
Index 0 is the sentinel value. It stores irrelated value, inorder to terminate the program in case of complex codings.
Structure Property : Always index 1 should be starting position.

2.Heap Order Property :

The property that allows operations to be performed quickly is a heap order property.

Mintree:

Parent should have lesser value than children.

Maxtree:

Parent should have greater value than children.

These two properties are known as heap properties

- Max-heap
- Min-heap

Min-heap:

The smallest element is always in the root node.Each node must have a key that is less or equal to the key of each of its children.
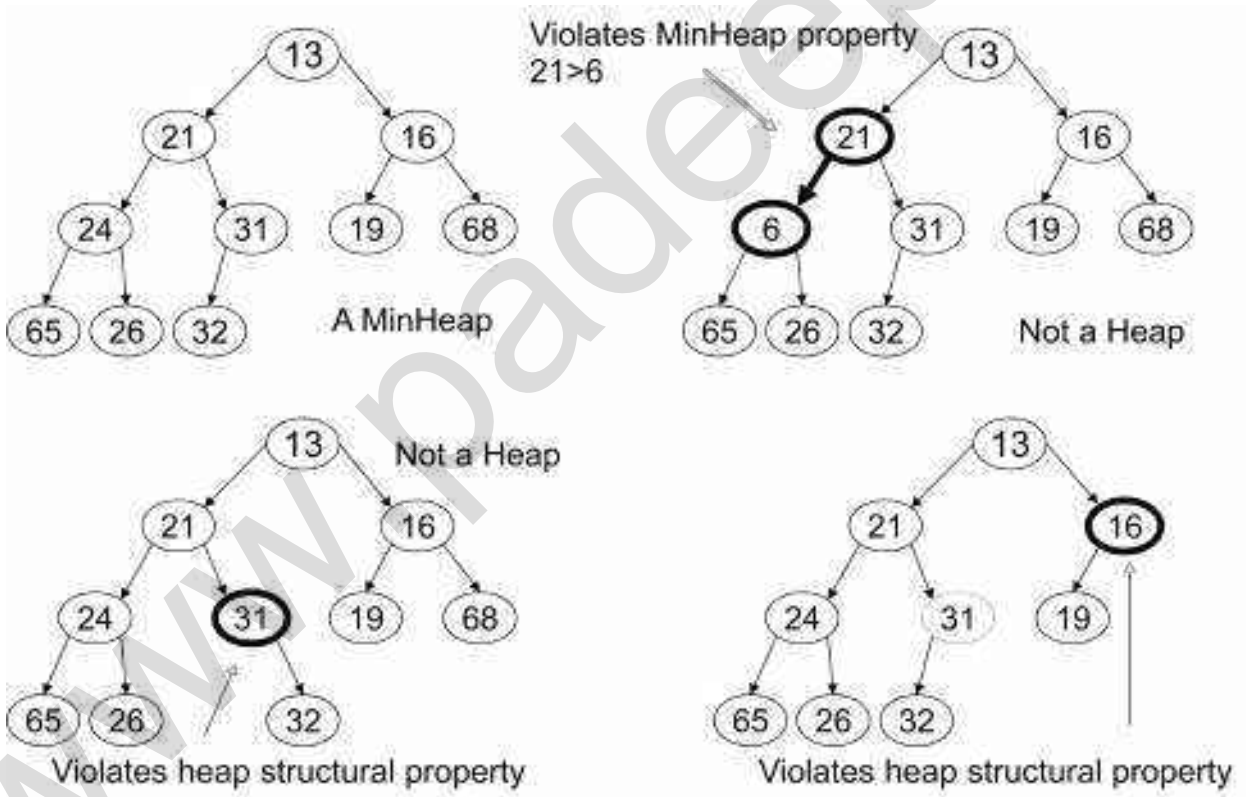
Examples

Max-Heap:

The largest Element is always in the root node.

Each node must have a key that is greater or equal to the key of each of its children.

Examples
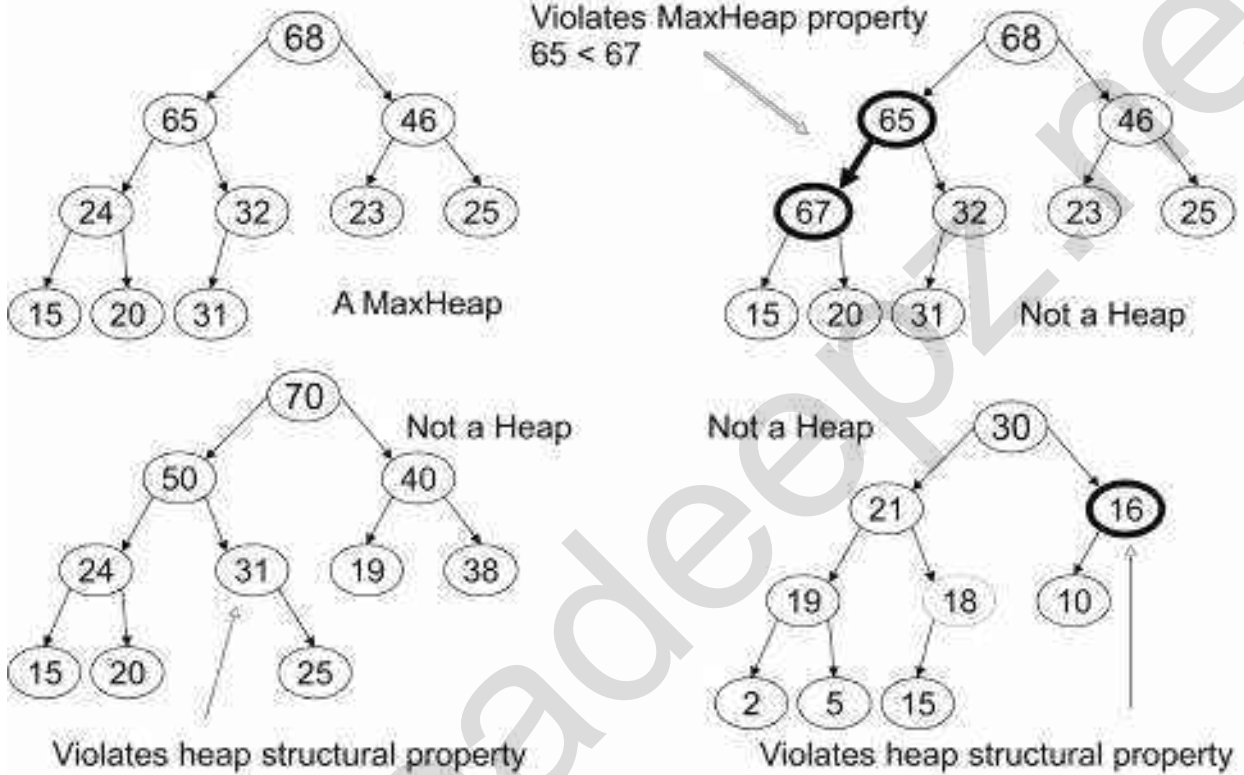
HEAP OPERATIONS:

There are 2 operations of heap

· Insertion
· Deletion

2.12.1 Insert:
Adding a new key to the heap

Rules for the insertion:

To insert an element X, into the heap, do the following:

Step1: Create a hole in the next available location , since otherwise the tree will not be complete.

Step2: If X can be placed in the hole, without violating heap order, then do insertion, otherwise slide the element that is in the hole's parent node, into the hole, thus, bubbling the hole up towards the root.

Step3: Continue this process until X can be placed in the hole.
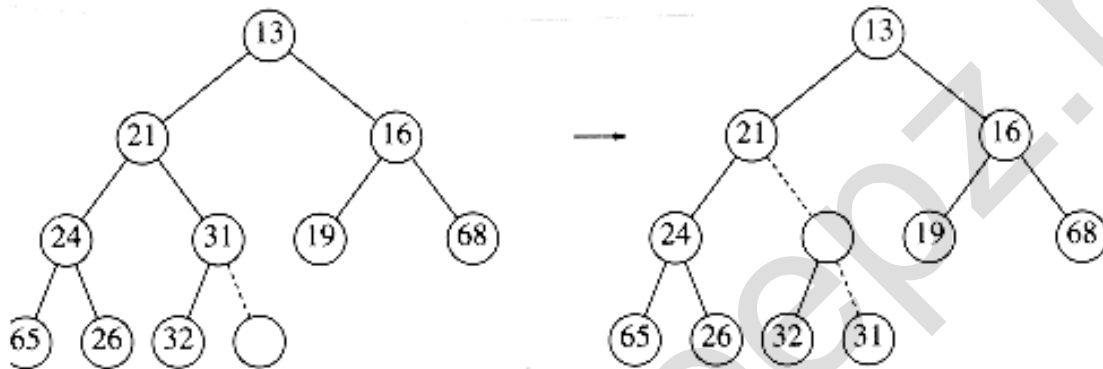
Attempt to insert 14: creating the hole, and bubbling the hole up
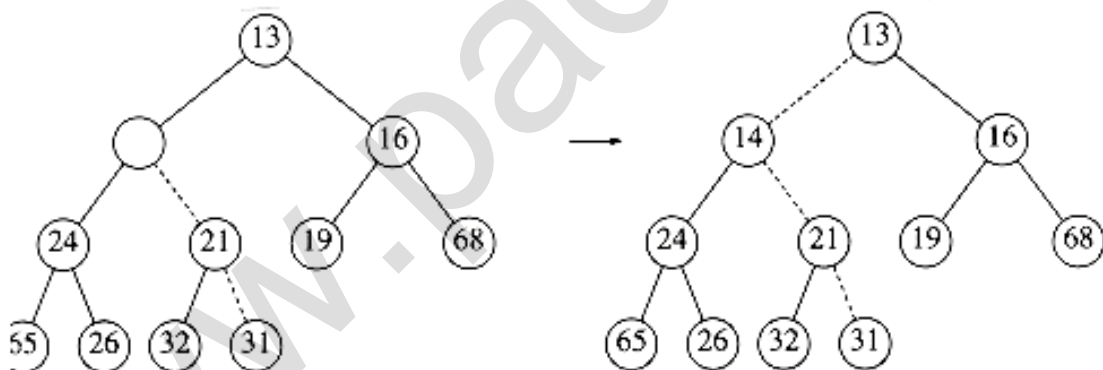
The remaining two steps to insert 14 in previous heap

Example Problem :

1.Insert- 18 in a Min Heap
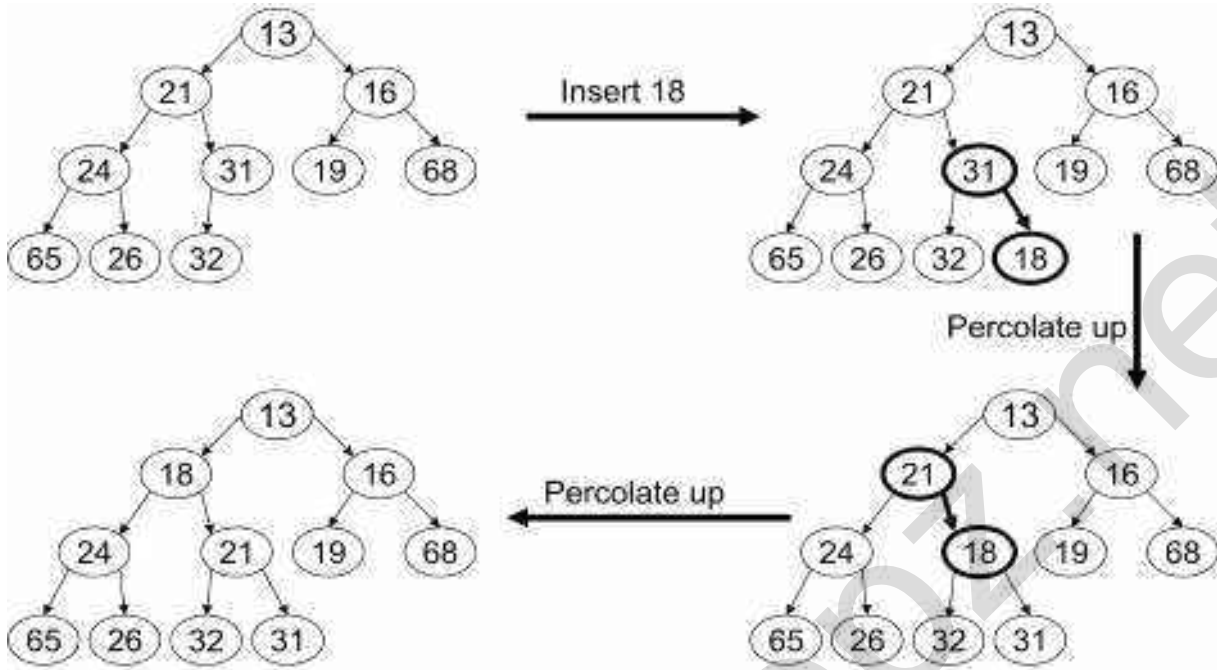
2. Insert the keys 4, 6, 10, 20, and 8 in this order in an originally empty max-heap
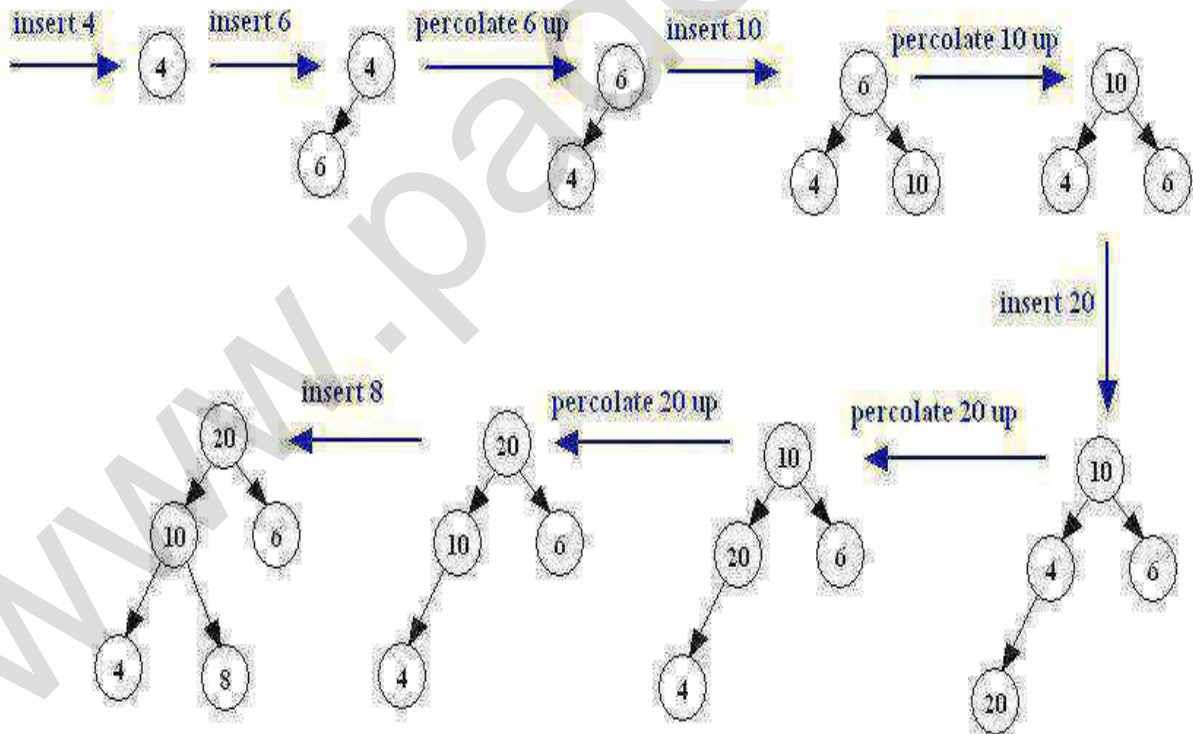
2.12.2 Delete-max or Delete-min:

Removing the root node of a max- or min-heap, respectively

Procedure for Deletemin :
* Deletemin operation is deleting the minimum element from the loop.
* In Binary heap | min heap the minimum element is found in the root.
* When this minimum element is removed, a hole is created at the root.
* Since the heap becomes one smaller, make the last element X in the heap to move somewhere in the heap.
* If X can be placed in the hole, without violating heap order property, place it , otherwise slide the smaller of the hole's children into the hole, thus , pushing the hole down one level.
* Repeat this process until X can be placed in the hole.
This general strategy is known as Percolate Down.
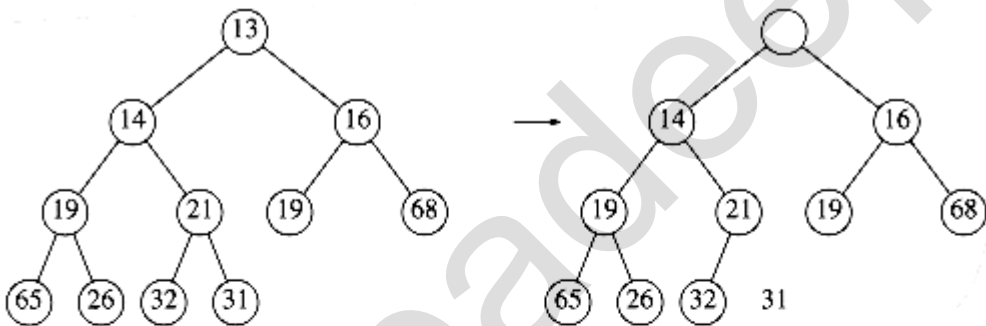
Creation of the hole at the root
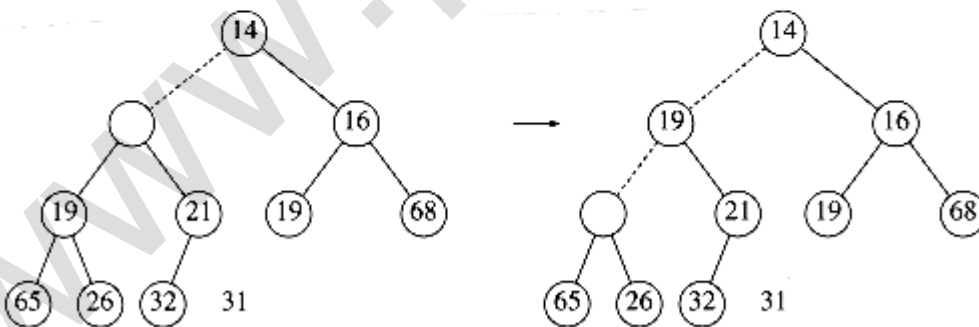
Next two steps in delete_min
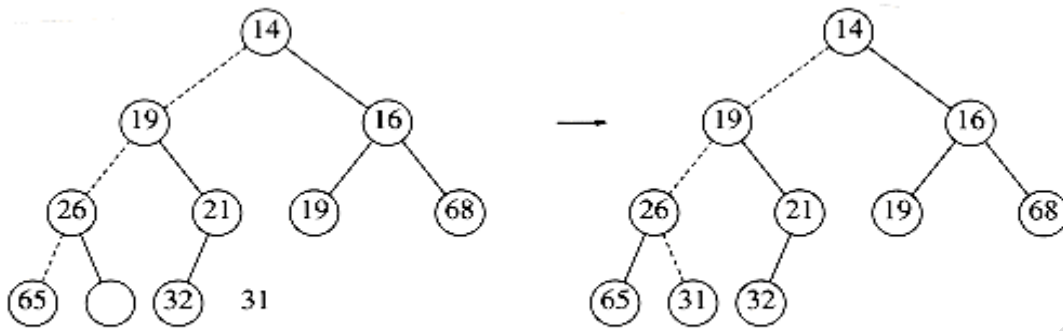
Last two steps in delete_min

EXAMPLE PROBLEMS :

1.DELETE MIN
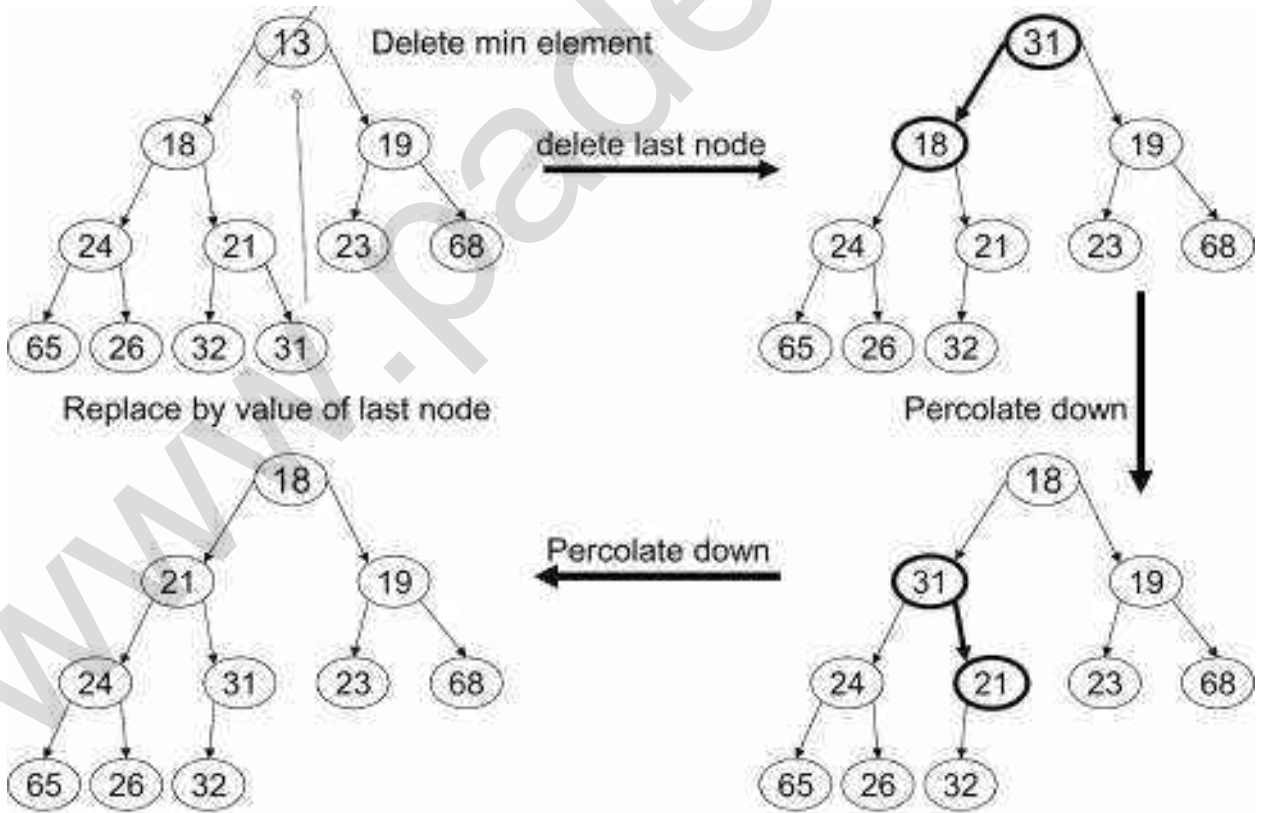
2. Delete Min -- 13
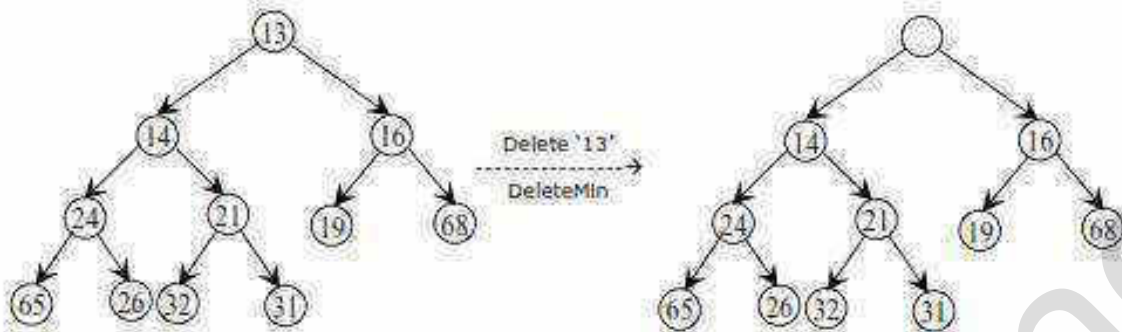
- ❖ The last value cannot be placed in the hole, because this would violate heap order.
- ❖ Therefore, we place the smaller child '14' in the hole, sliding the hole down one level.
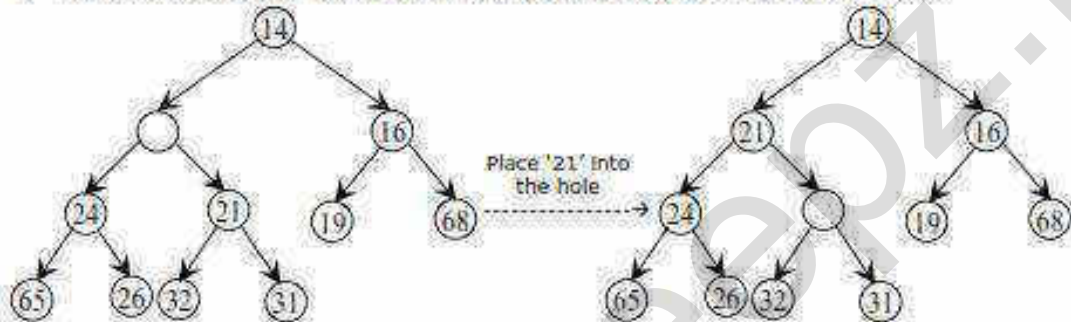
Now slide the hole down by inserting the value '31' in to the hole.

BINARY HEAP ROUTINES [Priority Queue]

Typedef struct heapstruct *priorityqueue;
Typedef int elementype;
Struct heapstruct
{
int capacity;
int size;

```
elementtype *element;
};
```

Declaration of Priority Queue
```
Priorityqueue initialize(int maxelement)
{
Priorityqueue H;
If(minpsize<maxelements)
Error("Priority queue size is too small");
H=malloc(sizeof(struct heapstruct));
If(H=NULL)
Fatalerror("Out of space");
/ * Allocate the array plus one extra for sentinel * /
H-->elements=malloc((maxelements+1)*sizeof(elementtype));
If(H-->elements==NULL)
Fatalerror("out of space");
H-->capacity=maxelements;
H-->size=0;
H-->elements[0]=mindata;
Return H;
}
/ * H-->elements[0]=sentinelvalue * /
```

Insert Routine
```
Void insert(elementtype X, priorityqueue H)
{
int i;
if(isfull(H))
{
Error("Priority queue is full");
Return;
}
```

```
For(i=++H-->size;H-->elements[i/2]>X;i=i/2)
H-->elements[i]=H-->elements[i/2];
H-->elements[i]=X;
}
```

Delete Routine

```
Elementtype deletemin(priorityqueue H)
{
int i,child;
elementtype minelement,lastelement;
if(isempty(H))
{
Error("Priority queue is empty");
Return H-->element[0];
}
Minelement=H-->element[1];
Lastelement=H-->element[H-->size--];
For(i=1;i*2<=H-->size;i=child)
{
/ *Find smaller child */
Child=i*2;
If(child!=H-->size && H-->elements[child++]<H-->elements[child])
{
Child++;
}
/ * Percolate one level * /
If(lastelement>H-->elements[child])
H-->element[i]=H-->elements[child];
Else
Break;
}
H-->element[i]=lastelement;
```

Return minelement;

}

Other Heap Operations

1. Decrease Key.
2. Increase Key.
3. Delete.
4. Build Heap.

1.Decrease Key :
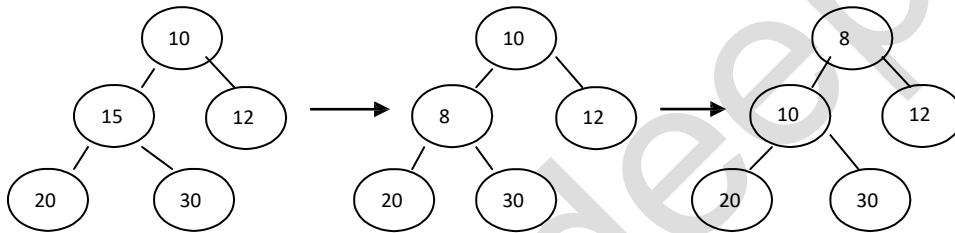
       The Decrease key(P,Δ,H) operation decreases the value of the key at position P, by a positive amount Δ. This may violate the heap order property, which can be fixed by percolate up Ex : decreasekey(2,7,H)

2.Increase Key :

       The Increase Key(P,Δ,H) operation increases the value of the key at position P, by a positive amount Δ. This may violate heap order property, which can be fixed by percolate down.
Ex : increase key(2,7,H)
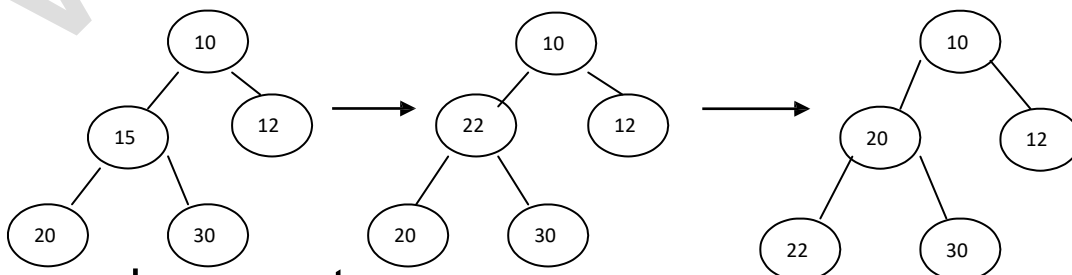
3. Delete :

The delete(P,H) operation removes the node at the position P, from the heap H. This can be done by,

Step 1: Perform the decrease key operation, decrease key(P,∞,H).
Step 2: Perform deletemin(H) operation.

Step 1:   Decreasekey(2, ∞,H)
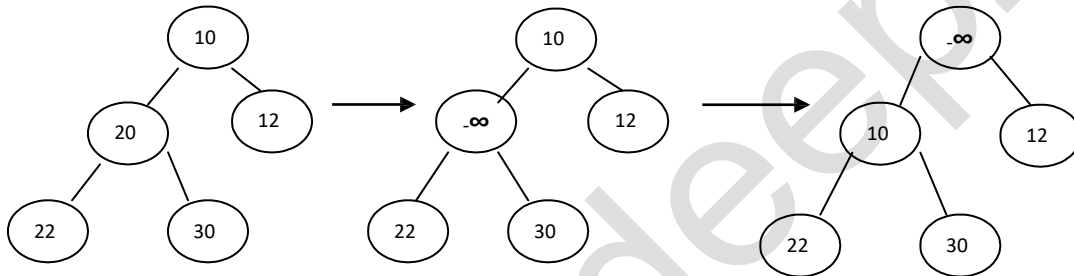
Step 2 : Deletemin(H)
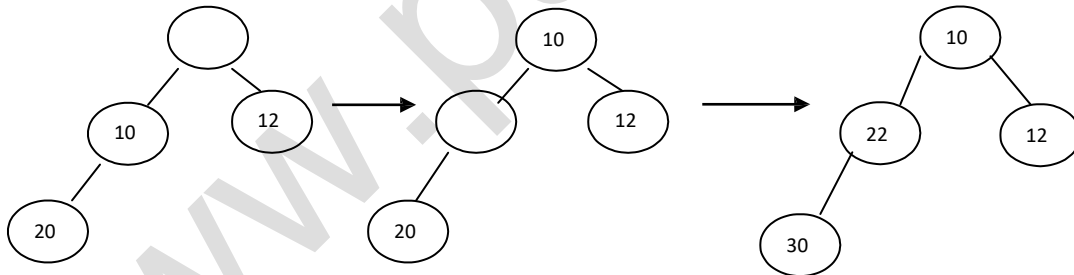
APPLICATIONS

The heap data structure has many applications

- Heap sort
- Selection algorithms

- Graph algorithms

Heap sort :
One of the best sorting methods being in-place and with no quadratic worst-case scenarios.

Selection algorithms:
Finding the min, max, both the min and max, median, or even the k-th largest element can be done in linear time using heaps.

Graph algorithms:
By using heaps as internal traversal data structures, run time will be reduced by an order of polynomial. Examples of such problems are Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

ADVANTAGE

The biggest advantage of heaps over trees in some applications is that construction of heaps can be done in linear time.

- It is used in
  o Heap sort
  o Selection algorithms
  o Graph algorithms

DISADVANTAGE

Heap is expensive in terms of

- safety
- maintenance
- performance

Performance :
Allocating heap memory usually involves a long negotiation with the OS.

Maintenance:

   Dynamic allocation may fail; extra code to handle such exception is required.

Safety :

       Object may be deleted more than once or not deleted at all .

B-TREES

Multi-way Tree

A multi-way (or m-way) search tree of order m is a tree in which

- Each node has at-most m subtrees, where the subtrees <u>may be empty</u>.
- Each node consists of at least 1 and at most m-1 distinct keys
- The keys in each node are sorted.
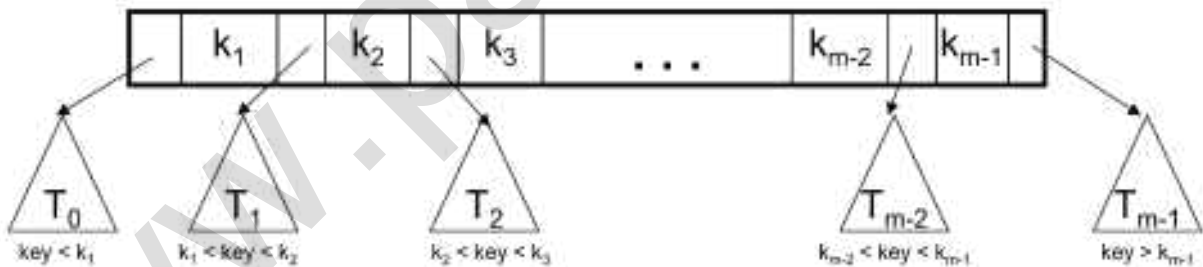
- The keys and subtrees of a non-leaf node are ordered as:
    - $T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-1}, T_{m-1}$ such that:
- All keys in subtree T0 are less than k1.
- All keys in subtree T_i , $1 <= i <= m - 2$, are greater than k_i but less than k_{i+1}.
- All keys in subtree T_{m-1} are greater than k_{m-1}
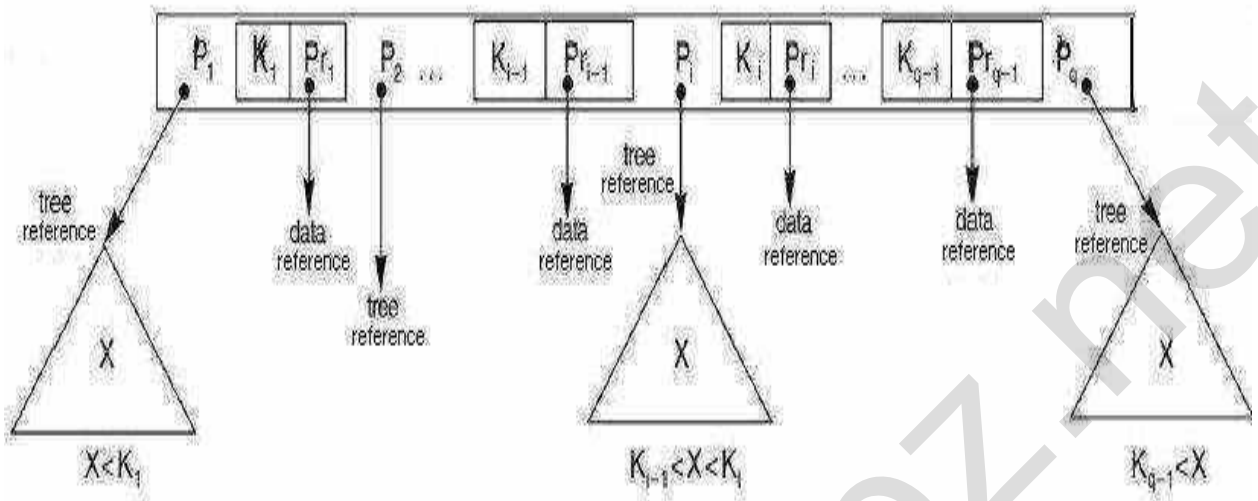
A **B-tree** of order m (or branching factor m), where m > 2, is either an empty tree or a multiway search tree with the following properties:

- The root is either a leaf or it has at least two non-empty subtrees and at most m non-empty subtrees.
- Each non-leaf node, other than the root, has at least ⌈m/2⌉ non-empty subtrees and at most m non-empty subtrees. (Note: ⌈x⌉ is the lowest integer > x).
- The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
- All leaf nodes are at the same level; that is the tree is perfectly balanced.

B-Tree Examples

Example: A B-tree of order 4
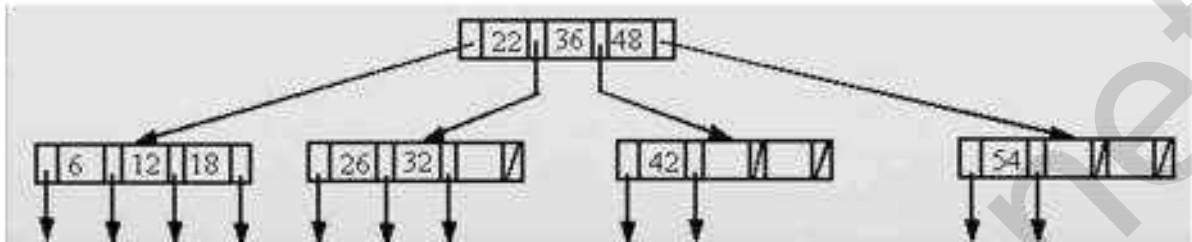
Example: A B-tree of order 5
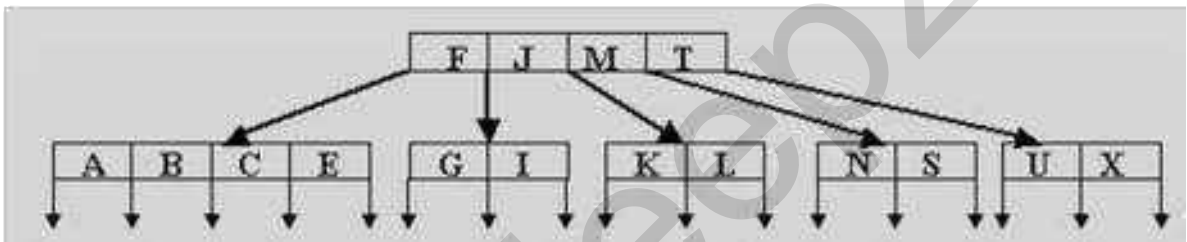
Insertion in B-Trees

OVERFLOW CONDITION:

   A root-node or a non-root node of a B-tree of order m overflows if, after a key insertion, it contains m keys.

Insertion algorithm:

   If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

- Note: Insertion of a key always <u>starts</u> at a leaf node.

Insertion in a B-tree of <u>odd</u> order

Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3
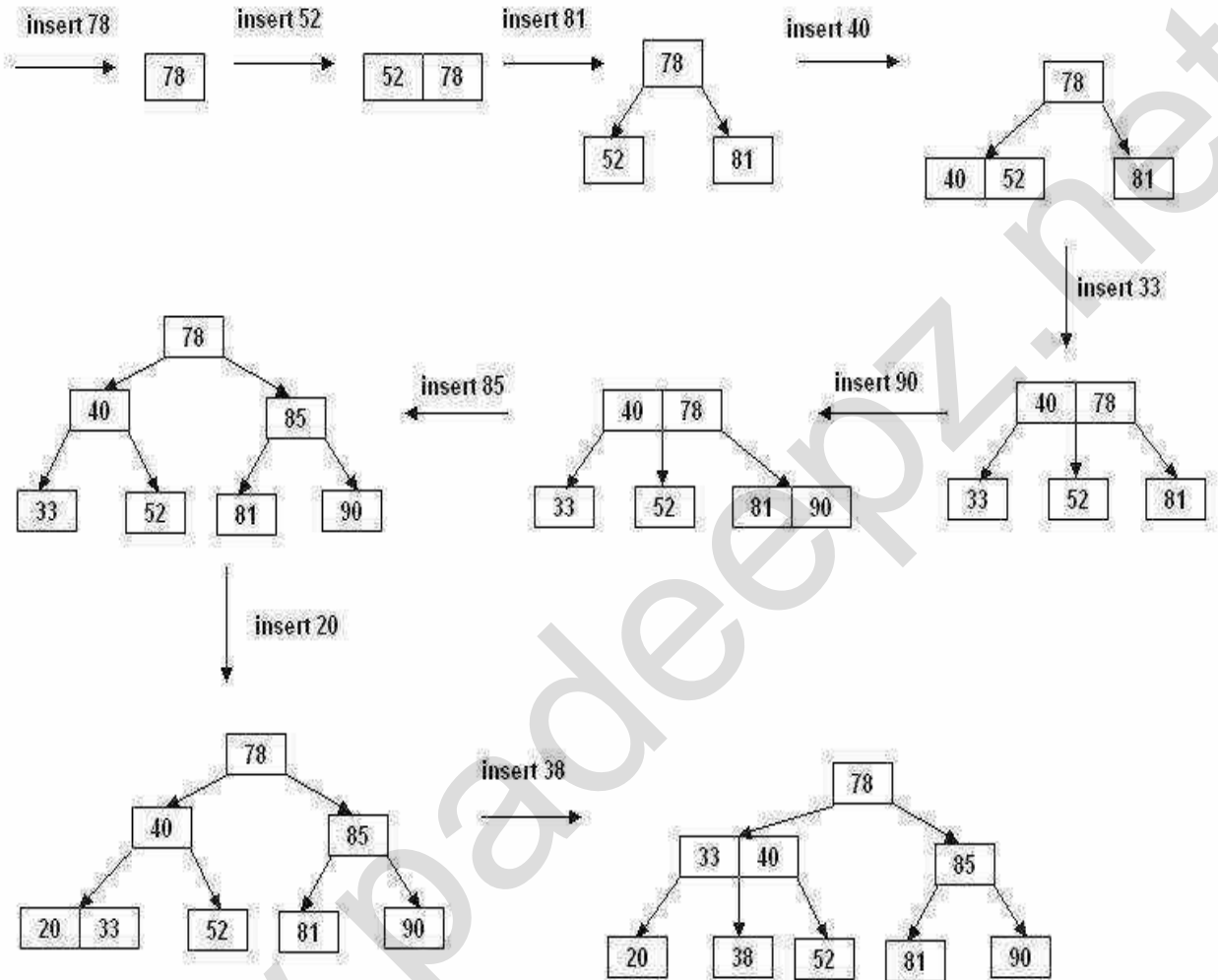
Insertion in a B-tree of <u>even</u> order

At each node the insertion can be done in two different ways:
- **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- **left-bias:** The node is split such that its left subtree has more keys than the right subtree.

Example: Insert the key **5** in the following B-tree of order **4**:

1.	Construct a binary tree whose preorder traversal is K L N M P R Q S T and in order traversal is N L K P R M S Q T

2.	(i) Define the height of a binary tree or subtree and also define a height balanced (AVL) tree.

The height of a tree or a sub tree is defined as the length of the longest path from the root node to the leaf.

A tree is said to be height balanced if all the nodes are having a balance factor -1,0 or 1. The balance factor is the height of the left sub tree minus height of the right sub tree

(ii) Mark the balance factor of each mode on the tree given in fig 7.1 and state whether it is height-balanced

(iii) Into the same tree given in 7(ii) above, insert the integer 13 and show the new balance factors that would arise if the tree is no rebalanced. Finally, carry the required rebalancing of the tree and show the new tree with the balance factors on each mode.

3. The maximum number of nodes in a binary tree of level k, k ≥ 1 is
       (A) $2^k + 1$
       (B) $2^k - 1$
       (C) 2^{k-1}
       (D) $2^{k-1} - 1$

[B]
Here k starts from 1

For the root node k=1, no. of nodes in root node is 1

For k=2, no. of nodes in the binary tree is 3,

For k=3, no. of nodes is 7

For k=4, no of nodes is 15.

………

………

So for any value k it is $2^k - 1$

4. Consider the height balanced tree T_1, with values stored at only the leaf nodes, shown in Fig. 4
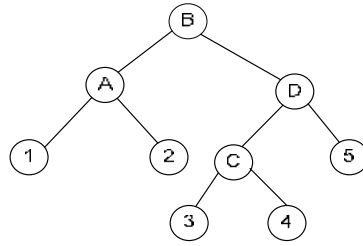
Fig 4

(i) Show how to merge to the tree T_1 elements from tree T_2 shown in Fig. 5 using node D of tree T_1
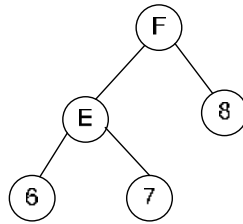
Fig 5

(ii) What is the time complexity of a merge operation on balanced trees T_1 and T_2 where T_1 and T_2 are of height h_1 and h_2 respectively, assuming rotation scheme are given. Give reasons

5. The number of different ordered trees with 3 nodes labeled Y, Y, Z are

   (A)   16
   (B)   8
   (C)   12
   (D)   24

Ans: 15

There are 5 tree unordered tree structures possible.

(a)   (b)   (c)   (d)   (e)
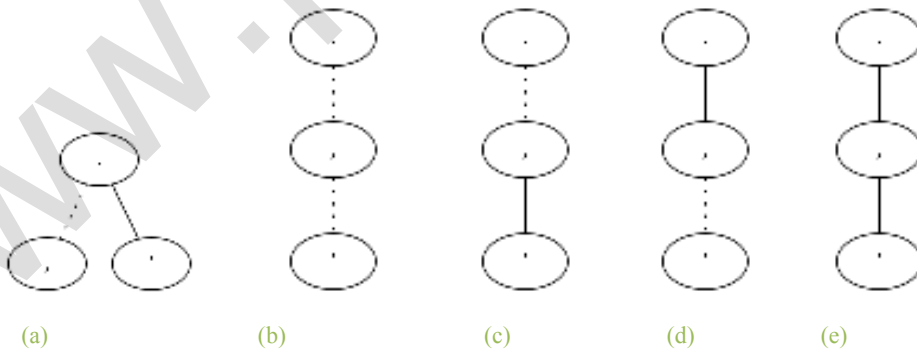
However, we want to order the nodes based on Y, Y and Z, there are 3 possibilities associated with every tree structure.

Hence, 5 * 3 = 15

6. Construct a binary tree whose preorder and inorder sequences are A B M H E O C P G J D K L I N F and H M C O E B A G P K L

D I N J F respectively, where A, B, C, D, E, ………. are the labels of the tree nodes. Is it unique?
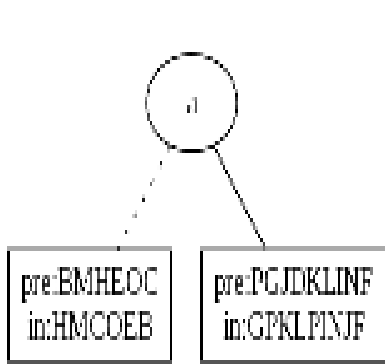
(a)
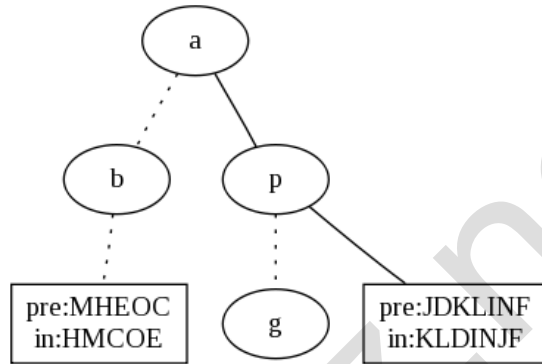
(b)
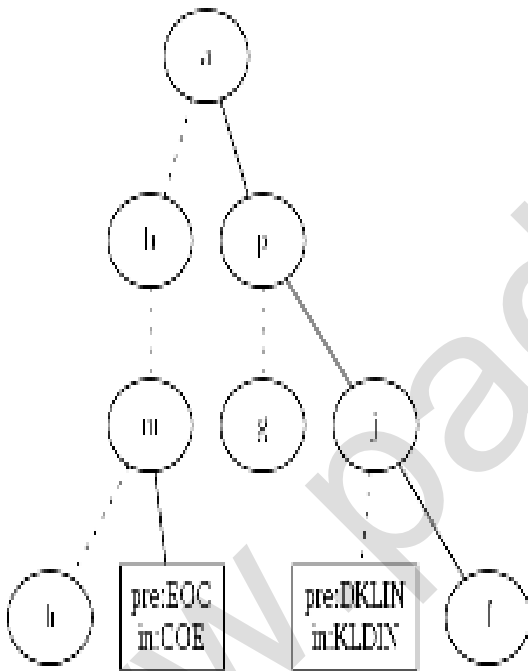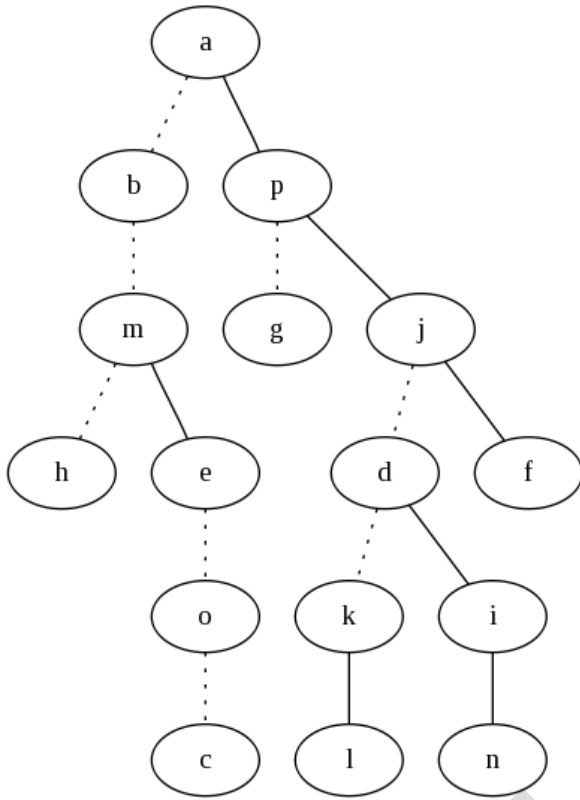
(c)
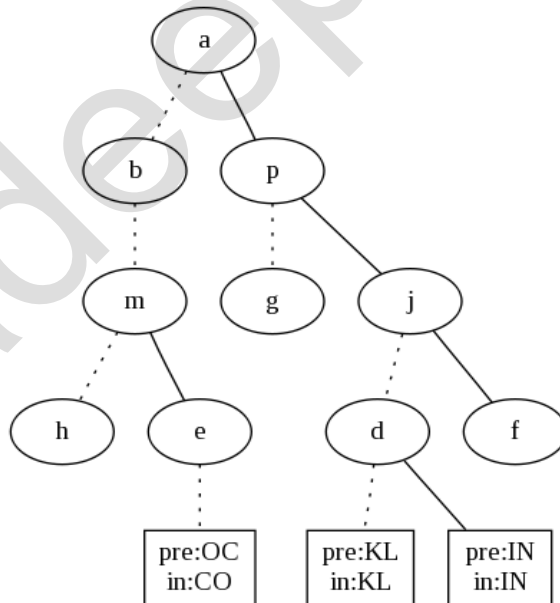
(d)
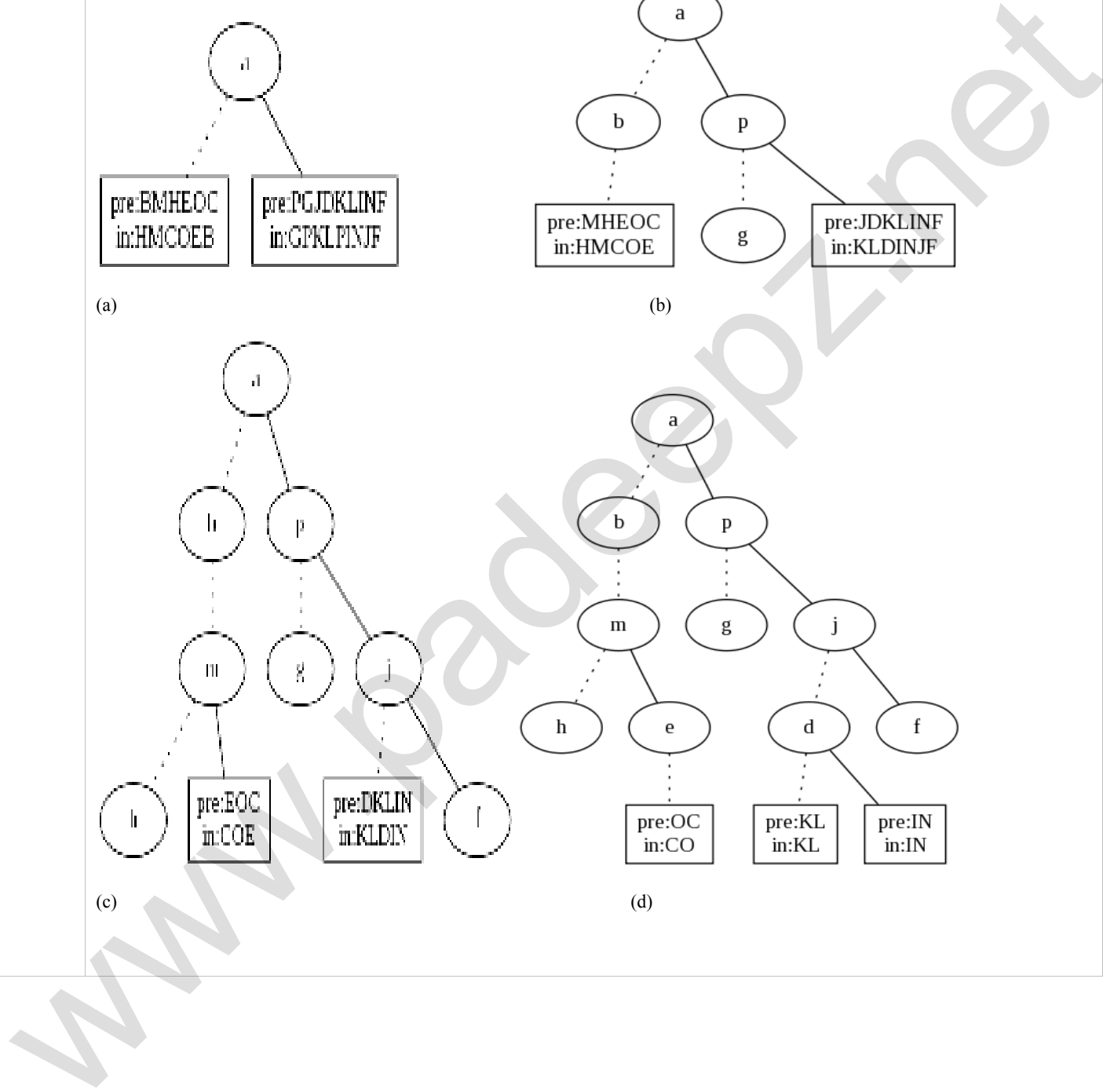
(e)

7. The weighted external path length of the binary tree in Fig. 2 is _____.
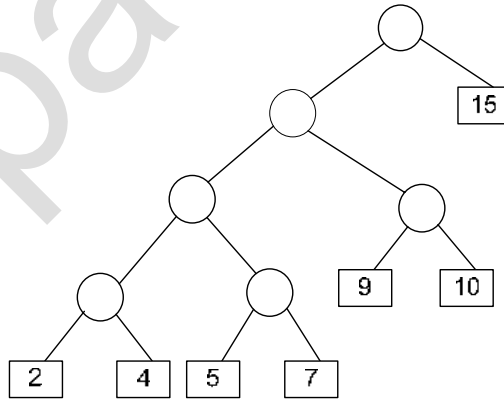
Fig 2

weighted external path = (2*4) + (4*4) + (5*4) + (7*4) + (9*3) + (10*3) + (15*1)

= 8 + 16 + 20 + 28 + 27 + 30 + 15

=144

8. If the binary tree in Fig. 3 is traversed in inorder, then the order in which the nodes will be visited is _____
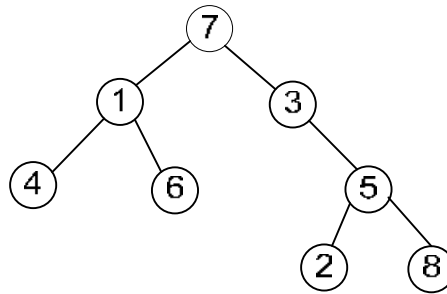
Fig 3

4 1 6 7 3 2 5 8

The in order sequence is left, root, right .

9.	Consider the binary tree in Fig. 7:

(a) What structure is represented by the binary tree?

(b) Give the different steps for deleting the node with key 5 so that the structure is preserved.

(c) Outline a procedure in the pseudo-code to delete an arbitrary node from such a binary tree with n nodes that preserves the structure. What is the worst case complexity of your procedure?

Fig. 7

(a) Min-Heap

(b)

i.   Swap(27,5)

ii.   Delete 5

iii.   Minheapify

(c)

Alogithm delMinHeap()

Step 1: let I th node is to be deleted.

Step 2: copy last element to this location

Step 3: remove last element of array

Step 4: perform Min-Heapify on I th node

Algorithm Min-Heapify()

A-array, i-ith node

Step 1: if 2*i<=size[A] and A[2i]<A[i]   /*left child*/

       Then smallest=2i

       Else smallest =i

Step 2: if 2i+1 <=size[A] and A[2i+1] < A[smallest]

       Then smallest=2i+1

Step3: If smallest != i

     Then swap A[i] and A[smallest]

Step 3: Min-Heapify(A,smallest)

---

10. if a tree has n_1 nodes of degree 1, n_2 nodes of degree 2, ... n_m nodes of degree m, give a formula for the number of terminal nodes n0 of the tree in terms of n_1, n_2, ..., n_m.

Sum of degree of all the nodes will be-$1*n_1+2*n_2.....m*n_m$-------(1)

We know sum of degree of all the nodes is equal to number of edges.-----(2)

And number of edges= number of total node-1.-------(3)

Total number of nodes= $n_0+n_1+n_2....n_m$.------------(4)

From 1,2,3,4 we have

$1*n_1+2*n_2......m*n_m= (n_0+n_1+n_2......n_m)-1$

We get

Ans: $n_0 = 1+[1*n_2(2-1) +2*n_3....+(m-1)*n_m]$

---

11. A 2-3 tree is a tree such that
     (a) all internal nodes have either 2 or 3 children
     (b) all paths from root to the leaves have the same length.
The number of internal nodes of a 2-3 tree having 9 leaves could be

     (A) 4
     (B) 5
     (C) 6
     (D) 7

A and D

At leaf-level:

	1. group the nodes into 3 sets of 3 …… it gives 4 internal nodes 2. group the nodes into 3 sets of 2 and 1 set of 3 …… it gives 7 internal nodes
12.	A *K*-ary tree is such that every node has either *K* sons or no sons. If *L* and *I* are the number of leaves and internal nodes respectively, then express *L* in terms of *K* and *I*. If there is only one 1 internal node then it have k leaf nodes. Adding one more internal node means making 1 leaf node as internal and adding k leaf nodes. So if there are I internal nodes it have **I*k-(I-1)** leaf nodes **L=K*(I-1)+1**

13.	A 3-ary tree is a tree in which every internal node has exactly three children. Use the induction to prove that the number of leaves in a 3-ary tree with n internal nodes is 2(n-1) + 3.
	When,
	
	n = 0 → l = 2(0 - 1) + 3 =1
	
	n = 1 → l = 2(1 - 1) + 3 = 3
	
	In this way let the rule be true for n = k
	
	Hence l = 2(k - 1) + 3
	
	For n = k + 1
	
	However, from observation: l(k + 1) = l(k) + 3 – 1
	
	i.e. l(k + 1) = 2(k - 1) + 3 + 3 -1
	
	        = 2k + 3
	
	        = 2((k + 1)-1) + 3
	
	Thus, by induction, the rule is true for all k.

14.	A complete n-ary tree is one in which every node has 0 or n sons. If x is the number of internal nodes of a complete n-ary tree, the number of leaves in it is given by (a) x (n – 1) + 1 (b) xn – 1 (c) xn + 1 (d) x (n + 1) [A] Degree of internal nodes (except root) = n + 1 Degree of root = n Degree of leaf nodes = 1 Also, No of edges = no. of nodes – 1 But n = x + L

Thus, no. of edges = x + L -1

From graph theory: sum of degree = 2 * (no. of edges)

Case 1: the tree has only 1 node.

Sum of degree = 2 * (no of edges) = 0

Case 2: The tree has more than one node.

(n + 1) * (x - 1) + n + L = x + L -1

Ans: L = x(n - 1) + 1

15.	
16.	The minimum number of interchanges needed to convert the array

89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70

into a heap with the maximum element at the root node is

(A) 0   (B) 1

(C) 2   (D) 3

Only element 70 violates the rule. Hence, it must be shifted to its proper position.

Step1: swap(10, 70)

Step2: swap(40, 70)

Hence, only 2 interchanges are required.

17.	Heap allocation is required for languages

(A) that support recursion   (B) that support dynamic data structures

(C) that use dynamic scope rules   (D) none of the above

[B] Dynamic data structures have the capacity to grow. Hence, they need some sort of memory allocation during runtime. The heap is required to provide this memory.

18.	A binary tree T has n leaf nodes. The number of nodes of degree 2 in T is

(a) $\log_2 n$
(b) n - 1
(c) n
(d) 2^n

[B] Total no. of nodes in the tree

n = n0 + n1 + n2

Where ni is the number of nodes with i children.

Total no. of edges in the tree

e = n0*0 + n1*1 + n2*2

But e = n - 1

n1 + 2*n2 = n0 + n1 + n2 − 1

n2 = n0 − 1

n2 = n - 1

---

19.   What is the number of binary trees with 3 nodes which when traversed in post-order give the sequence A, B, C? Draw all these binary trees.

Ans: 5

(a)   (b)   (c)   (d)   (e)

---
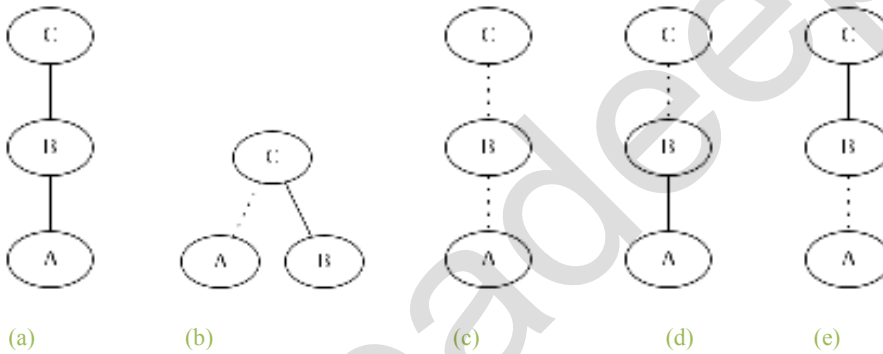
20.   In the balanced binary tree in Fig. 1.14 given below, how many nodes will become unbalanced when a node is inserted as a child of the node "g"?
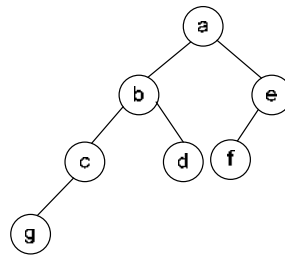
Fig. 1.14

(A)   1
(B)   3
(C)   7
(D)   8
   [B]

After insertion

Bf(a) = 2

Bf(b) = 2

Bf(c) = 2

Bf(d) = 0

Bf(e) = 1

Bf(f) = 0

Bf(g) = 1

Which of the following sequences denotes the post-order traversal sequence of the tree of FIG 1.14?

(A) f e g c d b a   (B) g c b d a f e

(C) g c d b f e a   (D) f e d g c b a
[C]
Post-order = Left sub-tree – right sub-tree – self

21.   A binary search tree is generated by inserting in order the following integers:

                50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24

        The number of nodes in the left subtree and right subtree of the root is respectively is

        (A) (4, 7)   (B) (7, 4)

        (C) (8, 3)   (D) (3, 8)

[B]

Nodes to in the left subtree are less than 50, while nodes in the right sub-tree are greater than 50.

 No. of nodes less than 50 = 7

No. of nodes greater than 50 = 4

22.   A binary search tree is used to locate the number 43. Which of the following probe sequences are possible and which are not? Explain.

(A)		61	52	14	17	40	43
(B)		2	3	50	40	60	43
(C)		10	65	31	48	37	43
(D)		81	61	52	14	41	43
(E)	17	77	27	66	18	43	

Possible:[A,C, D]

Not possible:

[B] – at 50 the left branch is taken. Hence, all nodes in the list after 50 must be less than it. But 60 is not similarly in[D]18 is smaller than 27

23.   A binary search tree contains the values 1, 2, 3, 4, 5, 6, 7, 8. The tree is traversed in pre-order and the values are printed out. Which of the following sequences is a valid output?

| | (A) 5 3 1 2 4 7 8 6 | (B) 5 3 1 2 6 4 8 7 |
| | (C) 5 3 2 4 1 6 7 8 | (D) 5 3 1 2 4 7 6 8 |

Valid: d

Invalid: a, b, c

A preorder list has the following format –

<root> <left sub-tree (smaller elements)> <right sub-tree (larger elements)>

This format is applied recursively to the sub-trees.

---

24. A size balanced binary tree is a binary tree in which for every node, the difference between the number of nodes in the left and right subtree is at most 1. The distance of a node from the root is the length of the path from the root to the node. The height of a binary tree is maximum distance of a leaf node from the root.

   (a) Prove, by using induction on h, that a size-balanced binary tree of height h contains at least 2^h nodes.

When

h = 0 ……… least no. of nodes = $2 ^ 0 = 1$

h = 1 ……… least no. of nodes = $2 ^ 1 = 2$

h = 2 ……… least no. of nodes = $2 ^ 2 = 4$

Assume that the rule is true for h = k

Then the min no. of nodes = $2 ^ k$ nodes

If we increase the height by 1 by adding a node, we must also add nodes to fill the (max level -1) level.

This would mean doubling the nodes

Thus $2^{(k+1)}$

   Hence, proved

   (b) In a size-balanced binary tree of height h ≥ 1, how many nodes are at distance h-1 from the root? Write only the answer without any explanation

   2^{h-1}

---

25. Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. the binary search tree uses the usual ordering on natural numbers. What is the inorder traversal sequence of the resultant tree?

   (A) 7 5 1 0 3 2 4 6 8 9
   (B) 0 2 4 3 1 6 5 9 8 7
   (C) 0 1 2 3 4 5 6 7 8 9
   (D) 9 8 6 4 2 3 0 1 5 7

   [C] In order traversal of the binary search tree is always in ascending order

26.	Which of the following statements is false?

(a) A tree with n nodes has (n-1) edges
(b) A labeled rooted binary tree can be uniquely constructed given its post-order and pre-order traversal results.
(c) A complete binary tree with n internal nodes has (n+1) leaves
(d) The maximum number of nodes in a binary tree of height h is $(2^{h+1} - 1)$

[B and D] With inorder,preorder or with inorder, postorder it is possible for a unique combination

Maximum number of nodes in a binary tree of height h is $(2^h - 1)$

27.	Draw the binary tree with the node labels a, b, c, d, e, f and g for which the inorder and post-order traversals result in the following sequences

Inorder   a f b c d g e

Postorder   a f c g e d b

(a)   (b)   (c)
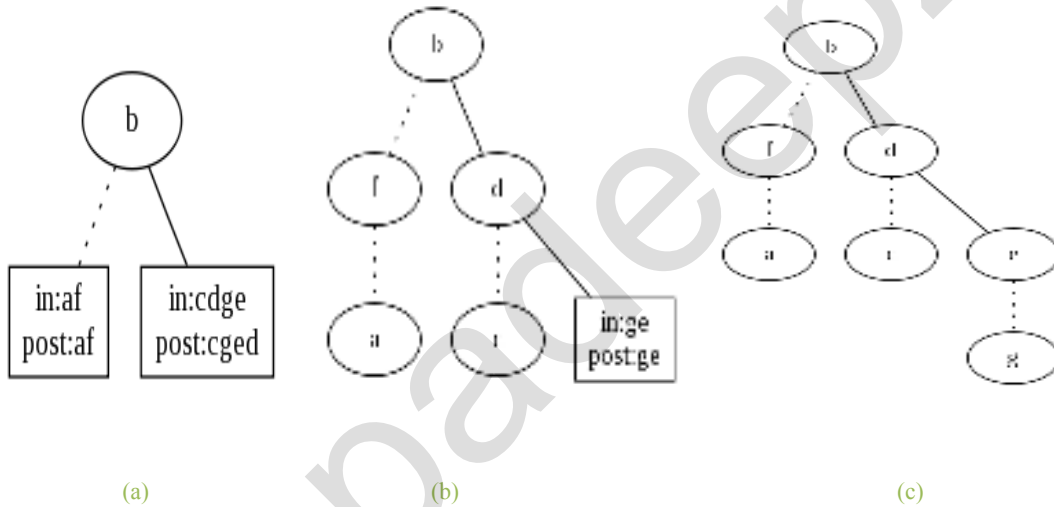
28.	Draw the min-heap that results from insertion of the following elements in order into an initially empty min-heap: 7, 6, 5, 4, 2, 3, 1. show the result after the deletion of the root of this heap.

(a)   (b)   (c)
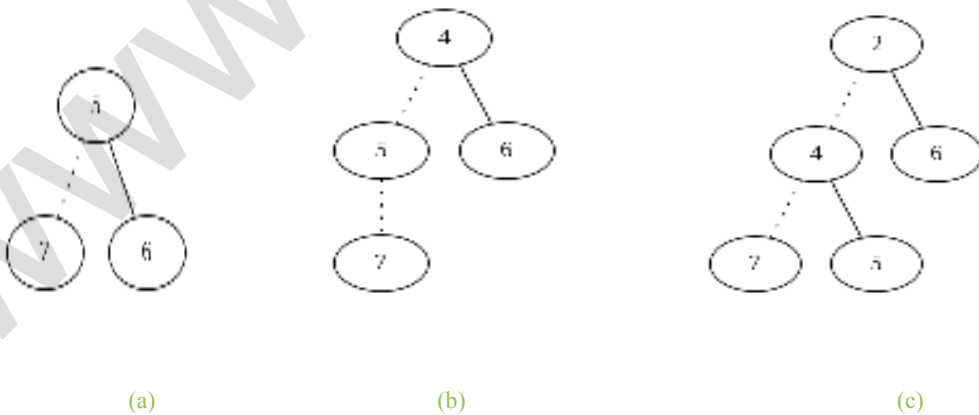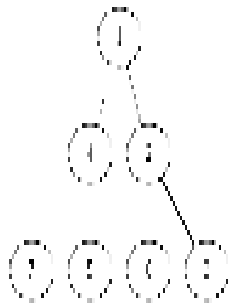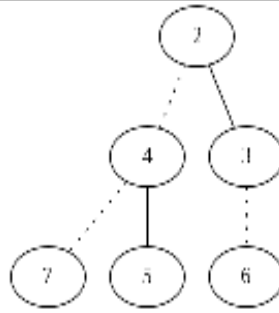
(d)   (e)

| 29. | The number of leaf nodes in a rooted tree of n node, with each node having 0 or 3 children is: |

     A.   n/2

     B.   (n – 1)/3

     C.   (n – 1)/2

     D.   (2n + 1)/3

[D]Degree of internal node (except root node) = 4

Degree of root = 3 or 0

Degree of leaf node = 1

No of edges = n – 1

From graph theory: sum of degree = 2 * (no. of edges)

Case 1: tree contains only root

Sum of degree = 2 * (no of edges) = 0

Case 2:

(n – L - 1) * 4 + 3 + L * 1 = 2 * (n - 1)

Ans: L = (2n + 1)/3

| 30. | In a binary tree, a full node is defined to be a node with 2 children. Use the induction on the height of the binary tree to prove that the number of full nodes plus one is equal to the numbers of leaves. |

1(root) element contains 2 children .no of full nodes=1,no. of leaves=2

2 full nodes have 3 leaves

3 full nodes have 4 leaves

………………………..
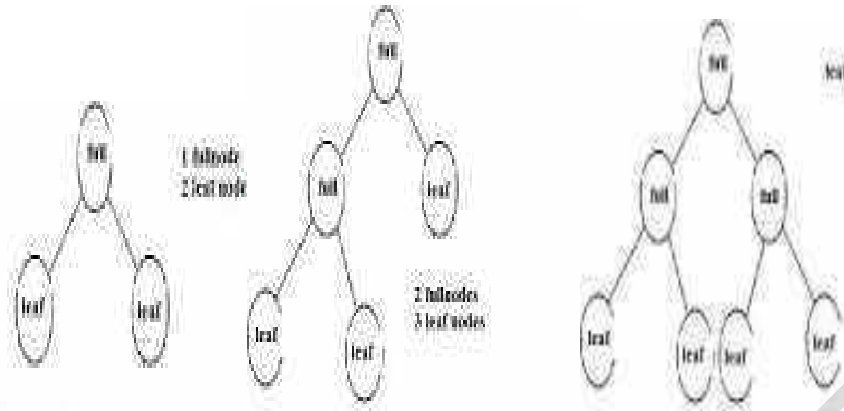
31. Consider the following nested representation of binary trees: (X Y Z) indicates Y Z are the left and right subtrees, respectively, of node X. Note that Y and Z may be NULL, or further nested. Which of the following represents a valid binary tree?

   (A) (1 2 (4 5 6 7))
   (B) (1 ((2 3 4) 5 6) 7)
   (C) (1 (2 3 4) (5 6 7))
   (D) (1 (2 3 NULL) (4 5))

[C] Every node contains two sub trees.

In [A] 4 contains 3 sub trees which violate the property.

[B] node 2 has 4 child 3 and 4 and 5 and 6

[D] node 4 right child is not specified NULL.

32. Let LASTPOST, LASTIN and LASTPRE denotes the last vertex visited in a post-order, inorder and pre-order traversal respectively, of a completely binary tree. Which of the following is always true?

   (A) LASTIN = LASTPOST
   (B) LASTIN = LASTPRE
   (C) LASTPRE = LASTPOST
   (D) None of the above

[B]

In-order = LNR

Pre-order = NLR

Post-order = LRN

The rightmost element will be the last element in both in and pre orders

33. (a) Insert the following keys one by one into a binary search tree in order specified.

   15, 32, 20, 9, 3, 25, 12, 1

   Show the final binary search tree after insertions.
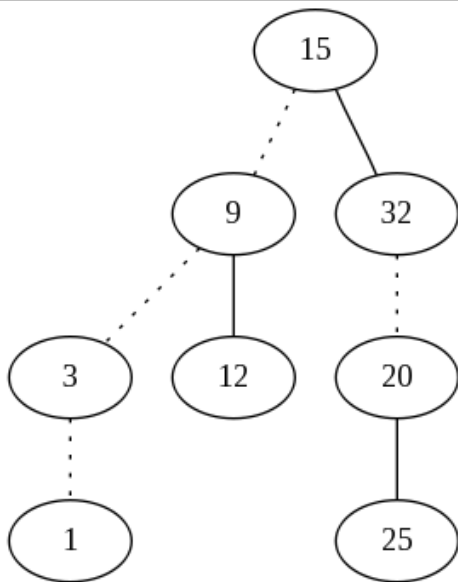
(b) Draw the binary search tree after deleting 15 from it.
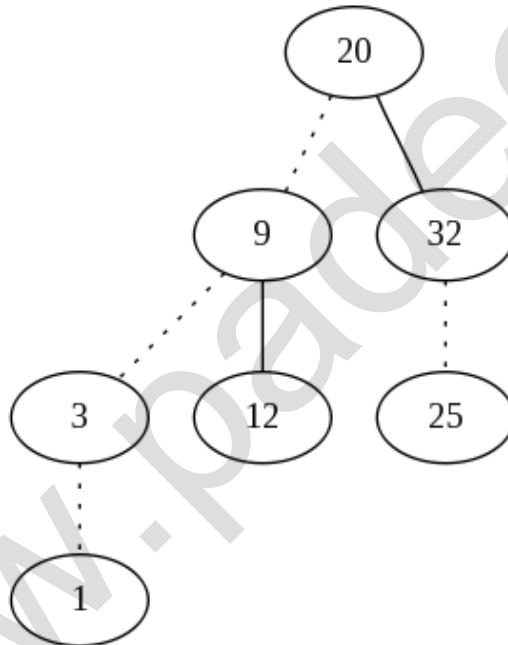
(c) Complete the statements S1, S2 and S3 in the following function so that the function computes the depth of a binary tree rooted at t.

```
typedef struct tnode {

    int key;

    struct tnode *left, *right;

} *Tree;


int  depth (Tree t)
```

```
                              {
                                  int x,y;
                                  if (t = = NULL) return 0;
                                  x = depth(t →left);
                S1:        _____;
                S2:        if(x > y) return _____;
                S3:        else return _____;
        }
```

S1: y=depth(t->right)

S2: x+1

S3: y+1

34.	A weight balanced tree is a binary tree in which for each node, the number of nodes in the left subtree is at least half and at most twice the number of nodes in the right subtree. The maximum possible height (number of nodes on the path from the root to the furthest leaf) of such a tree on n nodes is best described by which of the following?:

     A. $\log_2 n$
     B. $\log_{4/3} n$
     C. $\log_3 n$
     D. $\log_{3/2} n$

35.	Draw all binary trees having exactly three nodes labeled A, B, and C on which pre-order traversal gives the sequence C, B, A.

Ans: 5

   (a)   (b)   (c)   (d)   (e)
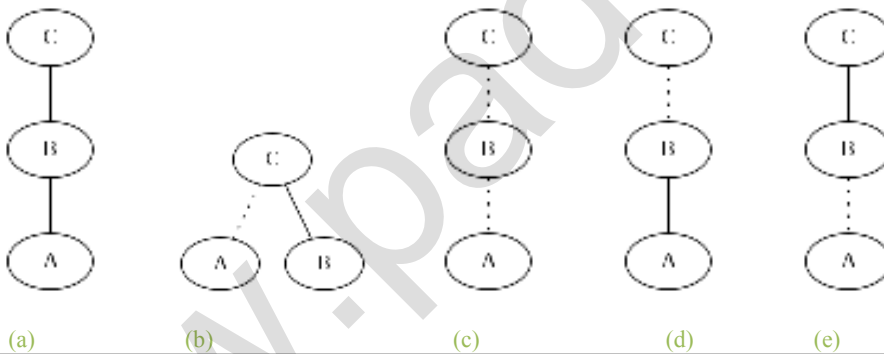
36.	In heap with n elements with the smallest element at the root, the 7th smallest element can be found in time

     (A) $\Theta(n \log n)$
     (B) $\Theta(n)$
     (C) $\Theta(\log n)$
     (D) $\Theta(1)$

In order to find the 7th element one must remove elements one by one from the heap, until the 7th smallest element is found.

As a heap may contain duplicate values, we may need to remove more then 7 elements. At the worst case we will have to remove all n elements.

   1 deletion takes O(log(n)). Hence 'n' deletes will take O(nlog(n))

37.	Construct a binary tree whose preorder traversal is K L N M P R Q S T and inorder traversal is N L K P R M S Q T

38. (i) Define the height of a binary tree or subtree and also define a height balanced (AVL) tree.

The height of a tree or a sub tree is defined as the length of the longest path from the root node to the leaf.

A tree is said to be height balanced if all the nodes are having a balance factor -1,0 or 1. The balance factor is the height of the left sub tree minus height of the right sub tree

(ii) Mark the balance factor of each mode on the tree given in fig 7.1 and state whether it is height-balanced

(iii) Into the same tree given in 7(ii) above, insert the integer 13 and show the new balance factors that would arise if the tree is no rebalanced. Finally, carry the required rebalancing of the tree and show the new tree with the balance factors on each mode.

39. The number of rooted binary trees with n nodes is,

    (A) Equal to the number of ways of multiplying (n+1) matrices.
    (B) Equal to the number of ways of arranging n out of 2n distinct elements.
    (C) Equal to $\dfrac{1}{(n+1)}\dbinom{2n}{n}$
    (D) Equal to n!

[C]

N=1 1 tree

N=2 2 trees

N=3 5 trees

N=4 14 trees

1, 2 ,5, 14…………………..are catlan numbers

40.	The maximum number of nodes in a binary tree of level k, k ≥ 1 is
	(E) $2^k + 1$
	(F) $2^k - 1$
	(G) 2^{k-1}
	(H) $2^{k-1} - 1$

41.	Construct a binary tree whose preorder and inorder sequences are A B M H E O C P G J D K L I N F and H M C O E B A G P K L D I N J F respectively, where A, B, C, D, E, ………. are the labels of the tree nodes. Is it unique?

(a)

(b)

(c)

(d)
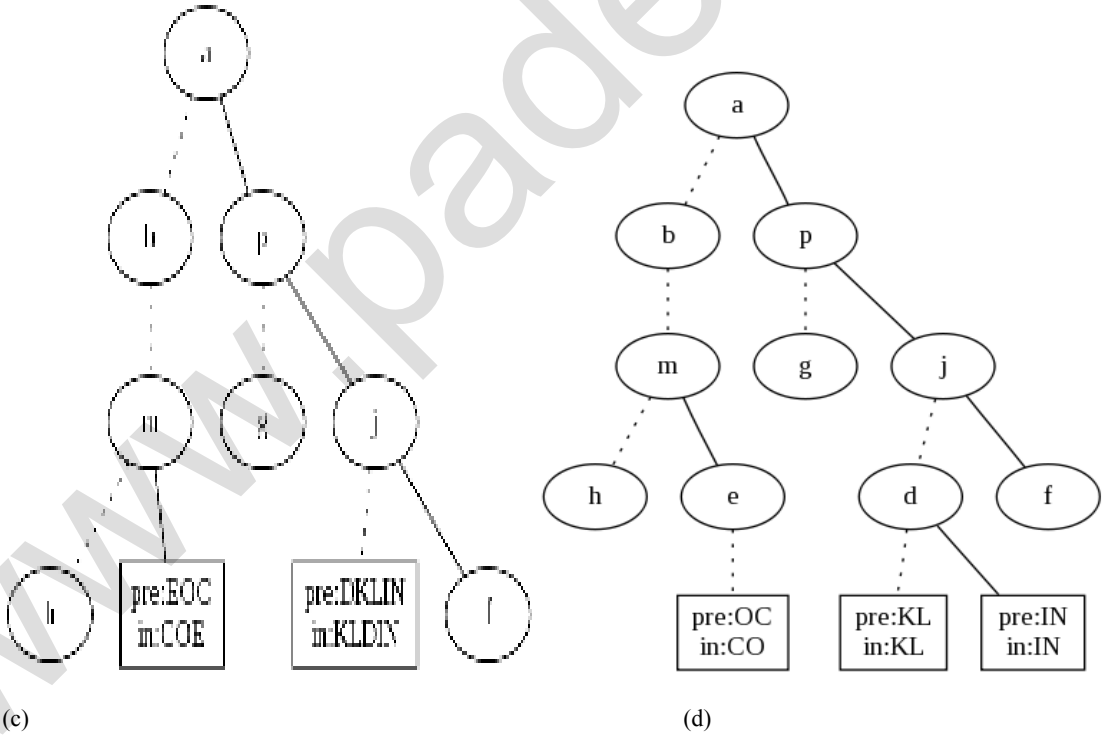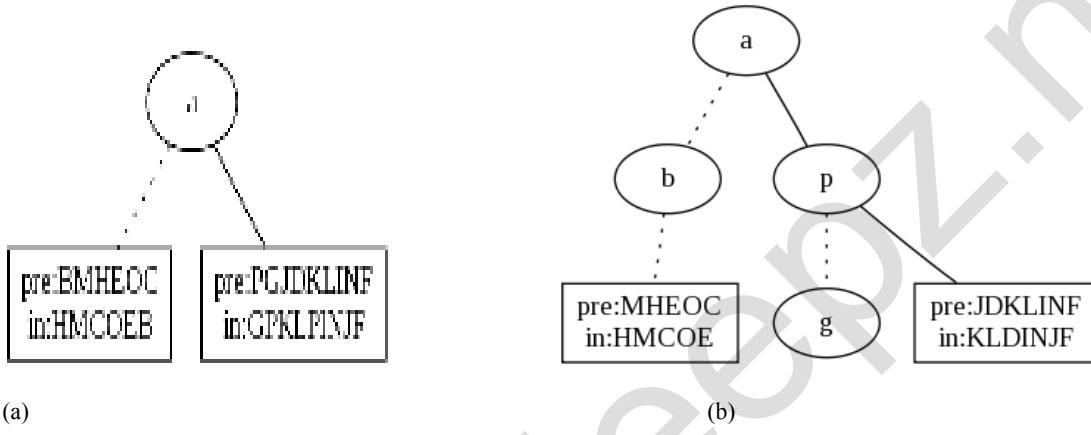
(e)

42.	The minimum number of comparisons required to sort 5 elements is _____.
	4 using insertion sort when numbers are in sorted order
43.	The weighted external path length of the binary tree in Fig. 2 is _____.
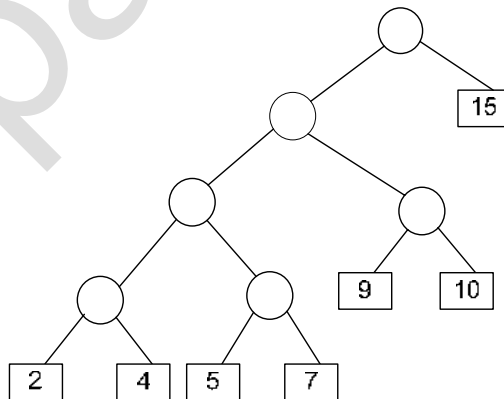
Fig 2

weighted external path = (2*4) + (4*4) + (5*4) + (7*4) + (9*3) + (10*3) + (15*1)

= 8 + 16 + 20 + 28 + 27 + 30 + 15

=144

44.	If the binary tree in Fig. 3 is traversed in inorder, then the order in which the nodes will be visited is _____

Fig 3

In-order traversal: 4 1 6 7 3 2 5 8

---

45. Consider the binary tree in Fig. 7:

   (d) What structure is represented by the binary tree?
   (e) Give the different steps for deleting the node with key 5 so that the structure is preserved.
   (f) Outline a procedure in the pseudo-code to delete an arbitrary node from such a binary tree with n nodes that preserves the structure. What is the worst case complexity of your procedure?
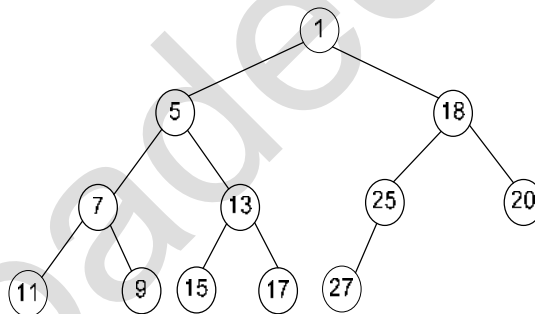
Fig. 7

(a)Min-Heap

(b)

   iv.   Swap(27,5)

   v.   Delete 5

   vi.   Minheapify

(c)

Alogithm delMinHeap()

Step 1: let I th node is to be deleted.

Step 2: copy last element to this location

Step 3: remove last element of array

Step 4: perform Min-Heapify on I th node

Algorithm Min-Heapify()

A-array, i-ith node

Step 1:   if 2*i<=size[A] and A[2i]<A[i]   /*left child*/

           Then smallest=2i

           Else smallest =i

Step 2: if 2i+1 <=size[A] and A[2i+1] < A[smallest]

           Then smallest=2i+1

Step3: If smallest != i

       Then swap A[i] and A[smallest]

Step 3: Min-Heapify(A,smallest)

---

46. Consider a binary max-heap implemented using an array.

Which one of the following array represents a binary max-heap?

        (A) { 25, 12, 16, 13, 10, 8, 14 }
        (B) { 25, 14, 13, 16, 10, 8, 12 }
        (C) { 25, 14, 16, 13, 10, 8, 12 }
        (D) { 25, 14, 12, 13, 10, 8, 16 }
        [C]

What is the content of the array after two delete operations on the correct answer to the previous question?

        (A) { 14, 13, 12, 10, 8 }
        (B) { 14, 12, 13, 8, 10}
        (C) { 14, 13, 8, 12, 10 }
        (D) { 14, 13, 12, 8, 10 }
         [D]

(A) 12 has child 13

(B) 14 has child 16

(D) 12 has child 16

Hence, they are not max-heaps.
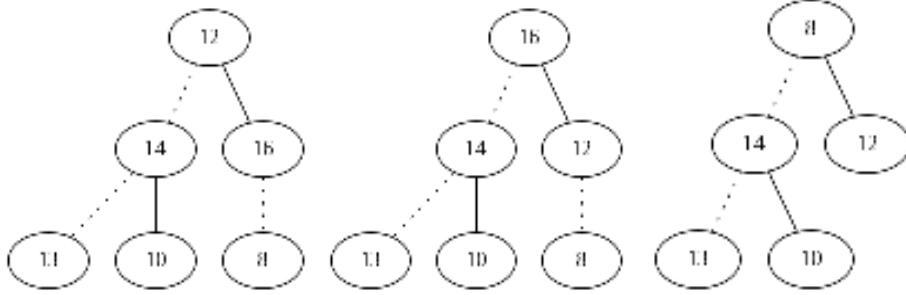
(a) (b) (c)
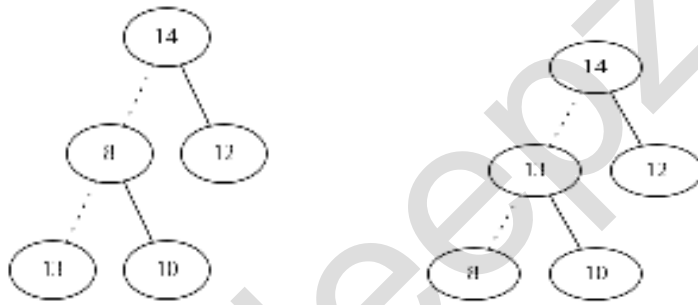
(d) (e)

47. What is the maximum height of any AVL-tree with 7 nodes? Assume that the height of a tree with a single node is 0.

(A) 2 (B) 3 (C) 4 (D) 5

[B]The maximum height of an AVL tree with n nodes is 1.44log n.

By substituting n=7 we get 4.03,since it is given that the height of a tree with a single node is 0.Therefore the answer is 3.

48. The following three are known to be the preorder, inorder and postorder sequences of a binary tree. But it is not known which is which.

I.   MBCAFHPYK
II. KAMCBYPFH
III. MABCKYFPH

Pick the true statement from the following.

(A) I and II are preorder and inorder sequences, respectively
(B) I and III are preorder and postorder sequences, respectively
(C) II is the inorder sequence, but nothing more can be said about the other two sequences
(D) II and III are the preorder and inorder sequences, respectively

[D]

II and III have the same last element. Thus they must be either pre-order or in-order.

Thus, I must be post-order.

But then K must be the root.

K occurs as the first element tin II. Thus, II must be pre-order.

Thus, III must be in-order.

49.	Which of the following is TRUE?

(A) The cost of searching an AVL tree is $\theta(\log n)$ but that of a binary search tree is $O(n)$

(B) The cost of searching an AVL tree is $\theta(\log n)$ but that of a complete binary tree is $\theta(n \log n)$

(C) The cost of searching a binary search tree is $O(\log n)$ but that of an AVL tree is $\theta(n)$

(D) The cost of searching an AVL tree is $\theta(n \log n)$ but that of a binary search tree is $O(n)$

[A]Searching in AVL tree has O(logn) complexity.
And searching for binary search tree is also O(n), when it is skewed and   is order of O(log(n)) when it is complete

50.	We have a binary heap on n elements and wish to insert n more elements (not necessarily one after another) into this heap. The total time required for this is

(A) $\Theta(\log n)$

(B) $\Theta(n)$

(C) $\Theta(n \log n)$

(D) $\Theta(n^2)$

[C]

For insertion of 1 node in heap will take O(log(n))

For insertion of n nodes will take O(nlog(n)).

51.	You are given the postorder traversal, P, of a binary search tree on the n elements $1, 2, ..., n$. You have to determine the unique binary search tree that has P as its postorder traversal. What is the time complexity of the most efficient algorithm for doing this?

(A) $\Theta(\log n)$

(B) $\Theta(n)$

(C) $\Theta(n \log n)$

(D) none of the above, as the tree cannot be uniquely determined.

[C] To get unique tree we must have pre order and in order or post order and inorder.
In order traversal of a BST is always sorted.
I=Sort P(using merge sort or heap sort which takes O(nlogn))
Using I and P construct the tree.

52.	When searching for the key value 60 in a binary search tree, nodes containing the key values 10, 20, 40, 50, 70, 80, 90 are traversed, not necessarily in the order given. How many different orders are possible in which these key values can occur on the search path from the root node containing the value 60?
	(A) 35 (B) 64 (C) 128 (D) 5040

53.	Consider the process of inserting an element into a *Max Heap,* where the *Max Heap* is represented by an array. Suppose we perform a binary search on the path from the new leaf to the root to find the position for the newly inserted element, the number of *comparisons* performed is:

(A) $\Theta(\log_2 n)$
(B) $\Theta(\log_2 \log_2 n)$
(C) $\Theta(n)$
(D) $\Theta(n \log_2 n)$
[B]
n = no of elements in the tree.

lgn = height of the tree i.e. path length of the newly inserted node to the root.

Complexity of binary search = O(lgn) where n is the no. of elements searched.

Thus, O(ln(lg(n)))

54. A complete *n*-ary tree is a tree in which each node has n children or no children. Let *I* be the number of internal nodes and *L* be the number of leaves in a complete *n*-ary tree. if *L* = 41, and *I* = 10, what is the value of n?

(A) 3 (B) 4 (C) 5 (D) 6

$L = I(n - 1) + 1$

Thus $n = (L - 1) / I - 1$

i.e $n = (41 - 1) / 10 - 1$

Ans: n = 3

55. The inorder and pre-order traversal of a binary tree are

d b e a f c g and a b d e c f g, respectively.

The post-order traversal of the binary tree is

(A) d e b f g c a
(B) e d b g f c a
(C) e d b f g c a
(D) d e f g b c a
[A]
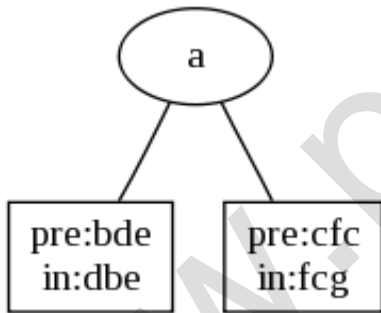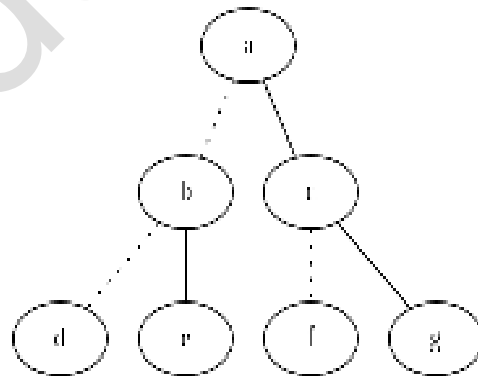
(a)   (b)

56. The maximum number of binary trees that can be formed with three unlabeled nodes is:

(A) 1 (B) 5 (C) 4 (D) 3

[B] Using formula (1/(n+1))* 2nCn , where n is the number of unlabeled node so 5

57. The height of a binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is:

(A) 2^h (B) 2^{h-1} - 1 (C) $2^{h+1} - 1$ (D) 2^{h+1}

Level 0: h = 0; nodes = 1

Level 1: h = 1; nodes = 2

Level 2: h = 2; nodes = 4

Level n: h = n; nodes = 2^n

Total no of nodes = 1 + 2 + 4 + … + 2^n = 2^(h+1) -1

58.	Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined?

(A) {10, 75, 64, 43, 60, 57, 55}   (B) {90, 12, 68, 34, 62, 45, 55}

(C) {9, 85, 47, 68, 43, 57, 55}   (D) {79, 14, 72, 56, 16, 53, 55}

[C] On arriving at node 47, we take the right sub-tree. The right sub-tree can contain only those nodes which are larger than 47. But 43 < 47

59.	Which of the following sequences of array elements forms a heap?

(A) {23, 17, 14, 6, 13, 10, 1, 12, 7, 5}   (B) {23, 17, 14, 6, 13, 10, 1, 5, 7, 12}

(C) {23, 17, 14, 7, 13, 10, 1, 5, 6, 12}   (D) {23, 17, 14, 7, 13, 10, 1, 12, 5, 7}

[C] In a max heap parent node is always greater than its children. Only [C] satisfies this property

60.	A scheme for sorting binary trees in an array X is as follows. Indexing of X starts at 1 instead of 0. The root is stored at X[1]. For a node stored at X[i], the left child, if any is stored in X[2i] and the right child, if any, in X[2i + 1]. To be able to store any binary tree on n vertices the minimum size of X should be

(A) $\log_2 n$ (B) n   (C) 2n + 1 (D) $2^n – 1$

[B] In a complete binary tree to store n nodes an array of size n is needed.

61.	In a binary max heap containing n numbers, the smallest element can be found in time

(A) $\Theta(n)$ (B) $\Theta(\log n)$ (C) $\Theta(\log \log n)$ (D) $\Theta(1)$

[A]The smallest element can be found in last level of tree. So finding smallest we have to perform search in last level and that search can be performed in O(n).So total complexity will be O(logn+n)= O(n).

62.	A binary search tree contains the numbers 1, 2, 3, 4, 5, 6, 7, 8. When the tree is traversed in pre-order and the values in each node printed out, the sequence of values obtained is 5, 3, 1, 2, 4, 6, 8, 7. If tree is traversed in post-order, the sequence obtained would be

(A) 8, 7, 6, 5, 4, 3, 2, 1
(B) 1, 2, 3, 4, 8, 7, 6, 5
(C) 2, 1, 4, 3, 6, 7, 8, 5
(D) 2, 1, 4, 3, 7, 8, 6, 5
[D]

(a)   (b)   (c)

---

63. In a binary tree, for every node the difference between the number of nodes in the left and right subtrees is at most 2. If the height of the tree is h . 0, then the minimum number of nodes in the tree is

(A) 2^{h-1} (B) $2^{h-1} + 1$ (C) $2^{h-1} - 1$ (D) 2^h

N= h+[1/2*[(h-2)*(h-3)]]

---

64. The numbers 1, 2, …, n are inserted in a binary search tree in some order. In the resulting tree, the right subtree of the root contains p nodes. The first number to be inserted in the tree must be

(A) 134   (B) 133   (C) 124   (D) 123

1-----r-----n

Where r is the value of the root.

No. of elements between n and r is n – r = p

r = n - p

---

65. In a complete k-ary tree, every internal node has exactly k children. The number of leaves in such a tree with n internal nodes is:

(A) nk
(B) (n-1)k +1
(C) n(k – 1) + 1
(D) n(k – 1)

[C]Degree of internal nodes (except root) = k + 1

Degree of root = k

Degree of leaf nodes = 1

Also,

No of edges = no. of nodes – 1

But total no of nodes = n + L (L = leaf nodes)

Thus, no. of edges = n + L -1

From graph theory: sum of degree = 2 * (no. of edges)

Case 1: the tree has only 1 node.

Sum of degree = 2 * (no of edges) = 0

Case 2: The tree has more than one node.

(k + 1) * (n - 1) + k + L = n + L -1

Ans: L = n(k - 1) + 1

66.	How many distinct binary search tree can be created out of 4 distinct keys?

    (A) 5
    (B) 14
    (C) 24
    (D) 42

[B] By formula $\dfrac{1}{(n+1)}\dbinom{2n}{n}$

67.	A Priority-Queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is given below:

10, 8, 5, 3, 2

Two new elements '1' and '7' are inserted in the heap in that order. The level-order traversal of the heap after the insertion of the elements is:

    (A) 10, 8, 7, 5, 3, 2, 1
    (B) 10, 8, 7, 2, 3, 1, 5
    (C) 10, 8, 7, 1, 2, 3, 5
    (D) 10, 8, 7, 3, 2, 1, 5
    [D]
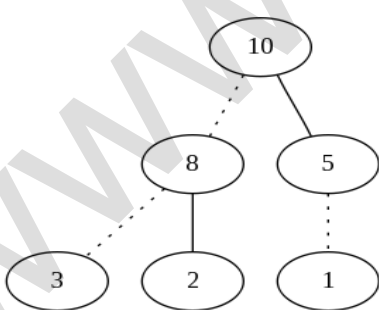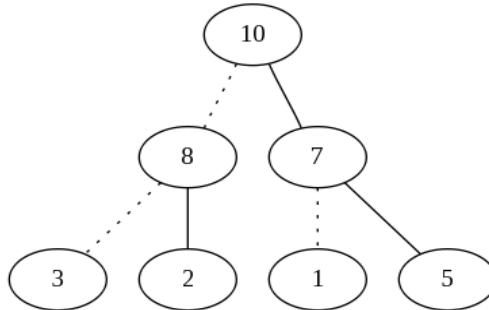
        (a)   (b)

68.	Which of the following binary trees has its inorder and preorder traversals as BCAD and ABCD, respectively?

(A)
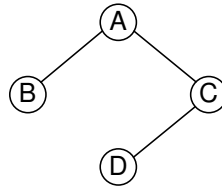
(B)
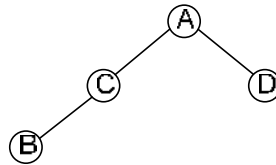
(C)
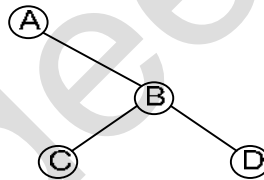
(D)
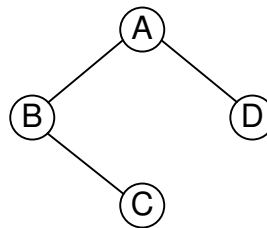
[D] With inorder and preorder,a tree can be uniquely identified.

69. An array of integers of size n can be converted into a heap by adjusting the heaps rooted at each internal node of the complete binary tree starting at the node $\lfloor (n-1)/2 \rfloor$, and doing this adjustment upto the root node (root node is at index 0) in the order $\lfloor (n-1)/2 \rfloor, \lfloor (n-3)/2 \rfloor, \ldots, 0$. The time required to construct a heap in this manner is

    (A) O(log n)
    (B) O(n)
    (C) O(n log log n)
    (D) O(n log n)

		[B] Refer Cormen for proof
70.		A program takes as input a balance binary search tree with n leaf nodes and computes the value of a function g(x) for each node x. if the cost of computing g(x) is min {no. of leaf nodes in left-subtree of x, no. of leaf-nodes in right-sub tree of x} then the worst time case complexity of the program is

(A) $\Theta(n)$
(B) $\Theta(n \log n)$
(C) $\Theta(n^2)$
(D) $\Theta(n^2 \log n)$

[B] The recurrence relation for the recursive function is

$T(N) = 2 * T(N/2) + n/2$

Where N is the total no. of nodes in the tree.

$T(N) = 2 * (2*T(N/2) + n/2) + n/2$

$= 4 * T(N/2) + 3(n/2)$

Solve this till T(1) i.e. till we reach the root.

$T(N) = c * T(N / 2^i) + (2*i - 1) * (n/2)$

Where i = lg(N)

$= lg((2n - 1) / 2)$

$O(c * T(N / 2^i) + (2*i - 1) * (n/2))$ reduces to

$O((2*i - 1) * (n/2))$

$O((2*(lg((2n - 1) / 2)) - 1) * (n/2))$ ....sub the value of i.

$O(n * ln(n))$

71.	Consider the following C program segment

```
struct CellNode{

    struct CellNode *leftChild;

    int element;.

    struct CellNode *rightChild;

    };


    int Dosomething(struct CellNode *ptr)

    {
```

```
                    int value = 0;

               if(ptr != NULL)

                    {     if(ptr -> leftChild != NULL)

                              value = 1 + DoSomething (ptr -> leftChild);

                    if(ptr -> rightChild != NULL)

                              value = max(value, 1 + DoSomething (ptr -> rightChild);

                    }

               }
```

The value returned by the function DoSomething when a pointer to the root of a non-empty tree is passed as argument is

(A) The number of leaf nodes in the tree
(B) The number of nodes in the tree
(C) The number of internal nodes in the tree
(D) The height of the tree

[D]Assuming that the function returns 'value'.The leaf nodes return 0. Each internal node adds 1 to the maximum returned value.

72. The elements 32, 15, 20, 30, 12, 25, 16 are inserted one by one in the given order into a maxheap. The resultant max heap is

[A]

(a) (b) (c)

(d) (e)

73.	Consider the label sequences obtained by the following pairs of traversals on a labeled binary tree. Which of these pairs identify a tree uniquely?

    (i)   Pre-order and post-order
    (ii)   Inorder and post-order
    (iii)   Pre-order and inorder
    (iv)   Level order and post-order

(A) (i)only
(B) (i), (iii)
(C) (iii)only
(D) (iv)only

(ii)   and (iii)

here [C] is also correct

74.	Level order traversal of a rooted tree can be done by starting the root and performing

(A) Pre-order traversal
(B) Inorder traversal
(C) Depth first search
(D) Breadth first search

[D] A breath first search is implemented by enqueueing the root and the repeating the following-

Step: dequeue a node and push its children in the queue from the left to right.

This will give a level-order traversal of the tree.

75.	The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)?

(A) 2
(B) 3
(C) 4
(D) 6

[A]
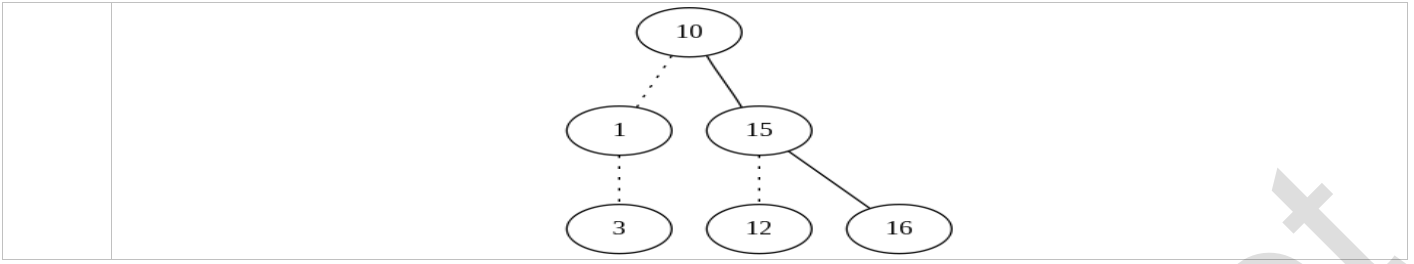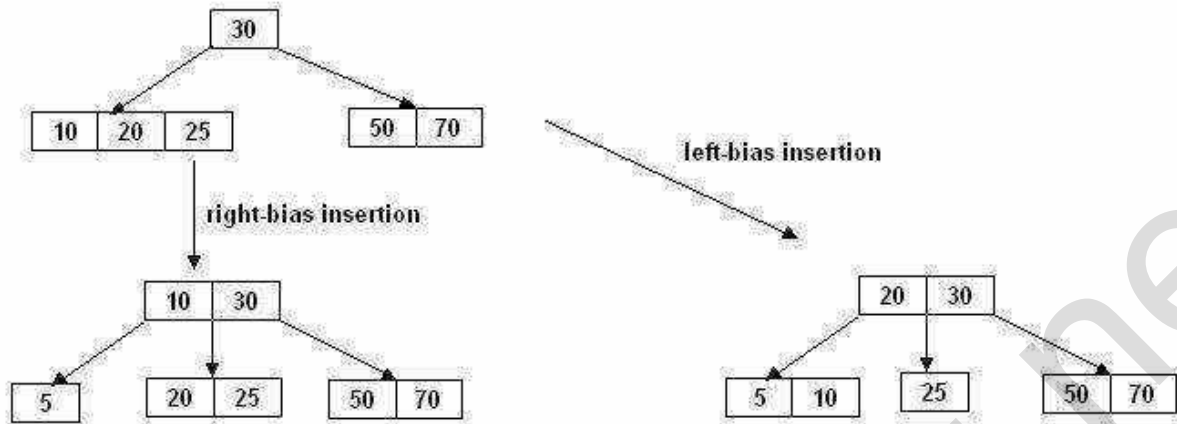
ADVANTAGES:
- B-trees are suitable for representing huge tables residing in secondary memory because:
    1. With a large branching factor m, the height of a B-tree is low resulting in fewer disk accesses.
    2. The branching factor can be chosen such that a node corresponds to a block of secondary memory.
    3. The most common data structure used for database indices is the B-tree. An **index** is any data structure that takes as input a property (e.g. a value for a specific field), called the search key, and *quickly* finds all records with that property.

Note: As m **increases the amount of computation at each node increases; however this cost is negligible compared to hard-drive accesses.**

Unit IV: Graphs: DEFINITION-REPRESENTATION OF GRAPH-TYPES OF GRAPH-BREADTH FIRST TRACERSAL-EPTH FIRST TRAVERSAL -TOPOLOGICAL SORT-BI-CONNECTIVITY-CUT VERTEX-EUCLER CIRCUITS-APPLICATIONS OF HEAP.

<u>**Graph:**</u> **-** A graph is data structure that consists of following two components.

 ¬ A finite set of vertices also called as nodes.

 ¬ A finite set of ordered pair of the form (u, v) called as edge.

(or)

A graph G=(V, E) is a collection of two sets V and E, where

V→ Finite number of vertices

E→ Finite number of Edges,

 Edge is a pair (v, w), where v, w ∈ V.

<u>Application of graphs:</u>

 ¬ Coloring of MAPS

 ¬ Representing network

  o Paths in a city

  o Telephone network o
  Electrical circuits etc.

 ¬ It is also using in social network
   including o LinkedIn
   o Facebook

<u>Types of Graphs:</u>

 ¬ Directed graph

 ¬ Undirected Graph

Directed Graph:

In representing of graph there is a directions are shown on the edges then that graph is called Directed graph.

That is,

A graph G=(V, E) is a directed graph ,Edge is a

pair (v, w), where v, w ∈ V, and the pair is ordered.
Means vertex 'w' is adjacent to v.

Directed graph is also called digraph.
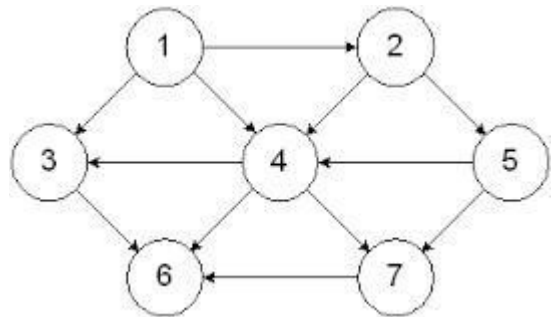
Undirected Graph:

In graph vertices are not ordered is called undirected graph. Means in which (graph) there is no direction (arrow head) on any line (edge).

A graph G=(V, E) is a directed graph ,Edge is a pair

(v, w), where v, w ∈ V, and the pair is not ordered.
Means vertex 'w' is adjacent to 'v', and vertex 'v' is adjacent to 'w'
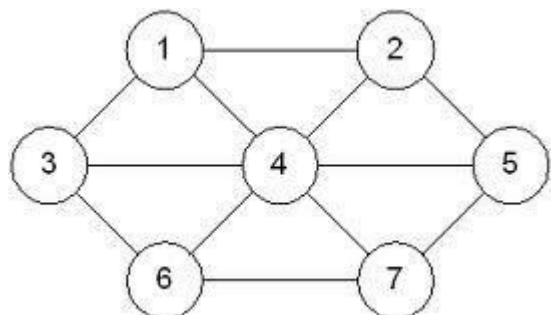
Note: in graph there is another component called weight/ cost.

Weight graph:
Edge may be weight to show that there is a cost to go from one vertex to another.
Example: In graph of roads (edges) that connect one city to another (vertices), the weight on the edge might represent the distance between the two cities (vertices).
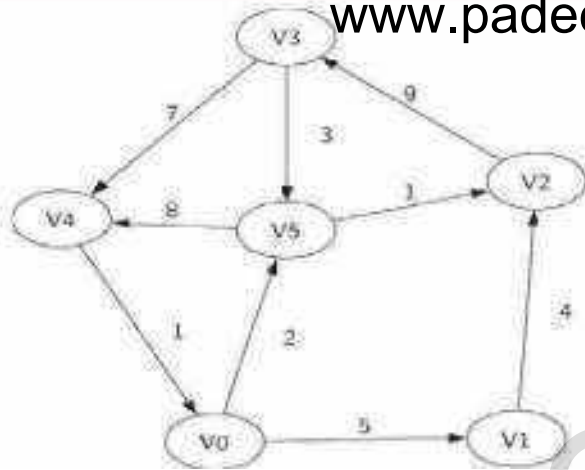
$$V = \{V0, V1, V2, V3, V4, V5\}$$

$$E = \left\{ \begin{array}{l} (v0,v1,5),(v1,v2,4),(v2,v3,9),(v3,v4,7),(v4,v0,1), \\ (v0,v5,2),(v5,v4,8),(v3,v5,3),(v5,v2,1) \end{array} \right\}$$

Difference between Trees and Graphs		
	Trees	**Graphs**
Path	Tree is special form of graph i.e. **minimally connected graph** and having only one path between any two vertices.	In graph there can be more than one path i.e. graph can have uni-directional or bi-directional paths (edges) between nodes
Loops	Tree is a special case of graph having no **loops**, no **circuits** and no self-loops.	Graph can have loops, circuits as well as can have **self-loops**.
Root Node	In tree there is exactly one root node and every **child** have only one **parent**.	In graph there is no such concept of **root** node.
Parent Child relationship	In trees, there is parent child relationship so flow can be there with direction top to bottom or vice versa.	In Graph there is no such parent child relationship.
Complexity	Trees are less complex then graphs as having no cycles, no self-loops and still connected.	Graphs are more complex in compare to trees as it can have cycles, loops etc
Types of Traversal	Tree traversal is a kind of special case of traversal of graph. Tree is traversed in **Pre-Order**, **In-Order** and **Post-Order**(all three in DFS or in BFS algorithm)	Graph is traversed by **DFS:** Depth First Search **BFS :** Breadth First Search **algorithm**
Connection Rules	In trees, there are many rules / restrictions for making connections between nodes through edges.	In graphs no such rules/ restrictions are there for connecting the nodes through edges.
DAG	Trees come in the category of **DAG : Directed Acyclic Graphs** is a kind of directed graph that have no cycles.	Graph can be **Cyclic or Acyclic**.
Different Types	Different types of trees are : **Binary Tree , Binary Search Tree, AVL tree, Heaps**.	There are mainly two types of Graphs :**Directed and Undirected graphs**.
Applications	Tree applications: sorting and searching like Tree Traversal & Binary Search.	Graph applications : Coloring of maps, in OR (**PERT & CPM**), algorithms, Graph coloring, job scheduling, etc.
No. of edges	Tree always has **n-1** edges.	In Graph, no. of edges depends on the graph.
Model	Tree is a **hierarchical model**.	Graph is a **network model**.
Figure		
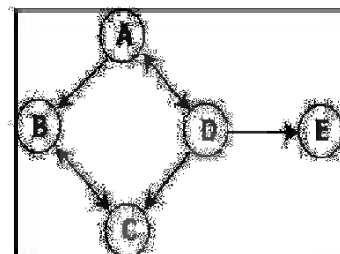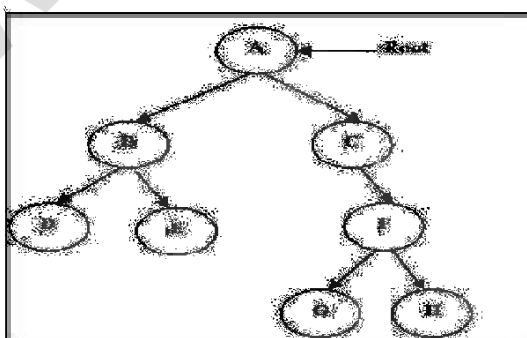
Other types of graphs:

Complete Graph:

A **complete graph** is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.

OR

If an undirected graph of n vertices consists of **n(n-1)/2** number of edges then the graph is called complete graph.

Example:

vertices	Edges	Complete graph	vertices	Edges	Complete graph
n=2			n=6	15	
n=3	3		n=7	21	
n=4	6		n=5	10	
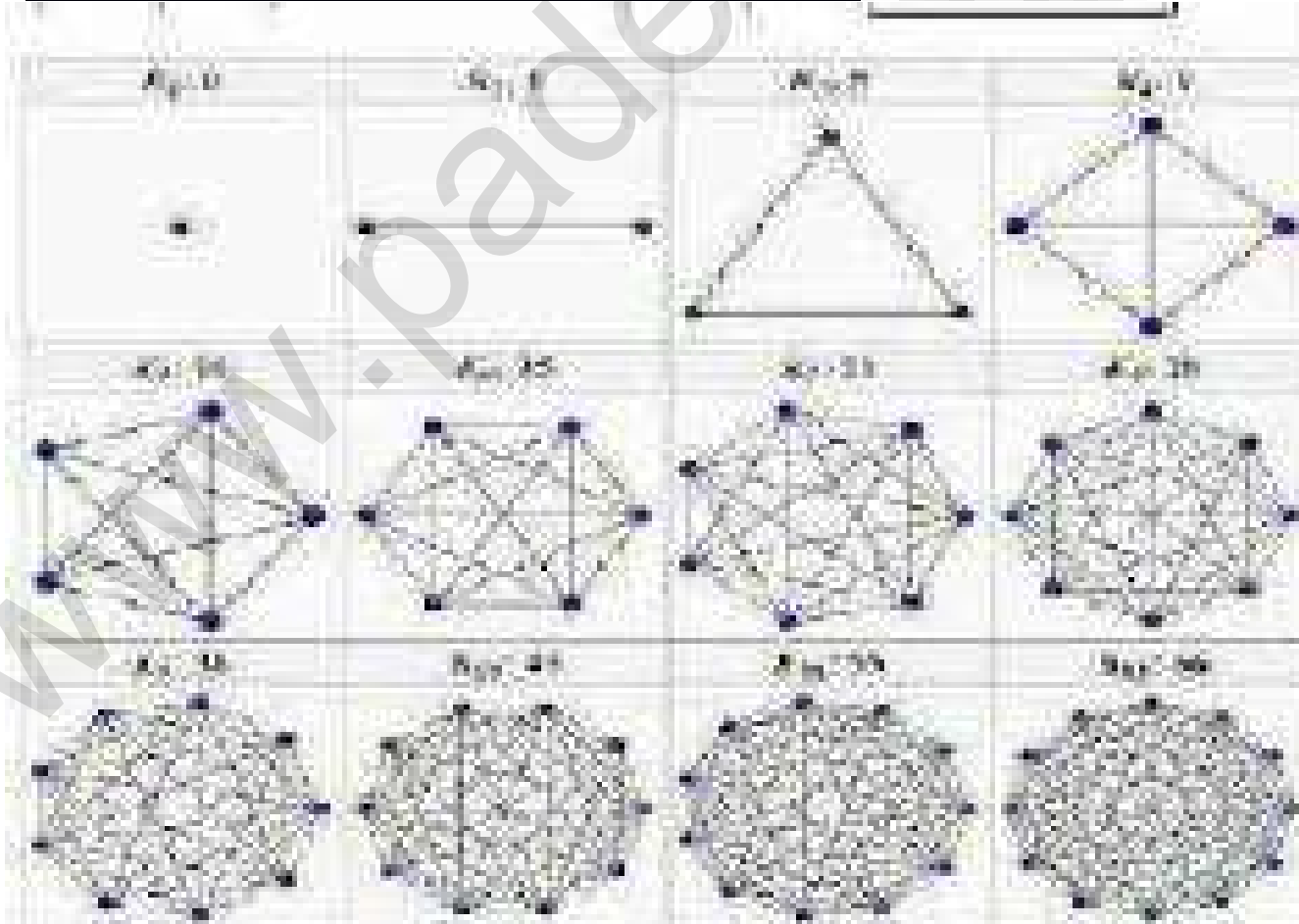
Sub graph:

A sub-graph G' of graph G is a graph, such that the set of vertices and set of edges of G' are proper subset of the set of vertices and set of edges of graph G respectively.
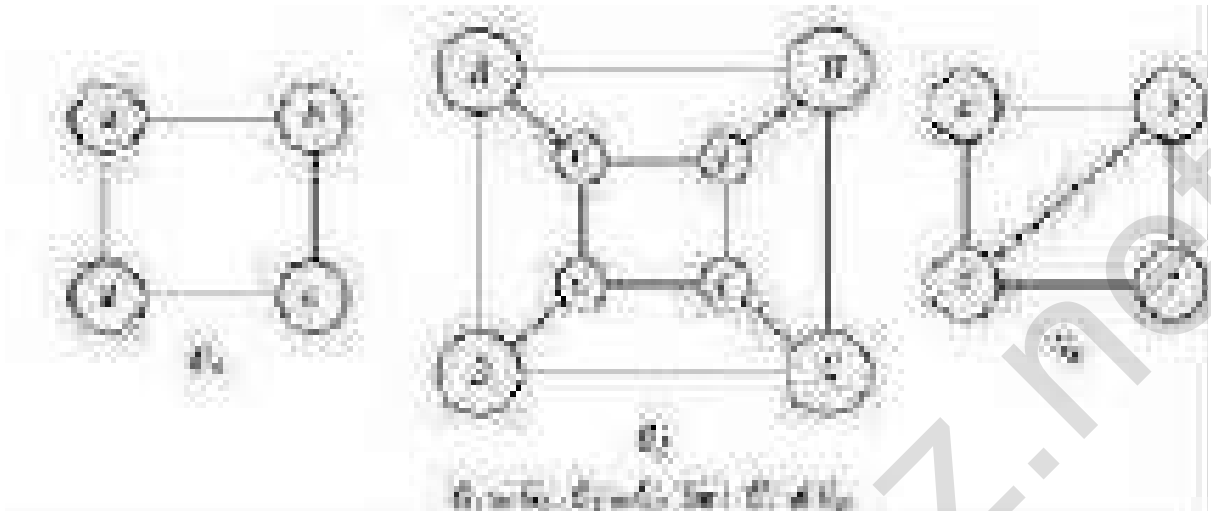
Connected Graph:

A graph which is connected in the sense of a topological space (study of shapes), i.e., there is a path from any point to any other point in the graph. A graph that is not connected is said to be disconnected.
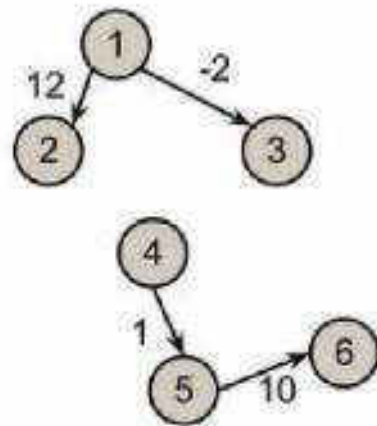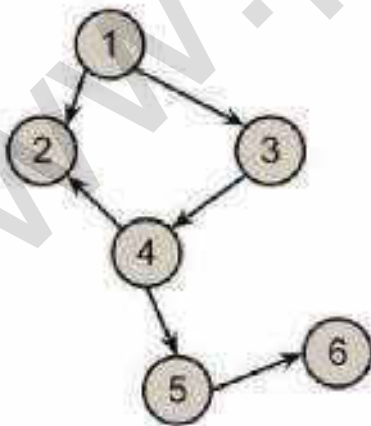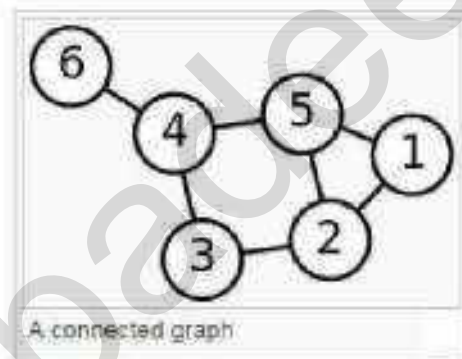
A connected graph

Connected

Disconnected

Path:

A path in a graph is a finite or infinite sequence of edges which connect a sequence of vertices. Means a path form one vertices to another vertices in a graph is represented by collection of all vertices (including source and destination) between those two vertices.

Cycle: A path that begins and ends at the same vertex.

Simple Cycle: a cycle that does not pass through other vertices more than once
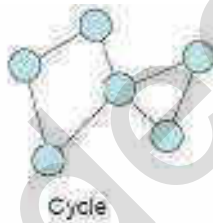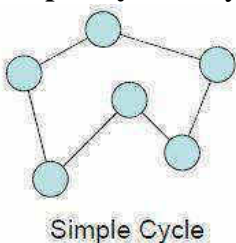
Degree:

The degree of a graph vertex v of a graph G is the number of graph edges which touch v. The vertex degree is also called the local degree or valency. Or

The degree (or valence) of a vertex is the number of edge ends at that vertex.
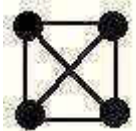
For example, in this graph all of the vertices have degree three.

In a digraph (directed graph) the degree is usually divided into the **in-degree** and the **out-degree**

- In-degree: The in-degree of a vertex v is the number of edges with v as their terminal vertex.
- Out-degree: The out-degree of a vertex v is the number of edges with v as their initial vertex.

TOPOLOGICAL SORT

A topological sort is a linear ordering of vertices in a **Directed Acyclic Graph** such that if there is a path from V_i to V_p, then V_j appears after V_i in the linear ordering.Topological sort is not possible if the graph has a cycle.

INTRODUCTION

- In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges.
- Every DAG has one or more topological sorts.
- More formally, define the partial order relation R over the nodes of the DAG such that xRy if and only if there is a directed path from x to y. Then, a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.

PROCEDURE

Step – 1 : Find the indegree for every vertex.

Step – 2 : Place the vertice whose indegree is 0, on the empty queue.

Step – 3 : Dequeue the vertex V and decrement the indegrees of all its adjacent vertices.

Step – 4 : Enqueue the vertex on the queue if its indegree falls to zero.

Step – 5 : Repeat from Step -3 until the queue becomes empty.

The topological ordering is the order in which the vertices dequeue.

Vertices	Indegree
1	0
2	0

GRAPH TRAVERSAL

Graph traversal is the Visiting all the nodes of a graph.

The traversals are :

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

BREADTH FIRST SEARCH

- The Breadth first search was one of the systematic approaches for exploring and searching the vertices/nodes in a given graph. The approach is called "breadth-first" because from each vertex 'v' that we visit, we search as broadly as possible by next visiting all the vertices adjacent to v.
- It can also be used to find out whether a node is reachable from a given node or not.
- It is applicable to both directed and undirected graphs.
- Queue is used in the implementation of the breadth first search.

ALGORITHM

```
Procedure bfs()
{
    //BFS uses Queue data structure
    Queue q=new LinkedList();
    q.add(this.rootNode);
    printNode(this.rootNode);
    rootNode.visited=true;
    while(!q.isEmpty())
    {
        Node n=(Node)q.remove(); Node child=null;
        while((child=getUnvisitedChildNode(n))!=null)
        {
            child.visited=true;
            printNode(child);
            q.add(child);
        }
    }
    //Clear visited property of nodes
    clearNodes();
}
```

Procedure

Step -1   Select the start vertex/source vertex. Visit the vertex and mark it
          as one (1) (1 represents visited vertex).

Step -2   Enqueue the vertex.

Step -3   Dequeue the vertex.

Step -4   Find the Adjacent vertices.

Step -5   Visit the unvisited adjacent vertices and mark the distance as 1.

Step -6   Enqueue the adjacent vertices.

Step -7   Repeat from Step – 3 to Step – 5 until the queue becomes empty.

○   ○

Ex : A ———— B
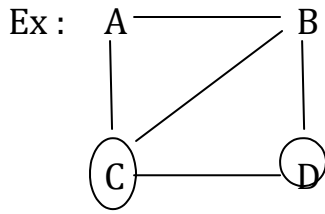
Vertices	Visited Vertices			
A	1			
B	~~0~~	1		
C	~~0~~	1		
D	~~0~~	1		
Enqueue	~~A~~	B	~~C~~	~~D~~
Dequeue	A	B	C	D
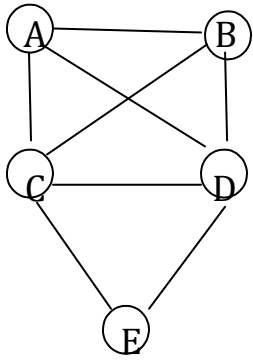
Vertices	Visited Vertices			
C	1			
A	~~0~~	1		
B	~~0~~	1		
D	~~0~~	1		
E	~~0~~	1		
Enqueue	~~C~~	~~A~~	~~B~~	~~D~~ ~~E~~
Dequeue	C	A	B	D E

Pseudo Code :

void BFS(Vertex S)

{

Vertex v,w;

Queue Q;

visited[s] = 1; enqueue(S,Q);

while(!IsEmpty(a))

{

```
v = Dequeue(Q);

print(v);

for each adjacent vertices w to v

if(visited[w]==0)

{

visited[w] = 1;

Enqueue(w,Q);

}

}

}
```

APPLICATIONS

Breadth-first search can be used to solve many problems in graph theory, for example.

- Finding all nodes within one connected component
- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes u and v (in an unweighted graph)
- Finding the shortest path between two nodes u and v (in a weighted graph: see talk page)
- Testing a graph for bipartiteness
- (Reverse) Cuthill–McKee mesh numbering
- Testing whether graph is connected.
- Computing a spanning forest of graph.
- Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists.
- Computing a cycle in graph or reporting that no such cycle exists.

Uses

- Testing whether graph is connected.
- Computing a spanning forest of graph.
- Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists.
- Computing a cycle in graph or reporting that no such cycle exists.

Depth first search	Breadth first search
1. Backtracking is possible from a dead end	1. Backtracking is not possible.
2. Vertices from which exploration is incomplete are processed in a LIFO order.	2. The vertices to be explored are organized as a FIFO queue.
3. Search is done in one particular direction at the time.	3. Search is done parallely in all possible direction.

DEPTH FIRST SEARCH

- Depth first search works by taking a node, checking its neighbors, expanding the first node it finds among the neighbors, checking if that expanded node is our destination, and if not, continue exploring more nodes.
- In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it.
- When all of v' s edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source.

- This entire process is repeated until all vertices are discovered.
- Stack is used in the implementation of the depth first search.

In DFS, the basic data structures for storing the adjacent nodes is stack.

Procedure:

Step -1   Select the start vertex.

Step -2   Visit the vertex.

Step -3   Push the vertex on to the stack.

Step -4   Pop the vertex.

Step -5   Find the adjacent vertices, and select any 1 of them.

Step -6   Repeat from Step – 4 to Step – 5 until the stack becomes empty.
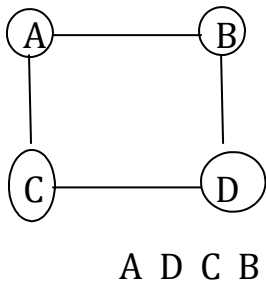
A D C B

Vertices	Visited Vertices
A	1
B	0 1
C	0 1
D	0 1

Stack   O/P

A

D

C

B
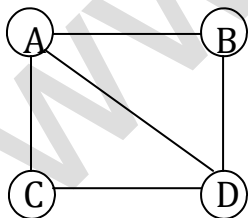
Vertices	Visited Vertices
A	1
B	0 1
C	0 1
D	0 1

Stack o/p   A B D C

Pseudo Code:

```
void DFS(Vertex S)

{

int top = 0;

Stack C;  vertex v,w;

visited [S] = 1

C[top] = S;

while(!IsEmpty(C))

{

v = pop(C);

print(V);

for each unknown adjacent vertices of V,

{

if(visited[w] == 0)

{

visited[w] = 1;

push(w,C);

}

}

}

}
```

(OR)

```
void DFS(vertex V)

{

visited[v] = 1;

push(v);

pop(v);

for each w adjacent to V

if(!visited[w])

Dfs(w);

}
```

ALGORITHM

```
Procedure dfs()
{
    //DFS uses Stack data structure
    Stack s=new Stack();
    s.push(this.rootNode);
    rootNode.visited=true;
    printNode(rootNode);
    while(!s.isEmpty())
    {
        Node n=(Node)s.peek();
        Node child=getUnvisitedChildNode(n);
        if(child!=null)
        {
            child.visited=true;
            printNode(child);
            s.push(child);
        }
        else
        {
            s.pop();
```

```
        }
    }
    //Clear visited property of nodes
    clearNodes();
}
```

Applications of DFS :

1. To check whether the undirected graph is connected or not.
2. To check if the connected undirected graph is bi – connected or not.
3. To check whether the directed graph is acyclic or not.

I.   **Undirected Graph :**

An undirected graph is connected if and only if a depth first search starting from any node visits every node.
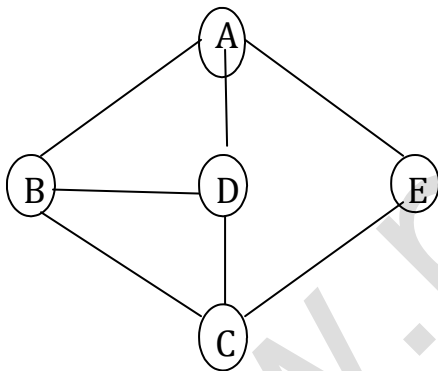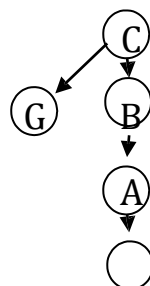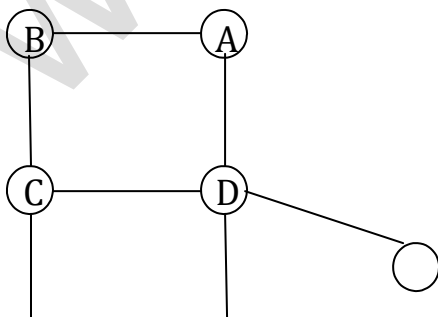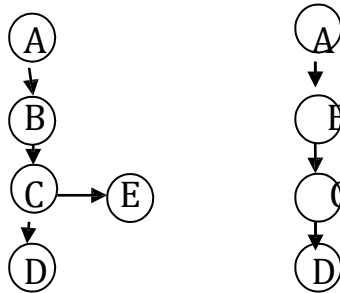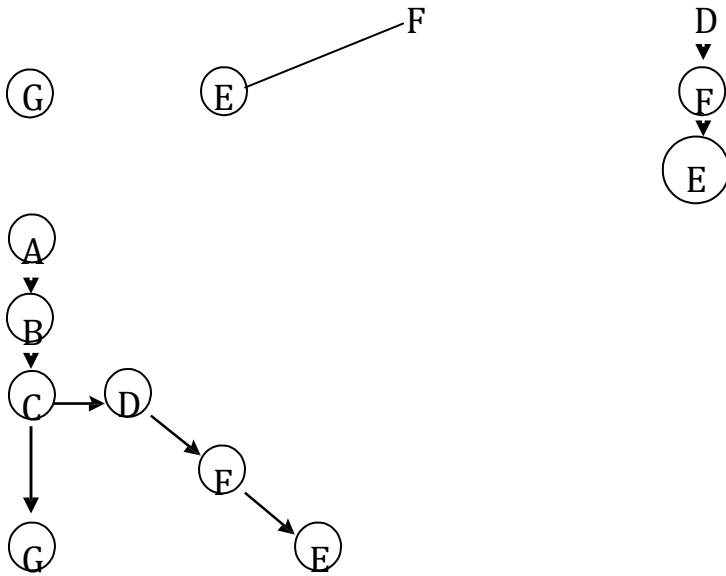
II.   Bi-Connectivity :

The vertices which are responsible for disconnection is called as Articulation points.

If in a connected undirected graph, the removal of any node does not affect the connectivities, then the graph is said to be biconnected graph.

1. Num $\boxed{Low(w) \geq Num(v)}$
2. Low

Calculation of Num and Low gives the articulation point.

F

G   E

D   (4,1)

E   (5,4)

F   (6,4)
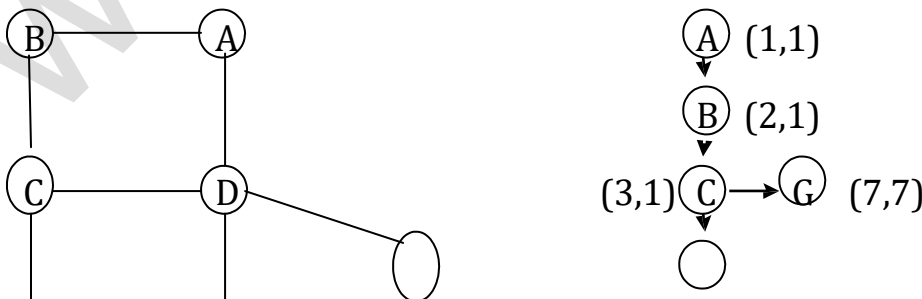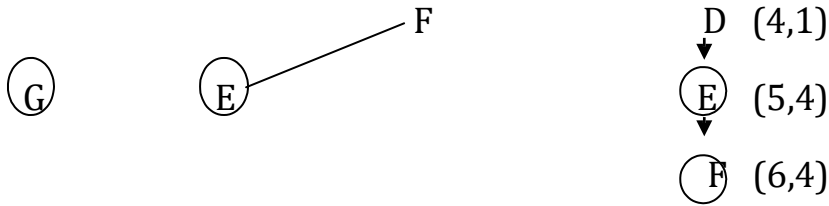
BICONNECTIVITY :

A connected undirected graph is biconnective if there are no vertices whose removal disconnects the rest of the graph.

- A biconnected undirected graph is a connected graph that is not broken into disconnected pieces by deleting any single vertex (and its incident edges).
- A biconnected directed graph is one such that for any two vertices v and w there are two directed paths from v to w which have no vertices in common other than v and w.
- If a is not bio-connected,the vertices whose removal would disconnect the graph is called articulation points.

DEFINITION

Equivalent definitions of a biconnected graph G:

- Graph G has no separation edges and no separation vertices
- For any two vertices u and v of G, there are two disjoint simple paths between u and v (i.e., two simple paths between u and v that share no other vertices or edges)
- For any two vertices u and v of G, there is a simple cycle

containing u and v.

Biconnected Components

Biconnected component of a graph G are:

- A maximal biconnected subgraph of G, or
- A subgraph consisting of a separation edge of G and its end vertices

Interaction of biconnected components

- An edge belongs to exactly one biconnected component
- A nonseparation vertex belongs to exactly one biconnected component
- A separation vertex belongs to two or more biconnected components

Articulation Point :

The vertices whose removal disconnects the graph are known as Articulation Points.

Steps to find Articulation Points :

(i)   Perform DFS, starting at any vertex.

(ii)   Number the vertex as they are visited as Num(V).

(iii)   Compute the lowest numbered vertex for every vertex V in the DFS tree, which we call as low(W), that is reachable from V by taking one or more tree edges and then possible one back edge by definition.

$$Low(V) = min(Num(V), Num(W), Low(W))$$

The Lowest Num(W) among all back edges V, W.

The Lowest Low(W) among all the tree edges V, W.

The root is an articulation if and only if (iff) it has more than two children.

Any vertex V other than the root is an Articulation point iff V has some child W such that **Low(W)≥Num (V)**
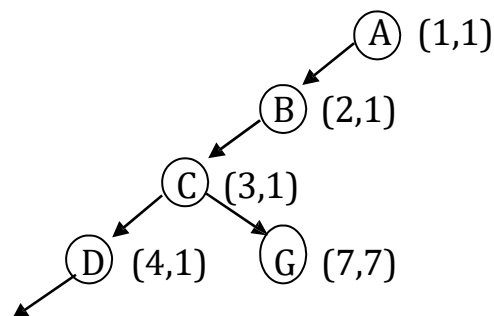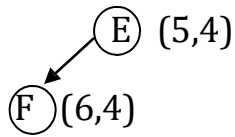
$$\text{Low(F)} = \min(\text{Num(V)}, \text{Num(W)}, \text{Low(W)})$$

$$= \min(6, 4, -1)$$

$$= 4$$

$$\text{Low(E)} = \min(5, 6, 4)$$

$$= 4$$

$$\text{Low(D)} = \min(4, 1, 4)$$

$$= 1$$

$$\text{Low(G)} = \min(7, -, -)$$

$$= 7$$

$$\text{Low(C)} = \min(3, (4,7), (1,7))$$

$$= 1$$

$$\text{Low(B)} = \min(2, 3, 1)$$

$$= 1$$

$$\text{Low(A)} = \min(1, 2, 1)$$

$$= 1$$

Example 2
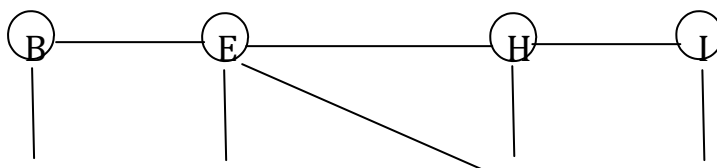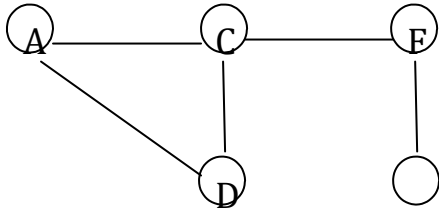
G
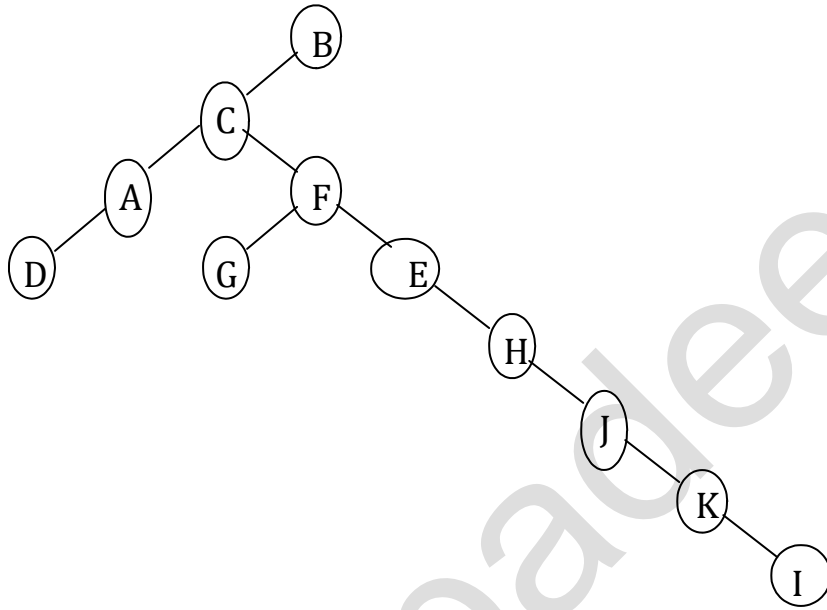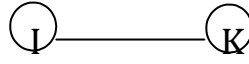
Visited

1	2	3	4
1	1	1	1
A	B	C	D

Parent[]

1	1	1	1
0	1	2	3

0	1	2	3
1	1	1	1
A	B	C	D

Num[]
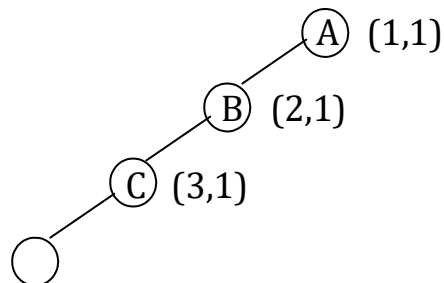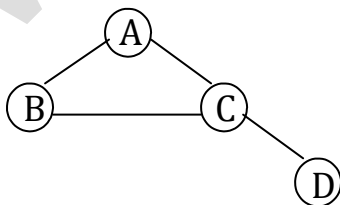
D   (4,4)

ALGORITHM

Routine to assign Num to Vertices

```
void AssignNum(Vertex V)
{
Vertex W;
int counter = 0;
Num[V] = ++counter;
visited[V] = True;
for each W adjacent to V
if(!visited[W])
{
parent[W] = V;
AssignNum(W);
}
}
```

Routine to compute low and to test for points

```
 void AssignLow(Vertex V)
{
Vertex W;
Low[V] = Num[V];                        /* Rule 1 */
for each W adjacent to V
{
if(Num[W] > Num[V])                      /* forward edge or free edge */
{ AssignLow(W)
if(low[W] >= Num[V])
printf("%v Articulation point is", V);
Low[V] = min(Low[V], Low[W]);           /* Rule 3 */
}
else
{
if(parent[V]! = W)              /* Back edge */
Low[V] =  min(Low[V], Num[W]);    /* Rule 2 */
```

```
}
}
}
```

APPLICATION

- Bio-Connectivity is a application of depth first search.
- Used mainly in network concepts.

BICONNECTIVITY ADVANTAGES

- Total time to perform traversal is m*inimum.*
- Adjacency lists are used
- Traversal is given by O(E+V).

DISADVANTAGES

- Have to be careful to avoid cycles
- Vertices should be carefully removed as it affects the rest of the graph.

EULER CIRCUIT

EULERIAN PATH

An Eulerian path in an undirected graph is a path that uses each edge exactly once. If such a path exists, the graph is called traversable or semi-eulerian.

EULERIAN CIRCUIT

An Eulerian circuit or Euler tour in an undirected graph is a cycle that uses each edge exactly once. If such a cycle exists, the graph is called

Unicursal. While such graphs are Eulerian graphs, not every Eulerian graph possesses an Eulerian cycle.
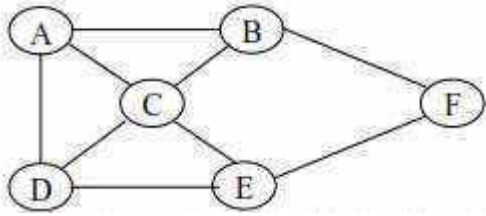
EULER'S THEOREM

Euler's theorem 1

- If a graph has any vertex of odd degree then it cannot have an Euler circuit.
- If a graph is connected and every vertex is of even degree, then it at least has one Euler   circuit.

Euler's theorem 2

- If a graph has more than two vertices of odd degree then it cannot have an Euler path.
- If a graph is connected and has just two vertices of odd degree, then it at least has one Euler path. Any such path must start at one of the odd-vertices and end at the other odd vertex.

ALGORITHM

Fleury's Algorithm for finding an Euler Circuit

1. Check to make sure that the graph is **connected and all vertices are of even degree**
2. Start at any vertex
3. Travel through an edge:
    - If it is **not a bridge for the untraveled part**, or
    - there is no other alternative
4. Label the edges in the order in which you travel them.
5. When you cannot travel any more, stop.

Fleury's Algorithm

1. pick any vertex to start .
2. from that vertex pick an edge to traverse .
3. darken that edge, as a reminder that you can't traverse it again .
4. travel that edge, coming to the next vertex .
5. repeat 2-4 until all edges have been traversed, and you are back at the starting vertex .

At each stage of the algorithm:

- the original graph minus the darkened (already used) edges = **reduced graph**
- **important rule**: *never cross a bridge* of the reduced graph *unless there is no other choice*

Note:

- the same algorithm works for *Euler paths*
- before starting, use Euler's theorems to check that the graph has an Euler path and/or circuit to find.

APPLICATION

Eulerian paths are being used in bioinformatics to reconstruct the DNA SEQUENCE from its fragments.

IMPORTANT SUMS

1. **Topological Sort**

Vertices	Indegree
V_1	0
V_2	1 0
V_3	1 0̶ 0
V_4	1 0
V_5	1 0
V_6	2 1 0
V_7	2 1 0
V_8	1 0
V_9	1 0
V_{10}	1 0
Enqueue	V̶₁̶ V̶₂̶ V̶₄̶ V̶₃̶ V̶₅̶ V̶₁̶₀̶ V̶₆̶ V̶₉̶ V̶₈̶ V̶₇̶
Dequeue	V_1 V_2 V_4 V_3 V_5 V_{10} V_6 V_9 V_8 V_7

Sorted Vertices :

V_1 V_2 V_4 V_3 V_5 V_{10} V_6 V_9 V_8 V_7
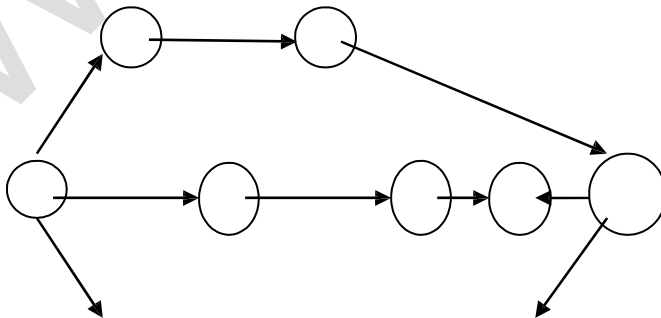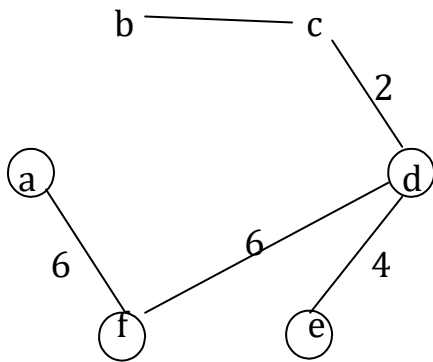
5. Write the pseudo code for Sorting algorithms

Pseudo Code for Topological Sort :

void Topsort(Graph G)

{

Queue Q;

vertex v,w; int counter = 0;

Q = Create Queue (Num Vertex);

Make Empty(Q);

for each vertex V

if (Indegree[V]==0)

      enqueue(V,Q);

while(!Isempty(Q))

{

V = Dequeue(Q);

Topnum[V]=++Counter;

for each w adjacent to V

```
if(--Indegree[w]==0)

Enqueue(w,0);

}

if(counter!=Num Vertex)

Error("Graph has a cycle");

Dispose Queue(Q);

}
```

Pseudo Code to Perform Unweighted Sort :

```
void unweighted(Table T)

{

Queue Q;

Vertex v,w;

Q = Create Queue(Num Vertex);

Make Empty(Q);

while(!IsEmpty(Q))

{

V = Dequeue(o);

T[V].Known = True;

for each w adjacent to v

if(!T[w].Known)

{
```

1. (a) How many binary relations are there on a set A with n elements?
   (b) How many one-to-one functions are there from a set A with n elements onto itself?
   (c) Show that the number of odd degree vertex in a finite graph is even.
   (d) Specify an adjacency-lists representation of the undirected graph.
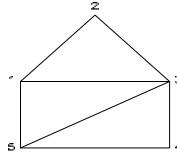
2. Answer the following question briefly.
   (i) If the transportation problem is solved using some version of the simplex algorithm, under what conditions will the solution always have integer values?

3. Write the adjacency matrix representation of the graph given in fig below:
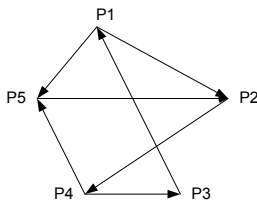
4. In the graph shown below, the depth-first spanning tree edges are marked with 'T'. Identify the forward, backward and cross edges.
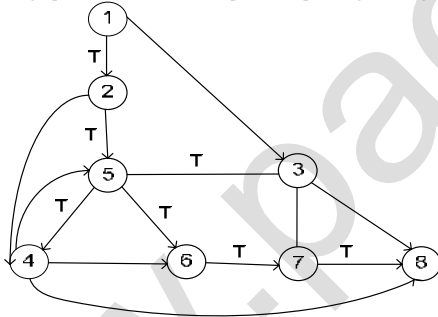
5. The maximum number of possible edges in an undirected graph with n vertices and k components is _____.

6. Show that all vertices in an undirected finite graph cannot have distinct degrees, if the graph has at least two vertices

7. Maximum number of edges in a planner graph with n vertices is _____.

8. Consider a simple connected graph G with n vertices and n edges (n>2). Then which of the following statements are true

   (a) G has no cycles
   (b) the graph obtained by removing any edges from G is not consider connected
   (c) G has at least one cycle
   (d) the graph obtained by removing any two edges from G is not consider connected
   (e) none of the above.

9. **Consider the following graph:**
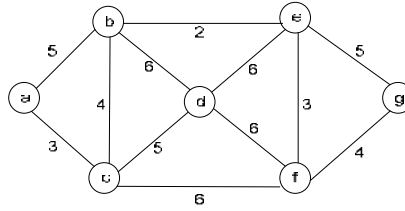
Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm?

(A) (b, e) (e, f) (a, c) (b, c) (f, g) (c, d)
(B) (b, e) (e, f) (a, c) (f, g) (b, c) (c, d)
(C) (b, e) (a, c) (e, f) (b, c) (f, g) (c, d)
(D) (b, e) (e, f) (b, c) (a, c) (f, g) (c, d)

10. Which one of the following is TRUE for any simple connected undirected graph with more that 2 vertices?

(A) No two vertices have the same degree.
(B) At least two vertices have the same degree.
(C) At least three vertices have the same degree.
(D) All vertices have the same degree.

11. Consider the following sequences of nodes for the undirected graph given below.
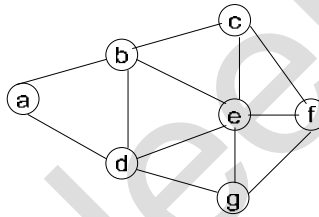
I.   a b e f d g c
II. a b e f c g d
III. a d g e b c f
IV. a d b c g e f

A Depth First Search (DFS) is started at node *a*. The nodes are listed in the order they are first visited. Which all of the above is (are) possible output(s)?

(A) I and III only
(B) II and III only
(C) II, III and IV only
(D) I, II and III only

12. For the undirected, weighted graph given below, which of the following sequences of edges represents a correct execution of Prim's algorithm to construct a Minimum Spanning Tree?
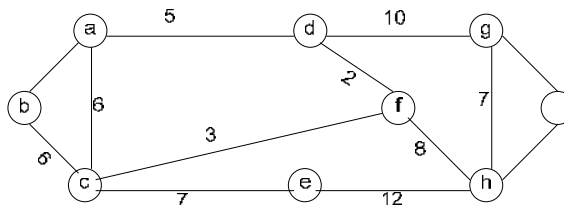
(D) (a, b), (d, f), (f, c), (g, i), (d, a), (g, h), (c, e), (f, h)
(E) (c, e), (c, f), (f, d), (d, a), (a, b), (g, h), (h, f), (g, i)
(F) (d, f), (f, c), (d, a), (a, b), (c, e), (f, h), (g, h), (g, i)
(G) (h, g), (g, i), (h, f), (f, c), (f, d), (d, a), (a, b), (c, e)

13. G is a simple undirected graph. Some vertices of G are of odd degree. Add a node v to G and make it adjacent to each odd degree vertex of G. The resultant graph is sure to be

    (A) regular
    (B) complete
    (C) Hamiltonian
    (D) Euler

14. What is the chromatic number of the following graph?
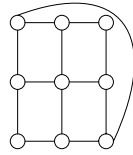
    (A) 2
    (B) 3
    (C) 4
    (D) 5

15.
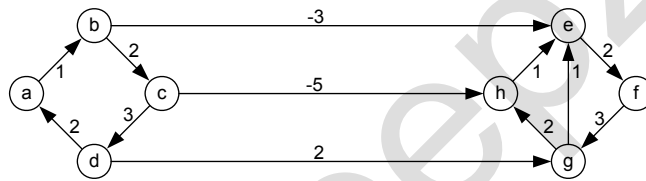
Dijkstra's single source shortest path algorithm when run from vertex a in the above graph, computes the correct shortest path distance to

    (A) only vertex a
    (B) only vertices a, e, f, g, h
    (C) only vertices a, b, c, d
    (D) all the vertices

16. G is a graph on n vertices and $2n-2$ edges. The edges of G can be partitioned into two edge-disjoint spanning trees. Which of the following is NOT true for G?

    (A) For every subset of k vertices, the induced subgraph has at most $2k-2$ edges
    (B) The minimum cut in G has at least two edges
    (C) There are two edge-disjoint paths between every pair of vertices
    (D) There are two vertex-disjoint paths between every pair of vertices

17. Which of the following statements is true for every planar graph on n vertices?

    (A) The graph is connected
    (B) The graph is Eulerian
    (C) The graph has a vertex-cover of size at most $3n/4$
    (D) The graph has an independent set of size at least $n/3$

18. The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is
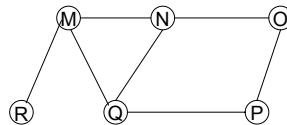
    (A) MNOPQR
    (B) NQMPOR
    (C) QMNPRO
    (D) QMNPOR

7

19. The most efficient algorithm for finding the number of connected components in an undirected graph on n vertices and m edges has time complexity

    (A) Θ (n)
    (B) Θ (m)
    (C) Θ (m + n)
    (D) Θ (mn)

20. What is the largest m such that every simple connected graph with n vertices and n edges contains at least m different spanning trees?

    (A) 1 (B) 2   (C) 3 (D) n

21. Consider the DAG with V = {1, 2, 3, 4, 5, 6}, shown below.
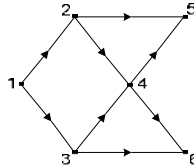
    Which one the following is **NOT** a topological ordering?

    (A) 1 2 3 4 5 6   (B) 1 3 2 4 5 6

    (C) 1 3 2 4 6 5   (D) 3 2 4 1 6 5

22. Let G be the non-planar graph with the minimum possible number of edges. Then G has

    (A) 9 edges and 5 vertices
    (B) 9 edges and 6 vertices
    (C) 10 edges and 5 vertices
    (D) 10 edges and 6 vertices

23. Consider the depth first search of an undirected graph with 3 vertices P, Q and R. Let discovery time $d(u)$ represent the time instant when the vertex u is first visited, and finis time $f(u)$ represent the time instant when the vertex u is last visited. Given that

    $d(P)$ = 5 units $f(P)$ = 12 units

    $d(Q)$ = 6 units $f(Q)$ = 10 units

    $d(R)$ = 14 units $f(R)$ = 18 units

    Which of the following statements is TRUE about the graph?

    (A) There is only one connected component
    (B) There are connected components, and P and R are connected
    (C) There are connected components, and Q and R are connected
    (D) There are connected components, and P and Q are connected

24. Which of the following is the correct decomposition of the directed graph given below into its strongly connected components?
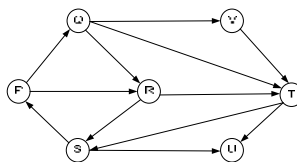
    (A) {P, Q, R, S}, {T}, {U}, {V}   (B) {P, Q, R, S, T, V}, {U}

    (C) {P, Q, S, T, V}, {R}, {U}   (D) {P, Q, R, S, T, U, V}

25. Consider the undirected graph G defined as follows. The vertices of G are bit strings of length n. we have an edge between vertex u and vertex v if and only if u and v differ in exactly one bit position (in other words, v can be obtained from u by flipping a single bit). The ratio of the chromatic number of G to the diameter of G is

(A) $1/2^{n-1}$ (B) $1/n$ (C) $2/n$ (D) $3/n$

26. If all the edge weights of an undirected graph are positive, then any subset of edges that connects all the vertices and has minimum total weight is a

(A) Hamiltonian cycle (B) grid   (C) hypercube   (D) tree

27. In a binary tree, the number of internal nodes of degree 1 is 5, and the number of internal nodes of degree 2 is 10. The number of leaf nodes in the binary tree is

(A) 10   (B) 11   (C) 12 (D) 15

28. The 2^n vertices of a graph G corresponds to all subsets of size n, for $n \geq 6$. Two vertices of G are adjacent if and only if the corresponding sets intersect in exactly two elements.

The number of connected components in G is

(A) 2   (B) n +2   (C) $2^{n/2}$ (D) $\dfrac{2^n}{n}$

The maximum degree of a vertex in G is

(A) $\binom{n/2}{2} 2^{n/2}$ (B) 2^{n-2} (C) $2^{n-3} \times 3$ (D) 2^{n-1}

The number of vertices of degree zero in G is

(A) 1   (B) n   (C) n + 1   (D) 2^n

29. Let *T* be a depth first search tree in an undirected graph *G*. Vertices u and v are leaves of this tree *T*. The degrees of both u and v in *G* are at least 2. Which one of the following statements is true?

(A) There must exist a vertex w adjacent to both u and v in G
(B) There must exist a vertex w whose removal disconnects u and v in G
(C) There must exist a cycle in G containing u and v
(D) There must exist a cycle in G containing u and v and all its neighbours in G

30. Consider the following graph
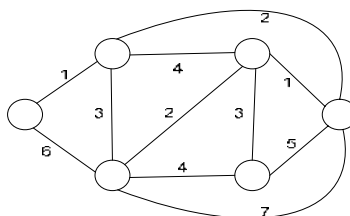
Which one of the following cannot be the sequence of edges added, in that order, to a minimum spanning tree using Kruskal's algorithm?

(A) (a-b), (d-f), (b-f), (d-c), (d-e)   (B) (a-b), (d-f), (d-c), (b-f), (d-e)

(C) (d-f), (a-b), (d-c), (b-f), (d-e)   (D) (d-f), (a-b), (b-f), (d-e), (d-c)

31.   To implement Dijkstra's shortest path algorithm on undirected graphs so that it runs in linear time, the data structure to be used is

(A) Queue   (B) Stack   (C) Heap   (D) B – Tree

32.   Consider weighted complete graph G on the vertex set $\{v_1, v_2, ..., v_n\}$ such that the weight of the edge (v_i, v_j) is $2\left|i - j\right|$. The weight of a minimum spanning tree of G is

(A) n – 1   (B) 2n – 2   (C) $\binom{n}{2}$ (D) n^2

33.   Let G be a directed graph whose vertex set is the set of number from 1 to 100. There is an edge from a vertex i to a vertex j iff either j = i + 1 or j = 3i. the minimum number of edges in a path in G from vertex 1 to 100 is

(A) 4   (B) 7   (C) 23   (D) 99

34.   Let G be a weighted undirected graph and e be an edge with maximum weight in G. Suppose there is minimum weight spanning tree in G containing edge e. which of the following statements is always true?

    (A)   There exists a cutest in G having all edges of maximum weight.
    (B)   There exists a cycle in G having all edges of maximum weight.
    (C)   Edge e can be contained in a cycle.
    (D)   All edges in G have same weight.

35.   In the following table, the left column contains the names of standard graph algorithms and the ight column contains the time complexities of the algorithms. Match each algorithm with its time complexity.

1: Bellman-Ford algorithm	A: $O(m \log n)$
2: Kruskal's algorithm	B: $O(n^3)$
3: Floyd-Warshall algorithm	C: $O(nm)$
4: Topological sorting	D: $O(n + m)$

    (A)   1→ C, 2→ A, 3→ B, 4→ D
    (B)   1→ B, 2→ D, 3→ C, 4→ A
    (C)   1→ C, 2→ D, 3→ A, 4→ B
    (D)   1→ B, 2→ A, 3→ C, 4→ D

36.   In depth first traversal of a graph G with n vertices, k edges are marked as tree edges. The number of connected components in G is

(A) k   (B) k + 1   (C) n – k – 1   (D) n – k

37.   **Statement for Linked Answer Questions 82a & 82b:**

    Let s and t be two vertices in a undirected graph G = (V, E) having distinct positive edge weights. Let [X, Y] be partition of V such that s Є X and t Є Y. Consider the edge e having the minimum weight amongst all those edges that have one vertex in X and one vertex in Y.

    (a)   The edge e must definitely belong to:

        (A)   The minimum weighted spanning tree of G
        (B)   The weighted shortest path from s to t
        (C)   Each path from s to t
        (D)   The weighted longest path from s to t

(b) Let the weight of an edge e denote the congestion on that edge. the congestion on a path is defined to be the maximum of the congestions on the edges of the path. We wish to find the path from s to t having minimum congestion. Which of the following paths is always such a path of minimum congestion?

(A) A path from s to t in the minimum weighted spanning tree
(B) A weighted shortest path from s to t
(C) An Euler walk from s to t
(D) A Hamiltonian path from s to t

38. Postorder traversal of a given binary search tree, T produces the following sequence of keys

10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29

Which one of the following sequences of keys can be the result of an inorder traversal of tree T?

(A) 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95
(B) 9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29
(C) 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95
(D) 95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29

39. Let G be a simple connected graph with 13 vertices and 19 edges. Then, the number of faces in the planar embedding of the graph is:

(A) 6
(B) 8
(C) 9
(D) 13

40. An undirected graph G has n nodes. its adjacency matrix is given by an n × n square matrix whose (i) diagonal elements are 0's and (ii) non-diagonal elements are 1's. Which one of the following is TRUE?

(A) Graph G has no minimum spanning tree (MST)
(B) Graph G has a unique MST of cost n-1
(C) Graph G has multiple distinct MSTs, each of cost n-1
(D) Graph G has multiple minimum spanning trees of different costs.

41. Consider the undirected graph below.
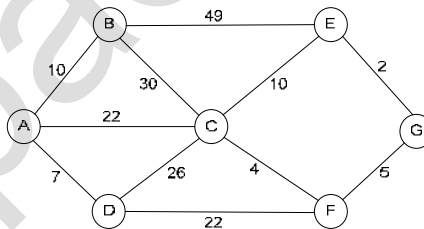
Using Prim's algorithm to construct a minimum spanning tree starting with node A, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree?

(A) (E, G), (C, F), (F, G), (A, D), (A, B), (A, C)
(B) (A, D), (A, B), (A, C), (C, F), (G, E), (F, G)
(C) (A, B), (A, D), (D, F), (F, G), (G, E), (F, C)
(D) (A, D), (A, B), (D, F), (F, C), (F, G), (G, E)

42. What is the number of vertices in an undirected connected graph with 27 edges, 6 vertices of degree 2, 3 vertices of degree 4 and remaining of degree 3?

(A) 10
(B) 11
(C) 18
(D) 19

43. What is the number of edges in an acyclic undirected graph with n vertices?

(A) n – 1

(B) n
(C) n + 1
(D) 2n – 2

44. Let $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ be connected graphs on the same vertex set V with more than two vertices. If $G_1 \cap G_2 = (V, E_1 \cap E_2)$ is not a connected graph, then the graph $G_1 \cup G_2 = (V, E_1 \cup E_2)$

(A) Cannot have a cut vertex
(B) Must have a cycle
(C) Must have a cut-edge (bridge)
(D) Has chromatic number strictly greater than those of G_1 and G_2

45. How many graphs on n labeled vertices exist which have at least $(n^2 - 3n)/2$ edges?

(A) $^{(n^2-n)/2}C_{(n^2-3n)/2}$

(B) $\sum_{k=0}^{(n^2-3n)/2} {}^{(n^2-n)}C_k$

(C) $^{(n^2-n)/2}C_n$

(D) $\sum_{k=0}^{n} {}^{(n^2-n)/2}C_k$

46. The minimum number of colours required to colour the following graph, such that no two adjacent vertices are assigned the same colour, is
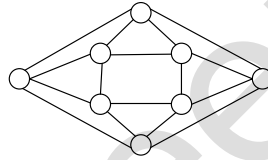
(A) 2
(B) 3
(C) 4
(D) 5

47. Suppose we run Dijkstra's single source shortest path algorithm on the following edge-weighted directed graph with vertex P as the source.
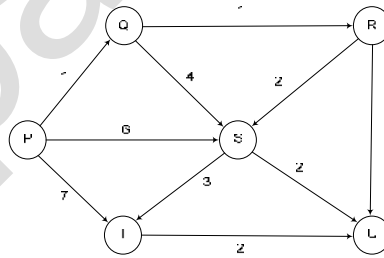
In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized?

(A) P, Q, R, S, T, U
(B) P, Q, R, U, S, T
(C) P, Q, R, U, T, S
(D) P, Q, T, R, U, S

48. A binary tree with $n > 1$ nodes has n_1, n_2 and n_3 nodes of degree one, two and three respectively. The degree of a node is defined as the number of its neighbours.

n_3 can be expressed as:

(A) $n_1 + n_2 - 1$

(B) $n_1 - 2$

(C) $\left\lceil \dfrac{n_1 + n_2}{2} \right\rceil$

(D) $n_2 - 1$

Starting with the above tree, while there remains a node v of degree two in the tree, add an edge between the two neighbours of v and then

remove *v* from the tree. How many edges will remain at the end of the process?

(A) $2 * n_1 - 3$

(B) $n_2 + 2 * n_1 - 2$

(C) $n_3 - n_2$

(D) $n_2 + n_1 - 2$

49.   Suppose the BST has been unsuccessfully searched for key 273. Which all of the above sequences list nodes in the order in which we could have encountered them in the search?

(A) II and III only

(B) I and III only

(C) III and IV only

(D) III only

Which of the following statements is TRUE?

(A) I, II and IV are inorder sequences of three different BSTs

(B) I is a preorder sequence of some BST with 439 as the root

(C) II is an inorder sequence of some BST where 121 is the root and 52 is a leaf

(D) IV is a postorder sequence of some BST with 149 as the root.

How may distinct BSTs can be constructed with 3 distinct keys?

(A) 4

(B) 5

(C) 6

(D) 9

50.   In an unweighted, undirected connect graph, the shortest path from a node S to every node is computed most efficiently, in terms of *time complexity,* by

(A) Dijkstra's algorithm starting from S.

(B) Warshall's algorithm.

(C) Performing a DFS starting from S.

(D) Performing a BFS starting from S.

51.   The following algorithm (written in pseudo-Pascal) works on a undirected graph G

Program *Explore* (G)

   procedure Visit (u)

    begin

      if Adj (u) is not empty

        { comment : Adj (u) is the list of edges incident to u}

      then

        begin

            select an edge from Adj (u);

            Let an edge e = (u, v);

            remove e from Adj (u) and Adj (v);

            Visit (v);

        end

      else

        mark u as a finished vertex and remove u from LIST;

        {Comment : LIST is the set of vertices in the graph}

```
            end;

        begin

            while LIST is not empty

                do

                    begin

                        Let v ε LIST;

                        Visit (v);

                    end

            end
```

NOTE: Initially Adj (u) is the list of all edges incident to u and LIST is the set of all vertices in the graph. They are globally accessible.

What kinds of sub graphs are obtained when this algorithm traverses the graphs G1 and G2 shown in Fig. 6 and Fig. 7, respectively?
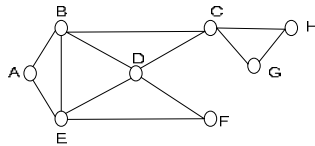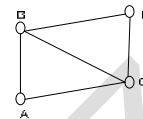
Fig. 6   **Fig. 7**

(a) What is the commonly known traversal of graphs that can be obtained from the sub graphs generated by *Program Explore*?
(b) Show that the time complexity of the procedure is O (v + e) for a graph with v vertices and e edges, given that each vertex can be accessed and removed from LIST in const time. Also show that all edges of the graph are traversed.

52.   The maximum number of possible edges in an undirected graph with n vertices and k components is _____.

53.   Kruskal's algorithm for finding a minimum spanning tree of a weighted graph G with n vertices and m edges has the time-complexity of:
          A. (n^2)
          B.   (m n)
          C.   (m + n)
          D.   (m log n)
          E. (m^2)

54.   Show that all vertices in an undirected finite graph cannot have distinct degrees, if the graph has at least two vertices

55.   Complexity of Kruskal's algorithm for finding the minimum spanning tree of an undirected graph containing n vertices and m edges, if the edges are sorted is _____.

56.   Maximum number of edges in a planner graph with n vertices is _____.

57.   A non-planer graph with minimum number of vertices has

          (A) 9 edges, 6 vertices
          (B) 6 edges, 4 vertices
          (C) 10 edges, 5 vertices
          (D) 9 edges, 5 vertices

58.   How many edges are there in a forest with p components having n vertices in all?

59.   An independent set in a graph is a subset of vertices such that no two vertices in the subset are connected by an edge. An incomplete scheme for greedy algorithm to find a maximum independent set in a tree is given below:

          V: = Set of all vertices in a tree;

          I: = φ do

                begin

                        select a vertex u Є V such that

                        _____ ;

                        V: = V –{u};

                        If u is such that

                        _____ then I: = I **U**{u}

                end;

                Output (I);

Complete the algorithm by specifying the property of vertex u in each case.   (4)

What is the time complexity of the algorithm?

60.   Consider a simple connected graph G with n vertices and n edges (n>2). Then which of the following statements are true

        (f)   G has no cycles
        (g)   the graph obtained by removing any edges from G is not consider connected
        (h)   G has at least one cycle
        (i)   the graph obtained by removing any two edges from G is not consider connected
        (j)   none of the above.

61.   The number of distinct simple graphs with upto three nodes is

                (A) 15
                (B) 10
                (C) 7
                (D) 9

62.   The number of edges in a regular graph of degree d and n vertices are ……………

63.   The minimum number of edges in a connected cyclic graph on n vertices is

                (A) n - 1
                (B) n
                (C) n + 1
              (D) None of the above

64.   How many minimum spanning trees does the following graph have? Draw them (weights are assigned to the edges).
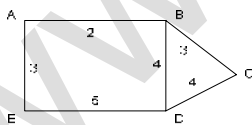
65.   Prove that in finite graph, the number of vertices of odd degree is always even.

66.   A complete undirected, weighted graph G is given on the vertex set {0, 1, ..,n-1} for any fixed 'n'. Draw the minimum spanning tree of G if

            (a)   The weight of the edge (u, v) is $|u – v|$
        (b)   The weight of the edge (u, v) is $u + v$

67.   Let G be the directed, weighted graph shown below in Fig. 17
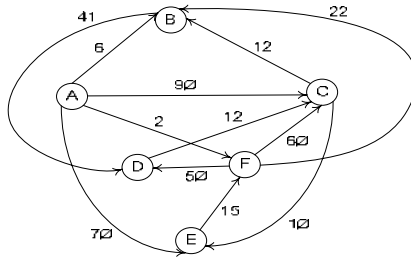
Fig. 17

We are interested in the shortest paths from A.

(a) Output the sequence of vertices identified by the Dijkstra's algorithm for single source shortest path when the algorithm is started at node A. (2)

(b) Write down the sequence of vertices in the shortest path from A to E. (2)

(c) What is the cost of shortest path from A to E?

68. Which of the following algorithm design techniques is used in finding all pairs of shortest distances in a graph?

(a) Dynamic programming (b) Backtracking

(c) Greedy (d) Divide and Conquer

69. Consider a graph whose vertices are points in the plane with integer co-ordinates (x,y) such that $1 \leq x \leq n$ and $1 \leq y \leq n$, where $n \geq 2$ is an integer. Two vertices $(x_1, y1)$ and $(x_2, y2)$ are adjacent iff $|x_1 - x_2| \leq 1v$ $|y_1 - y_2| \leq 1$. The weight of na edge $\{(x_1, y1), (x_2, y2)\}$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

(a) What is the weight of a minimum weight spanning tree in this graph? Write only the answer without any explanations. (2)

What is the weight of a maximum weight spanning tree in this graph? Write only the answer without any explanations.

70. Let G be a graph with 100 vertices numbered 1 to 100. Two vertices i and j are adjacent iff $|i - j| = 8$ or $|i - j| = 12$. The number of connected components in G is

(A) 8 (B) 4 (C) 12 (D) 25

71. The number of articulation points of the following graph is:
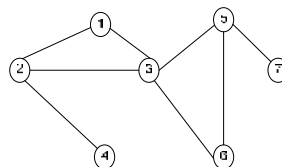
(A) 0
(B) 1
(C) 2
(D) 3

72. Let G be an undirected graph with distinct edge weights. Let e_{max} be the edge with maximum weight and e_{min} the edge with minimum weight.

Which of the following is **false**?

(A) Every minimum spanning tree of G must contain e_{min}
(B) If e_{max} is an minimum spanning tree, then its removal must disconnect G
(C) No minimum spanning tree contains e_{max}

16

(D) G has a unique minimum spanning tree

73. Let G be an undirected graph. Consider a depth first traversal G, and let T be the resulting depth first search tree. Let u be a vertex in G and v be the first new (unvisited) vertex visited after visiting u in the traversal. Which of the following is always true?

(A) {u,v} must be an edge in G, and u is descendent of v in T
(B) {u,v} must be an edge in G, and v is descendent of u in T
(C) If {u,v} is not an edge in G then u is leaf in T
(D) If {u,v} is not an edge in G then u and v must have the same parent in T

74. How many undirected graphs (not necessarily connected) can be constructed out of a given set And = {v_1, v_2, …, v_n} of n vertices?
A. n(n-1)/2
B. 2^n
C. n!
D. $2^{n(n-1)/2}$

75. Consider a weighted undirected graph with vertex set And = {n_1, n_2, n_3, n_4, n_5, n6} and edge set E = {(n_1, n_2, 2), (n_1, n_3, 8), (n_1, n_6, 3), (n_2, n_4, 4), (n_2, n_5, 12), (n_3, n_4, 7), (n_4, n_5, 9), (n_4, n_6, 4)} The third value in each tuple represents the weight of the edge specified in the tuple.

(a) Lists the edges of a minimum spanning tree of the graph.   (2)

(b) How many distinct minimum spanning tree does this graph have?

(c) Is the minimum among the edge weights of a minimum spanning tree unique over all possible minimum spanning trees of a graph?   (1)

(d) Is the maximum among the edge weights of a minimum spanning tree unique over all possible minimum spanning trees of a graph?

76. The minimum number of colors required to color the vertices of a cycle with n nodes in such a way that no two adjacent nodes have the same color is:

A. 2
B. 3
C. 4
D. $n - 2\lfloor n/2 \rfloor + 2$

77. The Maximum number of edges in a n –node undirected graph without self loops is

A. n2
B. n(n − 1)/2
C. n -1
D. (n+1)(n)/2

78. The number of distinct simple graphs with upto three nodes is

(E) 15
(F) 10
(G) 7
(H) 9

79. Fill in the blanks in the following template of an algorithm to compute all pairs shortest path lengths in a directed graph G with n*n adjacency matrix A. A[i, j] equals 1 if there is an edge in G from i to j, and 0 otherwise. Your aim in filling in the blanks

INITIALIZATION: For I = 1 … n

{For j = 1 … n

{If A[i,j] = 0 then P[i, j] = _____ else P[i,j] = _____;}

}

ALGORITHM: For i = 1 … n

{ For j = 1 … n

{ For k = 1 … n

{P[_, _] = min{ _____ };}

}

}

(a) Copy the complete line containing the blanks in the Initialization step and fill in the blanks. (1)
(b) Copy the complete line containing the blanks in the Initialization step and fill in the blanks. (3)
(c) Fill in the blank: The running time of the algorithm is O(_____)

80.   Let G be an arbitrary graph with n nodes and k components. If a vertex is removed from G, the number of components in the resultant graph must necessarily lie between

(A) k and n
(B) k − 1 and k + 1
(C) k − 1 and n − 1
(D) k + 1 and n − k

81.   Consider the following graph
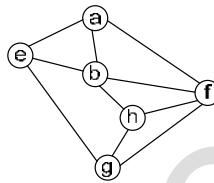
Among the following sequences

I   a b e g h f

II   a b f e h g

III   a b f h g e

IV   a f g h b e

Which are depth first traversals of the above graph?

(A) I, II and IV only
(B) I and IV only
(C) II, III and IV only
(D) I, III and IV only

82.   How many perfect matching are there in a complete graph of 6 vertices?

(A) 15
(B) 24
(C) 24
(D) 60

83.   Let G = (V, E) be an undirected graph with a subgraph G1 = (V1, E1). Weights are assigned to edges of G as follows.

A single-source shortest path algorithm is executed on the weighted graph (V, E, what) with an arbitrary vertex v1 of V1 as the source. Which of the following can always be inferred from the path costs computed?

(A) The number of edges in the shortest paths from v1 to all vertices of G
(B) G1 is connected.
(C) V1 forms a clique in G
(D) G1 is a tree

84. What is the weight of a minimum spanning tree of the following graph?

(A) 29
(B) 31
(C) 38
(D) 41
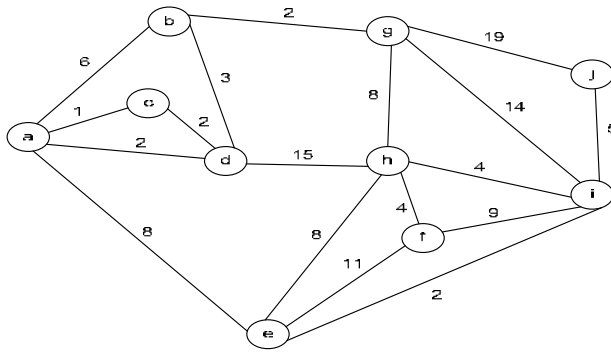
```
for each w adjacent to v

if(!T[w].Known)

{

if(!T[v].Dist + C_{v,w} < T[w].Dist)

{

/* update of w */

Decrease(T[w].Dist to T[v].Dist + C_{v,w});

T[w].path = v;

}

}

}

Dispose Queue(Q);

}
```

UNIT V SORTING, SEARCHING AND HASH TECHNIQUES

Sorting algorithms: Insertion sort - Selection sort - Shell sort - Bubble sort - Quick sort - Merge sort - Radix sort – Searching: Linear search –Binary Search Hashing: Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

SORTING:

Definition:

Sorting is a technique for arranging data in a particular order.

Order of sorting:

Order means the arrangement of data. The sorting order can be ascending or descending. The ascending order means arranging the data in increasing order and descending order means arranging the data in decreasing order.

Types of Sorting

- Internal Sorting
- External Sorting

Internal Sorting

Internal Sorting is a type of sorting technique in which data resides on main memory of computer. It is applicable when the number of elements in the list is small.

E.g. Bubble Sort, Insertion Sort, Shell Sort, Quick Sort., Selection sort, Radix sort

External Sorting

External Sorting is a type of sorting technique in which there is a huge amount of data and it resides on secondary devise(for eg hard disk,Magnetic tape and so no) while sorting.

E.g. Merge Sort, Multiway Merge Sort,Polyphase merge sort

Sorting can be classified based on

1.Computational complexity

2.Memory utilization

3.Stability

4.Number of comparisons.

ANALYSIS OF ALGORITHMS:

Efficiency of an algorithm can be measured in terms of:

Space Complexity: Refers to the space required to execute the algorithm

Time Complexity: Refers to the time required to run the program.

Sorting algorithms:

- Insertion sort
- Selection sort
- Shell sort
- Bubble sort
- Quick sort
- Merge sort
- Radix sort

INSERTION SORTING:

- The insertion sort works by taking elements from the list one by one and inserting them in the correct position into a new sorted list.
- Insertion sort consists of N-1 passes, where N is the number of elements to be sorted.
- The ith pass will insert the ith element A[i] into its rightful place among A[1],A[2],…,A[i-1].
- After doing this insertion the elements occupying A[1],…A[i] are in sorted order.

How Insertion sort algorithm works?
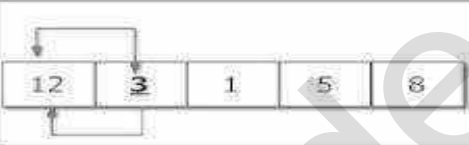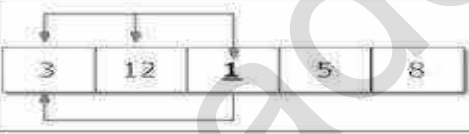
Step 1	12	3	1	5	8	Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2	3	12	1	5	8	Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3	1	3	12	5	8	Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4	1	3	5	12	8	Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
	0	1	3	8	12	Sorted Array in Ascending Order

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Insertion Sort routine:

```
void Insertion_sort(int a[ ], int n)
{
        int i, j, temp;
        for ( i = 0 ; i < n -1  ;  i ++ )
                {
                temp = a [ j ] ;
```

```
     for ( j = i ; j > 0  && a [ j -1  ] >  temp  ; j -- )
               {
                a[ j ] = a [ j – 1 ] ;

               }
     a[j]=temp;
   }}
```

Program for Insertion sort

```
 #include<stdio.h>
 void main( ){
  int n, a[ 25 ], i, j, temp;
  printf( "Enter number of elements \n" );
     scanf( "%d", &n );
  printf( "Enter %d integers \n", n );
  for ( i = 0; i < n; i++ )
     scanf( "%d", &a[i] );
  for ( i = 0 ; i < n; i++ ){
temp=a[i];
for (j=i;j > 0 && a[ j -1]>temp;j--)
       {
           a[ j ]   = a[ j - 1 ];
       }
       a[j]=temp;}
  printf( "Sorted list in ascending order: \n ");
  for ( i = 0 ; i < n ; i++)
     printf ( "%d \n ", a[ i ] );}
```

OUTPUT:

Enter number of elements
6
Enter 6 integers
20 10 60 40 30 15
Sorted list in ascending order:
 10
 15
 20
 30
 40
 60

Advantage of Insertion sort

- Simple implementation.

- Efficient for (quite) small data sets.

- Efficient for data sets that are already substantially sorted.

Disadvantages of Insertion sort

- It is less efficient on list containing more number of elements.
- As the number of elements increases the performance of the program would be slow.
- Insertion sort needs a large number of element shifts.

Selection Sort

Selection sort selects the smallest element in the list and place it in the first position then selects the second smallest element and place it in the second position and it proceeds in the similar way until the entire list is sorted. For "n" elements, (n-1) passes are required. At the end of the ith iteration, the ith smallest element will be placed in its correct position.

Selection Sort routine:

```
void Selection_sort( int a[ ], int n )
{
    int i , j , temp , position ;
        for ( i = 0 ; i < n – 1 ; i ++ )
        {
                position = i ;
                for (  j = i + 1 ; j < n ;  j ++ )
                {
                        if ( a[ position ] > a[ j ] )
                                position = j;}
                temp = a[ i ];
                a[ i ] = a[ position ];
                a[ position ] = temp;
        }}
```

How Selection sort algorithm works?

Figure: Selection Sort
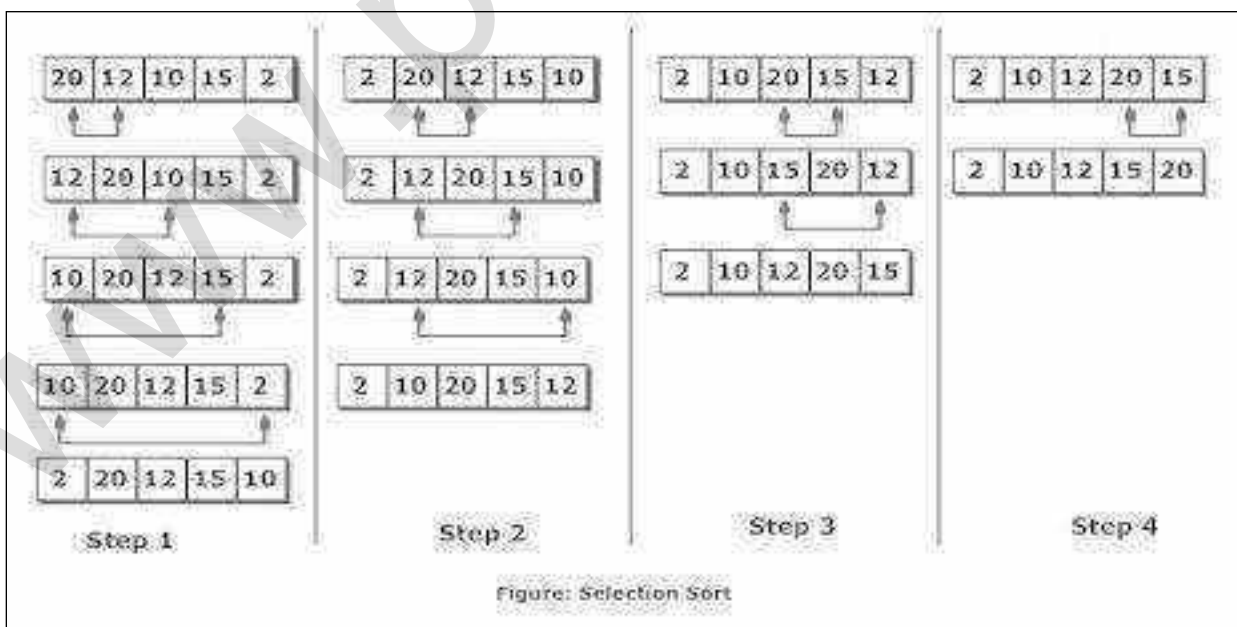
Program for Selection sort

```c
#include <stdio.h>
 void main( )
{
   int a [ 100 ] , n , i , j , position , temp ;
   printf ( "Enter number of elements \n" ) ;
       scanf ( "%d", &n ) ;
   printf ( " Enter %d integers \n ", n ) ;
   for ( i = 0 ; i < n ; i ++ )
      scanf ( "%d", & a[ i ] ) ;
   for ( i = 0 ; i < ( n - 1 ) ; i ++ )
     {
      position = i ;
      for ( j = i + 1 ; j < n ; j ++ )
      {
         if ( a [ position ] > a [ j ] )
         position = j ;
      }
     if ( position != i  )
     {
       temp = a [ i ] ;
       a [ i ] = a [ position ] ;
       a [ position ] = temp ;
     }
   }
   printf ( "Sorted list in ascending order: \n " ) ;
   for ( i = 0 ; i < n ; i ++ )
      printf ( " %d \n ", a[ i ] ) ;
 }
```
OUTPUT:

Enter number of elements
5
 Enter 5 integers
 8 3 9 5 1
Sorted list in ascending order:
 1
 3
 5
 8
 9

Advantages of selection sort

- Memory required is small.

- Selection sort is useful when you have limited memory available.

- Relatively efficient for small arrays.

Disadvantage of selection sort

- Poor efficiency when dealing with a huge list of items.

- The selection sort requires n-squared number of steps for sorting n elements.

- The selection sort is only suitable for a list of few elements that are in random order.

Shell Sort

- Invented by Donald shell.
- It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time.
- In shell sort the whole array is first fragmented into K segments, where K is preferably a prime number.
- After the first pass the whole array is partially sorted.
- In the next pass, the value of K is reduced which increases the size of each segment and reduces the number of segments.
- The next value of K is chosen so that it is relatively prime to its previous value.
- The process is repeated until K=1 at which the array is sorted.
- The insertion sort is applied to each segment so each successive segment is partially sorted.
- The shell sort is also called the Diminishing Increment sort, because the value of k decreases continuously

A Shell Sort with Increments of Three
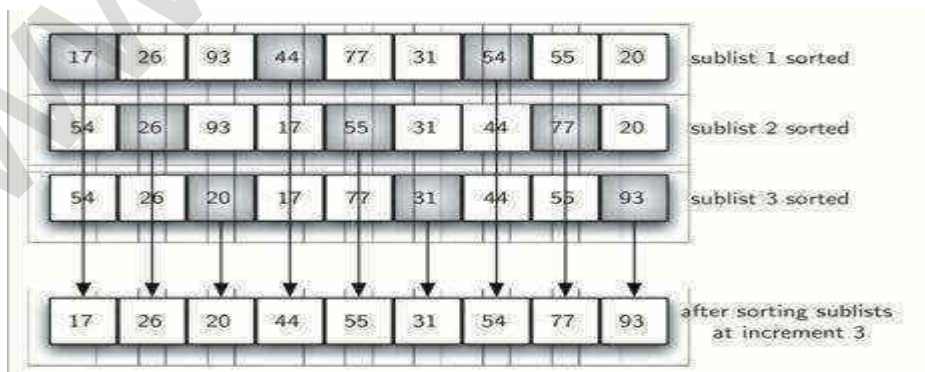
A Shell Sort after Sorting Each Sublist
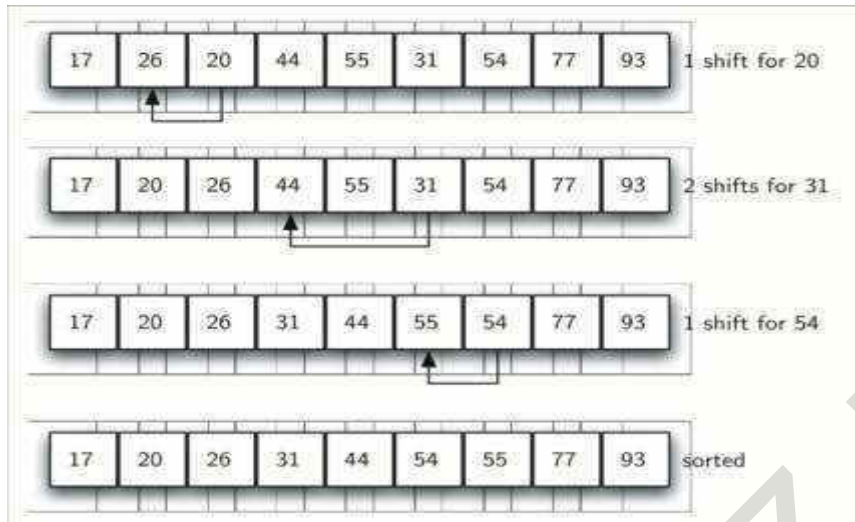
Shell Sort: A Final Insertion Sort with Increment of 1

Shell Sort routine:

```
void Shell_sort ( int a[ ], int n )
{
            int i, j, k, temp;
            for ( k = n / 2 ; k > 0 ;   k = k / 2 )
                 for ( i = k ;  i < n ; i + + )
                 {
                 temp = a [ i ] ;
            for ( j = i ; j > = k && a [ j – k ]  > temp ;  j  =   j – k )
                              {
                                   a [ j ] = a [ j – k ] ;
                              }
                          a [ j ] = temp ;
                 }
}
```

Program for Shell sort

```
#include<stdio.h>
 void main( )
{
 int n, a[ 25 ], i, j,k,temp;
 printf( "Enter number of elements \n" );
 scanf( "%d", &n );
  printf( "Enter %d integers \n", n );
 for ( i = 0; i < n; i++ )
     scanf( "%d", &a[i] );
                  for (k = n / 2 ; k>0 ;  k=k/ 2){
```

```
            for ( i = k ;  i < n ;  i ++ )
            {
             temp = a [ i ] ;
                 for (j = i ; j>= k && a [ j - k ]>temp ; j=j - k )
                                                  {
                                                              a [ j ] = a [ j - k ] ;
                                                  }
                                        a [ j ] = temp ;
            }
        }
 printf( "Sorted list in ascending order using shell sort: \n ");
  for ( i = 0 ; i < n ; i++)
     printf ( "%d\t ", a[ i ] );
  }
```

OUTPUT:

Enter number of elements
10
Enter 10 integers
81 94 11 96 12 35 17 95 28 58
Sorted list in ascending order using shell sort:
 11   12   17   28   35   58   81   94   95   96

//PROGRAM FOR SHELL USING FUNCTION

```
#include  < stdio.h >
 void main( )
{
   int a [ 5 ] = { 4, 5, 2, 3, 6 } , i = 0  ;
   ShellSort ( a, 5 ) ;
printf("Example using function");
   printf(  " After Sorting :" ) ;
   for ( i = 0 ; i < 5  ; i ++ )
      printf ( " %d ", a[ i ] ) ;
}
void ShellSort (int  a [ 5 ] , int  n )
{
     int i , j , k , temp ;
     for ( k = n / 2 ; k  >  0 ;  k  / = 2)
     {
         for ( i = k ;  i < n ;  i ++ )
         {
             temp = a [ i ] ;
            for ( j = i ; j > = k && a [ j – k ]  >  temp ; j =  j – k ){
                                     a [ j ] = a [ j – k ] ;
                               }
                         a [ j ] = temp ;  } }}
```

OUTPUT:

After Sorting : 2 3 4 5 6

Advantages of Shell sort

- Efficient for medium-size lists.

Disadvantages of Shell sort

- Complex algorithm, not nearly as efficient as the merge, heap and quick sorts

Bubble Sort

- Bubble sort is one of the simplest internal sorting algorithms.

- Bubble sort works by comparing two consecutive elements and the largest element among these two bubbles towards right at the end of the first pass the largest element gets sorted and placed at the end of the sorted list.

- This process is repeated for all pairs of elements until it moves the largest element to the end of the list in that iteration.

- Bubble sort consists of (n-1) passes, where n is the number of elements to be sorted.

- In 1st pass the largest element will be placed in the nth position.

- In 2nd pass the second largest element will be placed in the $(n-1)^{th}$ position.

- In $(n-1)^{th}$ pass only the first two elements are compared.
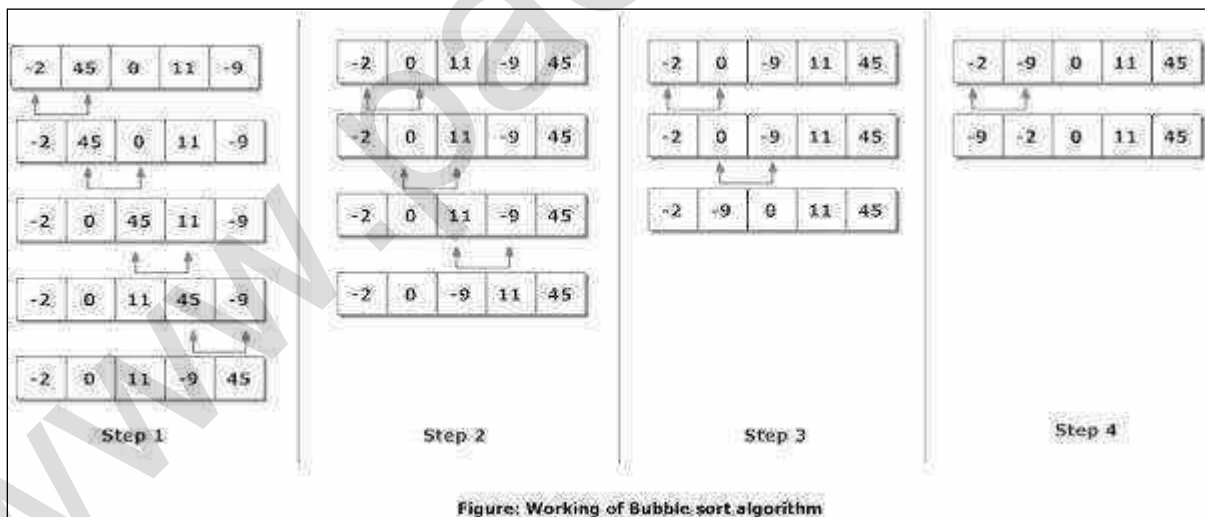
Figure: Working of Bubble sort algorithm

Bubble sort routine:

```
void Bubble_sort (int a [  ] , int n )
{
        int i, j, temp;
        for( i = 0; i < n - 1;  i++ ){
```

```
                for( j = 0; j < n – i - 1; j++ )
                {
                        if( a[ j ] > a [ j + 1 ] )
                        {
                                temp = a [ j ];
                                a[ j ] = a[ j + 1 ];
                                a[ j + 1 ] = temp;
                        }
                } }}
```

Program for Bubble sort
```
#include<stdio.h >
#include<conio.h >
void main( )
{
int a [ 20 ], i, j, temp, n ;
printf ("Enter the number of elements");
    scanf ("%d",&n);
printf("Enter the  numbers");
for(i=0;i < n ;i++)
scanf("%d",&a[i]);
for(i=0;i<n-1;i++)
{
for(j=0;j<n-i-1;j++)
{
if(a[j]>a[j + 1]){
                temp = a[ j ] ;
                a[ j ] = a[ j + 1 ] ;
        a[ j+ 1 ] = temp;
    }
}
}
printf("\nSorted array\t");
for(i=0;i<n;i++)
    printf("%d\t",a[i]);
}
```
OUTPUT:
Enter the number of elements5

Enter the numbers8 3 9 5 1

Sorted array   1   3   5   8   9

Advantage of Bubble sort
- It is simple to write
- Easy to understand
- It only takes a few lines of code.

Disadvantage of Bubble sort

- The major drawback is the amount of time it takes to sort.
- The average time increases almost exponentially as the number of table elements increase.

Quick Sort

- Quicksort is a divide and conquer algorithm.
- The basic idea is to find a "pivot" item in the array and compare all other items with pivot element.
- Shift items such that all of the items before the pivot are less than the pivot value and all the items after the pivot are greater than the pivot value.
- After that, recursively perform the same operation on the items before and after the pivot.
- Find a "pivot" item in the array. This item is the basis for comparison for a single round.
- Start a pointer (the left pointer) at the first item in the array.
- Start a pointer (the right pointer) at the last item in the array.

1. Assume A[0]=pivot which is the left. i.e pivot=left.
2. Set i=left+1; i.e A[1];
3. Set j=right. ie. A[6] if there are 7 elements in the array
4. If A[pivot]>A[i],increment i and if A[j]>A[pivot],then decrement j, Otherwise swap A[i] and A[j] element.
5. If i=j,then swap A[pivot] and A[j].

Step 4 :

Step 5 :

Step 6 :

Swap A[left] and A[right]

Step 6 :

Swap A[i] and A[pivot]

Step 7 :
This is the sorted list.

Quick Sort routine:

```
void Quicksort ( int a [  ], int left, int right )
{
        int i, j, p, temp;
        if ( left < right )
        {
                p = left;
                i = left + 1; j
                = right;
                while ( i < j )
                {
                        while ( a [ i ]  < =  a [ p ] )
                                i = i + 1;
                        while ( a [ j ]  >  a [ p ] )
                                j = j - 1;
                        if ( i < j )
                        {
                                temp = a [ i ];
                                a [ i ] = a [ j ];
                                a [ j ] = temp;
                        }
                }
                temp = a [ p ];
                a [ p ] =  a [ j ];
                a [ j ]  = temp;
                quicksort ( a, left, j - 1 );
                quicksort ( a, j + 1, right );
        } }
```

Program for Quick sort

```
#include<stdio.h>
void quicksort (int [10], int, int ) ;
void main( )
{
 int a[20], n, i ;
 printf("Enter size of the array:" );
     scanf("%d",&n);
         printf( " Enter the numbers :");
         for ( i = 0 ; i < n ; i ++ )
            scanf ("%d",&a[i]);
         quicksort ( a , 0  , n - 1 );
         printf ( " Sorted elements: " );
         for ( i = 0 ; i < n ; i ++ )
           printf ("%d\t",a[ i]);
 }
```

```
void quicksort ( int a[10], int left, int right )
{
 int p, j, temp, i ;
 if ( left < right )
 {
 p = left ;
 i = left ;
 j = right ;
 while ( i < j )
 {
   while(a[i]<= a[p] && i<right )
        i++ ;
   while ( a [ j ] > a [ p ] )
        j--;
   if ( i < j )
   {
        temp = a [ i ] ;
        a [ i ] = a [ j ] ;
        a[ j ] = temp ;
   }
 }
        temp = a [ p ] ;
        a [ p ] = a [ j ] ;
        a [ j ] =temp ;
 quicksort ( a , left , j - 1 ) ;
 quicksort ( a , j + 1 , right ) ;
}
}
```

OUTPUT:
Enter size of the array:8

Enter the numbers :40 20 70 14 60 61 97 30

Sorted elements: 14   20   30   40   60   61   70   97

Advantages of Quick sort

- Fast and efficient as it deals well with a huge list of items.

- No additional storage is required.

Disadvantages of Quick sort

- The difficulty of implementing the partitioning algorithm.

Merge Sort

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

Divide means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements.

If there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.

Conquer means sorting the two sub-arrays recursively using merge sort.

Combine means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.

- Otherwise, divide the unsorted array into two sub-arrays of about half the size.

- Use merge sort algorithm recursively to sort each sub-array.

- Merge the two sub-arrays to form a single sorted list.

(Divide and Conquer the array)
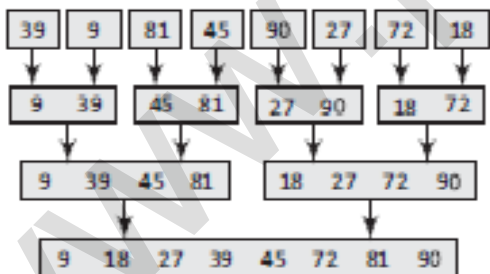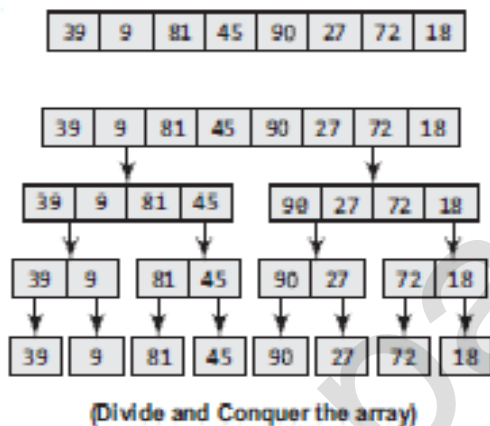
(Combine the elements to form a sorted array)

Merge Sort routine:

```
void Merge_sort (int a [  ] , int temp [  ] , int n )
{
    msort ( a , temp , 0 , n - 1 ) ;
}
```

```
void msort ( int a[ ] , int temp [  ] , int left , int right ){
              int center ;
              if( left < right ){
                     center = ( left + right ) / 2 ;
                     msort ( a , left , center ) ;
                     msort ( a , temp , center + 1 , right ) ;
                     merge ( a , temp , n , left , center , right ) ;
              }}
void  merge ( int a [  ] , int temp [  ] , int n , int left , int center , int right )
{
        int i = 0 , j , left_end  = center , center = center + 1 ;
        while( ( left < = left_end ) && ( center < = right ) )
        {
               if( a [ left ] < = a [ center ]  )
               {
                      temp [ i ] = a [ left ] ;
                      i + + ;
                      left + + ;
               }
               else
               {
                      temp [ i ] = a [ center ] ;
                      i + + ;
                      center + + ;
               }
        }
        while( left <= left_end )
        {
               temp [ I ] = a [ left ] ;
               left + + ;
               i + + ;
        }
        while( center < = right )
        {
               temp [ i ] = a [ center ] ;
               center + + ;
               i + + ;
        }
        for ( i = 0 ; i < n ;  i + + )
            print temp [ i ] ;
}
```

Program for merge sort

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
```

```c
int main()
{
   int a[30],n,i;
   printf("Enter no of elements:");
   scanf("%d",&n);
   printf("Enter array elements:");
   for(i=0;i<n;i++)
   scanf("%d",&a[i]);
   mergesort(a,0,n-1);
   printf("\nSorted array is :");
   for(i=0;i<n;i++)
   printf("%d ",a[i]);
   return 0;
}
void mergesort(int a[],int i,int j)
{
   int mid;
   if(i<j)
   {
      mid=(i+j)/2;
      mergesort(a,i,mid);        //left recursion mergesort(a,mid+1,j);
      //right recursion merge(a,i,mid,mid+1,j);    //merging of two
      sorted sub-arrays
   }}
void merge(int a[],int i1,int j1,int i2,int j2)
{
   int temp[50];    //array used for merging
   int i,j,k;
   i=i1;    //beginning of the first list
   j=i2;    //beginning of the second list
   k=0;
   while(i<=j1 && j<=j2)    //while elements in both lists
   { if(a[i]<a[j])
   temp[k++]=a[i++];
   else
   temp[k++]=a[j++];
   }
   while(i<=j1)    //copy remaining elements of the first list
   temp[k++]=a[i++];
   while(j<=j2)    //copy remaining elements of the second list
   temp[k++]=a[j++];
   //Transfer elements from temp[] back to a[]
   for(i=i1,j=0;i<=j2;i++,j++)
      a[i]=temp[j];
}
```

OUTPUT:

Enter no of elements:8
Enter array elements:24 13 26 1 2 27 38 15
Sorted array is :1 2 13 15 24 26 27 38

Advantages of Merge sort
- Mergesort is well-suited for sorting really huge amounts of data that does not fit into memory.
- It is fast and stable algorithm

Disadvantages of Merge sort

- Merge sort uses a lot of memory.
- It uses extra space proportional to number of element n.
- This can slow it down when attempting to sort very large data.

Radix Sort

- Radix sort is one of the linear sorting algorithms. It is generalized form of bucket sort. It can be performed using buckets from 0 to 9.
- It is also called binsort, card sort.
- It works by sorting the input based on each digit. In first pass all the elements are stored according to the least significant digit.
- In second pass the elements are arranged according to the next least significant digit and so on till the most significant digit.
- The number of passes in a Radix sort depends upon the number of digits in the given numbers.

Algorithm for Radix sort

Steps1: Consider 10 buckets (1 for each digit 0 to 9)

Step2: Consider the LSB (Least Significant Bit) of each number (numbers in the one''s

Place…. E.g., in 43 LSB = 3)

Step3: Place the elements in their respective buckets according to the LSB of each number

Step4: Write the numbers from the bucket (0 to 9) bottom to top.

Step5: repeat the same process with the digits in the $10^{"s}$ place (e.g. In 43 MSB =4)

Step6: repeat the same step till all the digits of the given number are consider.

43 27 31 15 37 80 03
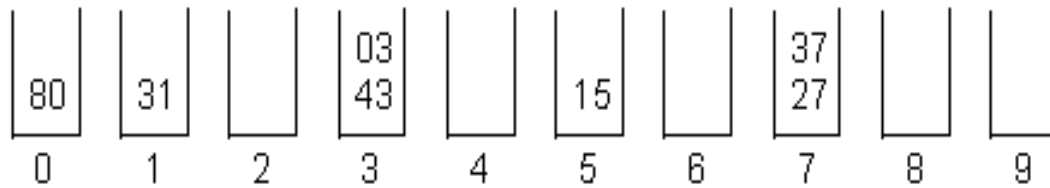
: 80 31 43 03 15 27 37
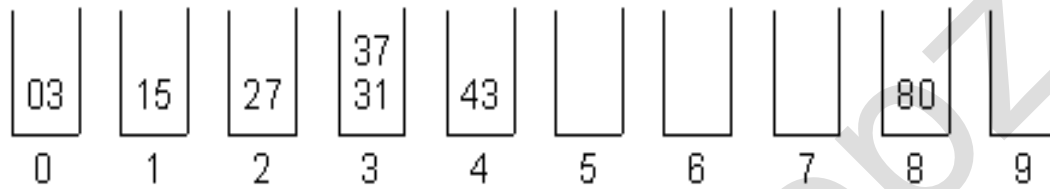
03 15 27 31 37 43 80

Sorted list of array : 3 15 27 31 37 43 80

Routine for Radix sort

void Radix_sort (int a [ ] , int n)
{
        int bucket [10] [5] , buck [10] , b [10] ;
        int i , j , k , l , num , div , large , passes ;
        div = 1 ;
        num = 0 ;
        large = a [0] ;
        for (i = 0 ; i < n ; i ++)
        {
                if (a[I] > large)
                {
                        large = a [i] ;
                }
                while (large > 0)
                {
                        num ++ ;

```
                                large = large / 10 ;
                      }
                   for ( passes = 0 ; passes < num ; passes ++ )
                   {
                           for ( k = 0 ; k < 10 ; k ++ )
                           {
                                   buck [ k ] = 0 ;
                           }
                           for ( i = 0 ; i < n ; i ++ )
                           {
                                   l = ( ( a [ i ] / div ) % 10 ) ;
                                   bucket [ l ] [ buck [ l ] ++ ] = a [ i ] ;
                           }
                           i = 0 ;
                           for ( k = 0 ; k < 10 ; k ++ )
                           {
                                   for ( j = 0 ; j < buck [ k ]  ; j ++ )
                                   {
                                           a [ i ++ ] = bucket [ k ] [ j ] ;
                                   }
                           }
                           div * = 10 ;
                   }
         }
}
```

Program for Radix sort

```
#include<stdio.h >
 void main( )
 {
    int a [ 5 ] = { 4, 5, 2, 3, 6 } , i = 0  ;
         void Radix_sort ( int a [  ] , int n );
         Radix_sort(a,5);
    printf(  " After Sorting :" ) ;
    for ( i = 0 ; i < 5  ; i ++ )
      printf ( " %d ", a[ i ] ) ;
}
void Radix_sort ( int a [  ] , int n )
{
        int bucket [ 10 ] [ 5 ] , buck [ 10 ] , b [ 10 ] ;
        int i , j , k , l , num , div , large , passes ;
        div = 1 ;
        num = 0 ;
        large = a [ 0 ] ;
        for ( i = 0 ; i < n ; i ++ ){
```

```
                if ( a[ i] > large )
                {
                        large = a [ i ] ;
                }
                while ( large > 0 )
                {
                        num ++ ;
                        large = large / 10 ;
                }
                for ( passes = 0 ; passes < num ; passes ++ )
                {
                        for ( k = 0 ; k < 10 ; k ++ )
                        {
                                buck [ k ] = 0 ;
                        }
                        for ( i = 0 ; i < n ; i ++ )
                        {
                                l = ( ( a [ i ] / div ) % 10 ) ;
                                bucket [ l ] [ buck [ l ] ++ ] = a [ i ] ;
                        }
                        i = 0 ;
                        for ( k = 0 ; k < 10 ; k ++ )
                        {
                                for(j=0 ; j<buck[k];j++ )
                                {
                                        a[i++]=bucket[ k ][ j ] ;
                                }
                        }
                        div*= 10 ;
                }
        }
}
```

Advantages of Radix sort:
- Fast and complexity does not depend on the number of data.
- Radix Sort is very simple.

Disadvantages of Radix sort:

- Radix Sort takes more space than other sorting algorithms, since in addition to the array that

  will be sorted, you need to have a sub list for each of the possible digits or letters.

- Since Radix Sort depends on the digits or letters, Radix Sort is also much less flexible than

  other sorts.

Analysis of Sorting algorithms:

S.No	Algorithm	Best Case Analysis	Average Case Analysis	Worst Case Analysis
1	Insertion sort	$O(N)$	$O(N^2)$	$O(N^2)$
2	Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
3	Shell sort	$O(N \log N)$	$O(N^{1.5})$	$O(N^2)$
4	Bubble sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
5	Quick sort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$
6	Merge sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
7	Radix or bucket or binsort sort or card sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$

SEARCHING

Searching is an algorithm, to check whether a particular element is present in the list.

Types of searching:-

- Linear search
- Binary Search

Linear Search

Linear search is used to search a data item in the given set in the sequential manner, starting from the first element. It is also called as sequential search

Linear Search routine:

```
void Linear_search ( int a[ ] , int n )
{
int search , count = 0 ;
for ( i = 0 ; i < n ; I ++ )
{
if ( a [ i ] = = search )
{
count ++ ;
}
}
if ( count = = 0 )
print  "Element not Present" ;
else
print "Element is Present in list" ;
}
```

Program for Linear search

```
#include < stdio.h >
void main(  )
{
int a [ 10 ] , n , i , search, count = 0 ;
printf ( " Enter the number of elements \t " ) ;
   scanf ( " %d " , & n ) ;
printf ( " \n Enter %d  numbers \n " , n ) ;
for ( i = 0 ;  i < n ; i ++ )
   scanf ( " %d " , & a [ i ] ) ;
printf ( " \n Array Elements \n " ) ;
for ( i = 0 ; i < n ; i ++ )
    printf ( " %d \t " , a [ i ] ) ;
printf ( " \ n \ n Enter the Element to be searched: \ t " ) ;
   scanf ( " % d " , & search ) ;
for ( i =0 ; i < n; i ++ )
   {
     if ( search = = a [ i ] )

        count ++ ;
   }
if ( count = = 0 )
   printf( " \n Element %d is not present in the array " , search ) ;
else
   printf ( " \n Element %d is present %d times in the array \n " , search , count ) ;
}
```

OUTPUT:
Enter the number of elements   5
Enter the numbers
20 10 5 25 100
Array Elements
20   10   5   25   100
Enter the Element to be searched: 25
Element 25 is present 1 times in the array

Advantages of Linear search:

- The linear search is simple - It is very easy to understand and implement;

- It does not require the data in the array to be stored in any particular order.

Disadvantages of Linear search:

- Slower than many other search algorithms.

- It has a very poor efficiency.

Binary Search

- Binary search is used to search an item in a sorted list. In this method , initialize the lower limit and upper limit.
- The middle position is computed as (first+last)/2 and check the element in the middle position with the data item to be searched.
- If the data item is greater than the middle value then the lower limit is adjusted to one greater than the middle value.Otherwise the upper limit is adjusted to one less than the middle value.

Working principle:

Algorithm is quite simple. It can be done either recursively or iteratively:

1. Get the middle element;
2. If the middle element equals to the searched value, the algorithm stops;
3. Otherwise, two cases are possible:
    o Search value is less than the middle element. In this case, go to the step 1 for the part of the array, before middle element.
    o Searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

Example 1.

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 > 6):   -1 5 6 18 19 25 46 78 102 114

Step 2 (middle element is 5 < 6):   -1 5 6 18 19 25 46 78 102 114

Step 3 (middle element is 6 == 6):   -1 5 6 18 19 25 46 78 102 114

Binary Search routine:

```
void Binary_search ( int a[ ] , int n , int search )
{
int first,  last,  mid ;
first = 0 ;
last = n-1 ;
mid = ( first + last ) / 2 ;
while ( first < = last )
{
if ( Search > a [ mid ]  )
first = mid + 1 ;
else if ( Search = = a [ mid ] )
{
print "Element is present in the list" ;
break ;
}
else {
```

```
last = mid – 1 ;
mid = ( first + last ) / 2 ;
}
if( first > last )
print "Element Not Found" ;
}
```

Program for Binary Search:

```
#include<stdio.h>
void main(  )
{
int a [ 10 ] , n , i , search, count = 0 ;
void Binary_search ( int a[ ] , int n , int search );
printf ("Enter the number of elements \t") ;
   scanf ("%d",&n);
printf("\nEnter the numbers\n") ;
for (i = 0; i<n;i++)
   scanf("%d",&a[i]);
printf("\nArray Elements\n") ;
for (i = 0 ; i < n ; i ++ )
printf("%d\t",a[i]) ;
printf ("\n\nEnter the Element to be searched:\t");
scanf("%d",&search );
Binary_search(a,n,search);
}
void Binary_search ( int a[ ] , int n , int search )
{
int first,  last,  mid ;
first = 0 ;
last = n-1 ;
mid = (first + last ) / 2 ;
while (first<=last )
{
if(search>a[mid])
first = mid + 1 ;
else if (search==a[mid])
{
printf("Element is present in the list");
break ;
}
else
last = mid - 1 ;
mid = ( first + last ) / 2 ;
}
if( first > last )
printf("Element Not Found");
}
```

OUTPUT:

Enter the number of elements   5

Enter the numbers

20 25 50 75 100

Array Elements

20   25   50   75   100

Enter the Element to be searched: 75

Element is present in the listPress any key to continue . . .

Advantages of Binary search:

- In Linear search, the search element is compared with all the elements in the array. Whereas in Binary search, the search element is compared based on the middle element present in the array.

- A technique for searching an ordered list in which we first check the middle item and - based on that comparison - "discard" half the data. The same procedure is then applied to the remaining half until a match is found or there are no more items left.

Disadvantages of Binary search:

- Binary search algorithm employs recursive approach and this approach requires more stack space.

- It requires the data in the array to be stored in sorted order.

- It involves additional complexity in computing the middle element of the array.

Analysis of Searching algorithms:

S.No	Algorithm	Best Case Analysis	Average Case Analysis	Worst Case Analysis
1	Linear search	O(1)	O(N)	O(N)
2	Binary search	O(1)	O(log N)	O(log N)

HASHING :

Hashing is a technique that is used to store, retrieve and find data in the data structure called Hash Table. It is used to overcome the drawback of Linear Search (Comparison) & Binary Search (Sorted order list). It involves two important concepts-

- ➢ Hash Table
- ➢ Hash Function

Hash table

- A **hash table** is a data structure that is used to store and retrieve data (keys) very quickly.
- It is an array of some fixed size, containing the keys.
- Hash table run from 0 to Tablesize – 1.
- Each key is mapped into some number in the range 0 to Tablesize – 1.
- This mapping is called Hash function.
- Insertion of the data in the hash table is based on the key value obtained from the hash function.
- Using same hash key value, the data can be retrieved from the hash table by few or more Hash key comparison.
- The **load factor** of a hash table is calculated using the formula:

    (Number of data elements in the hash table) / (Size of the hash table)

Factors affecting Hash Table Design

- Hash function
- Table size.
- Collision handling scheme

Simple Hash table with table size = 10

Hash function:

- It is a function, which distributes the keys evenly among the cells in the Hash Table.
- Using the same hash function we can retrieve data from the hash table.
- Hash function is used to implement hash table.
- **The integer value returned by the hash function is called hash key.**
- If the input keys are integer, the commonly used hash function is

$$H (key) = key \% Tablesize$$

```
typedef unsigned int index;
index Hash ( const char *key , int Tablesize )
{
    unsigned int Hashval = 0 ;
    while ( * key ! = „ \0 „ )
        Hashval + = * key ++ ;
    return ( Hashval % Tablesize ) ;
}
```

A simple hash function

Types of Hash Functions

1. Division Method
2. Mid Square Method
3. Multiplicative Hash Function
4. Digit Folding

1. Division Method:

- It depends on remainder of division.
- Divisor is Table Size.
- Formula is **(H (key) = key % table size)**

E.g. consider the following data or record or key (36, 18, 72, 43, 6) table size = 8

Assume a table with 8 slots:

Hash key = key % table size

4	= 36 % 8
2	= 18 % 8
0	= 72 % 8
3	= 43 % 8
6	= 6 % 8

[0] 72
[1]
[2] 18
[3] 43
[4] 36
[5]
[6] 6
[7]

2. Mid Square Method:

We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute $44^2=1,936$. Extract the middle two digit 93 from the answer. Store the key 44 in the index 93.

93   44

3. Multiplicative Hash Function:

Key is multiplied by some constant value.

Hash function is given by,

H(key)=Floor (P * (key * A))

P = Integer constant [e.g. P=50]

A = Constant real number [A=0.61803398987],suggested by Donald Knuth to use this

constant

E.g. Key 107

H(107)=Floor(50*(107*0.61803398987))

=Floor(3306.481845)

H(107)=3306

Consider table size is 5000

4. Digit Folding Method:

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash key value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43, 65, 55, 46, 01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1.

Collision:

If two more keys hashes to the same index, the corresponding records cannot be stored in the same location. This condition is known as collision.

Characteristics of Good Hashing Function:

- It should be Simple to compute.
- Number of Collision should be less while placing record in Hash Table.
- **Hash function with no collision → Perfect hash function.**
- Hash Function should produce keys which are distributed uniformly in hash table.
- The hash function should depend upon every bit of the key. Thus the hash function that simply extracts the portion of a key is not suitable.

Collision Resolution Strategies / Techniques (CRT):

If collision occurs, it should be handled or overcome by applying some technique. Such technique is called CRT.

There are a number of collision resolution techniques, but the most popular are:

- **Separate chaining (**Open Hashing**)**
- **Open addressing**. (Closed Hashing)
  - **Linear Probing**
  - **Quadratic Probing**
  - **Double Hashing**

Separate chaining (Open Hashing)

- Open hashing technique.
- Implemented using singly linked list concept.
- Pointer (ptr) field is added to each record.
- When collision occurs, a separate chaining is maintained for colliding data.
- Element inserted in front of the list.
  - **H (key) =key % table size**

Two operations are there:-

- Insert
- Find

Structure Definition for Node

```
typedef Struct node *Position;
Struct node
{
        int data;                                          defines the nodes
        Position  next;
};
```
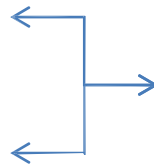
Structure Definition for Hash Table

```
typedef  Position  List;
struct Hashtbl
{                                          Defines the hash table which contains
        int Tablesize;                               array of linked list
        List * theLists;
};
```
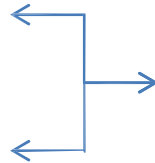
Initialization for Hash Table for Separate Chaining

```
Hashtable  initialize(int Tablesize)

{

       HashTable  H;

        int i;

        H = malloc (sizeof(struct HashTbl));              ➔ Allocates table

        H ➔ Tablesize = NextPrime(Tablesize);

        H➔the Lists=malloc(sizeof(List) * H➔Tablesize);   ➔ Allocates array of list

      for( i = 0; i < H ➔ Tablesize; i++ )

      {

      H ➔ TheLists[i] = malloc(Sizeof(Struct node));     ➔ Allocates list headers

      H ➔ TheLists[i] ➔ next = NULL;

      }

     return H;

}
```

Insert Routine for Separate Chaining

```
void insert (int Key, Hashtable H)

{

  Position P, newnode;                       *[Inserts element in the Front of the list always]*

  List L;
```

```
      P = find ( key, H );

      if(P = = NULL)

       {

              newnode = malloc(sizeof(Struct node));

              L = H → TheLists[Hash(key,Tablesize)];

              newnode → nex t= L → next;

              newnode → data = key;

             L → next = newnode;

         }}
```

Position find(int key, Hashtable H){

```
    Position P, List L;

    L = H →TheLists[Hash(key,Tablesize)];

    P = L → next;

    while(P != NULL && P → data != key)

          P = P → next;

    return P;}
```

        If two keys map to same value, the elements are chained together.

Initial configuration of the hash table with separate chaining. Here we use SLL(Singly Linked List) concept to chain the elements.

0		NULL
1		NULL
2		NULL
3		NULL
4		NULL
5		NULL
6		NULL
7		NULL
8		NULL
9		NULL

Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining.
The hash function is
H(key) = key % 10
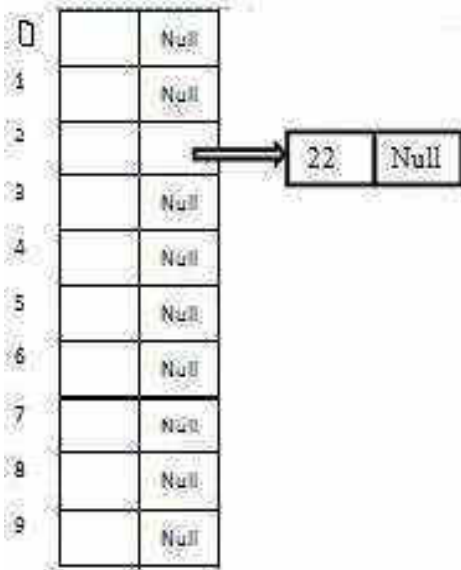
1. H(22) = 22 % 10 =2   **2. 84 % 10 = 4**
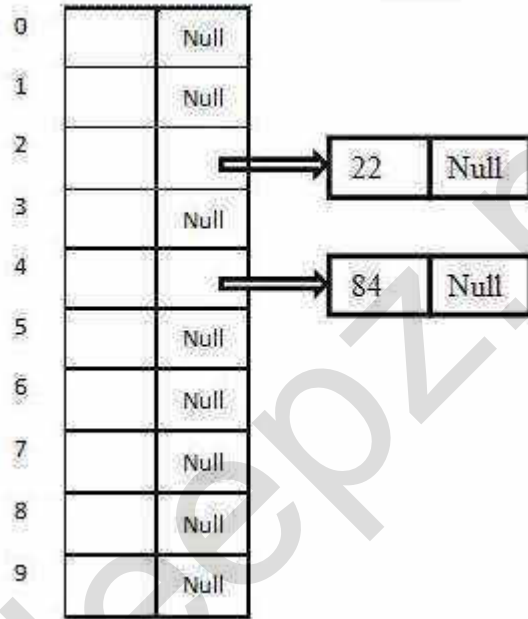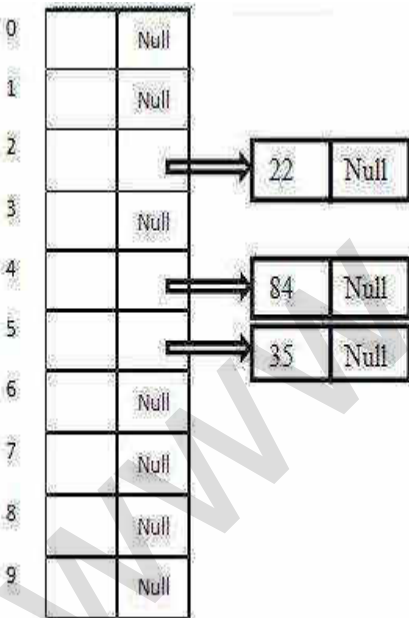
3.H(35)=35%10=5   **4. H(62)=62%10=2**
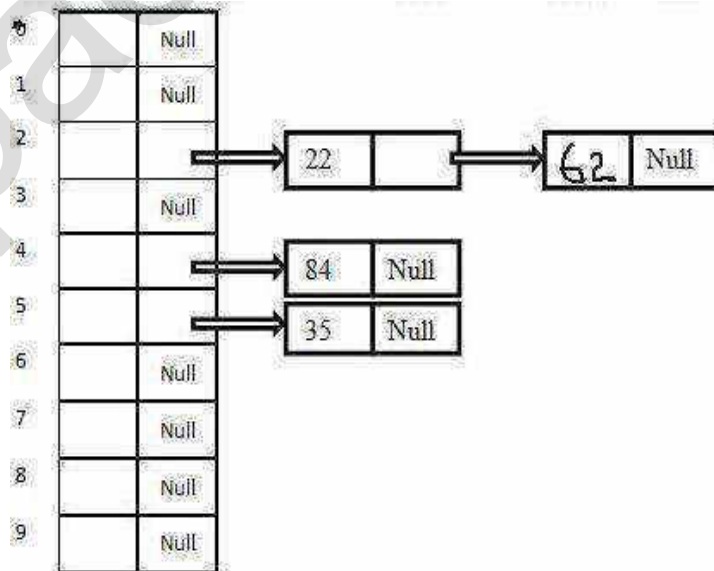
Advantages

      1. More number of elements can be inserted using array of Link List

Disadvantages

      1. It requires more pointers, which occupies more memory space.
      2.Search takes time. Since it takes time to evaluate Hash Function and also to traverse the List

Open Addressing

- Closed Hashing
- Collision resolution technique
- Uses $H_i(X)=(Hash(X)+F(i))$ mod Tablesize
- When collision occurs, alternative cells are tried until empty cells are found.
- Types:-
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- Hash function
    - $H(key) = key \% $ table size.
- Insert Operation
    - To insert a key; Use the hash function to identify the list to which the element should be inserted.
    - Then traverse the list to check whether the element is already present.
    - If exists, increment the count.
    - Else the new element is placed at the front of the list.

Linear Probing:

Easiest method to handle collision.

      Apply the hash function $H (key) = key \% $ table size
      $H_i(X)=(Hash(X)+F(i))$ mod Tablesize, where $F(i)=i$.

How to Probing:

- first probe – given a key k, hash to $H(key)$
- second probe – if $H(key)+f(1)$ is occupied, try $H(key)+f(2)$
- And so forth.

Probing Properties:

- We force $f(0)=0$
- The ith probe is to $(H (key) +f (i)) \%$ table size.
- If i reach size-1, the probe has failed.
- Depending on $f (i)$, the probe may fail sooner.
- Long sequences of probe are costly.

Probe Sequence is:

- $H (key) \% $ table size
- $H (key)+1 \% $ Table size
- $H (Key)+2 \% $ Table size

1. **H(Key)=Key mod Tablesize**

>   This is the common formula that you should apply for any hashing

>   If collocation occurs use Formula 2

2. **H(Key)=(H(key)+i) Tablesize**

Where i=1, 2, 3, …… etc

Example: - 89 18 49 58 69;   Tablesize=10

1.   H(89) =89%10

>   =9

2.   H(18) =18%10

>   =8

3.   H(49) =49%10

>   =9   ((coloids with 89.So try for next free cell using formula 2))

i=1 h1(49) = (H(49)+1)%10

>   = (9+1)%10

>   =10%10

>   =0

4.   H(58) =58%10

>   =8   ((colloids with 18))

i=1 h1(58) = (H(58) +1)%10

>   = (8+1) %10

>   =9%10

>   =9   =>Again collision

i=2   h2(58) =(H(58)+2)%10

>   =(8+2)%10

>   =10%10

>   =0   =>Again collision

	EMPTY	89	18	49	58	69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	
9		89	89	89	89	

Linear probing

insert(76)	Insert(93)	Insert(40)	Insert(47)	Insert(10)	Insert(55)
76%7=6	93%7=2	40%7=5	47%7=5	10%7=3	55%7=6
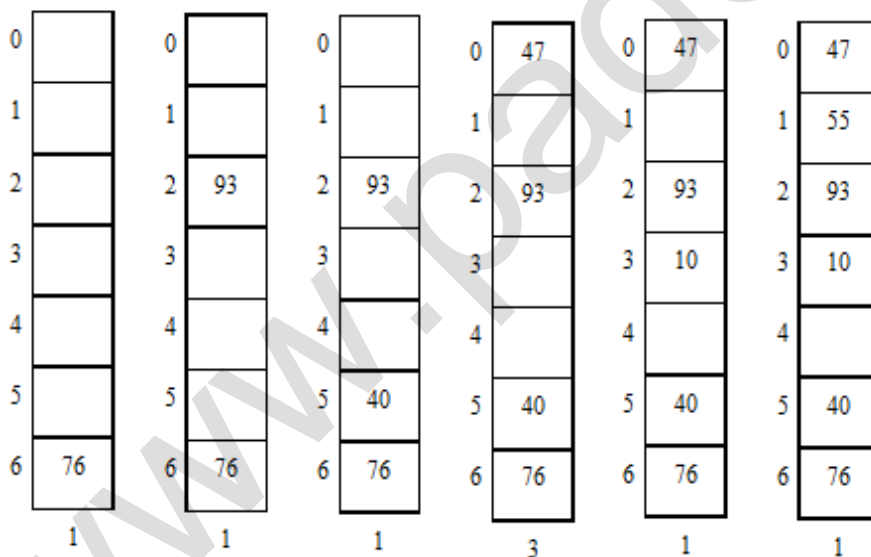
Quadratic Probing

To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move i^2 spots from the point of collision, where i is the number of attempts to resolve the collision.

- Another collision resolution method which distributes items more evenly.

- From the original index H, if the slot is filled, try cells H+1^2, H+2^2, H+3^2,.., H + i2 with wrap-around.

- $H_i(X)=(Hash(X)+F(i))$mod Tablesize,$F(i)=i^2$

- $H_i(X)=(Hash(X)+ i^2)$mod Tablesize

Insert 18, 89, 21   Insert 58   Insert 68

For **58**:
- H = hash(58, 10) = 8
- Probe sequence:
  i = 0, (8+0)% 10 = 8
  i = 1, (8+1) % 10 = 9
  i = 2, (8+4) % 10 = 2

For **68**:
- H = hash(68, 10) = 8
- Probe sequence:
  i = 0, (8+0)% 10 = 8
  i = 1, (8+1) % 10 = 9
  i = 2, (8+4) % 10 = 2
  i = 3, (8+9) % 10 = 7

Limitation: at most half of the table can be used as alternative locations to resolve collisions. This means that once the table is more than half full, it's difficult to find an empty spot. This new problem is known as secondary clustering because elements that hash to the same hash key will always probe the same alternative cells.

Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions forms the point of collision to insert.

There are a couple of requirements for the second function:

It must never evaluate to 0 must make sure that all cells can be probed.

$H_i(X)=(Hash(X)+i*Hash_2(X))$mod Tablesize

A popular second hash function is:

$Hash_2$ (key) = R - (key % R) where R is a prime number that is smaller than the size of the table.

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
  = 7 - (49 % 7)
  = 7 positions from [9]

Hash(58) = 58 % 10 = 8
  = 7 - (58 % 7)
  = 5 positions from [8]

Hash(69) = 69 % 10 = 9
  = 7 - (69 % 7)
  = 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Rehashing

Once the hash table gets too full, the running time for operations will start to take too long and may fail. To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

Advantage:

- A programmer doesn"t worry about table system.
- Simple to implement
- Can be used in other data structure as well

The new size of the hash table:

- should also be prime
- will be used to calculate the new insertion spot (hence the name rehashing)
- This is a very expensive operation! O(N) since there are N elements to rehash and the table size is roughly 2N. This is ok though since it doesn't happen that often.

The question becomes when should the rehashing be applied?

Some possible answers:
- once the table becomes half full
- once an insertion fails

- once a specific load factor has been reached, where load factor is the ratio of the number of elements in the hash table to the table size

Extendible Hashing

- Extendible Hashing is a mechanism for altering the size of the hash table to accommodate new entries when buckets overflow.

- Common strategy in internal hashing is to double the hash table and rehash each entry. However, this technique is slow, because writing all pages to disk is too expensive.

- Therefore, instead of doubling the whole hash table, we use a directory of pointers to buckets, and double the number of buckets by doubling the directory, splitting just the bucket that overflows.

- Since the directory is much smaller than the file, doubling it is much cheaper. Only one page of keys and pointers is split.
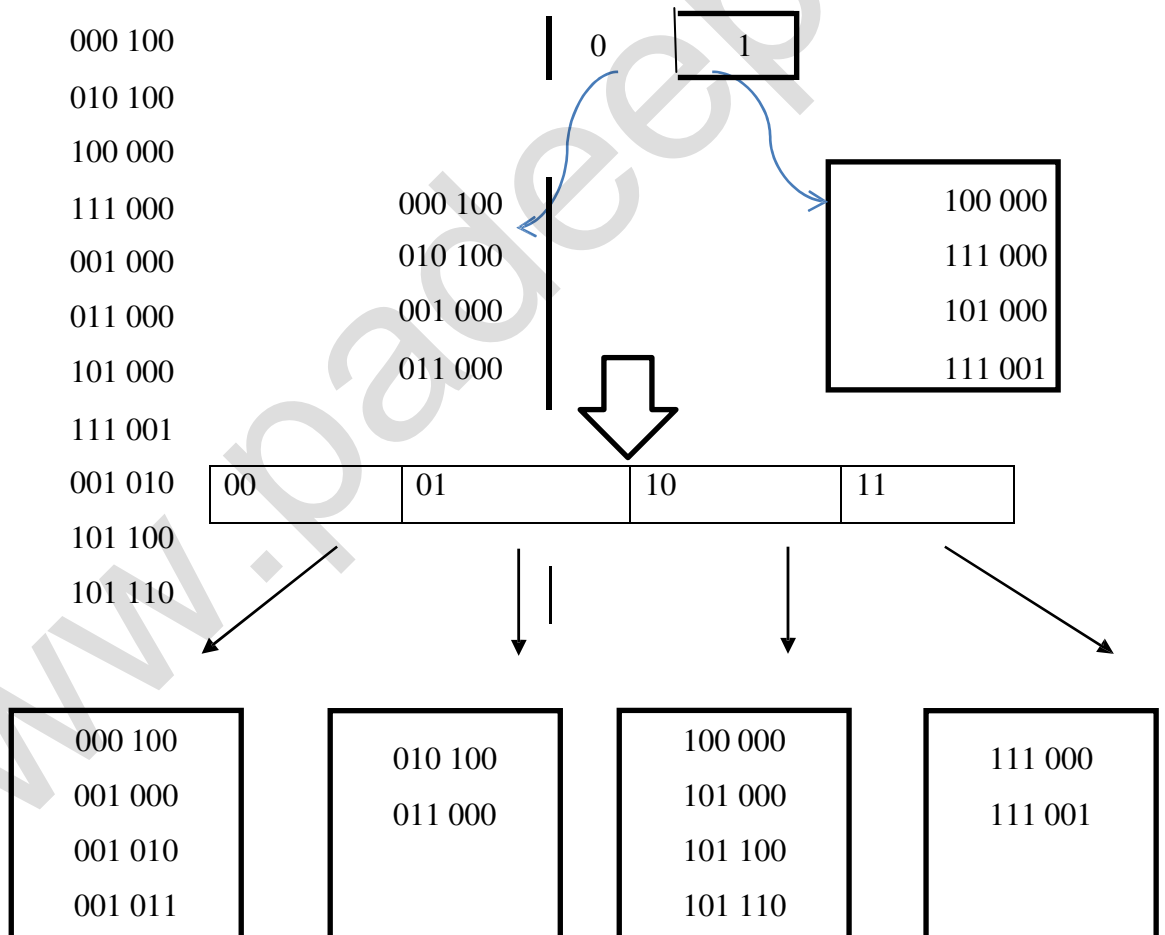
Arrays, Tables and Set

1. A hash table with ten buckets with one slot per bucket is shown in **Figure 1**, with the symbols S1 to S7 entered into it using some hashing function with linear probing. The worst case number of comparison required when the symbol being searched is not in the table is ……………………….

0	S7
1	S1
2	
3	S4
4	S2
5	
6	S5
7	
8	S6
9	S3

Fig. 1

2. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function $h(k) = k \bmod 10$ and linear probing. What is the resultant hash table?

3.

(A)

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(C)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(d)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5,15
6	
7	
8	18
9	

4. Consider a hash table of size 11 that uses open addressing with linear probing. Let $h(k) = k \bmod 11$ be a hash function used. A sequence of records with keys

43 36 92 87 11 4 71 13 14

is inserted into an initially empty hash table, the bins of which are indexed from zero to ten. What is the index of the bin into which the last record is inserted?

    (A) 3
    (B) 4
    (C) 6
    (D) 7

5.   The minimum number of comparisons required to determine if an integer appears more than $n/2$ times in a sorted array of n integers is

    (A) $\Theta(n)$
    (B) $\Theta(\log n)$
    (C) $\Theta(\log^* n)$
    (D) $\Theta(1)$

6.   Consider a hash function that distributes keys uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5.

          (A) 5   (B) 6   (C) 7 (D) 10

7.   Consider a hash table of size seven, with starting index zero, and a hash function (3x + 4) mod 7. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that – denotes an empty location in the table.

    (A) 8, -, -, -, -, -, 10
    (B) 1, 8, 10, -, -, -, 3
    (C) 1, -, -, -, -, -, 3
    (D) 1, 10, 8, -, -, -, 3

8.   A hash table contains 10 buckets and uses linear probing to resolve collisions. The key values are integers and the hash function used is key % 10. if the values 43, 165, 62, 123, 142 are inserted in the table, in what location would the key value 142 be inserted?

      (A) 2   (B) 3   (C) 4   (D) 6

9.   A program P reads in 500 integers in the range [0, 100] representing the scores of 500 students. It then prints the frequency of each score above 50. what would be the best way for P to store the frequencies?

    (A) An array of 50 numbers
    (B) An array of 100 numbers
    (C) An array of 500 numbers
    (D) A dynamically allocated array of 550 numbers

10.   Two matrices M_1 and M_2 are to be stored in arrays A and B respectively. Each array can be stored either in row-major or column-major order in contiguous memory locations. The time complexity of an algorithm to compute $M_1 \times M_2$ will be

    (A) Best if A is in row-major, and B is in column-major order
    (B) Best if both are in row-major order
    (C) Best if both are in column-major order
    (D) Independent of the storage scheme

11.   An advantage of chained hash table (external hashing) over open addressing scheme is

    (A) Worst case complexity of search operations is less
    (B) Space used is less
    (C) Deletion is easier
    (D) None of the above

12.   Consider a hash table with chaining scheme for overflow handling:

      (i)   What is the worst-case timing complexity of inserting n elements into such a table?

(ii) For what type of instances does this hashing scheme take the worst case time for insertion?

13. Let A be a n x n matrix such that the elements in each row and each column are arranged in ascending order. Draw a decision tree which finds 1st, 2nd and 3rd smallest elements in minimum number of comparisons

14. An array A contains n integers in locations A[0], A[1], ..., A[n-1]. It is required to shift the elements of the array cyclically to the left by K places, where 1 ≤ K ≤ n-1. An incomplete algorithm for doing this in linear time, without using another array is given below. Complete the algorithm in the blanks. Assume all variables are suitably declared.

```
min: = n;

i: = 0;

while _____ do

begin

        temp: = A[i];

        j: = i;

        while _____ do

        begin

                A[j]: = _____;

                j: = (j+K) mod n;

                if j < min then

                        min: = j;

        end;

        A[n + i – K) mod n]: = _____;

        i: = ...........................;

end;
```

15. An array A contains n integers in non-decreasing order, A[1] ≤ A[2] ≤ ....≤ A[n]. Describe, using Pascal-like pseudo code, a linear time algorithm to find i, j such that A[i] + A[j] = a given integer M, if such i, j exist.

16. A two dimensional array A[1..n][1..n] of integers is partially sorted if

For all i, j ε [1...., n-1]   A[i][j] < A[i][j+1] and.

                                                  A[i][j] < A[i+1][j]

Fill in the blanks:

The smallest item in the array is at A[i][j] where 1 = ☐   and   j = ☐ .

The smallest item is deleted. Complete the following O(n) procedure to insert item x (which is guaranteed to be smaller than any item in the last row or column) still keeping A partially sorted.   (4)

```
procedure          insert(x: integer);

var               i, j: integer;

begin

        (1)     i:= 1; j:= 1; A[i][j]:= x;

        (2)     while ( ☐ x>          ☐ x>

        (3)             if A[i+1][j] < A[i][j]     then begin

        (4)                     A[i][j]:= A[i+1][j]; i:= i+1;

        (5)     end

        (6)     else begin
```

(7)   [   ]

(8)   end

(9)   A[i][j]:=   [   ]

end

17.   Let A be a two dimensional array declared as follows:

A: array [1 … 10] [1 … 15] of integer;

Assuming that each integer takes one memory locations the array is stored in row-major order and the first element of the array is stored at location 100, what is the address of element A[i][j]?

(a) 15i + j + 84   (b) 15j + i + 84

(c) 10i + j 89   (d)10j+i+89

18.   Suppose you are given an array *s*[1 .. n] and a procedure *reverse*(s, i, j) which reverses the order of elements in *s* between positions i and j (both inclusive). What does the following sequence do, where $1 \le k < n$

19.   Consider the following declaration of a two dimensional array in C:
                Char a[100][100];

Assuming that the main memory is byte addressable and that the array is stored starting from address 0, the address of a[40][50] is

A.   4040
B.   4050
C.   5040
D.   5050