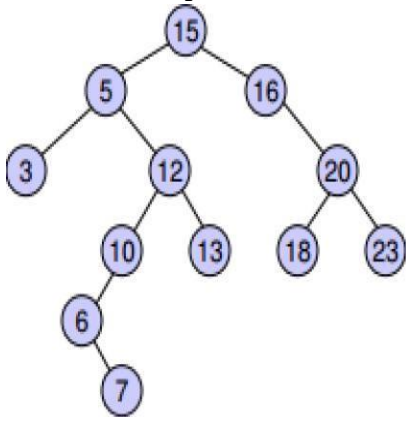
UNIT II:

Searching and Traversal Techniques:

Efficient non - recursive binary tree traversal algorithm, Disjoint set operations, union and find algorithms, Spanning trees, Graph traversals - Breadth first search and Depth first search, AND / OR graphs, game trees, Connected Components, Bi - connected components. Disjoint Sets-disjoint set operations, union and find algorithms, spanning trees, connected components and biconnected components.



Efficient non recursive tree traversal algorithms
in-order: (left, root, right) 3,5,6,7,10,12,13, 15, 16, 18, 20, 23
pre-order: (root, left, right) 15, 5, 3, 12, 10, 6, 7, 13, 16, 20, 18, 23
post-order: (left, right, root) 3, 7, 6, 10, 13, 12, 5, 18, 23,20,16, 65

Non recursive Inorder traversal algorithm

1. Start fiom the root. let's it is current.
2. Ifcurrent is not NULL. push the node on to stack.
3. Move to left child of current and go to step 2.
4. Ifcurrent is NULL, and stack is not empty, pop node from the stack.
5. Print the node value and change current to right child of current.
6. Go to step 2.

So we go on traversing all left node. as we visit the node. we will put that node into stack.remember need to visit parent after the child and as We will encounter parent first when start from root. it's case for LIFO :) and hence the stack). Once we reach NULL node. we will take the node at the top of the stack. last node which we visited. Print it.
Check if there is right child to that node. If yes. move right child to stack and again start traversing left child node and put them on to stack. Once we have traversed all node. our stack will be empty.

Non recursive postorder traversal algorithm

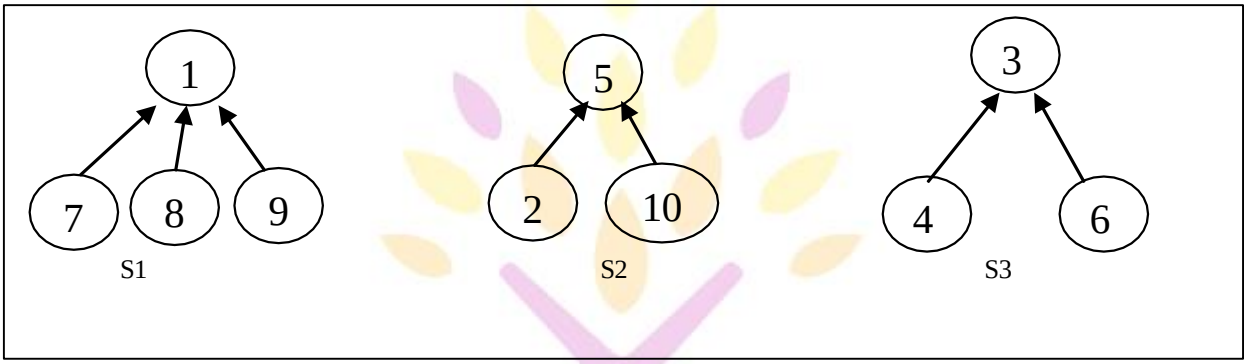
Left node. right node and last parent node.
1.1 Create an empty stack
1.1 Do Following while root is not NULL
a) Push root's right child and then root to stack.

b) Set root as root's left child.
1.2 Pop an item from stack and set it as root.
a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.
Ia) Else print root's data and set root as NULL.
1.3 Repeat steps 2.1 and 2.2 while stack is not empty.

Disjoint Sets: If S_i and S_j, $i \neq j$ are two sets, then there is no element that is in both S_i and S_j.
For example: n=10 elements can be partitioned into three disjoint sets,

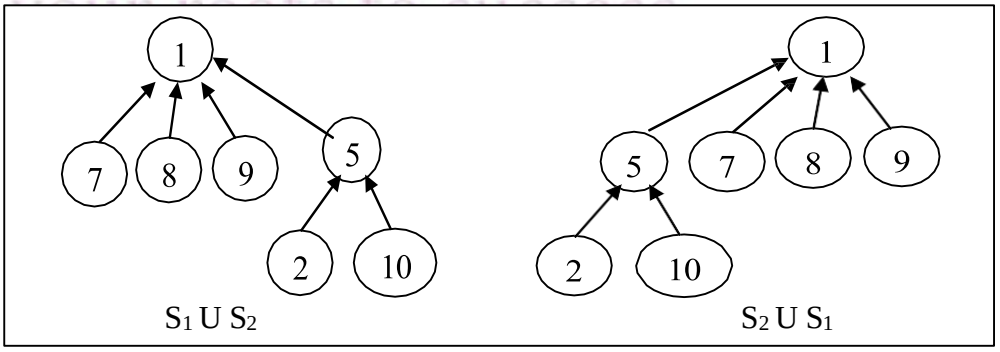
S_1= {1, 7, 8, 9}
S_2= {2, 5, 10}
S_3= {3, 4, 6}

Tree representation of sets:



Disjoint set Operations:
- Disjoint set Union
- Find(i)

Disjoint set Union: Means Combination of two disjoint sets elements. Form above example $S_1 \cup S_2 = \{1,7,8,9,5,2,10\}$
For $S_1 \cup S_2$ tree representation, simply make one of the tree is a subtree of the other.

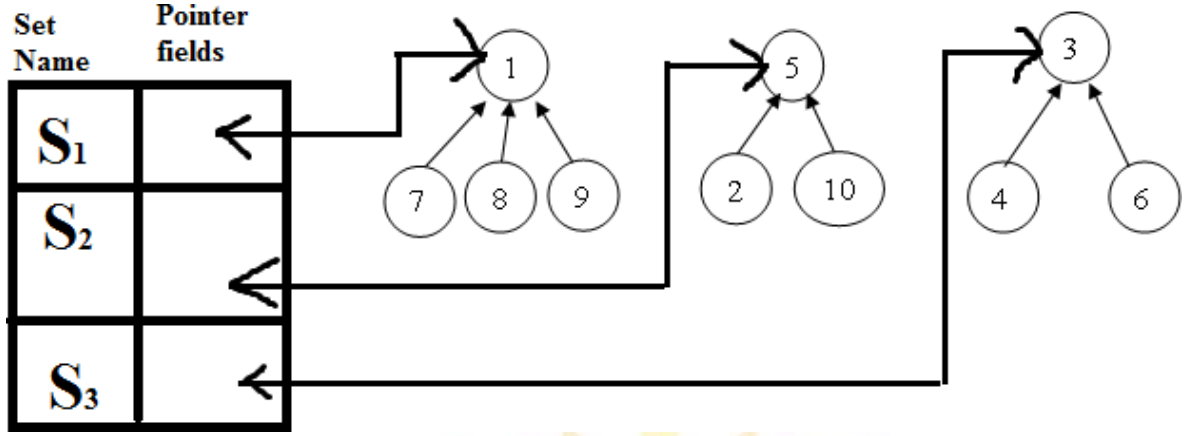


Find: Given element i, find the set containing i.
Form above example:
Find(4)→S_3

Find(1)→S_1
Find(10)→S_2

Data representation of sets:

Tress can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set.



For presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them.
For example: if we determine that element 'i' is in a tree with root 'j' has a pointer to entry 'k' in the set name table, then the set name is just **name[k]**

For unite (**adding or combine**) to a particular set we use FindPointer function.
Example: If you wish to unite to S_i and S_j then we wish to unite the tree with roots
FindPointer (S_i) and FindPointer (S_j)
FindPointer→ is a function that takes a set name and determines the root of the tree that represents it.
For determining operations:
Find(i)→ 1^{St} determine the root of the tree and find its pointer to entry in setname table.
Union(i, j)→ Means union of two trees whose roots are i and j.

If set contains numbers 1 through n, we represents tree node
P[1:n].
n→Maximum number of elements.

Each node represent in array

i	1	2	3	4	5	6	7	8	9	10
P	-1	5	-1	3	-1	3	1	1	1	5

Find(i) by following the indices, starting at i until we reach a node with parent value -1.
Example: Find(6) start at 6 and then moves to 6's parent. Since P[3] is negative, we reached the root.

Algorithm for finding Union(i, j):	Algorithm for find(i)

Algorithm Simple union(i, j)
{
P[i]:=j; // Accomplishes the union
}
	Algorithm SimpleFind(i)
	{
	While(P[i]≥0) do i:=P[i];
	return i;
	}

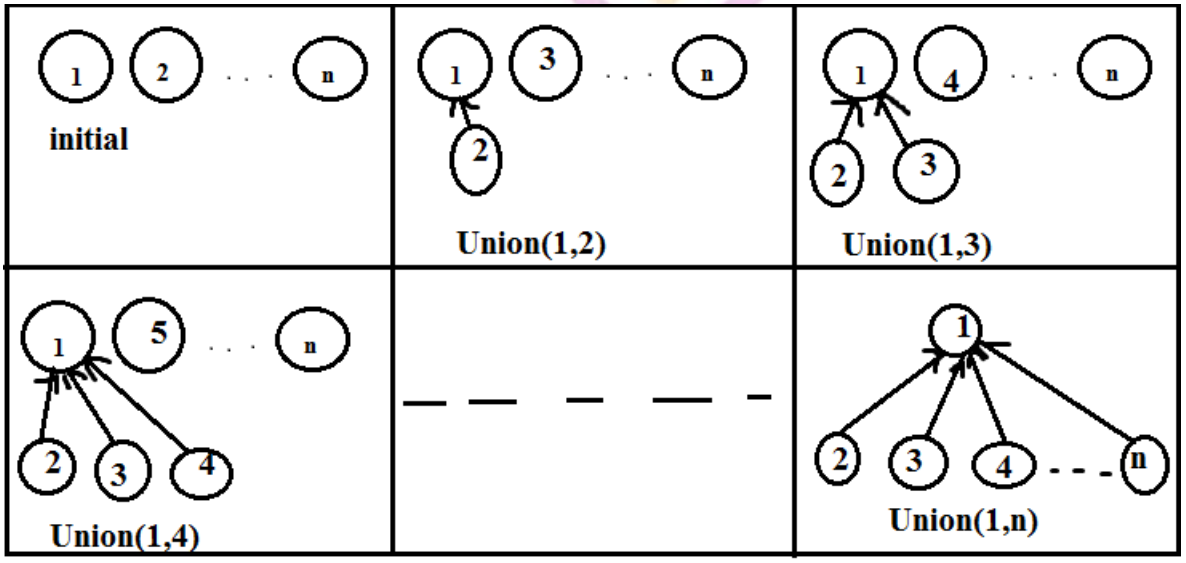
If n numbers of roots are there then the above algorithms are not useful for union and find.
For union of n trees→ Union(1,2), Union(2,3), Union(3,4),…..Union(n-1,n).
For Find i in n trees→ Find(1), Find(2),….Find(n).

Time taken for the union (simple union) is → O(1) (constant).
For the n-1 unions→ O(n).

Time taken for the find for an element at level i of a tree is → O(i).
For n finds → $O(n^2)$.

To improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. For this we use a weighting rule for union(i, j)

Weighting rule for Union(i, j):
If the number of nodes in the tree with root 'i' is less than the tree with root 'j', then make 'j' the parent of 'i'; otherwise make 'i' the parent of 'j'.



Tree obtained using the weighting rule

Algorithm for weightedUnion(i, j)

```
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j, i≠j
// The weighting 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 or equal nodes.
P[j] := i;
P[i] := temp;
}
}
```

For implementing the weighting rule, we need to know how many nodes there are in every tree.
For this we maintain a count field in the root of every tree.
i→ root node
count[i]→ number of nodes in the tree.
Time required for this above algorithm is O(1) + time for remaining unchanged is determined by using **Lemma.**

Lemma:- Let T be a tree with **m** nodes created as a result of a sequence of unions each performed using WeightedUnion. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.

NRCM
your roots to success.

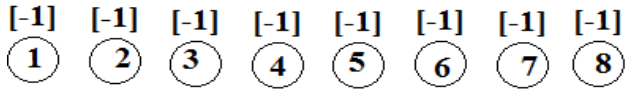
Collapsing rule: If 'j' is a node on the path from 'i' to its root and p[i]≠root[i], then set p[j] to root[i].

Algorithm for Collapsing find.

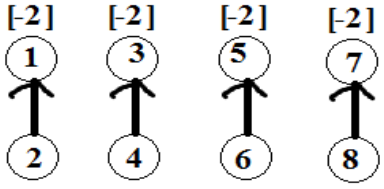
Algorithm CollapsingFind(i)
//Find the root of the tree containing element i.
//collapsing rule to collapse all nodes form i to the root.
{
r;=i;
while(p[r]>0) do r := p[r]; //Find the root.
While(i ≠ r) do // Collapse nodes from i to root r.
{
s:=p[i];
p[i]:=r;
i:=s;
}
return r;
}

Collapsing find algorithm is used to perform find operation on the tree created by WeightedUnion.

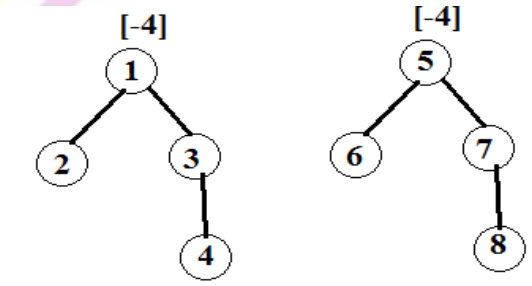
For example: Tree created by using WeightedUnion



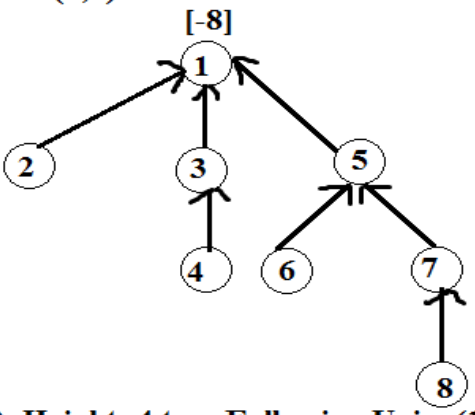
(a) initial height -1 tree



(b) Height -2 trees follwoing Union(1,2),(3,4),(5,6),(7,8)



(c) Height -3 trees following Union (1,3) and (5,7)



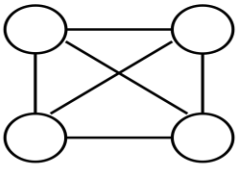
(d) Height -4 tree Following Union(1,5)

Now process the following eight finds: 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 to process all eight finds.

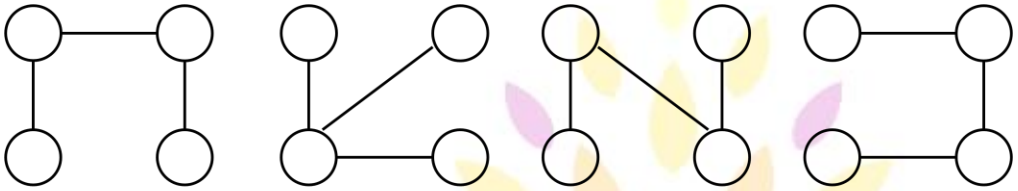
When CollapsingFind is uised the first Find(8) requires going up three links and then resetting two links. Total 13 movies requies for process all eight finds.

Spanning Tree:-

Let G=(V<E) be an undirected connected graph. A sub graph t=(V,E^1) of G is a spanning tree of G iff t is a tree.



A connected, undirected graph



Four of the spanning trees of the graph

Spanning Trees have many applications.
Example:-

It can be used to obtain an independent set of circuit equations for an electric network.
Any connected graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edges are trees. If nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is n-1.
There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

→Prim's Algorithm
→Kruskal's Algorithm

NRCM
your roots to success.

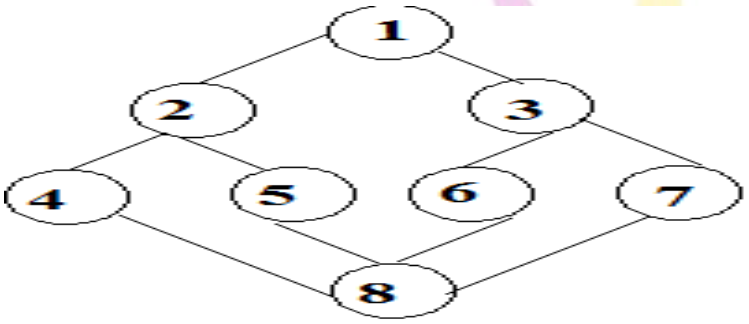
Prim's Algorithm: Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.

Kruskal's Algorithm: Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle.

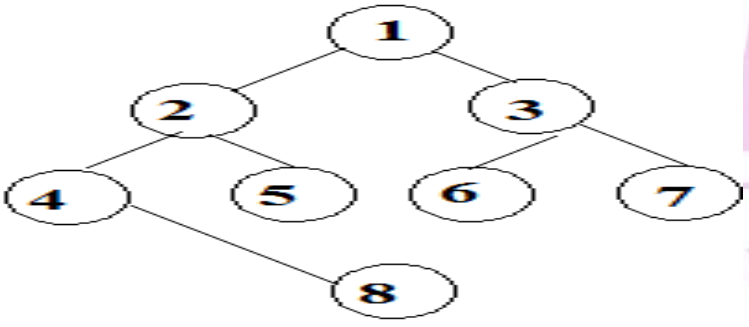
Connected Component:
Connected component of a graph can be obtained by using BFST (Breadth first search and traversal) and DFST (Dept first search and traversal). It is also called the spanning tree.

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.



Undirected Graph G



BFS Spanning tree

This is also called connected graph or spanning tree.
Spanning trees obtained using BFS then it called breadth first spanning trees.

Algorithm for BFS to convert undirected graph G to Connected component or spanning tree.

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)
}
```

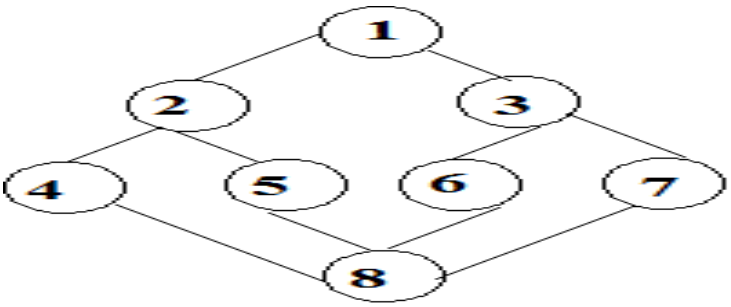
Maximum Time complexity and space complexity of G(n,e), nodes are in adjacency list.
$T(n, e)=\theta(n+e)$
$S(n, e)=\theta(n)$

If nodes are in adjacency matrix then
$T(n, e)=\theta(n^2)$
$S(n, e)=\theta(n)$

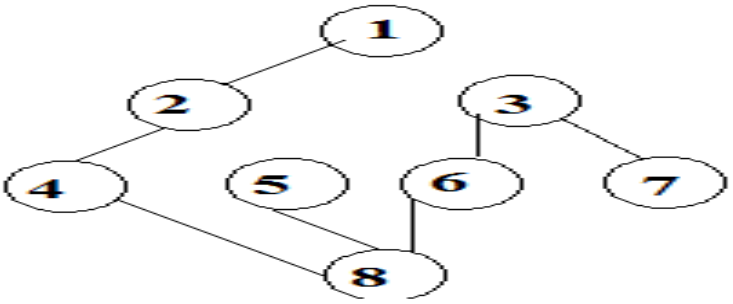
DFST(Dept first search and traversal).:

- 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.

NRCM
your roots to success.



Undirected Graph G



DFS(1) Spanning tree

Algorithm for DFS to convert undirected graph G to Connected component or spanning tree.

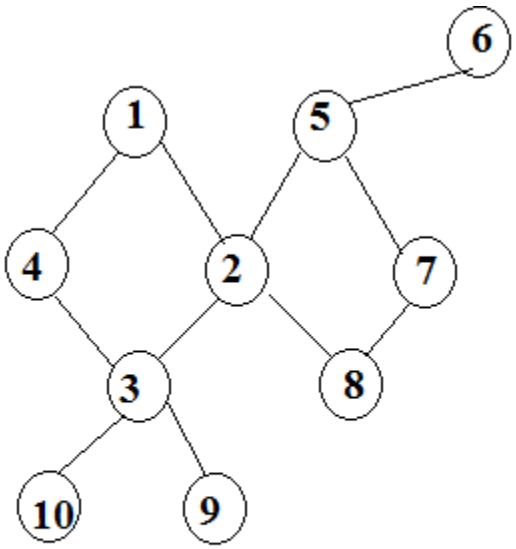
```
Algorithm dFS(v)
// a Dfs of G is begin at vertex v
// initially an array visited[] is set to zero.
//this algorithm visits all vertices reachable from v.
// the graph G, and array visited[] are global
{
Visited[v]:=1;
For each vertex w adjacent from v do
{
If (visited[w]=0) then DFS(w);
{
Add w 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
$T(n, e)=\theta(n^2)$
$S(n, e)=\theta(n)$

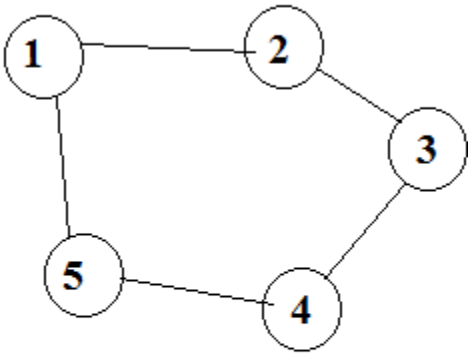
Bi-connected Components:

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.



Graph G1

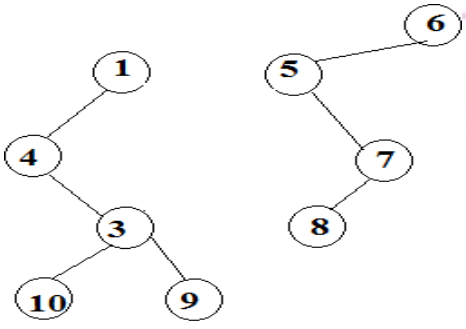
Not a biconnected graph



Graph Gb
Biconnected Graph

The presence of articulation points in a connected graph can be an undesirable(un wanted) feature in many cases.
For example
if G1→Communication network with
Vertex → communication stations.
Edges→ 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

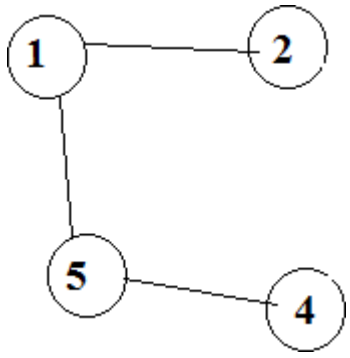


After deleting vertex (2) (Here 2 is articulation point)

If the graph is bi-connected graph (means no articulation point) then if any station i fails, we can still communicate between every two stations not including station i.

NRCM
roots to success.

From Graph Gb



There is an efficient algorithm to test whether a connected graph is biconnected. If the case of graphs 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.

NRCM
your roots to success.

NRCM
your roots to success.