DATA STRUCTURES

UNIT-1:

Data Structure:

It is a particular way of organizing and storing data in a computer. So that it can be accessed and modified efficiently. A data structure will have a collection of data and functions or operations that can be applied on the data.

TYPES OF DATA STRUCTURES

1. Primitive and Non-Primitive Data Structure

Primitive Data Structure defines a set of primitive elements that do not involve any other elements as its subparts. These are generally built-in data type in programming languages.

E.g.:- Integers, Characters etc.

Non-Primitive Data Structures are that defines a set of derived elements such as Arrays, Structures and Classes.

2. Linear and Non-Linear Data Structure

A Data Structure is said to be linear, if its elements from a sequence and each element have a unique successor and predecessor.

E.g.:- Stalk, Queue etc.

Non-Linear Data Structures are used to represent data that have a hierarchical relationship among the elements. In non-linear Data Structure every element has more than one predecessor and one successor.

E.g.:- Trees, Graphs

3. Static and Dynamic Data Structure

A Data Structure is referred as Static Data Structure, if it is created before program execution i.e., during compilation time. The variables of Static Data Structure have user specified name.

E.g.:- Array

Data Structures that are created at run time are called as Dynamic Data Structure. The variables of this type are known always referred by user defined name instead using their addresses through pointers.

E.g.:- Linked List

4. Sequential and Direct Data Structure

This classification is with respect to access operation associated with the data type. Sequential access means the locations are accessed in a sequential form.

E.g.:- To access $\boldsymbol{n^{th}}$ element, it must access preceding $\boldsymbol{n-1}$ data elements

E.g.:- Linked List

Direct Access means any element can access directly without accessing its predecessor or successor i.e., $\boldsymbol{n^{th}}$ element can be accessed directly.

E.g.:- Array

OPERATIONS ON DATA STRUCTURE

The basic operations that can be performed on a Data Structure are:

i) Insertion: - Operation of storing a new data element in a Data Structure.
ii) Deletion: - The process of removal of data element from a Data Structure.
iii) Traversal: - It involves processing of all data elements present in a Data Structure.
iv) Merging: - It is a process of compiling the elements of two similar Data Structures to form a new Data Structure of the same type.
v) Sorting: - It involves arranging data element in a Data Structure in a specified order.
vi) Searching: - It involves searching for the specified data element in a Data Structure.

COMPLEXITY OF ALGORITHMS

Generally algorithms are measured in terms of time complexity and space complexity.

1. Time Complexity

Time Complexity of an algorithm is a measure of how much time is required to execute an algorithm for a given number of inputs. And it is measured by its rate of growth relative to a standard function. Time Complexity can be calculated by adding compilation time and execution time. Or it can do by counting the number of steps in an algorithm.

2. Space Complexity

Space Complexity of an algorithm is a measure of how much storage is required by the algorithm. Thus space complexity is the amount of computer memory required during program execution as a function of input elements. The space requirement of algorithm can be performed at compilation time and run time.

ASYMPTOTIC ANALYSIS

For analysis of algorithm it needs to calculate the complexity of algorithms in terms of resources, such as time and space. But when complexity is calculated, it does not provide

the exact amount of resources required. Therefore instead of taking exact amount of resources, the complexity of algorithm is represented in a general mathematical form which will give the basic nature of algorithm.

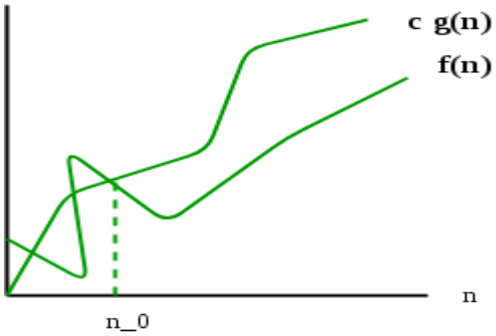
Thus on analyzing an algorithm, it derives a mathematical formula to represent amount of time and space required for the execution. The Asymptotic analysis of algorithm evaluates the performance of an algorithm in terms of input size It calculates how does the time taken by an algorithm increases with input size. It focuses on:

i) Analyzing the problem with large input size.
ii) Considers only leading terms of formula. Since the lower order term contracts lesser to the overall complexity as input grows larger.
iii) Ignores the coefficient of leading term.

ASYMPTOTIC NOTATION

The commonly used Asymptotic Notations to calculate the complexity of algorithms are:

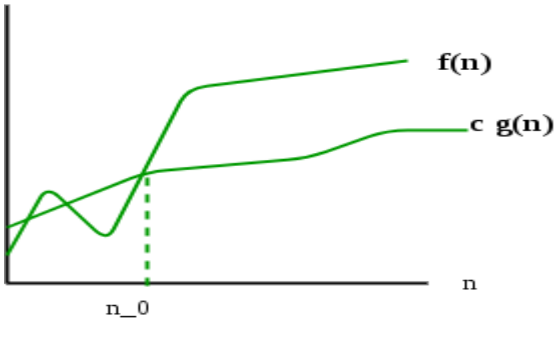
1. **Big Oh – O: -** The notation **O**(n) is the formal way to express the upper bound of an algorithm's complexity.



f(n) = O(g(n))

```
O(g(n)) = { f(n): there exist positive constants c and
                  n0 such that 0 <= f(n) <= c*g(n) for
                  all n >= n0}
```

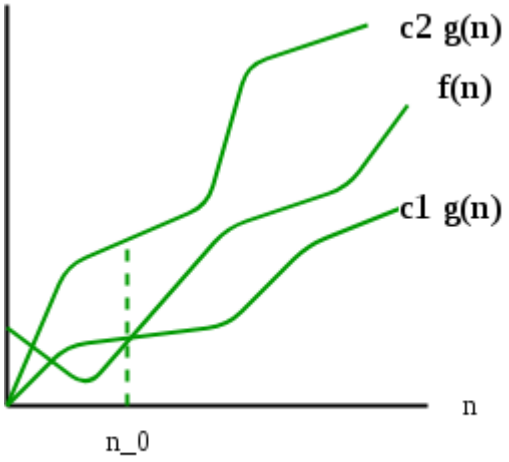
2. **Big Omega – Ω: -** The notation **Ω**(n) is the formal way of representing lower bound of an algorithm's complexity.



f(n) = Omega(g(n))

```
Ω (g(n)) = {f(n): there exist positive constants c and
                  n0 such that 0 <= c*g(n) <= f(n) for
                  all n >= n0}.
```

3. **Big Theta** – **Θ**: - The notation **Θ**($\boldsymbol{n}$) is the formal way to representing lower bound and upper bound of an algorithm's complexity.



f(n) = theta(g(n))

```
Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such
                 that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}
```

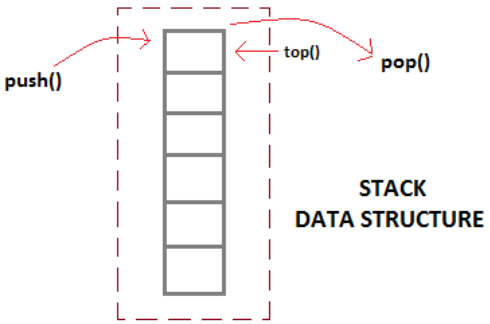
STACK

Stack is a linear Data Structure in which all the insertion and deletion operations are done at one end, called top of the stack and operations are performed in Last In First Out [**LIFO**] order. A stack can be implemented by using an array or by using linked list.

The array representation method needs to set the amount memory needed initially. So they are limited in size, i.e. it can't shrink or expand. Linked List representation uses Dynamic memory management method. So they are not limited in size i.e. they can shrink or expand.

The class declaration for stack using array is represented as:

```
class stack
 {
        int a[10], st;
        public:
        stack()
        {
        st =-1;
        }
        void push(int);
        void pop();
};
```



The operations performed on Stack are:

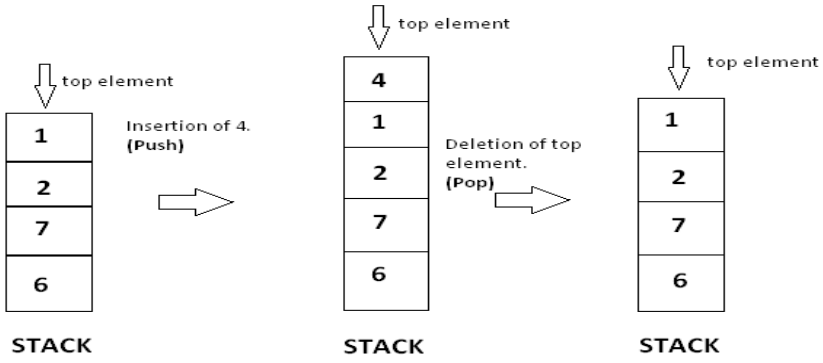
1. **Push Operation:** This function is used to store data into the stack. The Push operation performs following steps:
   - **i.** Check whether stack is full or not.
   - **ii.** If full, message that stack is full.
   - **iii.** If not full, it will increase the stack top by 1 and to that location new data is stored.

```
void stack :: push(int d)
{
        if (st>=size-1)
                cout<<"stack is full";
        else
                a[++st]=d;
}
```

2. **Pop Operation:** This function is used to remove data from stack. The Pop function performs following functions:
   - **i.** Check if stack is empty or not.
   - **ii.** If stack is empty, print a message.
   - **iii.** If stack is not empty, print the element at the stack top and decrement stack top by one.

```
void stack :: pop()
{
        if(st==-1)
                cout<<"stack is empty";
        else
                cout<<a[st--];
}
```

The Time complexity of push and pop operations are O(1).



Implement Stack ADT using array:

```
//Template class for Stack
#include<iostream>
using namespace std;
template<class T>
class stack
{
int size,top;
T *a;
public:
stack(int);
int isEmpty();
int isFull();
void push();
void pop();
};
template<class T>
stack<T>::stack(int s)
{
size=s;
a= new T[size];
top=-1;
}
```

```
template<class T>
int stack<T>::isEmpty()
{
if(top==-1)
return 1;
else
return 0;
}
template<class T>
int stack<T>::isFull()
{
if(top==size-1)
return 1;
else
return 0;
}
template<class T>
void stack<T>::push()
{
if(isFull())
{
cout<<"Stack is Full"<<endl;
}
else
{
int item;
cout<<"Enter the Data"<<endl;
cin>>item;
top++;
a[top]=item;
}}
```

```
template<class T>
void stack<T>::pop()
{
if(isEmpty())
cout<<"Stack is Empty"<<endl;
else
{
T x;
x=a[top];
top--;
cout<<"Data= "<<x<<endl;
} }
main()
{
stack<int>S(3);
int y=1,op,t;
while(y==1)
{
cout<<"Enter 1-PUSH 2-POP 3-EXIT"<<endl;
cin>>op;
switch(op)
{
case 1: S.push();
        break;
case 2: S.pop();
        break;
case 3: y=0;
        break;
default: cout<<"Invalid Option"<<endl;
} }
return 0; }
```

APPLICATIONS OF STACK

1) Converting infix expression to postfix and prefix expression.
2) Evaluating postfix expression.
3) Checking nested parenthesis.
4) Reversing a string.
5) Processing function calls.
6) Stimulating recursion.
7) Parsing.
8) Book tracking algorithm.

POLISH NOTATIONS

This gives two alternatives to represent an arithmetic expression called postfix and prefix notations. The fundamental property of polish notation is that the order in which the operations are to be performed is determined by positions of operators and operands in the expression. Hence the advantage is that parenthesis is not required while writing expressions in polish notations. It also overrides associatively of operators. The conventional way of writing expression is called infix because in binary operations the operators occur between the operands. In postfix notation the operator is written after its operands.

In postfix notation, the operator is written after its operands.

E.g.: AB+

In prefix notation, the operator precedes its operands.

E.g.: +AB

Infix	Postfix	Prefix
(A+B)*C	AB+C*	*+ABC

Need for Prefix and Postfix expressions

1) Need for parenthesis as in infix expression is overcome in postfix and prefix notations.
2) Priority of the operators is no longer relevant.
3) The order of evaluation depends on the position of the operator and not on priority and associativity.
4) Expression evaluation is simpler.

Algorithm for evaluating Postfix expression

1) Start
2) Let E denote postfix expression.
3) Let stacktop=-1.
4) While(1) do
Begin x=getnext(E)
If(x==#)
Then return
If x is an operand, then push(x) otherwise
Begin
op2=pop,
op1=pop,
op3=evaluate(op1,x,op2)
Push(op3)
End
End while
5) Stop.

Algorithm for Infix to Postfix conversion

1) Start
2) Let E be the Infix expression.
3) Scan expression E from left to right character by character till character is #.
ch= getnext(E)
4) While(ch!=#)
If (ch==')')
Then ch = pop();
While(ch!='(')
Display ch
ch=pop();
End while
If(ch==operand) display ch
If(ch==operator)
If(icp>isp), then push(ch)
otherwise
While(icp <=isp)
ch = pop()
display ch
end while
ch= getnext()
end While

5) If(ch==#)
Then while(!isempty())
ch=pop()
display ch
end while
6) Stop.

Q. Implement stack ADT for infix to postfix conversion

```
//PROGRAM FOR INFIX TO POSTFIX CONVERSION
#include<iostream>
#include<string.h>
using namespace std;
template<class T>
class stack
{
int size,top;
T *a;
public:
stack(int);
int isEmpty();
int isFull();
void push(T);
T pop();
T getNext(T*,int);
int isOperator(T);
int ICP(T);
int ISP(T);
};
template<class T>
T stack<T>::getNext(T w[], int i)
{
return w[i];
}
```

```
template<class T>
int stack<T>::isOperator(T x)
{
int i;
if(x=='+'||x=='-'||x=='*'||x=='/'||x=='^'||x=='(')
return 1;
else
return 0;
}
template<class T>
int stack<T>::ISP(T y)
{
if(y=='^')
        return 3;
else if(y=='*'||y=='/')
        return 2;
else if(y=='+'||y=='-')
        return 1;
else if(y=='(')
        return 0;
}
template<class T>
int stack<T>::ICP(T z)
{
if(z=='^'||z=='(')
        return 4;
else if(z=='*'||z=='/')
        return 2;
else if(z=='+'||z=='-')
        return 1;
}
```

```
template<class T>
stack<T>::stack(int s)
{
size=s;
a= new T[size];
top=-1;
}
template<class T>
int stack<T>::isEmpty()
{
if(top==-1)
return 1;
else
return 0;
}
template<class T>
int stack<T>::isFull()
{
if(top==size-1)
return 1;
else
return 0;
}
template<class T>
void stack<T>::push(T item)
{
if(!isFull())
a[++top]=item;
}
```

```
template<class T>
T stack<T>::pop()
{
if(!isEmpty())
return a[top--];
}
main()
{
stack<char>S(10);
char c[20],ch,t;
int i=0,l;
cout<<"Enter the INFIX expression ended with #"<<endl;
cin>>c;
ch=S.getNext(c,i++);
while(ch !='#')
{
        if(ch==')')
        {
                ch=S.pop();
                while(ch != '(')
                {
                cout<<ch;
                ch=S.pop();
                }
        }
        else if(S.isOperator(ch))
                {
                t=S.pop();
                if(S.ICP(ch)>S.ISP(t))
                {
                S.push(t);
```

```
                S.push(ch);
                }
                else
                {
                while(S.ICP(ch)<=S.ISP(t))
                {
                cout<<t;
                t=S.pop();
                }
                S.push(t);
                S.push(ch);
                }
                }
        else
                cout<<ch;

ch=S.getNext(c,i++);
}
                while(!S.isEmpty())
                cout<<S.pop();
cout<<endl;
return 0;
}
```

Q. Implement Post fix Evaluation

```
#include<iostream>
#include<string.h>
using namespace std;
template<class T>
class stack
{
int size,top;
T *a;
public:
stack(int);
int isEmpty();
int isFull();
void push(T);
T pop();
T getNext(T*,int);
int isOperator(T);
T eval(T,T,T);
};
template<class T>
T stack<T>::eval(T a,T b,T c)
{
T t;
switch(b)
{
case'+': t=a+c;
        return t;
        break;
case'-': t=a-c;
        return t;
```

```
        break;
case'*': t=a*c;
        return t;
        break;
case'/': t=a/c;
        return t;
} }
template<class T>
T stack<T>::getNext(T w[], int i)
{
return w[i];
}
template<class T>
int stack<T>::isOperator(T x)
{
int i;
if(x=='+'||x=='-'||x=='*'||x=='/'||x=='^'||x=='(')
return 1;
else
return 0;
}
template<class T>
stack<T>::stack(int s)
{
size=s;
a= new T[size];
top=-1;
}
template<class T>
int stack<T>::isEmpty()
{
```

```
if(top==-1)
return 1;
else
return 0;
}
template<class T>
int stack<T>::isFull()
{
if(top==size-1)
return 1;
else
return 0;
}
template<class T>
void stack<T>::push(T item)
{
if(!isFull())
a[++top]=item;
}
template<class T>
T stack<T>::pop()
{
if(!isEmpty())
return a[top--];
}
main()
{
stack<char>S(10);
char c[20],ch,op1,op2,op3;
int i=0;
cout<<"Enter the POSTFIX expression ended with #"<<endl;
```

```
cin>>c;
ch=S.getNext(c,i++);
while(ch !='#')
{
        if(S.isOperator(ch))
        {
        op2=S.pop();
        op1=S.pop();
        op3=S.eval(op1,ch,op2);
        S.push(op3);
        }
        else
        {
        ch=ch-'0';
        S.push(ch);
        }
ch=S.getNext(c,i++);
}
        if(ch=='#')
        {
        op3=S.pop();
        op3=op3+'0';
        cout<<op3<<endl;
        }
return 0;
}
```

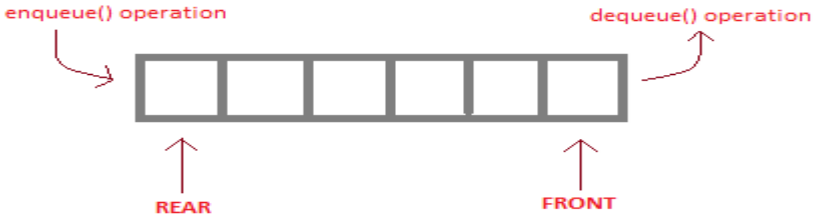
QUEUE

It's a linear Data Structure in which insertion is done at rear end and deletion at front end. The operations are performed in First In First Out order [**FIFO**]. This can be implemented by using array or by linked list. The array representation uses static memory allocation method, so they are limited in size i.e. they cannot shrink or expand. The linked

list representation uses dynamic memory management method, so they can shrink or expand.

The class declaration for queue using array is represented as:

```
class queue
{
        int a[10], f, r;
        public:
        queue()
        {
                f=-1;
                r=-1;
        }
        void enque(int);
        void deque();
};
```



The operations performed on queue are:

1. **Insertion or Enqueue:** This function is used to store data into the Queue. The operation perform the following steps:
    - **i.** Check whether Queue is full or not.
    - **ii.** If full, print a message that queue is full.
    - **iii.** If not full, then increment rear by 1 and to that location new data is inserted.

```
void queue :: enqueue(int d)
{
        if(r>=size-1)
                cout<<"Queue is full";
        else
                a[++r]=d;
}
```

2. **Deletion or Dequeue:** This function is used to remove data from the Queue. The Dqueue function performs the following steps:
    - **i.** Check whether Queue is empty or not.
    - **ii.** If empty, message that Queue is empty.
    - **iii.** If not empty, print the element represented by the front location and increment the front by 1.

```
void queue :: dequeue
{
        if((f==-1) || (front>rear))
                cout<<"Queue is empty";
        else
                cout<<a[f++];
}
```

Q. Implement Queue ADT using array

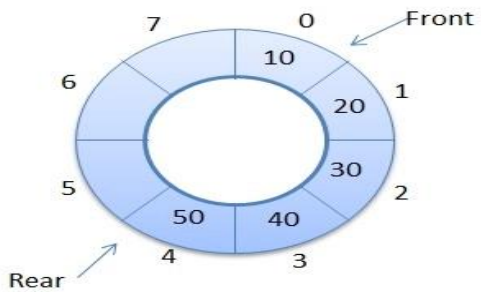
```
// Queue implementation using array

#include<iostream>

using namespace std;

template<class T>

class queue

{

int size,f,r;

T *a;

public:

queue(int);

int isEmpty();

int isFull();

void enqueue(T);

void dequeue();

};

template<class T>

queue<T>::queue(int s)

{

size=s;

a=new T[size];

f=-1;

r=-1;

};

template<class T>
```

```
int queue<T>::isFull()
{
if(r==size-1)
return 1;
else
return 0;
}
template<class T>
int queue<T>::isEmpty()
{
if(f==r)
return 1;
else
return 0;
}
template<class T>
void queue<T>::enqueue(T x)
{
if(isFull())
cout<<"Queue is Full"<<endl;
else
a[++r]=x;
}
template<class T>
void queue<T>::dequeue()
{
if(isEmpty())
cout<<"Queue is Empty"<<endl;
else
cout<<a[++f]<<endl;
}
```

```
int main()
{
queue<int> Q(5);
int y=1,op,t;
while(y==1)
{
cout<<"Enter 1.Enqueue 2.Dequeue 3.Exit"<<endl;
cin>>op;
switch(op)
{
case 1:  cout<<"Enter the Data"<<endl;
         cin>>t;
         Q.enqueue(t);
         break;
case 2:  Q.dequeue();
         break;
case 3:  y=0;
} }
return 0;
}
```

CIRCULAR QUEUE

It is a linear Data Structure in which operations are performed in **FIFO** order and last position is connected back to the first position to make a circle. A Circular Queue is also called as ring buffer. In a normal Queue it is possible to insert elements until queue becomes full, but once queue become full, it is impossible to add next data even if there is a space in front of queue. This situation can be overridden in Circular Queue.



Operations on Circular Queue

1. **Enqueue:** This function is used to insert an element to the circular queue. A new data is inserted at the rear position. Steps followed for an Enqueue:
   - **i.** Check whether Queue is full or not.
   - **ii.** If queue is full, display message that queue is full.
   - **iii.** If not full, front is set to zero, rear end is moded with the circular queue size after rear is incremented by 1 and then add the element to the location.

Enqueue function can be defined as:

```
void queue :: enqueue
{
        if((isfull())
                cout<<"Queue is full";
        else
        {
                if(f==-1)
                        f=0;
                        r=(r+1)%s;
                        a[r]=d;
         }
}
```

2. **Dequeue:** This function is used to delete an element from the circular queue. In circular queue elements are always deleted from different positions. Steps followed for a Dequeue are:
   - **i.** Check whether Queue is empty.
   - **ii.** If empty, give a message queue is empty.
   - **iii.** If not empty, the element at that location is displayed. Then front end and rear end is set to -1 if front is equal to rear otherwise front is moded by the queue size after front is incremented by 1.

Dequeue function can be defined as:

```
void queue :: dequeue
{
        if((isempty())
                cout<<"Queue is empty";
        else
        {
                cout<<a[f];
                if(f==r)
                {
                        f=-1;
                        r=-1;
                }
                else
                        f=(f+1)%s;
        } }
```

Implement Circular Queue ADT:

```
//Circular Queue implementation
#include<iostream>
using namespace std;
template<class T>
class queue
{
int size,f,r;
T *a;
public:
queue(int);
int isEmpty();
int isFull();
void enqueue(T);
void dequeue();
};
template<class T>
queue<T>::queue(int s)
{
size=s;
a=new T[size];
f=-1;
r=-1;
}
template<class T>
int queue<T>::isEmpty()
{
if(f>r||f==-1)
```

```
return 1;
else
return 0;
}
template<class T>
int queue<T>::isFull()
{
if((r+1)%size==f)
return 1;
else
return 0;
}
template<class T>
void queue<T>::enqueue(T x)
{
if(isFull())
cout<<"Queue is Full"<<endl;
else
{
if(f==-1)
{
f=0;
r=0;
a[r]=x;
}
else
{
r=(r+1)%size;
```

```
a[r]=x;
}}}
template<class T>
void queue<T>::dequeue()
{
if(isEmpty())
cout<<"Queue is Empty"<<endl;
else
{
cout<<a[f]<<endl;
if(f==r)
{
f=-1;
r=-1;
}
else
f=(f+1)%size;
} }
int main()
{
queue<int> Q(5);
int y=1,op,t;
while(y==1)
{
cout<<"Enter 1.Enqueue 2.Dequeue 3.Exit"<<endl;
cin>>op;
switch(op)
{
```

```
case 1: cout<<"Enter the data"<<endl;
        cin>>t;
        Q.enqueue(t);
        break;
case 2:  Q.dequeue();
        break;
case 3:  y=0;
} }
return 0;
}
```

DOUBLE ENDED QUEUE

Dequeue defines a data structure where elements can beaded or deleted at either at front end or rear end but no changes can be made elsewhere. It support both Stack like and Queue like capability. A Dequeue can be implemented as either by using an array or by using a linked list.

The operations associated with a double ended queue are:

1. **Enqueue Front:** This function adds element at the end of queue.
2. **Enqueue Rear:** This function adds element at the rear end of the queue.
3. **Dequeue Front:** This function delete element from the front end of the queue.
4. **Dequeue Rear:** This function delete element from the rear end of the queue.

For Stack implementation using this queue, the functions enqueue front and dequeue front are used as push and pop functions respectively. A Dequeue is useful where data to be stored has to be ordered, compact storage is needed and retrieval of data has to be faster.

Variations of Dequeue

1. **Input restricted Double Ended Queue: -** These types of Double ended queues allow insertions only at one end.

2. **Output restricted Double Ended Queue: -** These types of Double ended queues allow deletion from only one end.

PRIORITY QUEUES

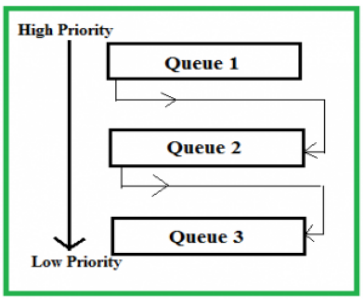
A priority Queue is a collection of a finite number of prioritised elements. Elements can be inserted in any order in the priority queue but when the element is removed, it is always the element with highest priority.

The following rules are applied to maintain the priority queue:

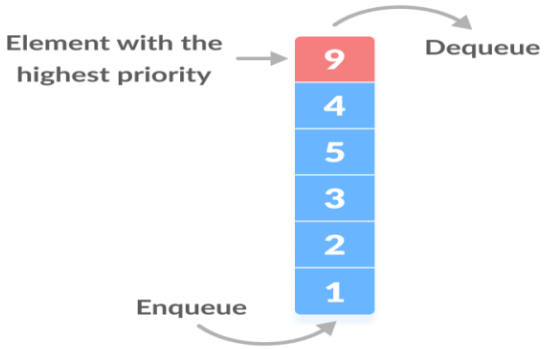
1. Element with highest priority is processed before any element of lower priority.
2. If there where elements with same priority, elements added first in the queue would get processed first.

There are two ways to implement priority queue:

1. **Implement separate queues for each priority**: - In this case elements are removed from front of the queue. Elements of the second queue are removed only when the first queue is empty. And elements from third queue are removed only when the second queue is empty.



2. **Implement by using a Structure or by a Class for the queue**: -Here each element in the queue has a data part and priority part of the element. The highest priority element is stored at the front of the queue and lowest priority element at the rear.



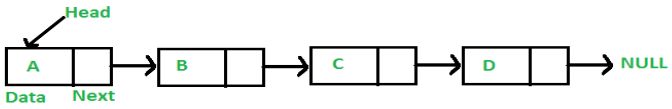
UNIT II

LINKED LIST

It is an ordered collection of data in which each element (node) contains data and links i.e. address to next element. The first node of the linked list is called head node which indicates the beginning of the linked list. The end node is called a tail node whose link will be assigned to null to indicate the end of linked list.

The operations that can be performed on a linked list are:

1. **Insertion of a new node**
2. **Deletion of a node**
3. **Traversal**



The class structure for a node can be defined as:

```
class node
{
        int data;
        node * next;
};
```

There are two types of linked list:

1. Singly linked list
2. Doubly linked list

Implement Singly Linked list ADT:

```
#include<iostream>
using namespace std;
class node
{
public:
int data;
node *next;};


class ll:public node
      {
      node *head,*tail;
      public:
      ll()
      {
      head=NULL;
      tail=NULL;
      }
```

```
            void create();
            void insert();
            void disp();
            void del();
      };

void ll::create()
            {
            node *temp;
            temp=new node;
            int n;
            cout<<"Enter the data";
            cin>>n;
            temp->data=n;
            temp->next=NULL;
                  if(head==NULL)
                  {
                  head=temp;
                  tail=head;
                  }
                  else
                  {
                  tail->next=temp;
                  tail=temp;
                  }
                  }
void ll::insert()
{
node *prev,*cur;
prev=NULL;
cur=head;
int count=1,pos,ch,n;
node *temp;
temp=new node;
cout<<"Enter the data"<<endl;
cin>>n;
temp->data=n;
temp->next=NULL;
cout<<"Enter 1.Insert as First node 2. Insert as Tail node 3. in
between node"<<endl;
cin>>ch;
switch(ch)
{
case 1:
      temp->next=head;
      head=temp;
      break;
case 2:
      tail->next=temp;
      tail=temp;
      break;
case 3:
      cout<<"Enter the position"<<endl;
      cin>>pos;
      while(count!=pos)
      {
      prev=cur;
```

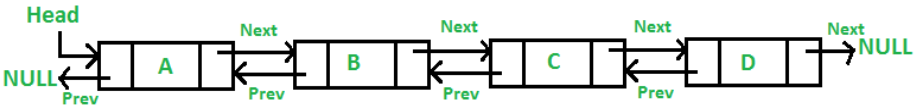
```
      cur=cur->next;
      count++;
      }
      if(count==pos)
      {
      temp->next=cur;
      prev->next=temp;
      }
      else
      cout<<"Unable to Insert";
}
}

void ll::del()
{
node *prev,*cur;
prev=NULL;
cur=head;
int count=1,pos,ch,n;
cout<<"Enter 1.Delete First node 2. Delete Tail node 3. Delete in
between node"<<endl;
cin>>ch;
switch(ch)
{
case 1:
      if(head!=NULL)
      {
      cout<<"Data deleted is "<<head->data<<endl;
      head=head->next;
      }
      else
      cout<<"Unable to Detete"<<endl;
      break;
case 2:
      while(cur!=tail)
      {
      prev=cur;
      cur=cur->next;
      }
      if(cur==tail)
      {
      cout<<"Data deleted is "<<cur->data<<endl;
      prev->next=NULL;
      tail=prev;
      }
      else
      cout<<"Unable to Detete"<<endl;
      break;
case 3:
      cout<<"Enter the position"<<endl;
      cin>>pos;
      while(count!=pos)
      {
      prev=cur;
      cur=cur->next;
      count++;
      }
```

```
      if(count==pos)
      {
      cout<<"Data deleted is "<<cur->data<<endl;
      prev->next=cur->next;
      }
      else
      cout<<"Unable to Detete"<<endl;
}
}


void ll::disp()
{
node *temp;
temp=head;
if(temp==NULL)
cout<<"Empty List"<<endl;
while(temp!=NULL)
{
cout<<temp->data<<" ";
temp=temp->next;
}

}
int main()
{
ll L;
int op,y=1;
while(y==1)
{
cout<<"Enter 1.Create 2.Insert 3.Delete 4.Display 5.Exit"<<endl;
switch(op)
{
case 1: L.create();
           break;
case 2: L.insert();
           break;
case 3: L.delete();
           break;
case 4: L.disp();
           break;
case 5: y=0;
           break;
default: cout<<"Invalid Option"<<endl;
}
return 0;
}
```

DOUBLY LINKED LIST (DLL)

In a single linked list each node provides information about where the next node is located. It has no knowledge about where the previous node is located. This causes difficulty to access $(i-1)^{th}$ or $(i-2)^{th}$ nodes from i^{th} node. In order to access such node it has to be traverse from head node. For handling such difficulties, doubly linked list

(**DLL**) is introduced where each node contains two links, one to its predecessor and the other to its successor.



The node class for the DLL can be declared as:

```
class node
{
        int data;
        node * next;
        node * prev;
};
```

For inserting a new node in DLL the following steps has to be followed:

Step 1: - node1 → next = temp

Step 2: - node2 → prev = temp

Step 3: - temp → next = node2

Step 4: - temp → prev = node1

For deleting a node from doubly linked list:

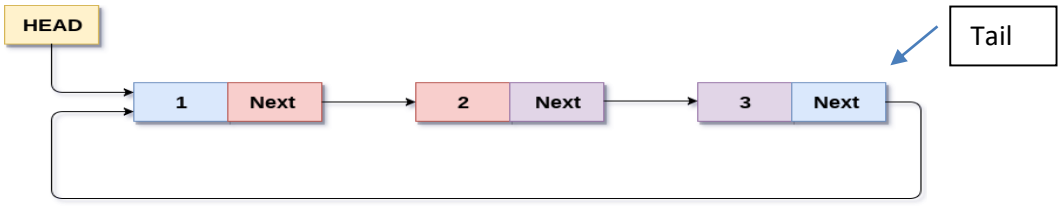
Step 1: - (cur → prev) → next = cur → next

Step 2: - (cur → next) → prev = cur → prev

CIRCULAR LINKED LIST

In circular linked list the last mode contains the address of first mode instead of null. This change will make last node point to first node of the list.

In this case of linked list, from any node of list it is possible to research any other node in the case and it keeps traversal procedure an unending one. Circular linked list is usually used for memory management.



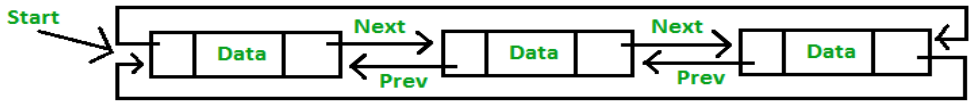
Circular Singly Linked List

DOUBLY CIRCULAR LINKED LIST

In doubly linked list the last node link is set to the first node of the list and first node link, previous link is set to the last node of the list.

Tail →next = head

Head →prev = tail



DYNAMIC MEMORY ALLOCATION

The process of allocating memory at runtime is known as dynamic memory allocation. The special area in memory called heap is reserved for dynamic allocation. Any new dynamic variable created by a programme consumes some memory in heap. The heap is a pool of memory from which new operator allocates memory. The memory allocated from system heap by using new operator can be deallocated by using delete operator. The users can dynamically allocate and deallocate memory for any built-in or userdefined data structure by using new and delete operators respectively.

1. **New Operator**

   The new operator creates a new dynamic object of a specified type and returns a pointer those points to this new object. If it fails to create the desired object then it will return zero.

   E.g.: - My type *ptr;

   Ptr = new My type;

2. **Delete Operator:**

   The object created using new operator exits till it is explicitly deleted or till the programme runs. To dynamically destroy an allocated object and free the space occupied by the object, the delete operator is used.

   E.g.: - Delete ptr;

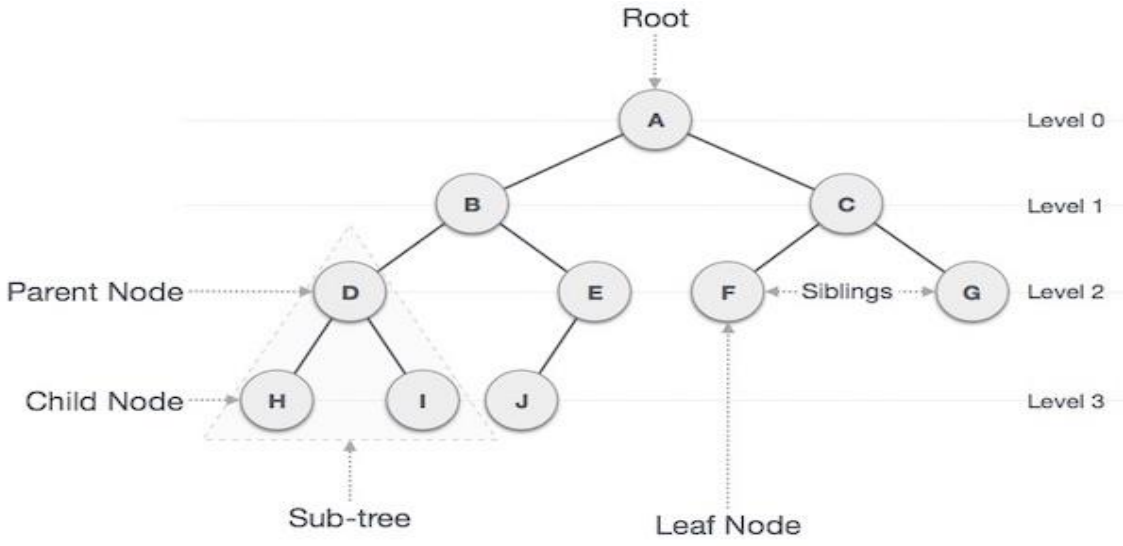
   The delete operator eliminates the dynamic variable and returns the memory that it has occupied in the heap. The memory can then reuse to create new dynamic variables.

UNIT III

TREES

It is non-linear data structure which contains a set of nodes and edges that have parent-child relationship. The starting node of a tree is called root node. Every other node other than root node have parent node. Each nodes of a tree can have any number of children.

E.g.: -



Tree Terminology

1. **Root**

   In a tree data structure first node is known ad root node. Any tree must have a root node and there must be only one root node foe a tree.

   E.g.: - In the above tree node A is called root node.

2. **Edge**

   The connecting link between any two nodes is called as edge. In a tree with n no. of nodes there will be maximum of (n-1) edges.

   E.g.: - Edge AB connects node A and B

3. **Parent Node**

   The node which is predecessor of any node is called as parent node.

   Eg:- B is parent of D and E and A is the parent of B and C.

4. **Child Node**

   Node which is successor of any node is called child node. In a tree any parent node can have any number of child nodes. The entire node except root node is child node.

   Here B and C are children of A.

5. **Leaf Node**

   Nodes which do not have a child are called as leaf nodes. They are also called as terminal nodes.

   E.g.: - Here H,I,J,F,G are leaf nodes.

6. **Intermediate Node / Internal Node**

   The nodes which have at least one child is called internal node. These are also called as non-terminal nodes.

   Eg: - B and C

7. **Sibling**

   Nodes which belong to same parent are called siblings.

   E.g.: - D and E

8. **Degree**

   The total number of children of a node is called degree of that node.

   E.g.: -

Node	Degree
A,B,C,D	2
E	1
H,I,J,F,G	0

9. **Level**

   In a tree, each step from top to bottom is called as level. The level count starts with zero from root node and incremented by 1 at each step.

   E.g.: - A at level 0,

   B,C at level 1

10. **Height**

    Total number of edges from leaf node to a particular node in the largest path is called height of node. The height of root node is considered as height of 3.

    E.g.: - Height of node B is 2

11. **Depth**

    Total number of edges from root node to a particular is called the depth of a node.

    The depth of the root node is zero. Eg:- Depth of J is 3

12. **Path**

    Path is the sequences of edges from one node to another node.

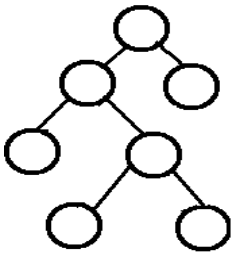
    Eg: - The path between nodes A to D is sequences of edges AB and BD respectively.

BINARY TREE

It is a tree data structure where each node has almost two children called left child and right child.

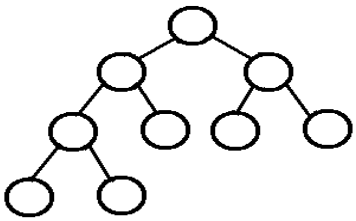
There are different types of binary tree.

1. **Full Binary Tree**
   In this type of binary tree every node has exactly two children or not.
   E.g.: -



2. **Complete Binary Tree**
   A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called as complete binary tree.

   E.g:-



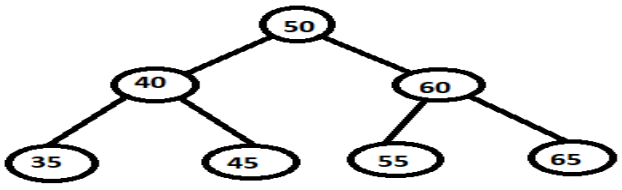
Properties of Binary Tree

1. Maximum number of nodes at level **L** of a binary tree is $\mathbf{2^{L-1}}$, **L > 0**.
2. Maximum number of nodes in a binary tree with depth **K** is $\mathbf{2^{K-1}}$, **K > 0**.

BINARY SEARCH TREE (BST)

It is a Binary Tree with following properties:

1. Left sub tree of a node have key values (Data part) less than its parent node.
2. Right sub tree of a node haver key values greater than its parent node.
   E.g.: -



RECURSSION

The process in which the function calls itself directly or indirectly is called as recursion and corresponding function is called as recursive function. Using recursive function certain problems such as tree traversal can be solved easily. In a recursive function,

the solution to base case is provided and solution of bigger problem is expressed in terms of smaller problems.

```
int fact (int n)
{
if(n<=1)
return 1;
else
return n*fact(n-1);
}
```

In this example the base case for n<=1 is defined and larger value of number can be solved by converting it to smaller one till base case is reached.

TREE TRAVERSAL

The process of visiting nodes of a tree is called as tree traversal. There are three different methods of tree traversal.

i) In-order Traversal

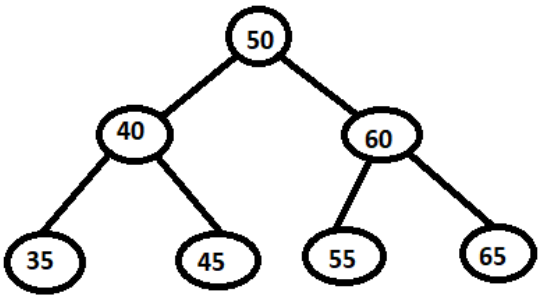
In this type of tree traversal left child of a node is first visited then root node is visited and finally right node is visited. The algorithm follows recursive method represented as:

Step 1: - Recursively traverse left sub tree.

Step 2: - Visit root node.

Step 3: - Recursively traverse right sub tree.

E.g.**:** -



The output sequence will be 35 40 45 50 55 60 65

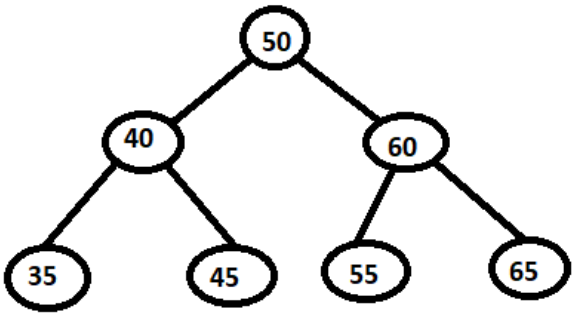
ii) Pre-order Traversal

In this type of tree traversal root node is first visited then left child is visited and finally right child of node is visited.

Step 1: - Visit root node.

Step 2: - Recursively traverse left sub tree.

Step 3: - Recursively traverse right sub tree.



E.g.: -

The output sequence will be 50 40 35 45 60 55 65

iii) Post-order Traversal

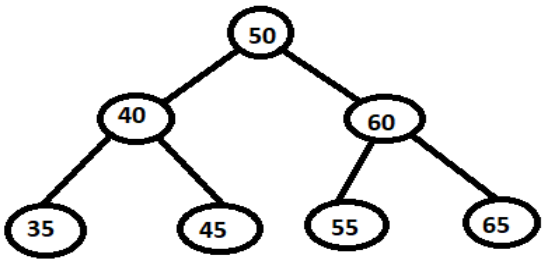
In this type of tree traversal left child of the node is first visited then right child of the node is visited and finally root node is visited.

Step 1: - Recursively traverse left sub tree.

Step 2: - Recursively traverse right sub tree.

Step 3: - Visit root node.

E.g.: -



The output sequence will be 35 45 40 55 65 60 50

Implement a binary search tree and write down the functions for tree traversal

```
#include<iostream>
using namespace std;
class node
{
public:
int data;
node *left,*right;
};
class bst:public node
{
public:
node *root;
bst()
{
root=NULL;
}
```

```
void create();
void inorder(node*);
void preorder(node*);
void postorder(node*);
};

void bst::create()
{
node *temp;
temp=new node;
cout<<"Enter the Data"<<endl;
cin>>temp->data;
temp->left=NULL;
temp->right=NULL;
if(root==NULL)
root=temp;
else
{
node *p;
p=root;
while(1)
{
if(temp->data<p->data)
{
if(p->left==NULL)
{
p->left=temp;
break;
}
else if(p->left!=NULL)
p=p->left;
}
else if(temp->data>p->data)
{
if(p->right==NULL)
{
p->right=temp;
break;
}
else if(p->right!=NULL)
p=p->right;
}}}}


void bst::inorder(node *m)
{
if(m!=NULL)
{
inorder(m->left);
cout<<m->data<<" ";
inorder(m->right);
}
}
void bst::preorder(node *m)
{
if(m!=NULL)
{
```

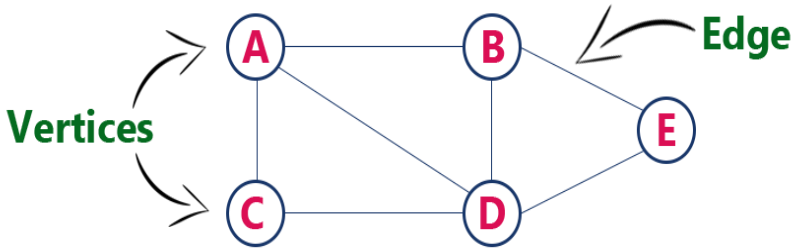
```
cout<<m->data<<" ";
inorder(m->left);
inorder(m->right);
}}
void bst::postorder(node *m)
{
if(m!=NULL)
{
inorder(m->left);
inorder(m->right);
cout<<m->data<<" ";
}}

int main()
{
bst B;
int y=1,ch;
while(y==1)
{
cout<<"Enter the Choice"<<endl;
cout<<"Enter 1.Create 2.Inorder 3.Preorder 4.Postorder
5.Exit"<<endl;
cin>>ch;
switch(ch)
{
case 1: B.create();
     break;
case 2:    B.inorder(B.root);
     break;
case 3:    B.preorder(B.root);
     break;
case 4:    B.postorder(B.root);
     break;
case 5:    y=0;
     break;
default: cout<<"Invalid Option"<<endl;
}}
return 0;
}
```

UNIT IV

GRAPH

Graph is a pictorial representation of set of objects where some pairs of objects are connected by links. The interconnected objects are represented by point called vertices. And the links that connect the vertices are called edges. Thus the graph is pair of sets (V, E) where V is set of vertices the graph and E is set of edges.

E.g.:



In the above graph a can be represents as (V, E) where

V = { A, B, C, D, E }

E = { $AB, BE, AD, AC, CD, BD, DE$ }

OPERATIONS ON GRAPH

1. Add Vertex: Addition of a new vertex to the graph
2. Add Edge: Addition of a new edge between 2 vertices
3. Display Vertex: Display vertex of a graph

GRAPH TERMINOLOGY

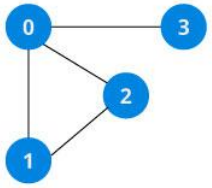
1. **Vertex:** Each node of a graph is termed as vertex.
2. **Edge:** Edge represents the link between two vertices.
3. **Adjacency:** Two vertices are adjacent if they are connected to each other through an edge.
4. **Path:** Path represents a sequence of edges between two vertices.

GRAPH REPRESENTATION

1. **Adjacent Matrix**

   The adjacency matrix in a graph is a matrix with rows and columns, represented by graph vertices with 1 or 0 in the position V_{ij} according to whether V_i or V_j has a link or not. In such a matrix when i^{th} row and j^{th} column element is 1, then there will be an

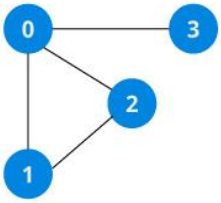
edge between i and j vertex. When there is no edge, the value will be zero. The above graph can be represented is as follows:

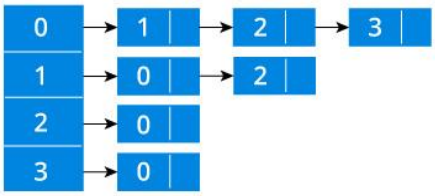


	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

2. Adjacent List

The adjacent list of a graph is linked with one list per vertex. It will have a head list which contains all the vertices in a sequential order and each element in its linked list represents the other vertices that form an edge with the vertex. The adjacency list representation of above graph is as follows:



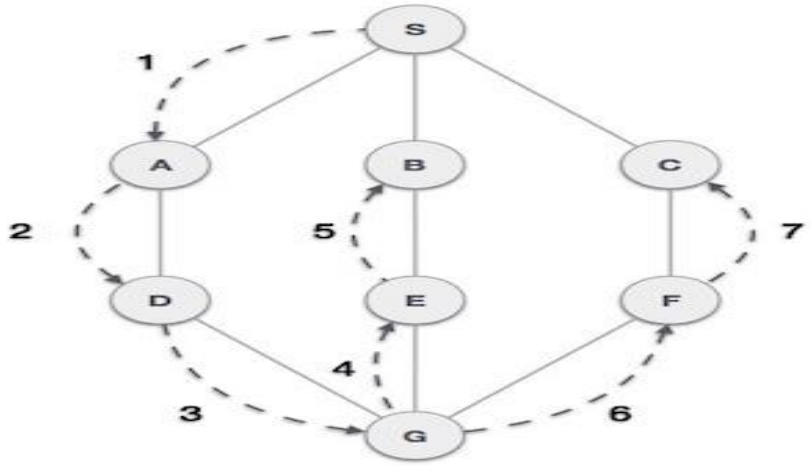


GRAPH TRAVERSAL METHODS

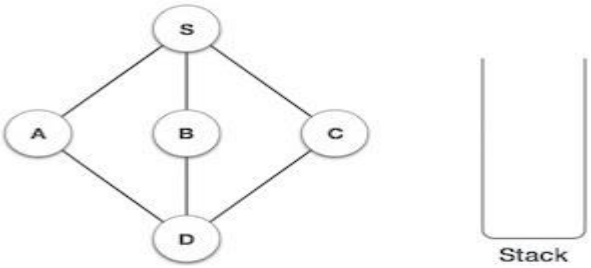
1. Depth First Search (DFS)

This algorithm traverses a graph in a depth ward direction and uses a stack to remember the next vertex to be visited when a dead end occurs i.e. for back tracking.

Step 1: Define a stack of size with total number of vertices in the graph.
Step 2: Select any vertex as starting point for traversal visit that vertex and push it on to the stack.
Step 3: Visit any one of the adjacent vertex of the vertex which at the top of stack and is not visited and push it on to the stack.
Step 4: Repeat step 3 until there are no vertex to visit from vertices on the stack.
Step 5: When there is no vertex to visit then use back tracking and pop one vertex from the stack.
Step 6: Repeat steps 3, 4 and 5 until stack becomes empty.
Step 7: When stack become empty it will give the sequence of vertices visited.
E.g.: - Consider the graph below

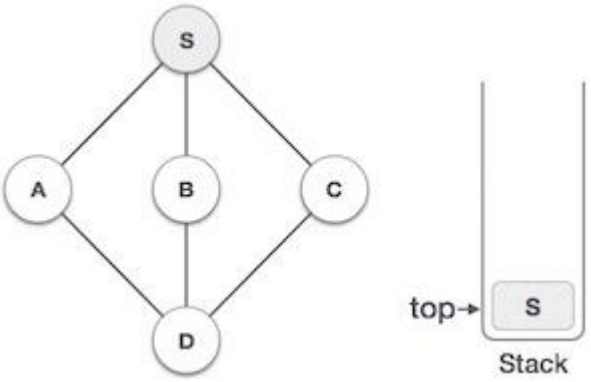


Step 1:



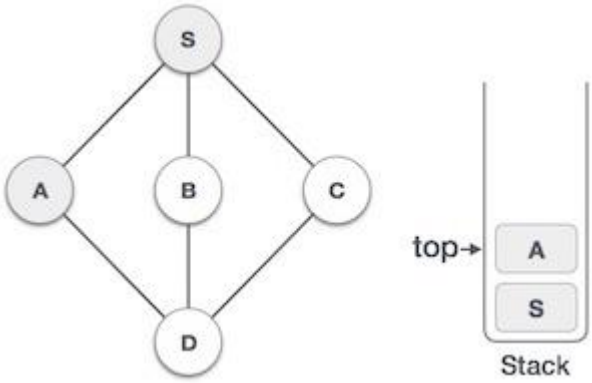
Initialize the stack

Step 2:



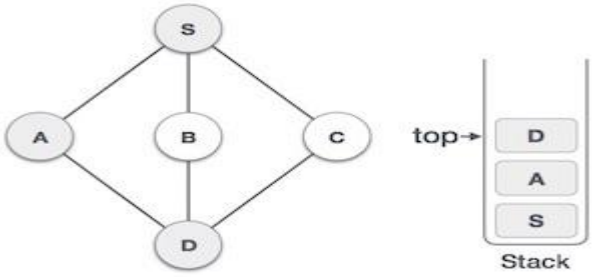
Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

Step 3:



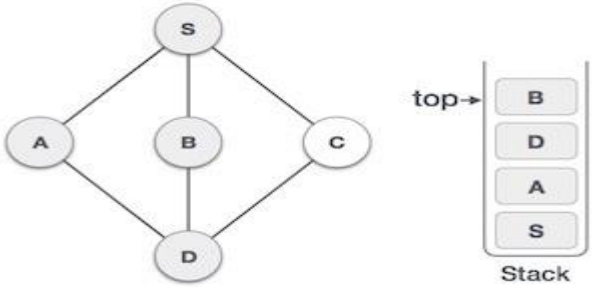
Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

Step 4:



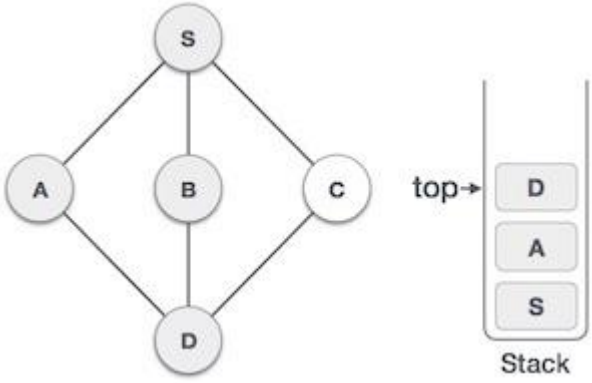
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

Step 5:



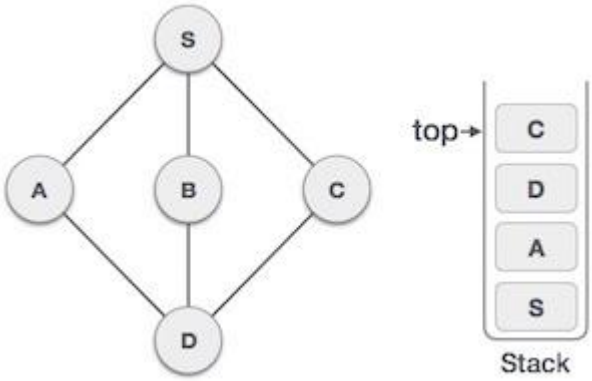
We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

Step 6:



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

Step 7:



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

Step 8:

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Q. Implement depth first search algorithm

```
#include<iostream>
using namespace std;
class graph
{
int n,c,s;
int ag[10][10],a[10];
public:
void matrix();
void dfs();
};
void graph::matrix()
{
int i,j;
cout<<"Enter the no. of Nodes"<<endl;
      cin>>n;
      for(i=0;i<n;i++)
      {
            cout<<"Enter the Node"<<endl;
            cin>>a[i];
      }

      for(i=0,j=1;j<n,i<n;j++,i++)
      {
            ag[0][j]=a[i];
      }
      for(i=1,j=0;i<n,j<n;i++,j++)
      {
```

```
            ag[i][0]=a[j];
      }
                        cout<<"Enter the adjacency matrix"<<endl;
            for(i=1;i<=n;i++)
            {
                  for(j=1;j<=n;j++)
                  {
                        cout<<"Enter 1 if edge is present otherwise 0
between  "<<ag[i][0]<<" "<<ag[0][j]<<endl;
                        cin>>ag[i][j];
                  }}


            cout<<"ADJACENCY MATRIX  OF UNDIRECTED GRAPHis---"<<endl;
            for(i=0;i<=n;i++)
            {
                  for(j=0;j<=n;j++)
                  {     if(i==0&&j==0)
                        {
                              cout<<" ";
                        }
                        else
                        {
                              cout<<" "<<ag[i][j];
                        }
                  }
            cout<<endl;
            }
}
void graph::dfs()
{
int i,j,t,x,y,m,k,top=-1;
      int stack[10],visited[10];
      cout<<"Enter the node from which node from which traversal
start"<<endl;
      cin>>x;
      stack[++top]=x;
      for(i=0;i<n;i++)
      {
            if(a[i]==x)
            {
                  k=i;
                  break;
            }
      }
      visited[k]=1;
      cout<<x<<" ";
      do
      {
            for(i=1;i<=n;i++)
            {
                  if(ag[i][0]==stack[top])
                        {
                         y=0;
                         for(j=1;j<=n;j++)
                         {
                          if(ag[i][j]==1)
```

```
                        {
                         for(t=0;t<n;t++)
                         {
                         if(ag[0][j]==a[t])
                         break;
                         }
                            if(visited[t]!=1)
                            {
                            stack[++top]=a[t];
                            cout<<a[t]<<" ";
                            visited[t]=1;
                            y++;
                            break;
                            }
                       }

                              }}}
                  if(y==0)
                  {
                  top=--top;
                  }
                      m=0;
                      for(i=0;i<n;i++)
                      {
                       if(visited[i]!=1)
                       {
                        ++m;
                        }
                        }
      }while(m!=0);

}


int main()
{
graph g;
g.matrix();
g.dfs();
return 0;
}
```

2. Breadth First Search (BFS)

This algorithm traverses a graph in breadth wise direction and is utilising a queue for processing vertices.

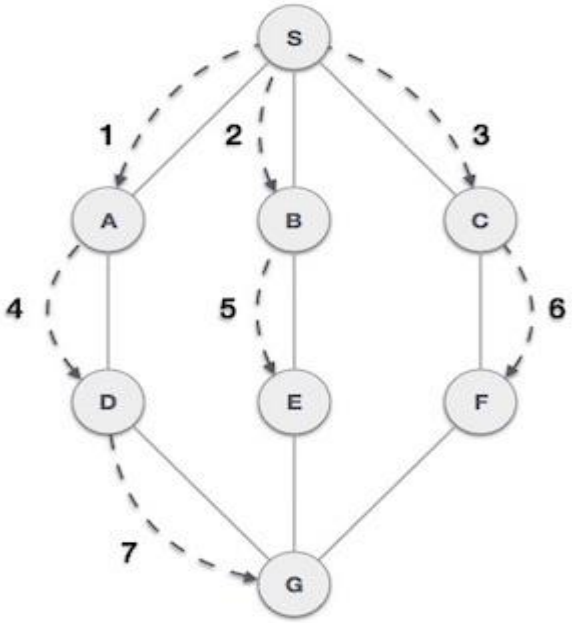
Step 1: Define a sequence of size with total number of vertices in a graph.

Step 2: Select any vertex as starting point for the traversal and visit that vertex. Insert the newly selected vertex into the queue.

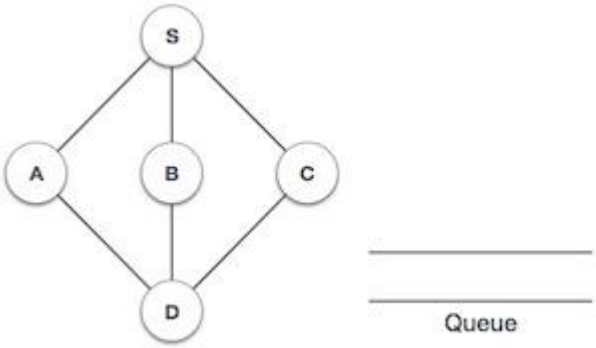
Step 3: Visit all adjacent vertex that vertex which is at the front of the queue and is not visited, insert that vertex on to queue.

Step 4: Repeat step 3 until there is no new vertex to be visited from queue.

Step 5: When queue is empty an algorithm produces BFS sequence.

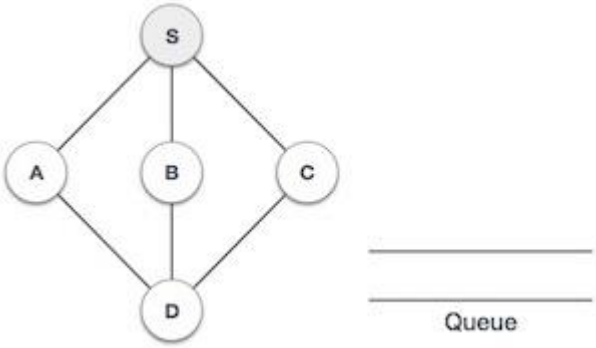


Step 1:



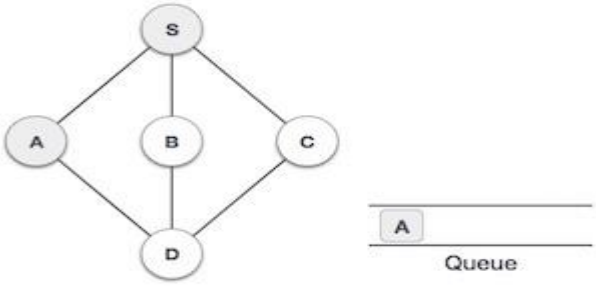
Initialize Queue

Step 2:



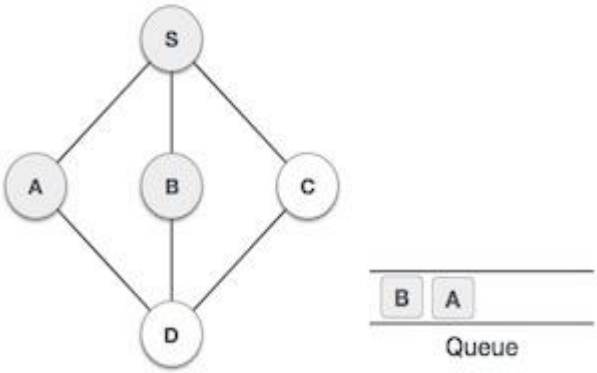
We start from visiting **S** (starting node), and mark it as visited.

Step 3:



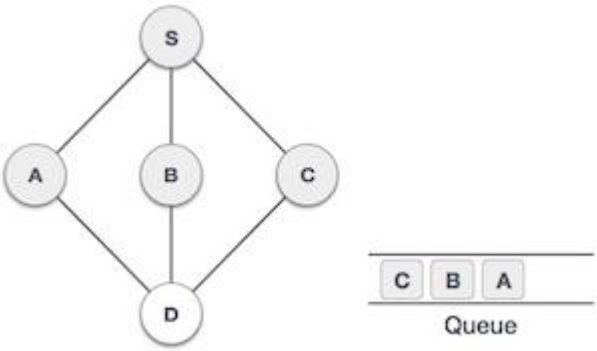
We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

Step 4:



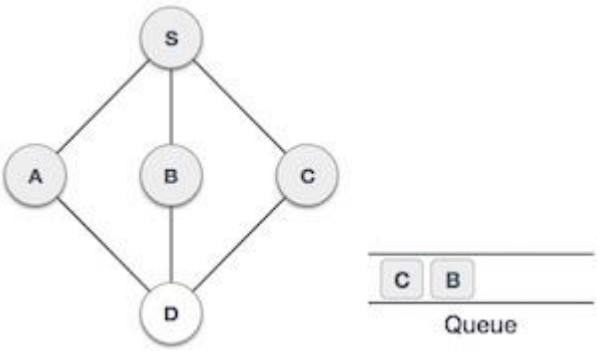
Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

Step 5:



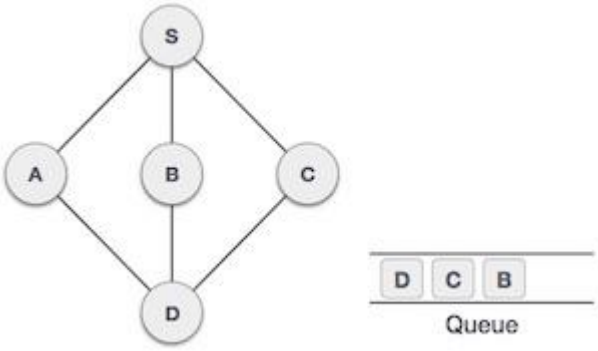
Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

Step 6:



Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

Step 7:



From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

Step 8:

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Q. Implement breadth first search algorithm

```
#include<iostream>
using namespace std;
class graph
{
int n,c,s;
int ag[10][10],a[10];
public:
void matrix();
void bfs();
};
void graph::matrix()
{
int i,j;
cout<<"Enter the no. of Nodes"<<endl;
      cin>>n;
      for(i=0;i<n;i++)
      {
            cout<<"Enter the Node"<<endl;
            cin>>a[i];
      }

      for(i=0,j=1;j<n,i<n;j++,i++)
      {
            ag[0][j]=a[i];
      }
      for(i=1,j=0;i<n,j<n;i++,j++)
      {
            ag[i][0]=a[j];
      }

            cout<<"Enter the adjacency matrix"<<endl;
```

```
            for(i=1;i<=n;i++)
            {
                  for(j=1;j<=n;j++)
                  {
                        cout<<"Enter 1 if edge is present otherwise 0
between  "<<ag[i][0]<<" "<<ag[0][j]<<endl;
                        cin>>ag[i][j];
                  }
            }


            cout<<"ADJACENCY MATRIX  OF UNDIRECTED GRAPHis----
"<<endl;
            for(i=0;i<=n;i++)
            {
                  for(j=0;j<=n;j++)
                  {     if(i==0&&j==0)
                        {
                              cout<<" ";
                        }
                        else
                        {
                              cout<<" "<<ag[i][j];
                        }
                  }
            cout<<endl;
            }
}

void graph::bfs()
{
 int i,j,t,y,m,x,k,front=-1,rear=-1;

 int queue[10],visited[10];
 cout<<"Enter the node from which nodes are printed"<<endl;
 cin>>x;
queue[++front]=x;
rear=front;

for(i=0;i<n;i++)
 {
  if(a[i]==x)
  {
      k=i;
      break;
      }
  }
  visited[k]=1;
  cout<<x<<" ";
  do{

        for(i=1;i<=n;i++)
        {
            if(ag[i][0]==queue[front])
            {
             y=0;
             for(j=1;j<=n;j++)
```

```
            {
             if(ag[i][j]==1)
             {
                 for(t=0;t<n;t++)
                 {
                  if(ag[0][j]==a[t])
                  {
                   break;
                   }
                  }
                  if(visited[t]!=1)
                  {
                   queue[++rear]=a[t];
                   cout<<a[t]<<" ";
                   visited[t]=1;
                   y++;

                   }}}}}
                   if(y==0)
                   {
                       front=front+1;
                       }
                       m=0;
                       for(i=0;i<n;i++)
                       {
                        if(visited[i]!=1)
                        {
                         ++m;
                         }
                         }
                        }while(m!=0);

}
int main()
{
graph g;
g.matrix();
g.bfs();
return 0;
}
```

Floyed Warshall algorithm

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Pseudocode for Floyed Warshal algorithm is

Create a|V|x|V| matrix //It represents the distance between every pair of vertices as given

For each cell (I,j) in M do

If i==j

M[i][j]=0 // For all diagonal elements , value=0

```
If(I,j) is an edge in E
        M[i][j]=weight(I,j)  //If there exists a direct edge between the values, value= weight of edge
else
        M[i][j]=infinity  //if there is no direct edge between the vertices, value=infinity
for k from 1 to |V|
  for i from 1 to |V|
    for j from 1 to |V|
        if M[i][j]>M[i][k]+M[k][j]
        M[i][j]=M[i][k]+M[k][j]
```

SEARCHING

It is a technique that helps to find a place of a given element in a list. Any search option said to successful if element to be search is found. Otherwise search option is unsuccessful. The standard technics for search options are:

1. **Linear Search**
   This is the simplest searching technique where search on a given array is done by traversing the array from beginning till the desired element is found. The time complexity of linear search is order of n where n is number of elements in array.
2. **Binary Search**
   This search technique is done on a list of sorted elements. Here, search operation starts by comparing the search element with middle element in the list. If search element is smaller than the middle element then repeat the same process for the left sub list of the middle element. If search element is larger then, repeat the same process for the right sub list of the search element. Repeat the process until the search element is found in the list or until left with a sub list of only one element. If that element does not match the search element then, the desired element is not found in the list. The time complexity of binary search is order of $\log n$.

Q. Implement a function for binary search

```
void BinarySearch()
{
                int count, i, arr[15], num, first, last, middle;
                cout<<"Enter the number of elements";
     cin>>count;
                for (i=0; i<count; i++)
```

```
                {
                        cout<<"Enter number "<<(i+1)<<": ";
        cin>>arr[i];
                }
                cout<<"Enter the number to search:";
    cin>>num;
                first = 0;
                last = count-1;
                middle = (first+last)/2;
                while (first <= last)
                {
                  if(arr[middle] < num)
                  {
                      first = middle + 1;
                  }
                  else if(arr[middle] == num)
                  {
                      cout<<num<<" found in the array at the location "<<middle+1<<"\n";
        break;
    }
    else {
        last = middle - 1;
    }
    middle = (first + last)/2;
}
if(first > last)
                {
                  cout<<num<<" not found in the array";
                } }
```

SORTING

Sorting refers to operation for arranging set of data in ascending or descending order according to relations among data items.

Categories of Sorting

1. **Internal Sorting**
   If the data item to be sorted can be adjusted at a time in primary memory then it is called an internal sorting.

2. **External Sorting**
   When data items that are to be sorted cannot be accommodated to the main memory at the same time and quotient of data elements have to be kept in secondary memory then such category of sorting is called as external sorting.

3. **Bubble sort**
   It compares the entire element one by one and sorts them based on their values. If an array of elements are sorted in ascending order by using bubble sort by comparing first element of an array to the second if first element is greater than second element to interchange two elements. The time complexity of bubble sort is in the order of n^2.

4. **Quick Sort**

QuickSort is a Divide and Conquer algorithm. The time complexity in worst case is $O(n^2)$ and average case is O(n log n). It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quick sort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Partition Algorithm:

It starts from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if it find a smaller element, then swap current element with arr[i]. Otherwise ignore current element.

```
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}


void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout<<arr[i]<<endl;
}

int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    cout<<"Sorted array: "<<endl;
    printArray(arr, n);
    return 0;
}
```

Eg:-

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes:0 1 2 3 4 5 6

low = 0, high =6, pivot = arr[h] = 70

Initialize index of smaller element, **i =-1**

Traverse elements from j = low to high-1

j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0

arr[] = {10, 80, 30, 90, 40, 50, 70}

// No change as i and j are same

j = 1 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70Swapped

Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after it.