UNIT-II

DEARCHING AND TRAVERSAL TECHNIQUES

Disjoint Set Operations

Set:

A set is a collection of distinct elements. The Set can be represented,for examples, asS1={1,2,5,10}.

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 S2are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are

1. Union
2. Find

Disjoint setUnion:

If Si and Sj are two 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 US2={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

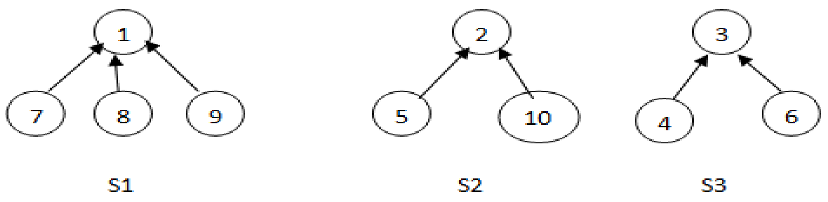
Set Representation:

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:

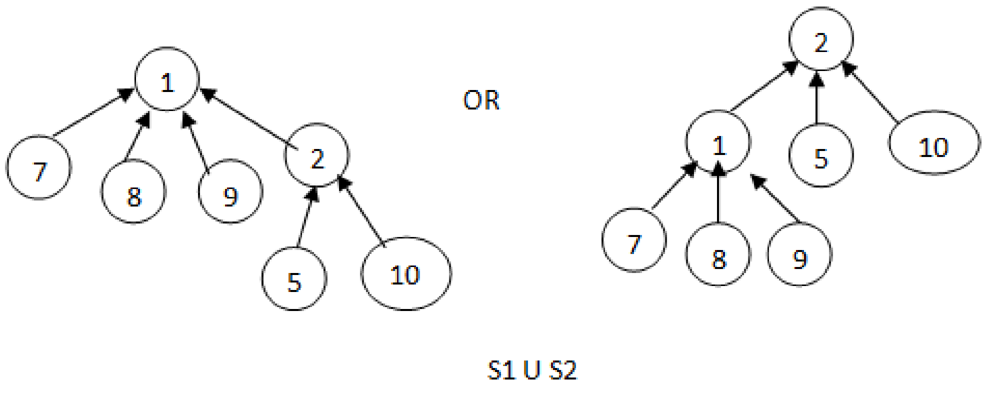
S1={1,7,8,9}   S2={2,5,10}   s3={3,4,6}

Then these sets can be represented as



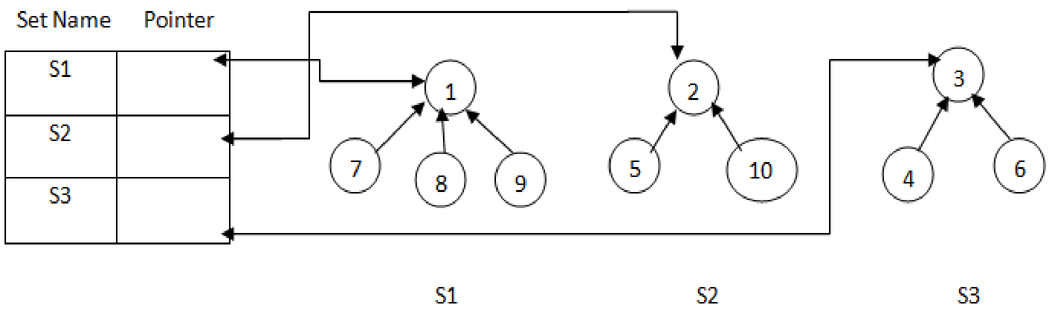
Disjoint Union:

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.

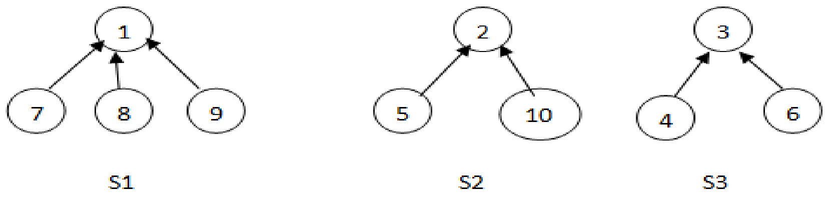


Union and Find Algorithms:

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 shownbelow.



I	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
P	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

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;
        }
```

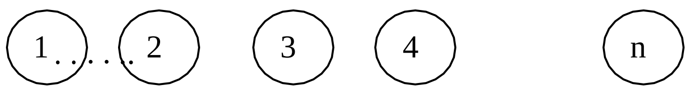
Algorithm for find operation:

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.

```
Algorithm SimpleFind(i)
{
        while( P[i]≥0)
        i:=P[i];
        returni;
}
```

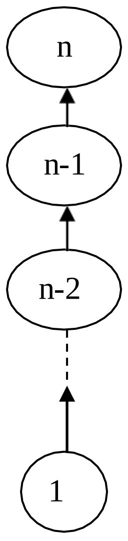
Analysis of SimpleUnion(i,j) and SimpleFind(i):

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.

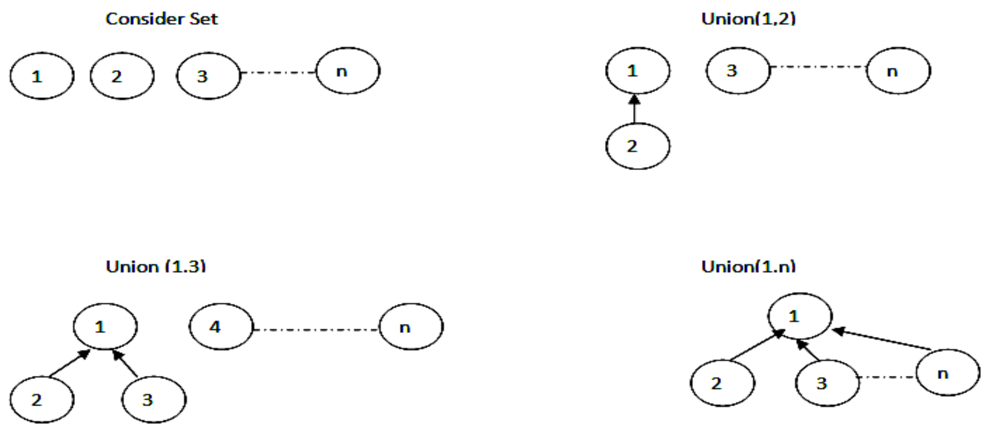


Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time O(n).And for the sequence of Find operations it will take

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

Weighting rule forUnion:

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.



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 rooti.
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.

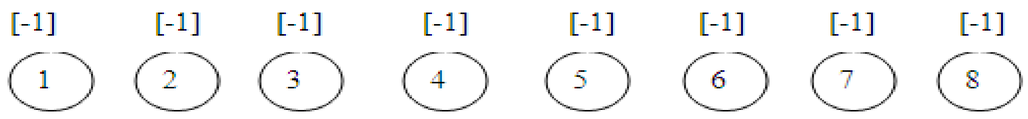
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j, i≠j using the weighted rule

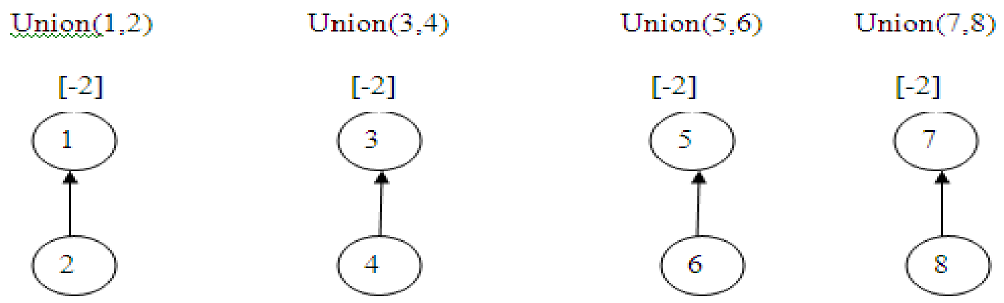
// P[i]=-count[i] andp[j]=-count[j]

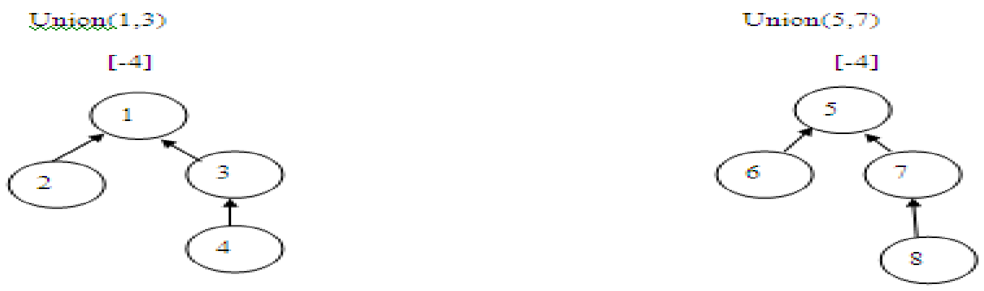
```
{
        temp:=P[i]+P[j];
        if (P[i]>P[j])then
        {
          // i has fewer nodes
              P[i]:=j;
              P[j]:=temp;
         }
         else
         {
          // j has fewer nodes
              P[j]:=i;
              P[i]:=temp;
         }
}
```

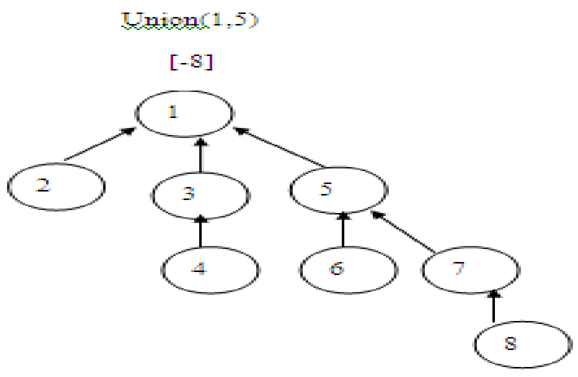
Collapsing rule for find:

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 CoIlapsingFind(i)
// Find the root of the tree containing element i. Use the
```

```
    // collapsing rule to collapse all nodes from i to the root.
     {        r := i;
     while  (p[r]  >0) do
      r := p[r]; / Find the root,
        while (i< r) do // Collapse nodes from i to root r,
               r:=p[i];
               return r;
 }
```

SEARCHING

Search means finding a path or traversal between a start node and one of a set of goal nodes. Search is a study of states and their transitions.

Search involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal.

Techniques for Traversal of a Binary Tree:

A binary tree is a finite (possibly empty) collection of elements. When the binary tree is not empty, it has a root element and remaining elements (if any) are partitioned into two binary trees, which are called the left and right subtrees.

There are three common ways to traverse a binary tree: Preorder, Inorder, postorder In all the three traversal methods, the left sub tree of a node is traversed before the right sub tree. The difference among the three orders comes from the difference in the time at which a node is visited.

Inorder Traversal:

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.

The algorithm for preorder traversal is as follows:

```
  treenode =record

 {
          Type data;       //Type is the data type of data.
           Treenode *lchild, *rchild;
}
Algorithm inorder(t)
 // t is a binary tree. Each node of t has three fields: lchild, data, and rchild.
 {
          If( t ≠0)then

          {

                    inorder  (t→ lchild);
                    visit(t);
                    inorder (t →rchild);
```

```
    }
}
```

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:

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

The algorithm for preorder traversal is as follows:

Algorithm Preorder (t)

```
// t is a binary tree. Each node of t has three fields; lchild, data, and rchild.

{
    If( t ≠0)then
    {
        visit(t);
        Preorder (t→lchild);
        Preorder
        (t→rchild);
    }
}
```

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:

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

The algorithm for preorder traversal is as follows:

Algorithm Postorder (t)

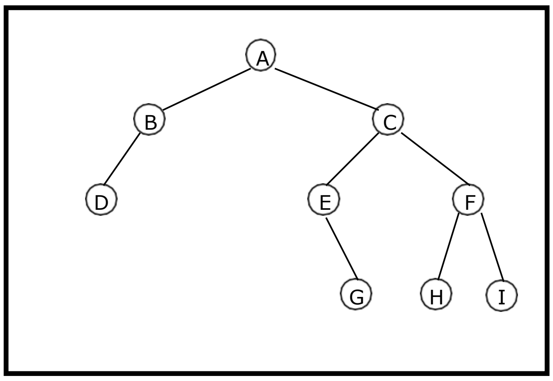
```
// t is a binary tree. Each node of t has three fields : lchild, data, and rchild.

{
    If( t ≠0)then
    {
        Postorder(t→  child);
        Postorder(t→rchild);
        visit(t);
    }   }
```

Examples for binary tree traversal/search technique:

Example1:

Traverse the following binary tree in pre, post and in-order.



Preordof the vertices: A, B, D, C, E, G, F, H, I.

Post order of the vertices: D, B, G, E, H, I, F, C, A.

Inorder of the vertices: D, B, A, E, G, C, H, F, I

Binary Tree Pre,Post and In-order Traversing

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.

The algorithm for inorder Non Recursive traversal is asfollows:

Algorithm **inorder**()

```
{
        stack[1] = 0
        vertex =root
top:    while(vertex ≠0)
        {
```

```
        push the vertex into the
        stack            vertex
        =leftson(vertex)

    }

    pop the element from the stack and make it as vertex

    while(vertex ≠0)

    {

        print the vertex node
        if(rightson(vertex)
        ≠0)

        {

            vertex           =
            rightson(vertex)   goto
            top

        }

        pop the element from the stack and made it as vertex

    }

}
```

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 ≠ 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) $\neq$0)
            vertex :=leftson(vertex)
else
            pop the element from the stack and made it as vertex
}
 }
```

Postorder 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.

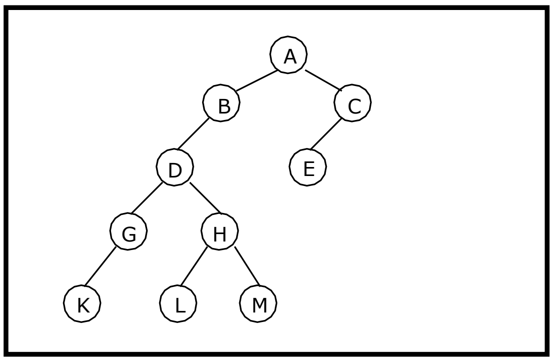
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

The algorithm for postorder Non Recursive traversal is as follows:

```
   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
        }
}
```

Example1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



- Preorder traversal yields: A, B, D, G , K, H, L, M , C , E
- Postorder t raversal yields: K, G , L, M , H, D, B, E, C , A
- Inorder traversal yields: K, G , D, L, H, M , B, A, E, C

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.

Current vertex	**Stack**	**Processed nodes**	**Remarks**
A	0		PUSH0
	0 A B D GK		PUSH the left most path ofA
K	0 A B DG	K	POPK
G	0 A BD	KG	POP G since K has no right son
D	0 AB	K GD	POP D since G has no right son
H	0 AB	K GD	Make the right son of D as vertex
H	0 A B HL	K GD	PUSH the leftmost path of H
L	0 A BH	K G DL	POPL
H	0 AB	K G D LH	POP H since L has no right son
M	0 AB	K G D LH	Make the right son of H as vertex
	0 A BM	K G D LH	PUSH the left most path of M
M	0 AB	K G D L HM	POPM
B	0A	K G D L H MB	POP B since M has no right son
A	0	K G D L H M BA	Make the right son of A as vertex
C	0 CE	K G D L H M BA	PUSH the left most path of C
E	0C	K G D L H M B AE	POPE
C	0	K G D L H M B A EC	Stop since stack is empty

Postorder 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.

Curren t	**Stack**	**Processed nodes**	**Remark**
A	0		PUSH0
	0 A -C B D -H GK		PUSH the left most path of A with a -ve for right sons
	0 A -C B D-H	KG	POP all +ve nodes K and G
H	0 A -C BD	KG	Pop H
	0 A -C B D H -ML	KG	PUSH the left most path of H with a -ve for right sons
	0 A -C B D H-M	K GL	POP all +ve nodes L
M	0 A -C B DH	K GL	PopM
	0 A -C B D HM	K GL	PUSH the left most path of M with a -ve for rightsons
	0 A-C	K G L M H DB	POP all +ve nodes M, H, D
C	0A	K G L M H DB	PopC
	0 A CE	K G L M H DB	PUSH the left most path of C with a -ve for rightsons
	0	K G L M H D B E	POP all +ve nodes E, C andA
	0		Stop since stack isempty

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.

Current vertex	**Stack**	**Processednodes**	**Remarks**
A	0		PUSH0
	0 CH	A B D GK	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0C	A B D GK	POPH

	0 CM	A B D G K HL	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0C	A B D G K HL	POPM
	0C	A B D G K H LM	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no leftpath
C	0	A B D G K H LM	PopC
	0	A B D G K H L M CE	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son
	0	A B D G K H L M CE	Stop since stack is empty

Subgraphs and SpanningTrees:

Subgraphs: A graph G' = (V', E') is a subgraph of graph G = (V, E) iff V' ⊆V and E'⊆ E.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v. If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

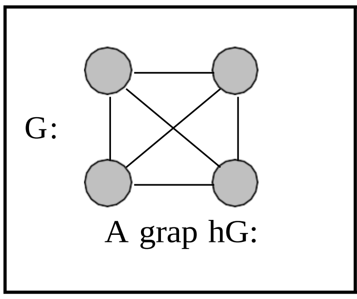
Lemma 1: *Let T be a spanning tree of a graph G. Then*

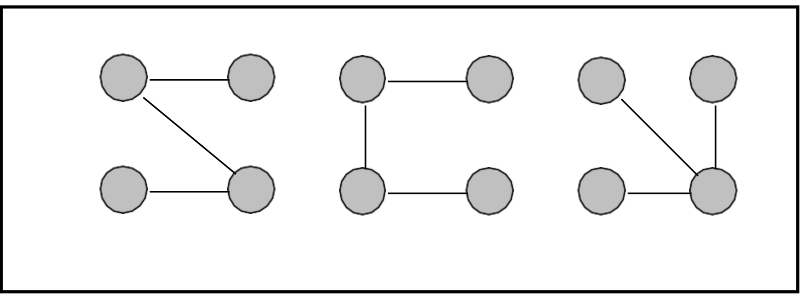
1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T isn-1.*

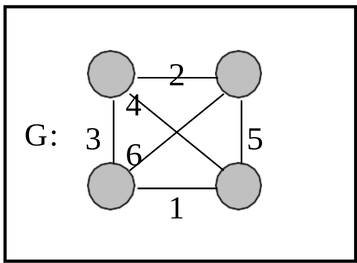
Minimum Spanning Trees(MST):

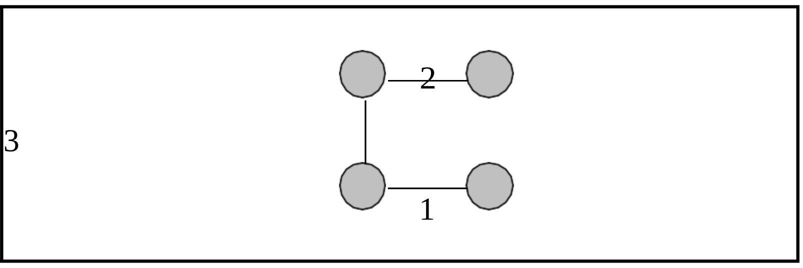
A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree w (T) is the sum of weights of all edges in T. The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.









A weighted graphG: Theminimalspanningtreefromweightedgraph G:

Examples:

To explain the Minimum Spanning Tree, let's consider a few real-world examples:

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum(i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm:

The algorithm for finding the MST, using the Kruskal’s method is as follows:

Algorithm Kruskal (E, cost, n,t)

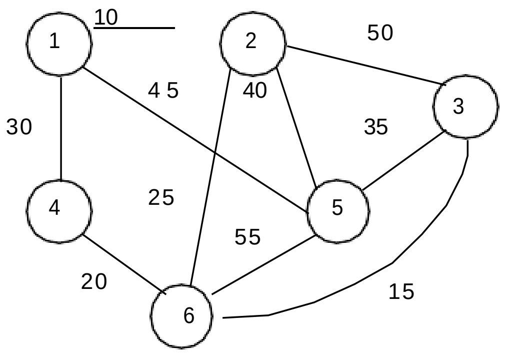
```
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). ‘t’ is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
        Construct a heap out of the edge costs using heapify; for
        i := 1 to n do parent [i] :=-1;
                                        // Each vertex is in a different set.
        i := 0; mincost :=0.0;
        while ((i < n -1) and (heap not empty))do
        {
                Delete a minimum cost edge (u, v) from the heap and re-
                heapify using Adjust;
                j := Find (u); k := Find(v); if
                (j □k)then
                {
                        i := i +1;
                        t [i, 1] := u; t [i, 2] := v; mincost
                        :=mincost +  cost [u,v]; Union
                        (j,k);
                }
        }
        if (i □n-1) then write ("no spanning tree"); else
        return mincost;
}
```

Running time:

- The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n +e).
- We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity

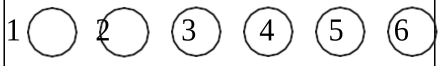
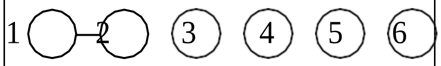
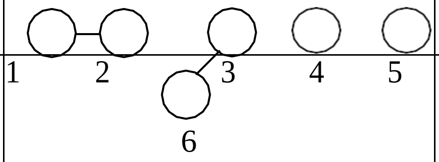
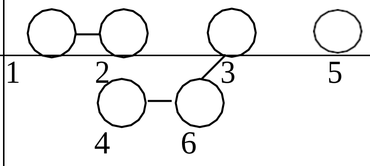
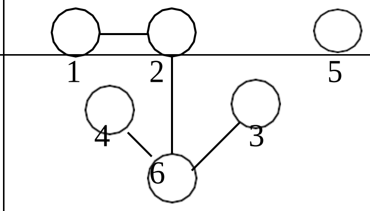
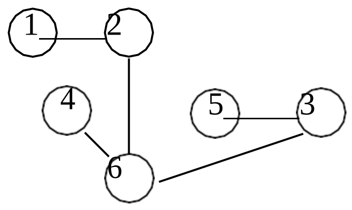
Example1:


1
2
3
4
5
6
10
50
4 5
40
30
35
25
55
20
15

Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1,2)	(3,6)	(4,6)	(2,6)	(1,4)	(3,5)	(2,5)	(1,5)	(2,3)	(5,6)

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by a set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need to check whether u and v are in the same vertex set. If so, then a cycle is created. If not then no cycle is created. Hence two **Find**s on the vertex sets suffice. When an edge is included in T, two components are combined into one and a **union** is to be performed on the two sets.

Edge	Cost	Spanning Forest	Edge Sets	Remarks
			{1}, {2}, {3}, {4}, {5},{6}	
(1, 2)	10		{1, 2}, {3},{4}, {5},{6}	The vertices 1and 2 are in different sets, so the edge Is combined
(3, 6)	15		{1, 2}, {3, 6}, {4},{5}	The vertices 3and 6 are in different sets, so the edge Is combined
(4, 6)	20		{1, 2}, {3, 4, 6}, {5}	The vertices 4and 6 are in different sets, so the edge is combined
(2, 6)	25		{1, 2, 3, 4, 6}, {5}	The vertices 2and 6 are in different sets, so the edge is combined
(1, 4)	30	Reject		The vertices 1and 4 are in the same set, so the edge is rejected
(3, 5)	35		{1, 2, 3, 4, 5,6}	The vertices 3and 5 are in the same set, so the edge is combined

MINIMUM-COST SPANNING TREES: PRIM'SALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

Algorit

hm

Algorit

hm

Prim

(E,

cost,

n,t)

```
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j]is
// either a positive real number or □if no edge (i, j)exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost [k,l];
    t [1, 1] := k; t [1, 2] :=l;
    for  i :=1 to n do                    //Initialize near if
        (cost [i, l] < cost [i, k]) then near [i] :=l;
        else near [i]  :=  k;
    near [k] :=near [l] :=0;
```

```
        for  i:=2 to n - 1do                                    // Find n - 2 additional edges fort.
        {
                Let j be an index such that near [j] □0 and
                cost [j, near [j]] is minimum;
                t [i, 1] := j; t [i, 2] := near [j]; mincost :=
                mincost + cost [j, near [j]]; near [j] :=0
                for   k:= 1 to n do                              // Update near[].
                        if ((near [k] □0) and (cost [k, near [k]] > cost [k, j])) then near
                                [k] :=j;
        }
        return mincost;
}
```

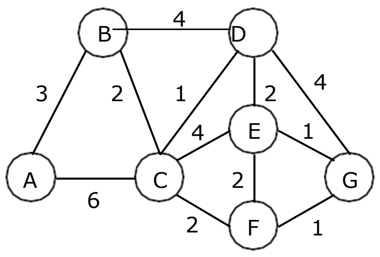
Running time:

We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get O (n^2) time when we implement dist with array, O (n + E□ log□n) when we implement it with a heap.

For each vertex u in the graph we dequeue it and check all its neighbors in O (1 + deg (u)) time.

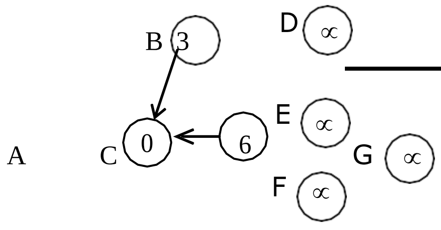
EXAMPLE1:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



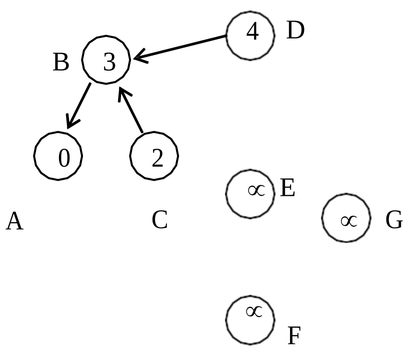
The stepwise progress of the prim's algorithm is as follows:

Step1:



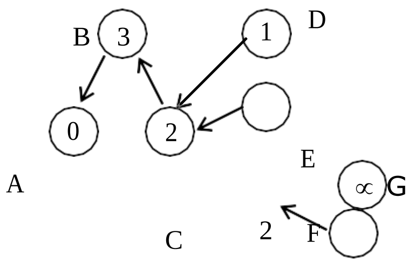
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step2:



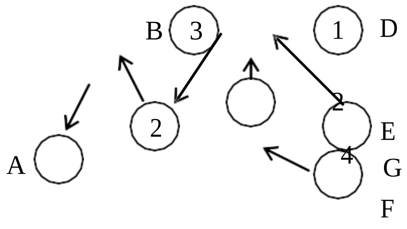
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∝	∝	∝
Next	*	A	B	B	A	A	A

Step3:



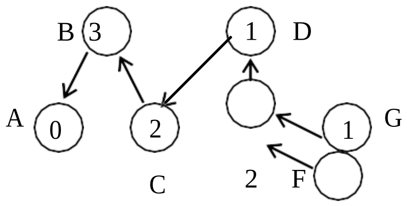
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∝
Next	*	A	B	C	C	C	A

Step4:



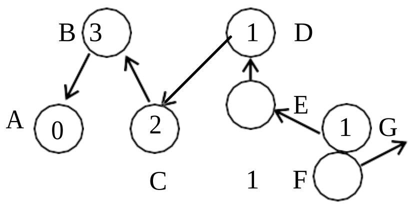
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step5:



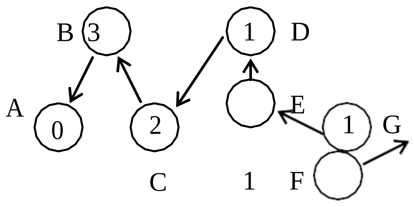
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

Step7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

GRAPH ALGORITHMS

Basic Definitions:

- **Graph G** is a pair (V, E), where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote n := |V|, m :=|E|.
- Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If (u, v) □E, then vertices u, and v are adjacent.
- We can assign weight function to the edges: $w_G(e)$ is a weight of edge e □E. The graph which has such function assigned is called **weighted graph**.
- **Degree** of a vertex v is the number of vertices u for which (u, v) □E (denote deg(v)). The number of **incoming edges** to a vertex v is called **in–degree** of the vertex (denote indeg(v)). The number of **outgoing edges** from a vertex is called **out-degree** (denote outdeg(v)).

Representation of Graphs:

Consider graph G = (V, E), where V= {v_1,v_2,….,v_n}.

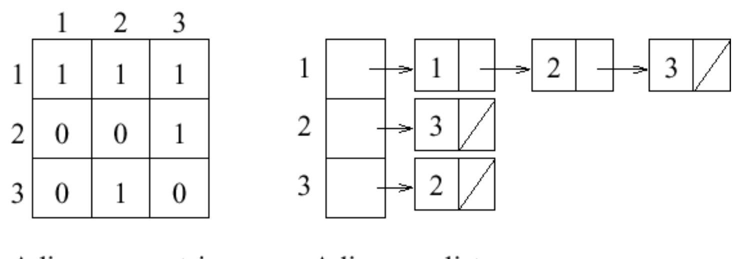
Adjacency matrix represents the graph as an n x n matrix A = ($a_{i,j}$),where

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

We may consider various modifications. For example for weighted graphs, we may have

Where default is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then default can be ∞, meaning value larger than any other value).

Adjacency List: An array Adj [1 . . . . n] of pointers where for $1 \leq v \leq n$, Adj [v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.



Adjacency matrix Adjacency list

Paths and Cycles:

A path is a sequence of vertices (v_1, v_2, ... , v_k), where for all i, (v_i, v_{i+1}) $\in$E. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices (v_1, v_2,............, v_k, $v_{k+1} = v_1$), where for all i, (v_i, v_{i+1}) $\in$E and all vertices in the cycle are distinct except pair v_1,v_{k+1}.

Techniques forgraphs:

Given a graph G = (V, E) and a vertex V in V (G) traversing can be done in two ways.

1. Depth first search
2. Breadth first search

Connected Component:

Connected component of a graph can be obtained by using BFST (Breadth first search and traversal) and DFST (Dept first search and traversal). It is also called the spanning tree.

BFST (Breadth first search and traversal):

In BFS we start at a vertex V mark it as reached (visited).The vertex V is at this time said to be unexplored (not yet discovered).A vertex is said to been explored (discovered) by visiting all vertices adjacent from it.All unvisited vertices adjacent from V are visited next.The first vertex on this list is the next to be explored.Exploration continues until no unexplored vertex is left. These operations can be performed by using Queue.

This is also called connected graph or spanning tree.

Spanning trees obtained using BFS then it called breadth first spanning trees

```
Algorithm BFS(v)
// a bfs of G is begin at vertex v
// for any node I, visited[i]=1 if I has already been visited.
// the graph G, and array visited[] are global
{
U:=v; // q is a queue of unexplored vertices.
Visited[v]:=1;
Repeat{
For all vertices w adjacent from 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 U from q; //Get 1st unexplored vertex.
} Until(false)
}
```

Maximum Time complexity and space complexity of G(n,e), nodes are in adjacency list.

$T(n, e)=\theta(n+e)$

$S(n, e)=\theta(n)$

If nodes are in adjacency matrix then

$T(n, e)=\theta(n^2)$

$S(n, e)=\theta(n)$

DFST(Dept first search and traversal).:

DFS different from BFS. The exploration of a vertex v is suspended (stopped) as soon as a new vertex is reached.In this the exploration of the new vertex (example v) begins; this new vertex has been explored, the exploration of v continues. Note: exploration start at the new vertex which is not visited in other vertex exploring and choose nearest path for exploring next or adjacent vertex.

```
Algorithm dFS(v)
// a Dfs of G is begin at vertex v
// initially an array visited[] is set to zero.
//this algorithm visits all vertices reachable from v.
// the graph G, and array visited[] are global
{
Visited[v]:=1;
For each vertex w adjacent from v do
{
If (visited[w]=0) then DFS(w);
{
```

```
            Add w to q; // w is unexplored
            Visited[w]:=1;
            }

}
```

Maximum Time complexity and space complexity of G(n,e), nodes are in adjacency list.

$T(n, e)=\theta(n+e)$

$S(n, e)=\theta(n)$

If nodes are in adjacency matrix then

$T(n, e)=\theta(n^2)$

$S(n, e)=\theta(n)$

Bi-connected Components:

A graph G is biconnected, iff (if and only if) it contains no articulation point (joint or junction).

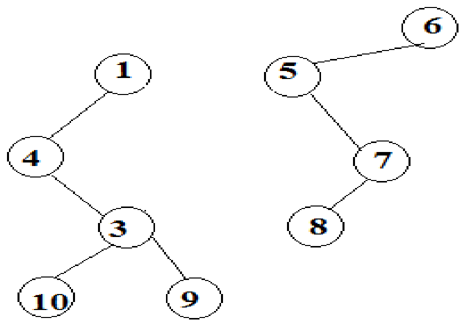
A vertex v in a connected graph G is an articulation point, if and only if (iff) the deletion of vertex v together with all edges incident to v disconnects the graph into two or more none empty components.

The presence of articulation points in a connected graph can be an undesirable(un wanted) feature in many cases.

For example

if G1 $\rightarrow$ Communication network with
Vertex $\rightarrow$ communication stations.
Edges $\rightarrow$ Communication lines.

Then the failure of a communication station I that is an articulation point, then we loss the communication in between other stations. F
Form graph G1



After deleting vertex (2)

There is an efficient algorithm to test whether a connected graph is biconnected. If the case ofgraphs that are not biconnected, this algorithm will identify all the articulation points.

Once it has been determined that a connected graph G is not biconnected, it may be desirable(suitable) to determine a set of edges whose inclusion makes the graph biconnected.