UNIT 2

1.Efficient Non-Recursive Binary Tree Traversal Algorithms.
2.Set,Disjoint Set and Operations
3.Union and find Algorithms
4. Spanning trees
5.Graph Traversals
6.AND/OR Graph
7.GAME TREE(tic-tac-toi)
8.Connected Components
9.Bi-connected Components

In computer science, a search algorithm is an algorithm that retrieves information stored within some data structure. Search algorithms can be classified based on their mechanism of searching. Linear search algorithms check every record for the one associated with a target key in a linear fashion. Binary, or half interval searches, repeatedly target the center of the search structure and divide the search space in half. Comparison search algorithms improve on linear searching by successively eliminating records based on comparisons of the keys until the target record is found, and can be applied on data structures with a defined order. Digital search algorithms work based on the properties of digits in data structures that use numerical keys. Finally, hashing directly maps keys to records based on a hash function. Searches outside of a linear search require that the data be sorted in some way.

Search functions are also evaluated on the basis of their complexity, or maximum theoretical run time. Binary search functions, for example, have a maximum complexity of O(log(n)), or logarithmic time. This means that the maximum number of operations needed to find the search target is a logarithmic function of the size of the search space.

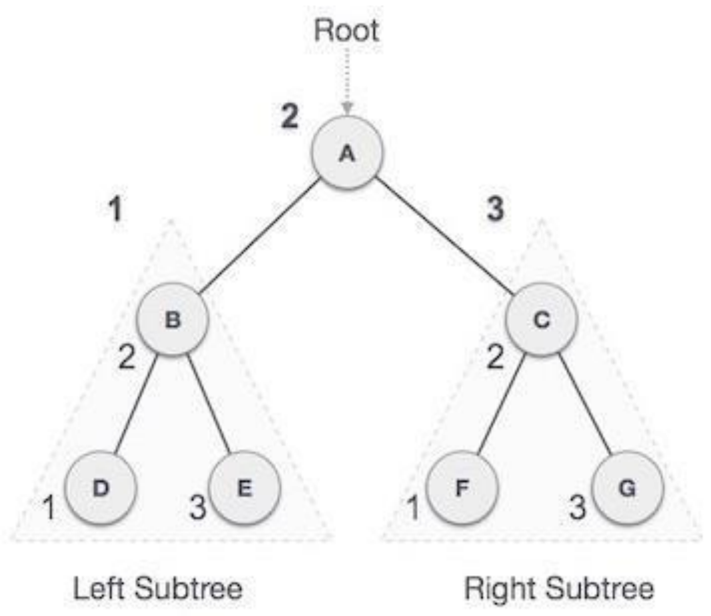
In computer science, graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal.

Let us learn in detail about these very important topics in the following pages…

Binary tree :

Definition: A **binary tree** is a tree data structure in which each node has atmost two children,which are referred to as the left child and the right child. (Or) Binary tree is a finite set of nodes which is either empty or consists of root and two disjoint binary trees called as left sub tree and right sub tree

Example:



Binary tree Traversals:

Traversing the tree means visiting each node exactly once there are two **binary tree traversals algorithms** they are:

1.) Recursive **binary tree traversal algorithms** 2.) **Non-Recursive binary tree traversal algorithms**

1.) Recursive binary tree traversal algorithms:

There are 3traversal techniques used for traversing binary tree they are:

1.) In-order traversal 2.) Pre-order traversal 3.) Post-order traversal

While traversing the binary tree we have to treat each node and its sub trees in the same manner. we will use some notations for tree traversals .L- stands for move left ,D- stands for print the data at current node and R- Stands for move right. Here there are six possible combinations of L, R, D such as LDR, LRD, DLR, DRL, RLD, and RDL. But from computing point of view we will have three different ways of traversing a tree. Those three combinations are LDR, DLR and LRD are called as In-order traversal, Pre-order traversal and Post-order traversal.

1. In-order Traversal :

In this traversal method, the left sub tree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a sub tree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

Left node	Root node	Right node

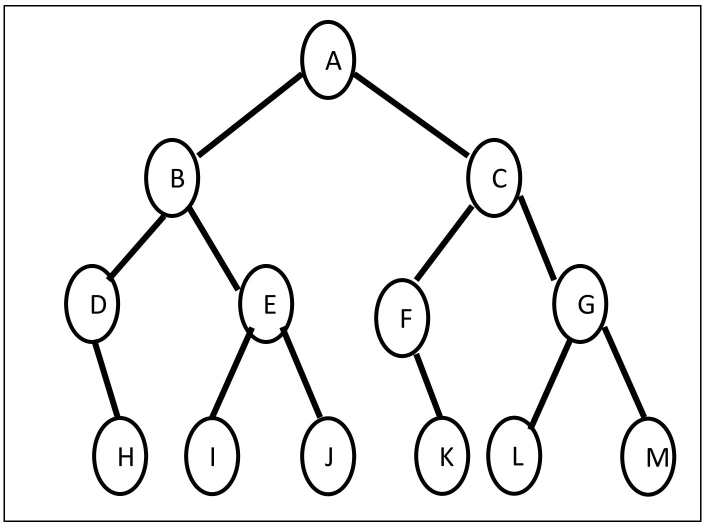
In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

Algorithm:

```
treenode = record
{
        Type   data;                    //Type is the data type of data. Treenode *lchild; treenode *rchild;
}
algorithm inorder (t)
// t is a binary tree. Each node of t has three fields: lchild, data, andrchild.
{
        if t □ 0 then
        {
                inorder (t □ lchild); visit (t);
                inorder (t □ rchild);
        }
}
```

Example:



LEFTNODE	ROOT NODE	RIGHT NODE
L_N	A	R_N
B	A	R_N
L_N B R_N	A	R_N
L_N D R_N B R_N	A	R_N
DHBL_NER_N	A	R_N
DHBIEJ	A	L_NCR_N
DHBIEJ	A	L_NFR_NCR_N

DHBIEJ	A	FKCL$_N$GR$_N$
DHBIEJ	A	FKCLGM
DHBIEJAFKCLGM		

2 Preorder Traversal:

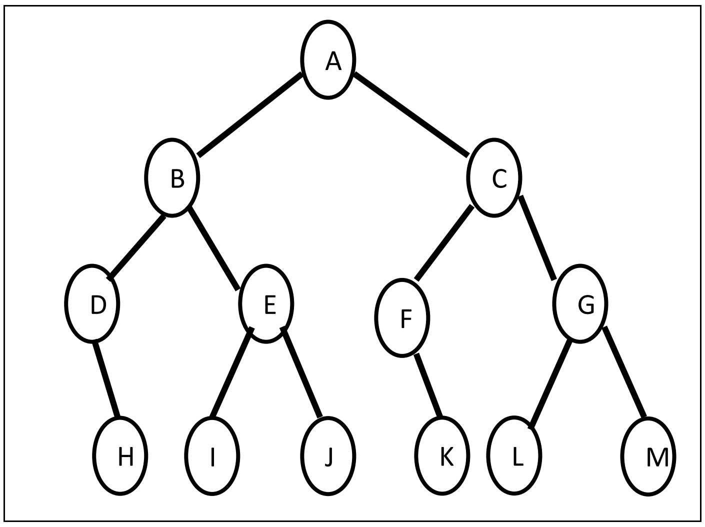
In a preorder traversal, each node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

Root node	Left node	Right node

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

Algorithm:

```
Algorithm Preorder (t)
// t is a binary tree. Each node of t has three fields; lchild, data, andrchild.
{
        if t □ 0 then
        {
                visit (t);
                Preorder (t □ lchild); Preorder (t □ rchild);
        }
}
```



ROOT NODE	LEFTNODE	RIGHT NODE
A	L_N	R_N
A	B	R_N
A	BL_NR_N	R_N
A	$B(DL_NR_N)R_N$	R_N
A	$BDH(EL_NR_N)$	R_N
A	BDHEIJ	CL_NR_N
A	BDHEIJ	$C(FL_NR_N)R_N$
A	BDHEIJ	$CFK(GL_NR_N)$
A	BDHEIJ	CFKGLM
A BDHEIJCFKGLM		

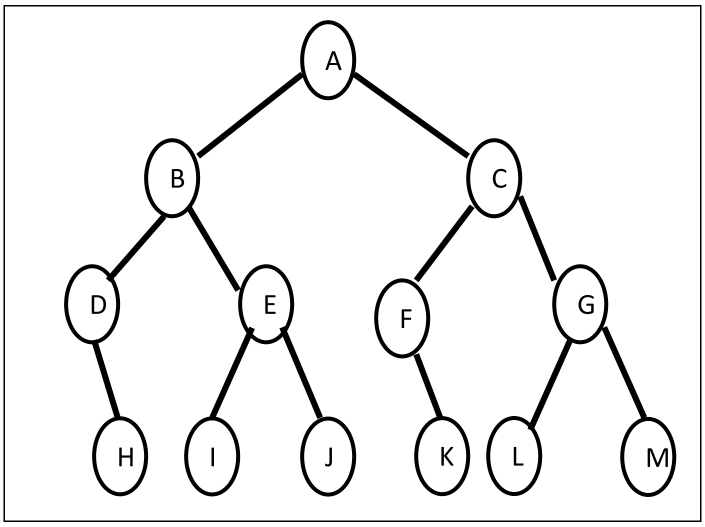
3 Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

Root node	Left node	Right node

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

```
Algorithm Postorder (t)
// t is a binary tree. Each node of t has three fields : lchild, data, andrchild.
{
        if t □ 0 then
        {
                Postorder (t □ lchild); Postorder (t □ rchild); visit(t);
        }
}
```



LEFTNODE	RIGHT NODE	ROOT NODE
L_N	R_N	A
B	R_N	A
L_NR_NB	R_N	A
$L_NR_NDR_NB$	R_N	A
HDL_NR_NEB	C	A
HDIJEB	L_NR_NC	A
HDIJEB	$L_NR_NFR_NC$	A
HDIJEB	KFL_NR_NGC	A
HDIJEB	KFLMGC	A
HDIJEB KFLMGCA		

2.)Non-Recursive binary tree traversal algorithms:

At first glance, it appears we would always want to use the flat traversal functions since the use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

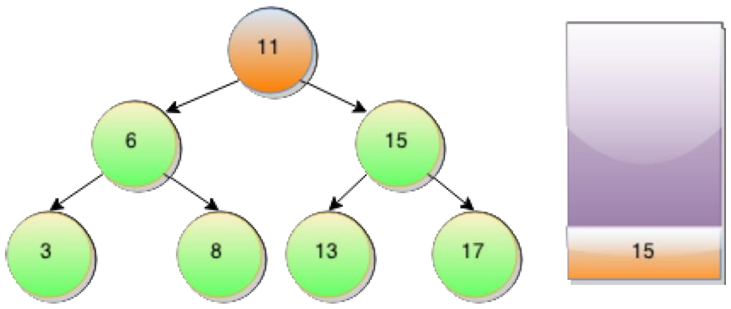
Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

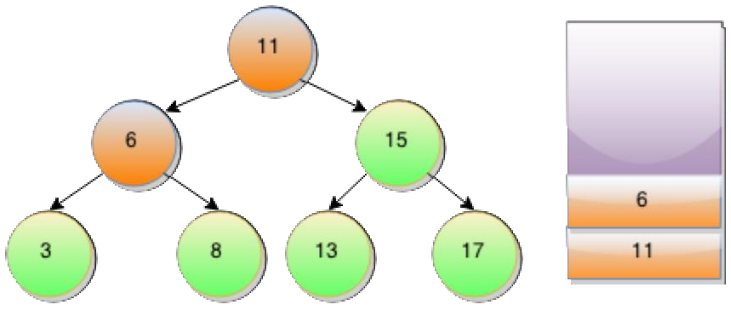
1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

he algorithm for inorder Non Recursive traversal is as follows:

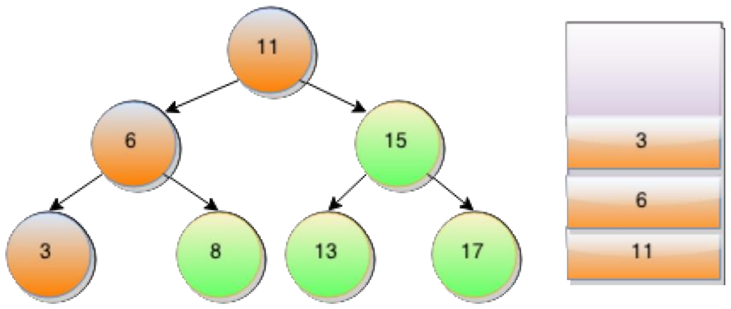
```
Algorithm inorder()
{
        stack[1] = 0 vertex = root
top:    while(vertex $\neq$ 0)
        {
                push the vertex into the stack vertex = leftson(vertex)
        }

        pop the element from the stack and make it as vertex

        while(vertex $\neq$ 0)
        {
                print the vertex node if(rightson(vertex) $\neq$ 0)
                {
                        vertex = rightson(vertex) goto top
                }
                pop the element from the stack and made it as vertex
        }
}
```



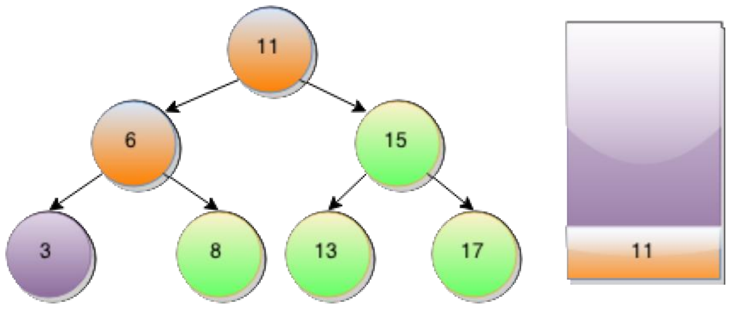
Push node onto stack and move towards le



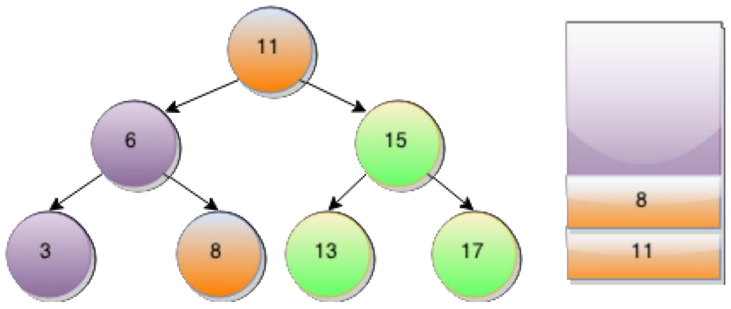
Push node onto stack and move towards le



Now, when going down towards left tree, NULL is encountered and hence pop from the stack



Since there is no right child of 3, we will again pop from stack



There is right child of 6 hence push that to stack

2 Preorder Traversal:

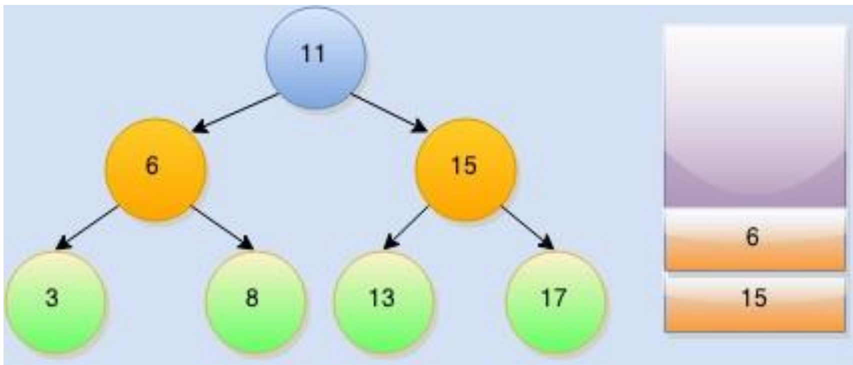
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.

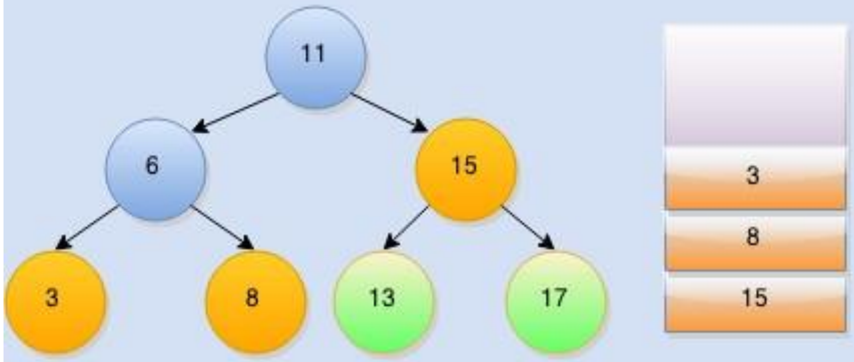
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

The algorithm for preorder Non Recursive traversal is as follows:

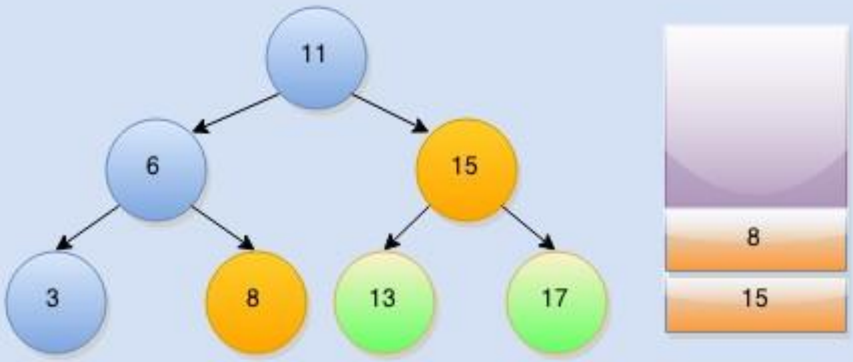
```
Algorithm preorder( )
{
    stack[1]: = 0 vertex := root. while(vertex ≠ 0)
    {
        print vertex node if(rightson(vertex) ≠ 0)
            push the right son of vertex into the stack. if(leftson(vertex) ≠ 0)
            vertex := leftson(vertex)  else

  } pop the element from the stack and made it as vertex


}
```



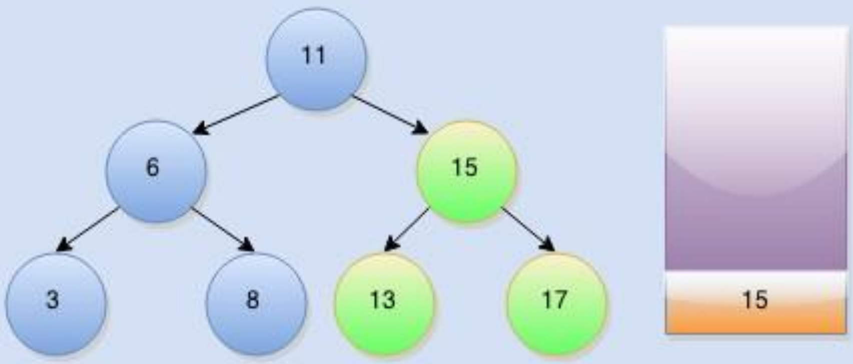
Visit 11 and put right and left node on stack



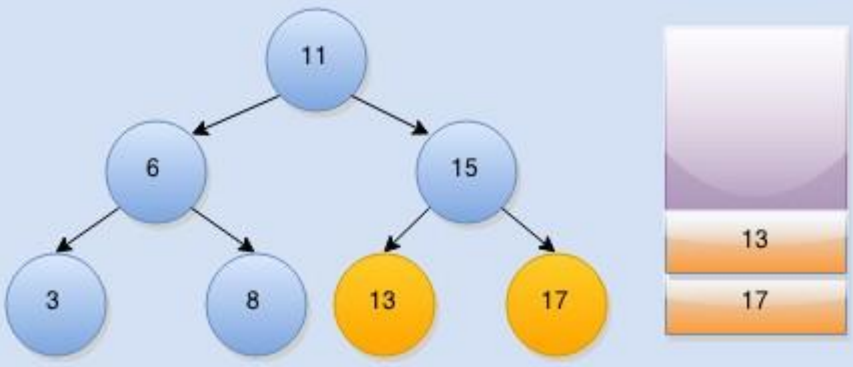
Pop 6, visit it and put right and left node on stack



Pop 3, visit it. Since there is no left or right child, no need to push anything



Pop 8, visit it. Since there is no left or right child, no need to push anything



Pop 15, visit it. Push right and left node onto stack

3 Post order Traversal:

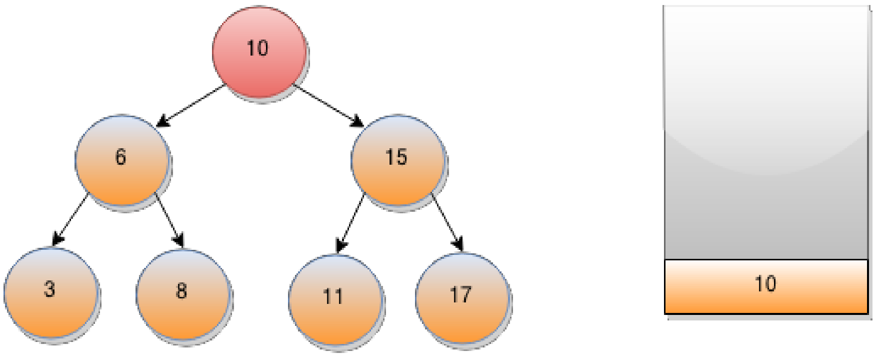
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

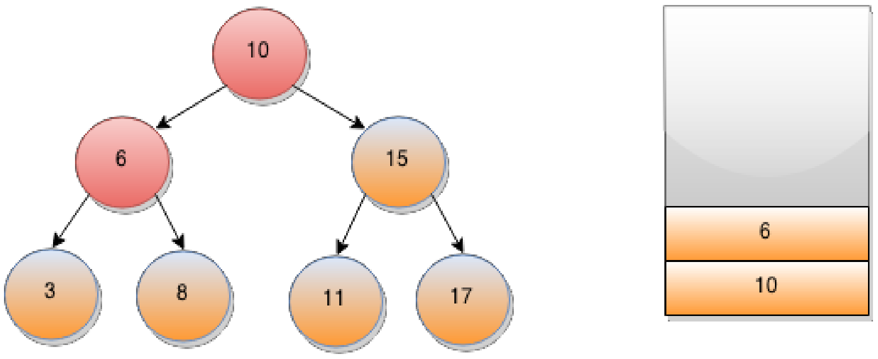
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Algorithm :

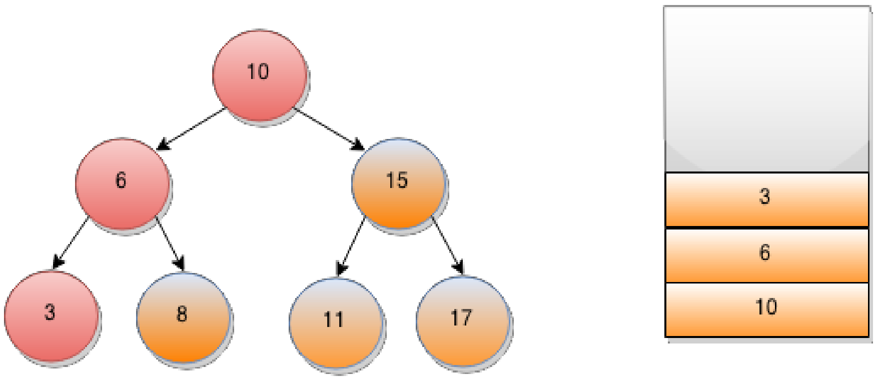
```
Algorithm postorder( )
{
        stack[1] := 0 vertex := root

 top: while(vertex ≠ 0)
        {
                push vertex onto stack if(rightson(vertex) ≠ 0)
                        push -(vertex) onto stack vertex := leftson(vertex)
        }
        pop from stack and make it as vertex while(vertex > 0)
        {
                print the vertex node
                pop from stack and make it as vertex
        }
        if(vertex < 0)
        {
                vertex := -(vertex) goto top
                }
                }
```



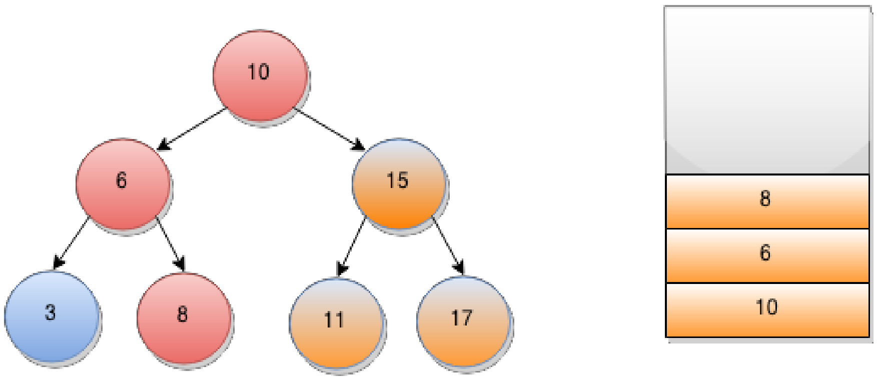
Push node onto stack



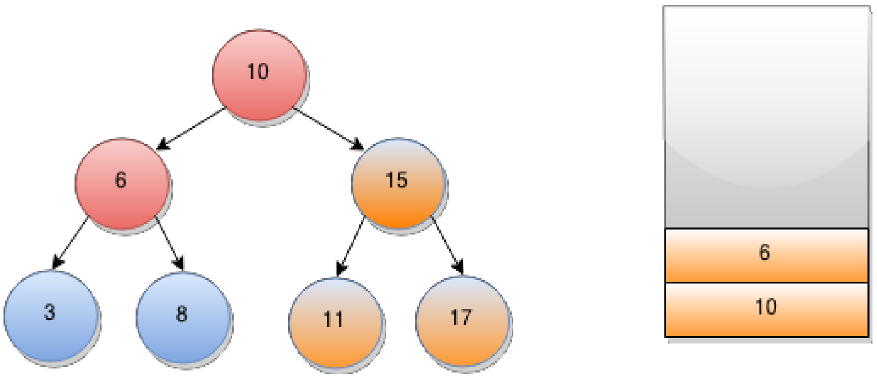
Push left node on to stack if it is there, 6 is pushed



Push left node on to stack if it is there, 3 is pushed



Since 3 is leaf node, it is marked as visited, and move up; at 6 we are coming up from left child, we will put right child of node on to stack



Now, node 8 is again leaf node, it is marked as visited and move up from there.

2 Disjoint Set Operations:

Set:

A set is a collection of distinct elements. The Set can be represented, for examples, S1={1,2,5,10}.

Operations on sets:

There are different types of operations available for sets.

1. **Member(a,S):**
   There is a possibility that a∈S (or) a∉S.
   If a∈S print 'Yes' otherwise print 'No'.
   Example: S={1,2,3,4}
   Member(2,s) ⇒ means 2∈S , print Yes.
   Member(2,s) ⇒ means 6∉S, print No.

2. **Insert(a,S):**
   It means to add the element to the set.
   Here 'a' is element , S is set.
   Example: S={1,2,3}
   Insert 4 ⇒ means S={1,2,3}U{4}
   ={1,2,3,4}
3. **Delete(a,S):**
   It means to remove the element from set.
   Example: S={1,2,3,4}
   Delete(4,S) ⇒ {1,2,3,4}-{4}
   ={1,2,3}
4. **Union(S1,S2,S3):**
   The union of two **sets** is the **set** containing all elements belonging to either one of the **sets** or to both, denoted by the symbol ∪.
   If we conclude that S3=S1US2, we assume that S1 and 2 are disjoint sets.
   Example: S1={1,2}
   S2={3,4}
   S3=S1US2={1,2}U{3,4}={1,2,3,4}

5. **Find(a):**
   Print the name of the set in which 'a' is currently member.
   Example: S1={1,2}
   S2={3,4}
   Find(4)⇒ return S2 because 4∈S2 ie. Name of the set.

6. **Split(a,S):**
   This operation can partition the set 'S' into 2 sets ie. S1 and S2.
   Example: S={1,2,3,4,5,6}
   Split(3,S) then S1={1,2,3}
   S2={4,5,6}

7. **Min(S):**
   It prints the smallest element of the set ‘S’.
   Example: S={2,4,6}
   Min(S)=2.

8. **Max(S):**
   It prints the largest element of the set ‘S’.
   Example: S={2,4,6,8}
   Max(S)=8.

Disjoint Sets:

The disjoints sets are those do not have any common element. For example S1= {1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are

1. Union
2. Find

1 Disjoint set Union:

If Si and Sj are tow disjoint sets, then their union Si U Sj consists of all the elements x such that x is in Si or Sj.

Example:

S1={1,7,8,9} S2={2,5,10}
S1 U S2={1,2,5,7,8,9,10}

Find: Given the element I, find the set containing i.

Example:

S1={1,7,8,9} S2={2,5,10} s3={3,4,6}

Then,

Find(4)= S3 Find(5)=S2 Find97)=S1

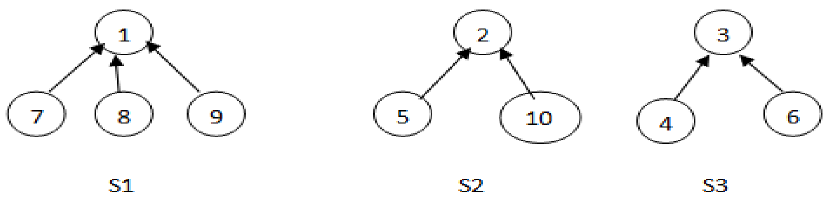
Set Representation:

The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

Example:

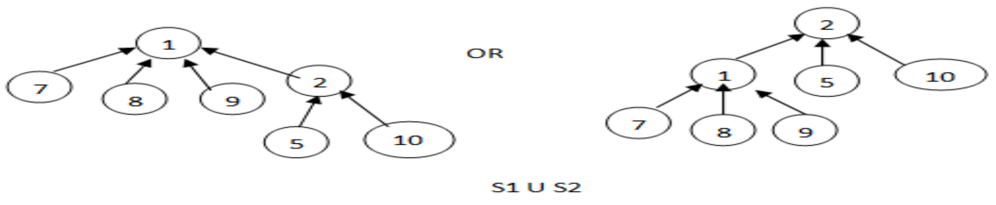
S1={1,7,8,9} S2={2,5,10} s3={3,4,6}

Then these sets can be represented as



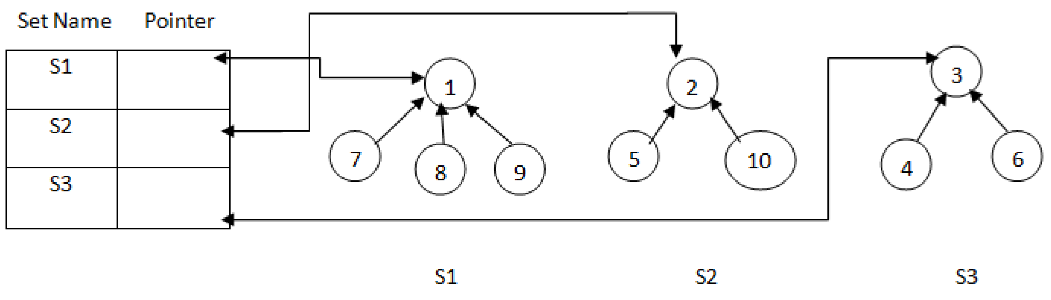
Disjoint Union:

To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other. Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following.



Find:

To perform find operation, along with the tree structure we need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.

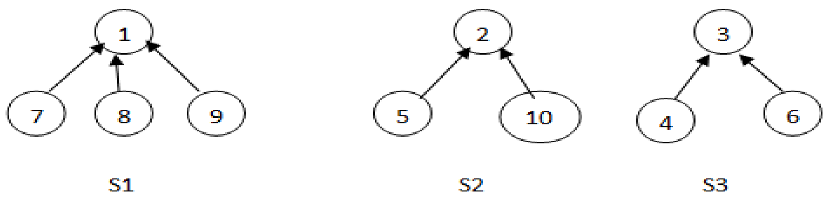


3 Union and Find Algorithms:

In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be '-1'.

Example:

For the following sets the array representation is as shown below.



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	-1	-1	3	2	3	1	1	1	2

1 Union Algorithm:

To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

```
Algorithm SimpleUnion(i,j)
      {
            P[i]:=j;
}
```

For example, let us consider an array. Initially parent array contains zero's.

0	0	0	0	0	0	0	
Child⟵ 1	2	3	4	5	6	7	↖parent

1) Union (1,3) → ①⟵③

0	0	1	0	0	0	0
1	2	3	4	5	6	7

2) Union (2,5) → ①⟵③

②⟵⑤

0	0	1	0	2	0	0
1	2	3	4	5	6	7

3) Union (1,2) →
$$\begin{tikzcd} ① & ③ \arrow[l] \\ ② \arrow[u] & ⑤ \arrow[l] \end{tikzcd}$$

0	0	1	0	2	0	0
1	2	3	4	5	6	7

Let us process the following sequence of union-find operations: →

Union (1,2); Union (2,3); Union (3,4);............................... ;Union (n-1,n);

i.e., Find(1), Find(2), Find(3),................ Find(n).

This sequence results in the degenerate tree of diagram

$$\begin{tikzcd} ① \arrow[r] & ② \arrow[r] & ③ \arrow[r] & ④ \arrow[r] & \cdots\cdots \end{tikzcd}$$

Since the time taken for a union is constant, the n-1 union s can be processed in time O(n).

. ∴ Time complexity of union algorithm is O(n).

2 Find Algorithm:

The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at I until it reaches a node with parent value -1.

Find (i) implies that if finds the root node of i^{th} node, in other words it returns the name of the set.

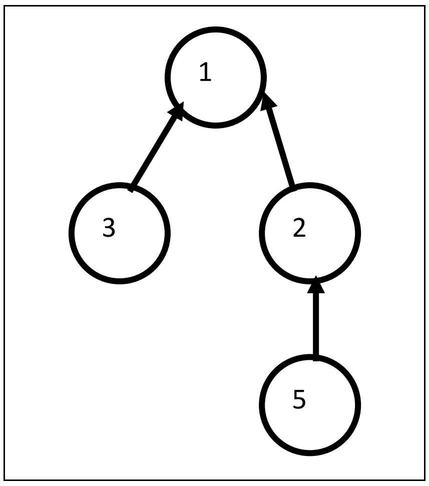
Eg:- union (1,3) →
$$\begin{tikzcd} ① \\ ③ \arrow[u] \end{tikzcd}$$

Find(3)=1 since its parent is 1 i.e., root node.

Algorithm:-

```
Algorithm find(i)
{
integer I,j;
while(parent (j)>0 )
do j←parent(j)
repeat
return j;
}
```

EXAMPLE:



0	1	1	0	2	0
1	2	3	4	5	6

Find (5) j=5

While P(j)>0 that is, P(5)>0

⇨ 2>0 (true)
There fore j=2

While P(2) => 1>0 (true)

There fore j=1

While P(1) => 0>0 (false)

return j;

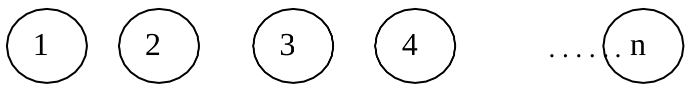
that is 1.

Therefore 1 is root node of node 5

The time complexity of find algorithm in nXn i.e O(n^2)

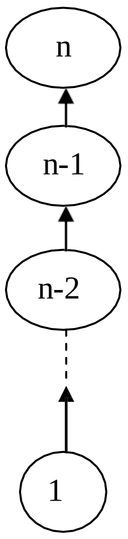
3 Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets



Then if we want to perform following sequence of operations Union(1,2) , Union(2,3)……. Union(n-1,n) and sequence of Find(1), Find(2)……… Find(n).

The sequence of Union operations results the degenerate tree as below.



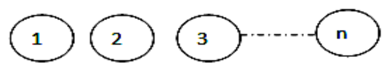
Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time O(n). And for the sequence of Find operations it will take time

complexity of $O(\sum_{i=1}^{n} i) = O(n^2)$.

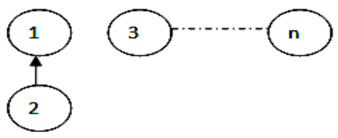
We can improve the performance of union and find by avoiding the creation of degenerate tree by applying weighting rule for Union.

4 Weighting rule for Union:

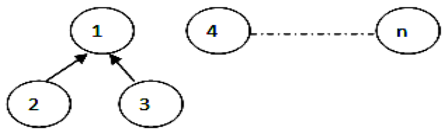
If the number of nodes in the tree with root I is less than the number in the tree with the root j, then make ‘j’ the parent of i; otherwise make ‘i' the parent of j.



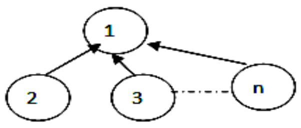
Consider Set



Union(1,2)



Union (1,3)



Union(1,n)

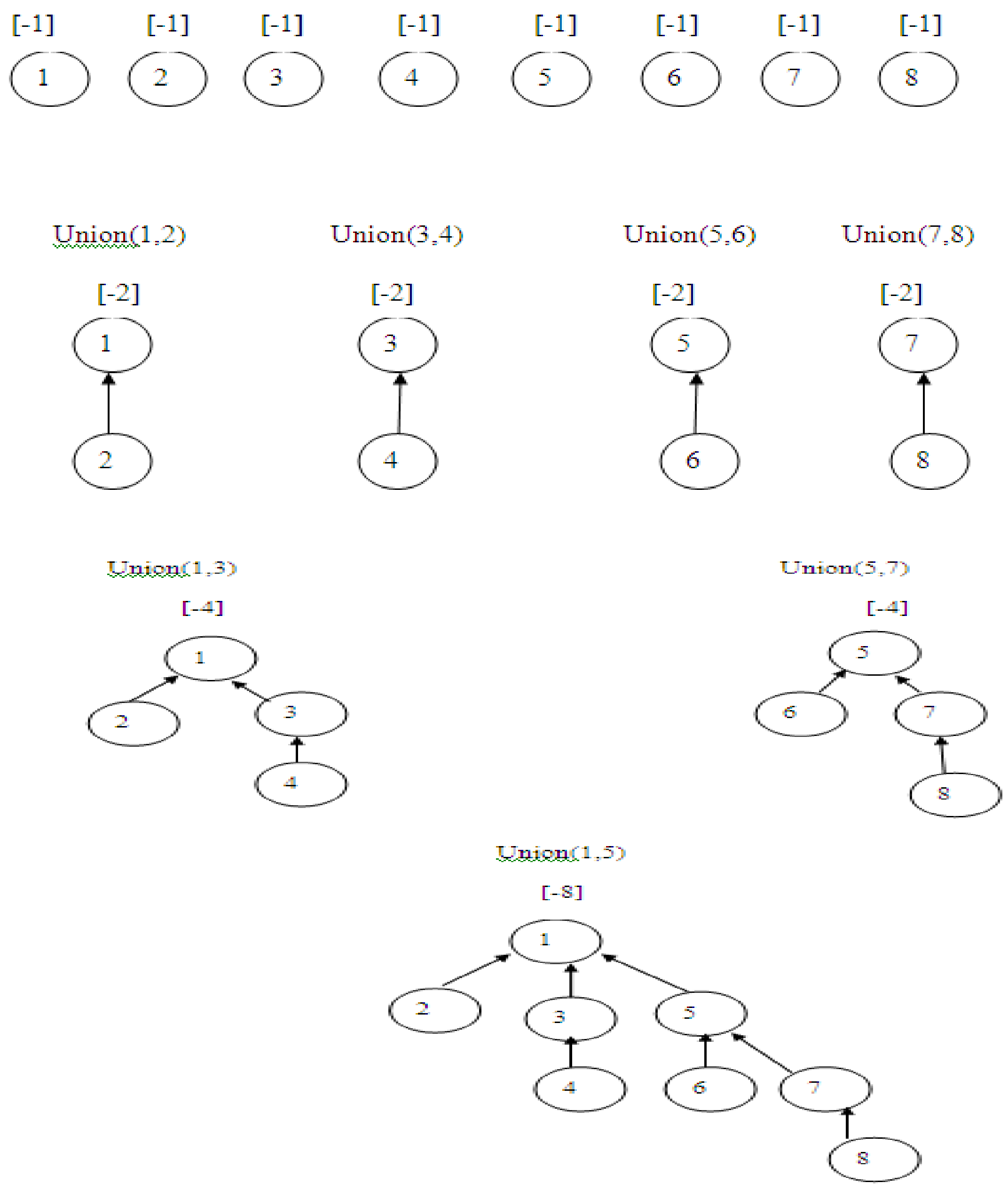
To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain "count" field in the root of every tree. If 'i' is the root then count[i] equals to number of nodes in tree with root i. Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

Algorithm WeightedUnion(i,j)

```
 //Union sets with roots i  and j , i≠j using the weighted rule

 // P[i]=-count[i] and p[j]=-count[j]

 {

          temp:= P[i]+P[j];
          if (P[i]>P[j]) then
          {
            // i has fewer nodes P[i]:=j;
                P[j]:=temp;
           }
           else
           {
            // j has fewer nodes P[j]:=i;
                P[i]:=temp;
           }

}
```

5 Collapsing rule for find:

If j is a node on the path from i to its root and p[i]≠root[i], then set P[j] to root[i]. Consider the tree created by WeightedUnion() on the sequence of $1 \le i \le 8$.
Union(1,2), Union(3,4), Union(5,6) and Union(7,8)



Now process the following eight find operations Find(8),

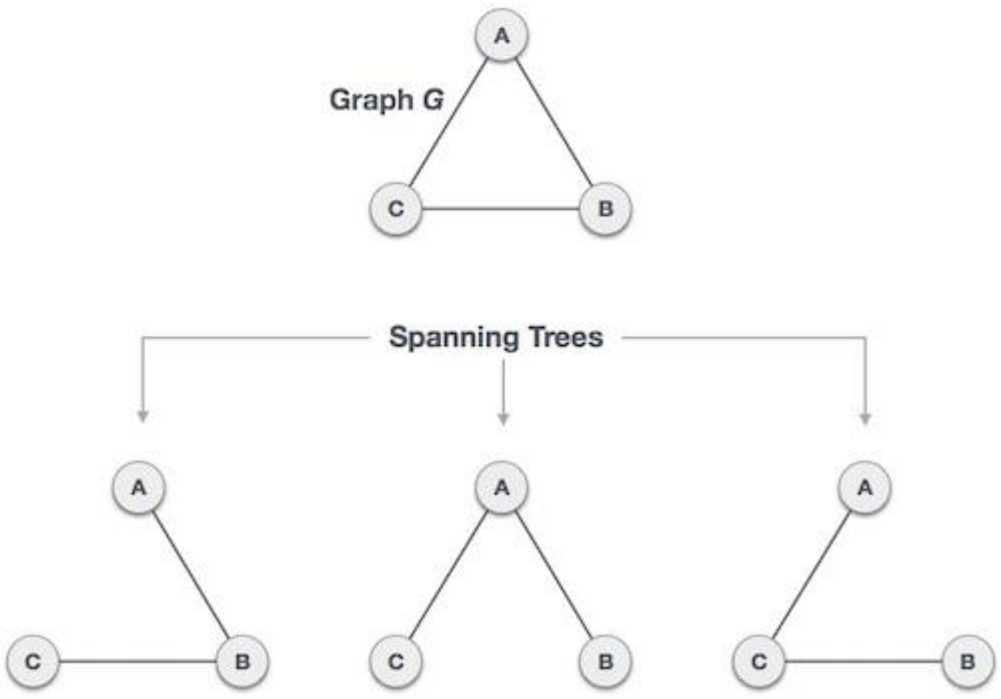
Find(8)……………………..Find(8)

If SimpleFind() is used each Find(8) requires going up three parent link fields for a total of 24 moves . When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i
// use the collapsing rule to collapse all nodes from i to root.
{
    r:=i;
    while(P[r]>0) do r:=P[r]; //Find root while(i≠r)
     {
        //reset the parent node from element i to the root s:=P[i];
        P[i]:=r;
        i:=s;
     }
}
```

4 Spanning Tree : A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

1 General Properties of Spanning Tree :

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

2 Mathematical Properties of Spanning Tree :

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

3 Application of Spanning Tree :

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

4.Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

5.Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here −

- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms

5 Graph traversals

A graph is a collection of vertices and edges and it is represented by G (V, E).

```
Here   G->Graph
       V->Vertices
       E->Edges
```

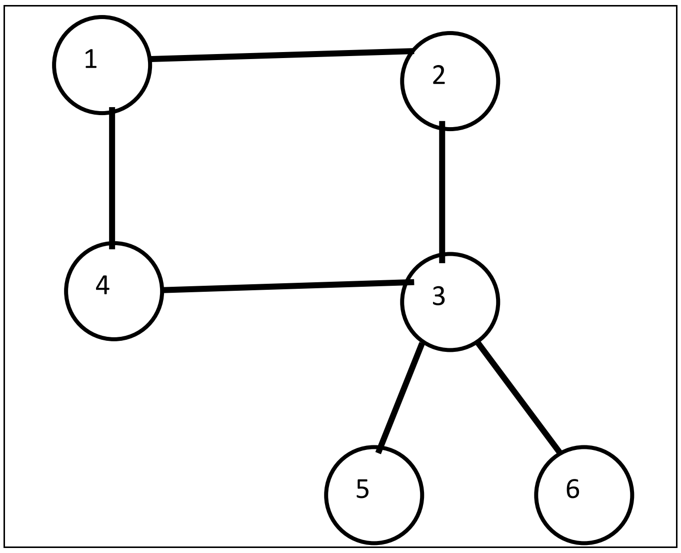
Here we implement 2 graph traversals.

1. BFS(Breadth First Search)
2. DFS(Depth First Search)

1 BFS (Breadth First Search)

```
Algorithm BFS (V)
//A breadth first search of G is carried out beginning at vertex V.
For any node i, visited[i] =1 if i has already been visited. The graph
G and array visited [] are global, visited [] is initialized to zero
{
    u := v; // q is a queue of unexpected vertices
    Visited [v]:=1;
    Repeat
    {
        For all vertices w adjacent u do.
        {
            If (visited[w] =0) then
            {
                    Add w to q; // w is unexplored
                     Visited [w]:=1;
                   }
                 }
                 If q is empty then return; // no unexplored vertex
                 Delete the next element, u from q; // get first
unexplored
              } Until (false);
  }
```

Example:



Our search operation start with vertex 1.

Queue is

1					

- Find out adjacent vertices of 1 and insert into queue here 2 and 4 are adjacent vertices of 1.After inserting 2 and 4 into queue we get queue

1	2	4			

- Apply delete operation on queue. After applying deletion operation 1 will be deleted so BFS searching order is 1.

~~1~~	2	4			

- Now find out adjacent vertices of 2 and insert into queue here 3 is adjacent vertex of 2.After inserting 3 in queue, we get

~~1~~	2	4	3		

- After applying deletion operation on queue 2 will be deleted so BFS searching order is 2.

~~1~~	~~2~~	4	3		

- There is no adjacent vertex of 4, we apply deletion operation 4 will be deleted.

~~1~~	~~2~~	~~4~~	3		

- Now find out adjacent vertices of 3 and insert into the queue here 5 and 6 are adjacent vertices of 5.

~~1~~	~~2~~	~~4~~	3	5	6

- After deletion operation on queue. After applying deletion operation 3 will be deleted so BFS searching order is 3.

~~1~~	~~2~~	~~4~~	~~3~~	5	6

- There is no adjacent vertices of 5 so by applying deletion operation 5 will be deleted.

~~1~~	~~2~~	~~4~~	~~3~~	~~5~~	6

- There are no adjacent vertices of 6 so by applying deletion operation 6 will be deleted.

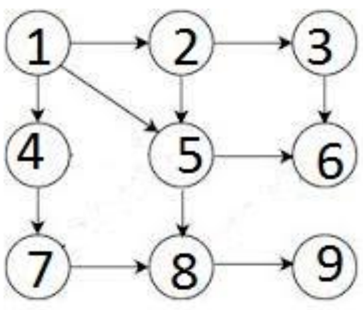
~~1~~	~~2~~	~~4~~	~~3~~	~~5~~	~~6~~

- Therefore BFS searching is 1 2 3 4 5 6

2 Depth First Search (DFS) :-

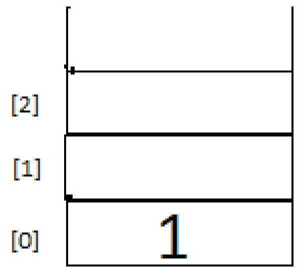
```
Algorithm DFS (V)
 // Given an undirected (directed) graph G = (V,E) with n vertices and
an array visited[] initially set to zero , this algorithm visits all vertices
reachable from V,G  and visited [] are global
{
  Visited [v]:=1;
   For each vertex w adjacent from v do
    {
       If (visited [w] =0) then DFS (W);
      }
}
```

Example:-

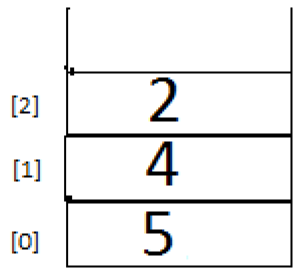


- Push starting node into stack.

Stack = 1

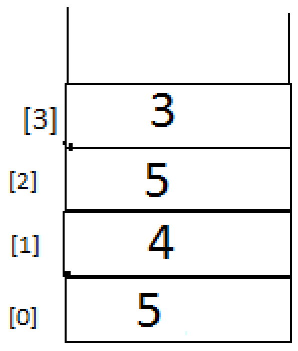


- Pop node 1 from the stack, traverse it push all the unvisited adjacent nodes 5, 4, 2 of the Pop element on stack.



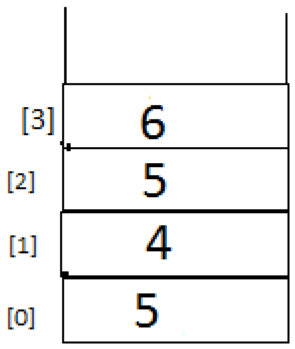
Top= 2 stack= 5, 4, 2 traversals=1

- Pop the element node 2 from the stack, traverse it and push all its unvisited adjacent nodes 5,3 on stack.



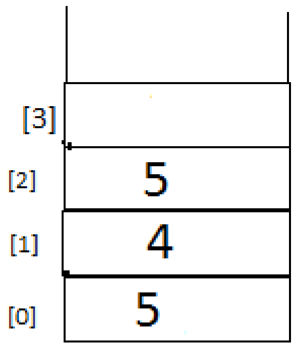
Stack=5, 4, 5,3 traversal=1,2 top =3

- Pop the element node 3 from the stack, traverse it and push its unvisited adjacent node 6 onto the stack.



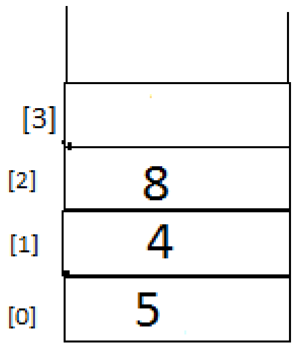
Stack=5, 4, 5, 6 traversal =1, 2, 3 top =3

- Pop the element node 6 from the stack, traverse it node 6 has no adjacent nodes so nothing is pushed in the stack.



Stack=5, 4, 5 traversal=1, 2, 3, 6 top=2

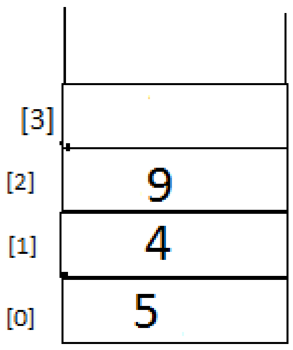
- Pop the element node 5 from stack, traverse it and push its unvisited adjacent node 8 on the stack, node 6 is its adjacent node but it has been visited so it is not pushed in the stack.



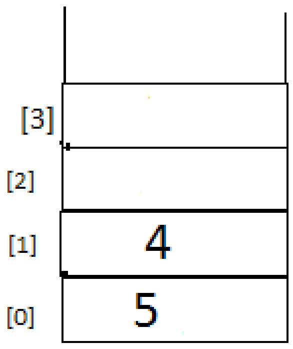
Stack=5, 4, 8 traversal=1, 2, 3, 6, 5 top=2

- Pop the element node 8 from the stack, traverse it and push its unvisited adjacent node 9 on the stack.

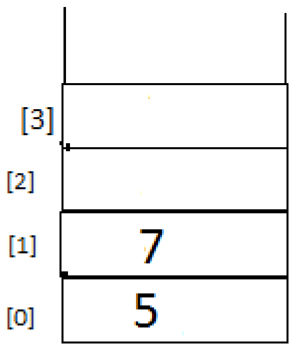
Stack=5, 4, 9 traversal=1, 2, 3, 6, 5, 8 top=2



- Pop the element node 9 from stack, traverse it node 9 has no adjacent nodes so nothing is pushed.
  Stack=5, 4 traversal=1, 2, 3, 6, 5, 8, 9 top=1

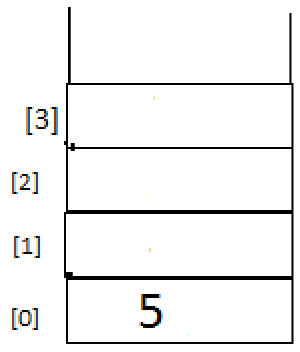


- Pop the element node 4 from the stack, traverse it and push its unvisited adjacent node 7 on the stack.
  Stack=5, 7 traversal=1, 2, 3, 6, 5, 8, 9, 4
  top=1



- Pop the element node 7 from stack, traverse it its unvisited adjacent node 8 has been visited so it is not pushed.
  Stack=5 traversal=1, 2, 3, 6, 5, 8, 9, 4, 7

Top=0

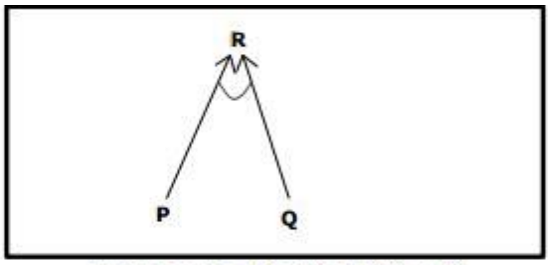


- The element node 5 is already in traversal so ignore it.

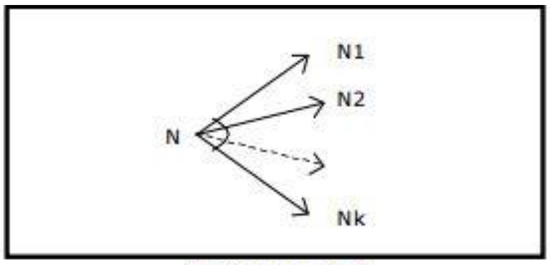
AND/OR GRAPH:

And/or graph is a specialization of hypergraph which connects nodes by sets of arcs rather than by a single arcs. A hypergraph is defined as follows:
A hypergraph consists of:
N, a set of nodes,
H, a set of hyperarcs defined by ordered pairs, in which the first implement of the pair is a node of N and the second implement is the subset of N.
An ordinary graph is a special case of hypergraph in which all the sets of decendent nodes have a cardinality of 1.

Hyperarcs also known as K-connectors, where K is the cardinality of the set of decendent nodes. If K = 1, the descendent may be thought of as an OR nodes. If K > 1, the elements of the set of decendents may be thought of as AND nodes. In this case the connector is drawn with individual edges from the parent node to each of the decendent nodes; these individual edges are then joined with a curved link.
And/or graph for the expression P and Q -> R is follows:



Expression for P and Q -> R



A K-Connector

or The K-connector is represented as a fan of arrows with a single tie is shown above. The and/or graphs consists of nodes labelled by global databases. Nodes labelled by compound databases have sets of successor nodes. These successor nodes are called AND nodes, to process the compound database to termination, all the compound databases must be processed to terminatioexample consider, consider a boy who collects stamps (M). He has for the purpose of exchange a winning conker (C), a bat (B) and a small toy animal (A). In his class there are friends who are also keen collectors of different items and will make the following exchanges.

1. 1 winning conker (C) for a comic (D) and a bag of sweets (S).

2. 1 winning conker (C) for a bat (B) and a stamp (M).

3. 1 bat (B) for two stamps (M, M). 4. 1 small toy animal (A) for two bats (B, B) and a stamp (M).

The problem is how to carry out the exchanges so that all his exchangable items are converted into stamps (M). This task can be expressed more briefly as:

1. Initial state = (C, B, A)

2. Transformation rules:

a. If C then (D, S)

b. If C then (B, M)

c. If B then (M, M)

d. If A then (B, B, M)

3. The goal state is to left with only stamps (M, ... , M).

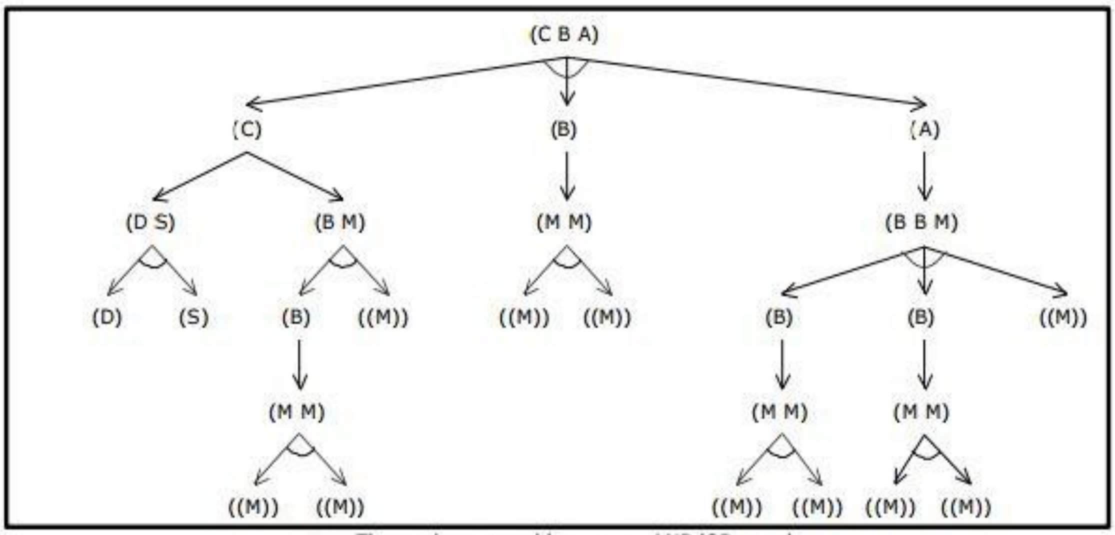
The figure shows that, a lot of extra work is done by redoing many of the transformations. This repetition can be avoided by decomposing the problem into subproblems. There are two major ways to order the components:

1. The components can either be arranged in some fixed order at the time they are generated (or).

2. They can be dynamically reordered during processing.

The more flexible system is to reorder dynamically as the processing unfolds. It can be represented by and/or graph. The solution to the exchange problem will be: Swap conker for a bat and a stamp, then exchange this bat for two stamps. Swap his own bat for two more stamps, and finally swap the small toy animal for two bats and a stamp. The two bats can be exchanged for two stamps.

The previous exchange problem, when implemented as an and/or graph looks as follows:



The exchange problem as an AND/OR graph

Game Trees

In this chapter we look at using trees for game playing, in particular the problems of searching game trees. We will classify all games into the folowing three groups

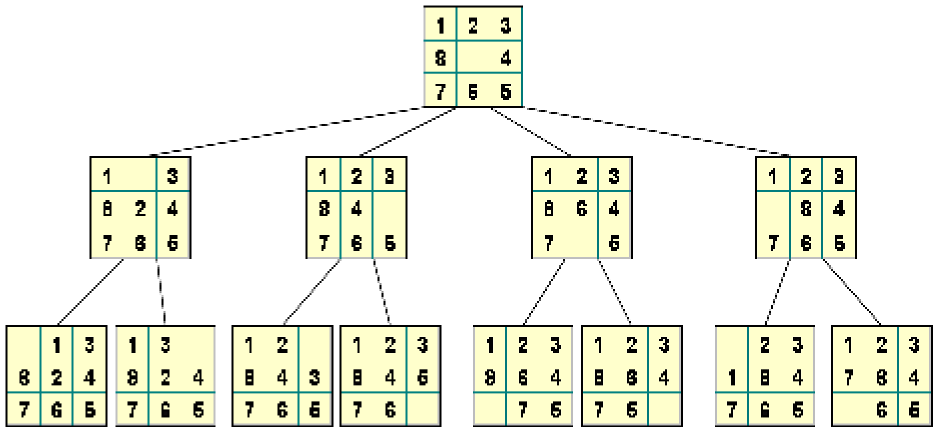
1. Single-player pathfinding problems.
   - Rubik's Cube
   - Sliding puzzle.
   - Travelling Salesman Problem.
2. Two-player games.
   - Chess
   - Checkers
   - Othello
3. Constraint satisfaction problems.
   - Eight Queens
   - Sudoku

Each game consists of a problem space, an initial state, and a single (or a set of) goal states. A problem space is a mathematical abstraction in a form of a tree:

- the root represents current state
- nodes represent states of the game
- edges represent moves
- leaves represent final states (win, loss or draw)

For example, in the 8-Puzzle game

- nodes: the different permutations of the tiles.
- edges: moving the blank tile up, down, right or left.



For some problems, the choice of a problem space is not so obvious. One general rule is that a smaller representation, in the sense of fewer states to search, is often better then a larger one. A problem space is characterized by two major factors.

The branching factor - the average number of children of the nodes in the space.

- The eight puzzle has a branching factor of 2.13
- Rubik’s cube has a branching factor of 13.34
- Chess has a branching factor of about 35

The solution depth - the length of the shortest path from the initial node to a goal node.

The size of a solution space:

- Tic-Tac-Toe is 9! = 362,880
- 8-puzzle - 9!/2
- Checkers - 10^40
- Chess - 10^120 (40 moves, 35 branch factor - 35^(2*40))

How to search for a move?

Brute-Force Searches

- Breadth-First Search (BFS)

  BFS expands nodes in order of their depth from the root.

  Implemented by first-in first-out (FIFO) queue.

  BFS will find a shortest path to a goal.

  Time/Space Complexity - branching factor b and the solution depth d. Generate all the nodes up to level d.

  total number of nodes = 1 + b + b^2 + ... + b^d = O(b^d)

  BFS will exhaust the memory in minutes.

- Depth-First Search (DFS)

  Implemented by LIFO stack

Space Complexity is linear in the maximum search depth.

DFS generate the same set of nodes as BFS - Time Complexity is O(b^d)

The first solution DFS found may not be the optimal one.

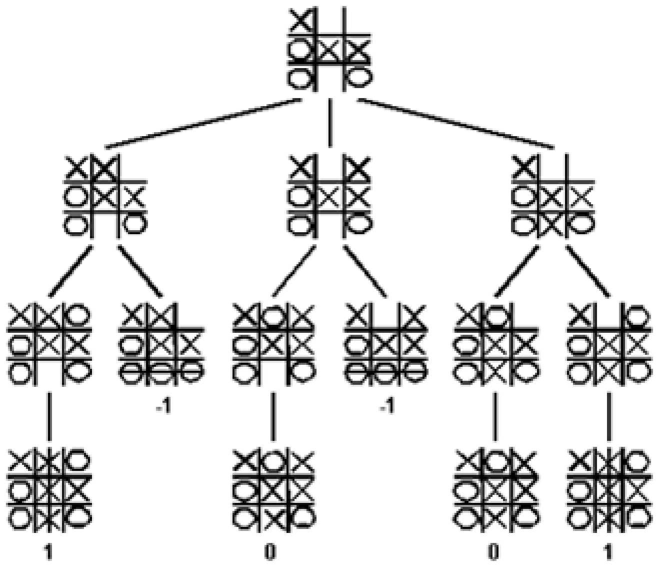
On infinite tree DFS may not terminate.

- Depth-First Iterative-Deepening

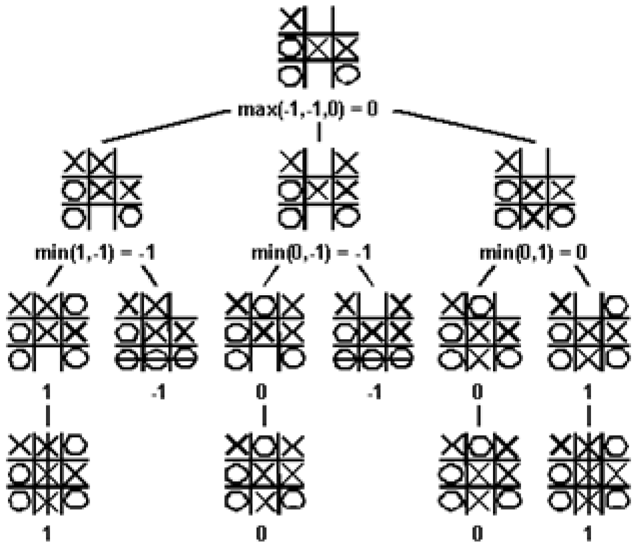
  First performs a DFS to depth one. Than starts over executing DFS to depth two and so on

Minimax

We consider games with two players in which one person's gains are the result of another person's losses (so called zero-sum games). The minimax algorithm is a specialized search algorithm which returns the optimal sequence of moves for a player in an zero-sum game. In the game tree that results from the algorithm, each level represents a move by either of two players, say A- and B-player. Below is a game tree for the tic-tac-toe game



The minimax algorithm explores the entire game tree using a depth-first search. At each node in the tree where A-player has to move, A-player would like to play the move that maximizes the payoff. Thus, A-player will assign the maximum score amongst the children to the node where Max makes a move. Similarly, B-player will minimize the payoff to A-player. The maximum and minimum scores are taken at alternating levels of the tree, since A and B alternate turns.



The minimax algorithm computes the minimax decision for the leaves of the game tree and than backs up through the tree to give the final value to the current state.

Heuristic Search

So far we have looked at search algorithms that can in principle be used to systematically search the whole search space. Sometimes however it is not feasible to search the whole search space - it's just too big. In this case we need to use heuristic search. The basic idea of heuristic search is that, rather than trying all possible search paths, we focus on paths that seem to be getting us closer to the goal state. Of course, we generally can't be sure that we are really near the goal state, but we might be able to have a good guess. Heuristics are used to help us make that guess.

To use heuristic search we need an *evaluation function* that rankes nodes in the search tree according to some criteria (for example, how close we are to the target). This function provides a quick way of guessing.

Best First Search.
The search is similar to BFS, but instead of taking the first node it always chooses a node with the best score, according to an evaluation function. If we create a good evalution function, best first search may drastically cut down the amount of search time.

Here are the two most important properties of a heuristic function:

- it must provide an accurate estimator of the cost to reach a goal.
- it must be cheap to compute.
- it always must be a lower bound on actual solution cost.

A* algorithm
Best first search doesn't take into account the cost of the path from a start state to the current state. So, we may find a solution but it may be not a very good solution. There is a variant of best first search known as A* which attempts to find a solution which minimizes the total cost of the solution path. This algorithm combines advantages of breadth first search with advantages of best first search.

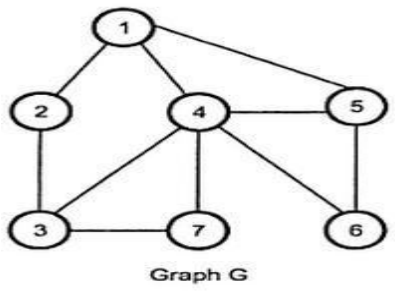
In the A* algorithm the score assigned to a node is a combination of the cost of the path so far A(S) and the estimated cost E(S) to solution.

$$H(S) = A(S) + E(S)$$

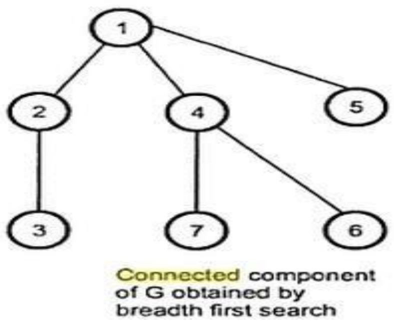
The A* algorithm looks the same as the best first algorithm, but we use this slightly more complex evaluation function.

.

Connected component: If G is connected undirected graph, then we can visit all the vertices of the graph in the first call to BFS. The sub graph which we obtain after traversing the graph using BFS represents the connected component of the graph.

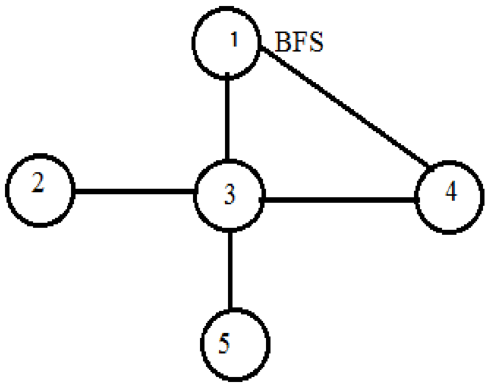


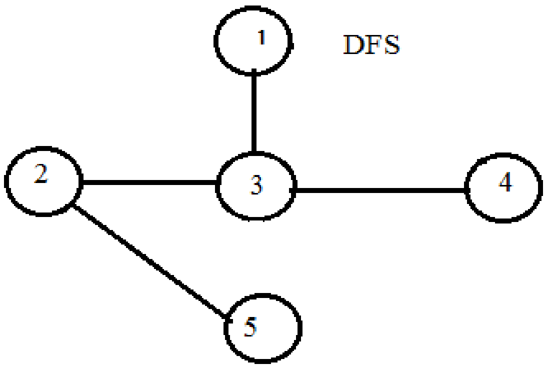
Graph G



Thus BFS can be used to determine whether G is connected. All the newly visited vertices on call to BFS represent the vertices in connected component of graph G. The sub graph formed by theses vertices make the connected component.

Spanning tree of a graph: Consider the set of all edges (u, w) where all vertices w are adjacent to u and are not visited. According to BFS algorithm it is established that this set of edges give the spanning tree of G, if G is connected. We obtain depth first search spanning tree similarly
These are the BFS and DFS spanning trees of the graph G



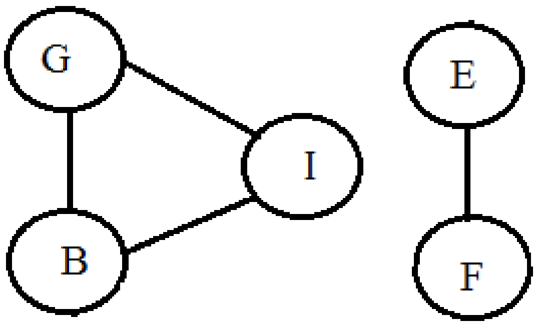


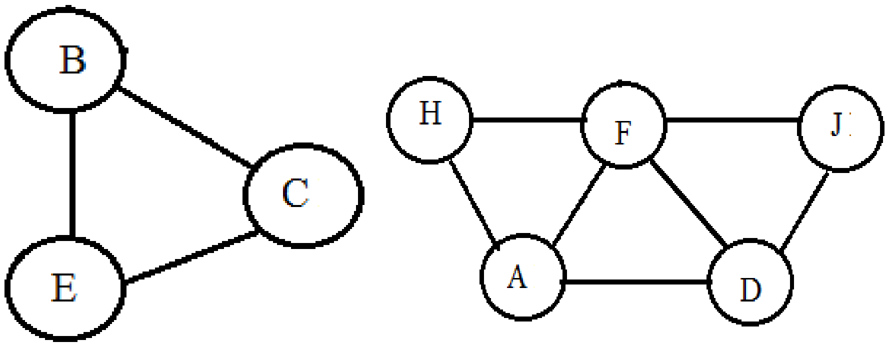
Bi-connected Components

A connected undirected graph is said to be bi-connected if it remains connected after removal of any one vertex and the edges that are incident upon that vertex.
In this we have two components.

i. Articulation point: Let G= (V, E) be a connected undirected graph. Then an articulation point of graph 'G' is a vertex whose articulation point of graph is a vertex whose removal disconnects the graph 'G'. It is also known as "cut point".

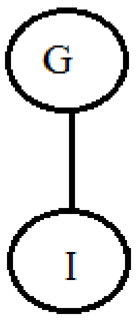
ii. Bi-connected graph: A graph 'G' is said to be bi-connected if it contains no-articulation point.

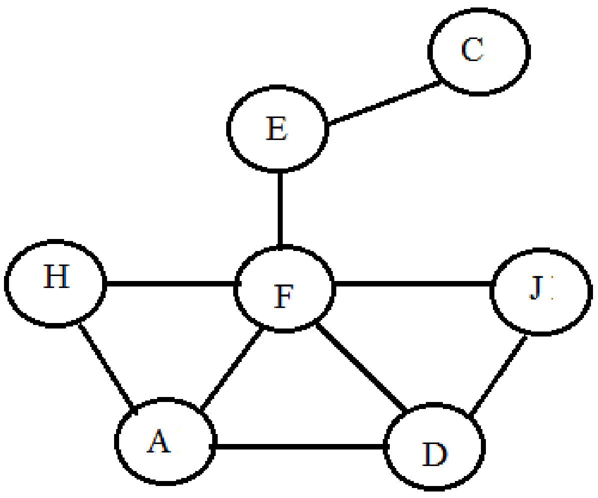




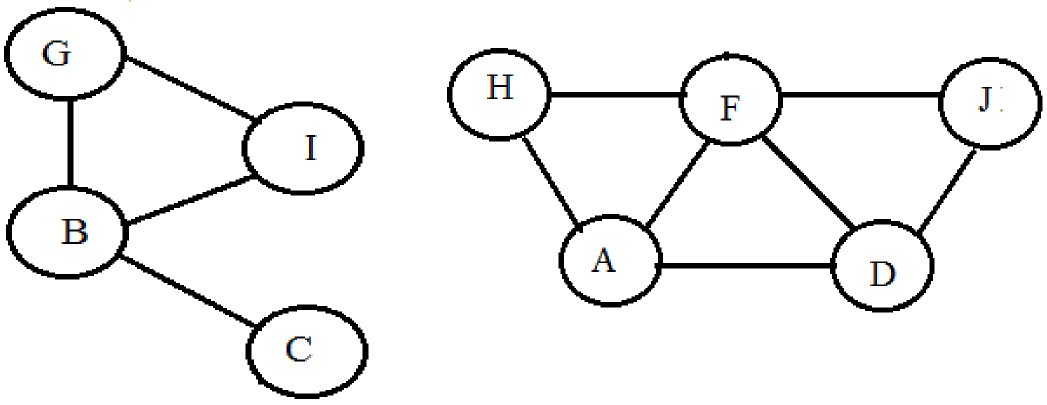
Articulation points for the above undirected graph are B, E, F

i) After deleting vertex B and incident edges of B, the given graph is divided into two components

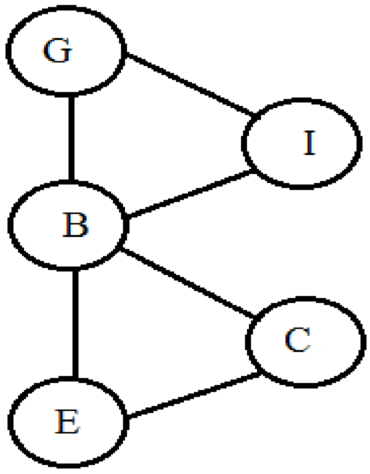


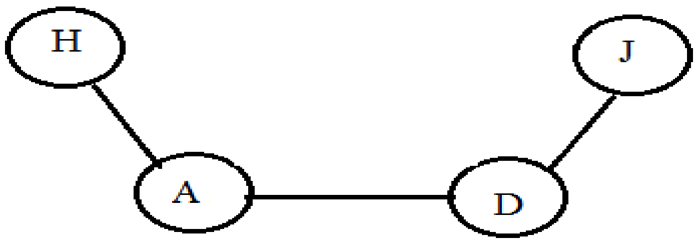


ii) After deleting the vertex E and incident edges of E, the resulting components are



iii) After deleting vertex F and incident edges of F, the given graph is divided into teo components.





Note: If there exists any articulation point, it is an undesirable feature in communication network where joint point between two networks failure in case of joint node fails.

Algorithm to construct the Bi- Connected graph

1. For each articulation point ‘a’ do

2. Let B1, B2, B3 ….Bk are the Bi-connected components

3. Containing the articulation point ‘a’

4. Let Vi E Bi, Vi # a i<=i<=k

5. Add (Vi,Vi+1) to Graph

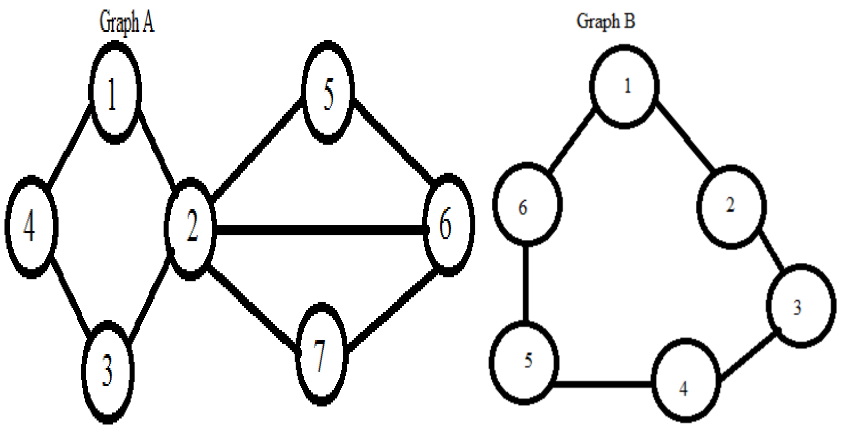
G. Vi-vertex belong Bi

Bi-Bi-connected component

i- Vertex number 1 to k

a- articulation point

Exercise


Graph A
1
5
4
2
6
3
7
Graph B
1
6
2
3
5
4