UNIT-2

3/2/15

UNIT - 2

SEARCHING AND TRAVERSING TECHNIQUES

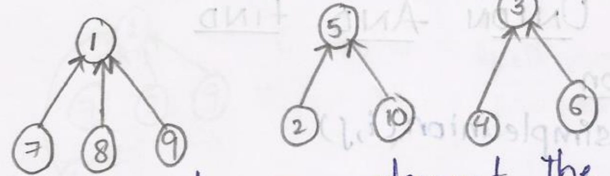
→ **DISJOINT SET OPERATION :-**

For $n=10$ the elements can be partitioned into 3 disjoint sets. $S_1 = \{1,7,8,9\}$

$S_2 = \{2,5,10\}$

$S_3 = \{3,4,6\}$

Each set is represented as a tree where each set is link the nodes from children to the parent.

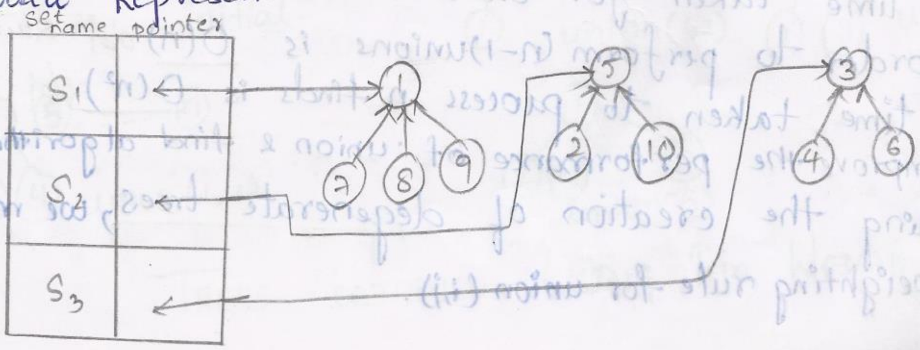


Root node can be any element. The operations to be performed on this sets are i) Disjoint Set Union :- if $S(i)$ and $S(j)$ are two disjoint set then their union $S_i \cup S_j$ = all elements x such that x is in S_i or S_j. i.e., $S_1 \cup S_2 = \{1,2,5,7,8,9,10\}$

ii) find (i) : Given the element i, find set containing i i.e., determines the root of tree containing element i. The determination of i is done until we reach a node with parent value -1.

iii) Data Representation And Array Representation of Sets:

a) Data Representation



b) Array Representation

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

In data lepeesentation the sets are represented in form of set table which contains two columns set name and the pointer which points to the tree representations of sets

In array representation the set elements are numbered from 1 to n i,e., P[1:n] where n is max. of n elements. The ith element of this allay represents tree node that contains element i.

2M

→* ALGORITHM FOR UNION AND FIND

i) Algorithm for union

```
Algorithm   simpleunion(i,j)
{
        P[i]: = j;
}
Algorithm simpleFind (i)
{
        while (P[i] >, 0) do
            i: = P[i];
            return i;
}
```

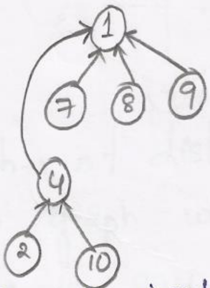
Sequence of union & Find operation

*· union(1,2) , union(2,3), union(3,4), union(4,5)...........u(n-1,n)
find(1) , find(2)........find(n).

- → The time taken for one union is constant.
- → In order to perform (n-1)unions is O(n) obj Ques.
- → The time taken to process n finds is $O(n^2)$ obj Ques.
- → To improve the performance of union & find algorithm by avoiding the creation of degenerate trees, we make use of weighting rule for union (i,j).

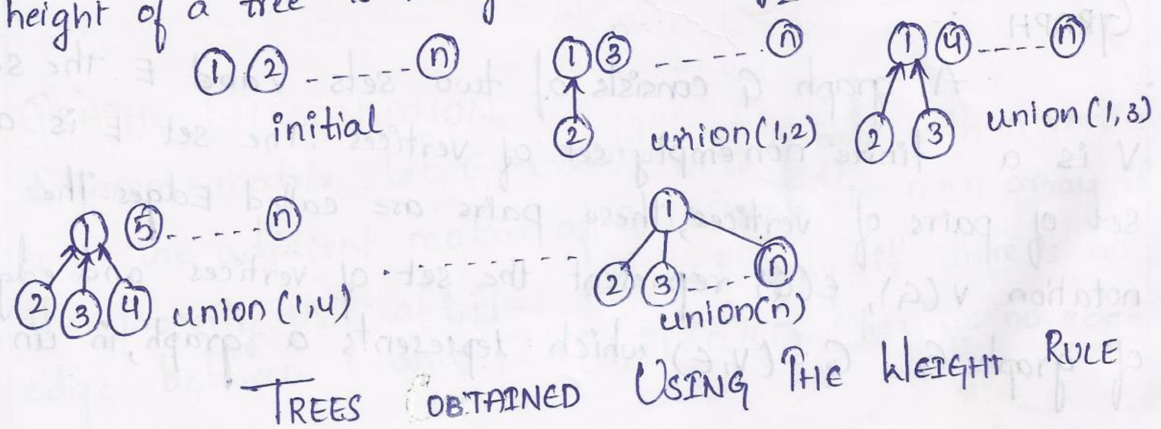
Degenerating trees

iv) **Weighting rule for union(i,j)** :- If the number of nodes in the tree with the root 'i' is less than the number of nodes in the tree with root 'j', make 'j' as parent of 'i' otherwise 'i' as the parent of 'j'.



In order to implement the weighting rule, wee need to know how many nodes are three in each tree. We use a field called count in order to find the number of nodes in that tree.

Note : let T be a tree with m nodes created as a result of sequence of unions each performed using weight union. The height of a tree is not greater than $\log_2 m + 1$



TREES OBTAINED USING THE WEIGHT RULE

Algorithm for weighted rule :

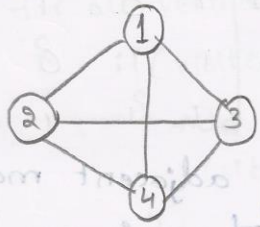
```
Algorithm weightedunion (i, j)
{
    temp: = P[i]+P[j];
    if(P[i]>P[j]) then
    {
        P[i] = j;
        P[j] = temp;
    }
    else {
        P[j]: = i;
        P[i]= temp;
    }
}
```

v) <u>Collapsing</u> <u>rule</u> :- If j is anode on path from i to its root and P[i] ≠ root[i], then set P[j] to root [i]

```
Algorithm collapsing Find(i)
{
    r: = i;
    while( p[r]> 0)do
      r: =p[r]    // to find root
    while (i≠r) do   // collapse nodes from i to root r.
    {
        S: =P[i]
        P[i]: = r;
        i: = s;
    }
    return r;
}
```

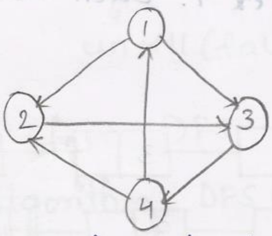
GRAPH :-

A graph G consists of two sets V and E. the set V is a finite nonempty set of vertices. The set E is a set of pairs of vertices, these pairs are called Edges. The notation V(G), E(G) represent the set of vertices and edges of graph G. G = (V, E) which repesents a graph, in undirected graph the pair of vertices represents any edge in an unorder.



(1,2), (1,3) (1,4)
(2,1) (2,3) (2,4)
(3,1) (3,2) (3,4)
(4,1) (4,2), (4,3)

In a directed graph, each edge is represented by directed pair.

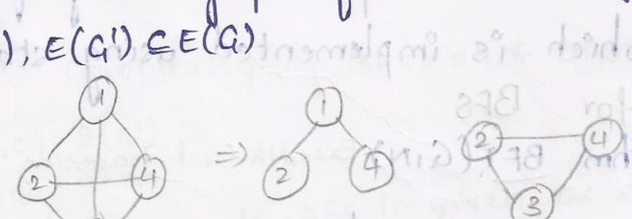


The number of distinct unordered pairs with P(U,V) in addition U ≠ V in graph with n vertices is $n(n+1)/2$ i,e., max. no. of edges in any n vertex, of undirected graph.

SUB GRAPH :-

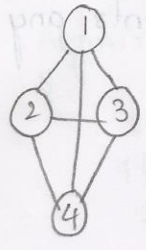
A subgraph of G is a su graph G' such that $V(G') \subseteq V(G)$, $E(G') \subseteq E(G)$

Eg:



GRAPH REPRESENTATION

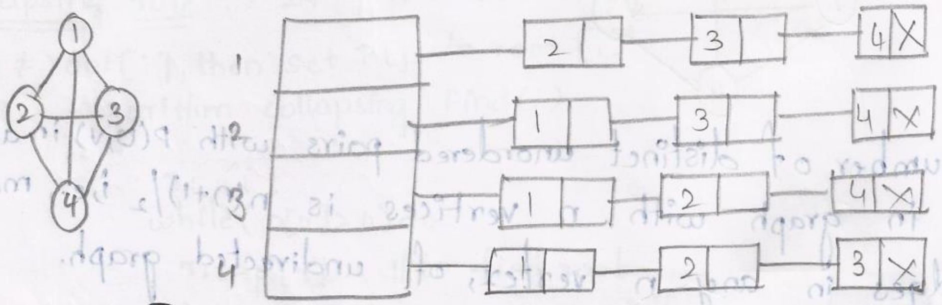
i) Adjacent matrix : let $G = (V, E)$ be a graph with n vertices $n \geq 1$ the adjacent matrix of G is a 2-D $n \times n$ array if the property that $a[i,j] = 1$ if and only if there is an edge between i and j. $a[i,j] = 0$ if there is no edge.



Adjacent Matrix =

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

ii) Adjacent list: The 'n' rows of adjacent matrix are represented as n linked list. There is one list for each vertex. There is a node in list i represent the vertexes that are adjacent from vertex i. Each nodes have atleast a field vertex and link.



GRAPH TRAVERSALS :

There are two types of traversing graph.

i) BFS - which is implemented using queue data structure

ii) DFS — which is implemented using stack data structure.

→ Algorithm for BFS

```
Algorithm BFT(G,N) Breadth First Traversing
{
    for i: = 1 to n do
        visited [i]=0;
    for i: = 1 to n do
        if (visited [i] = 0)
        then
            BFS(i)
}
Algorithm BFS (V) Breadth first Search
{
    u: = v;
```

```
visited [v]: =1;
repeat
{
  for all vertices w adjacent from u do
  {   if(visited [w]= o) then
      {  Add w to q;
         visited [w] :=1;
      }
  }   if q is empty then return;
      Delete u from q;
  }
  untill (false);
}
```

→ Algorithm for DFS

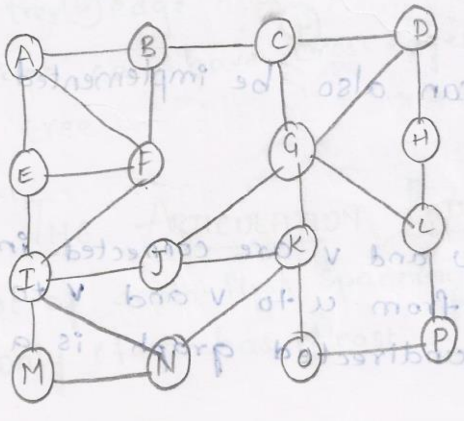
```
Algorithm DFS (v)
{
  visited [v]: =1;
  for each vertex w adjacent from v do
  {
    if (visited [w] = 0)
    then
    DFS (w); same as above
  }
}
```

The time complexity of BFS and DFS is given by

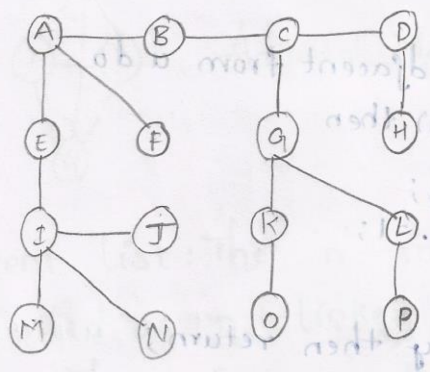
$T(n,e) = \theta(n^2)$

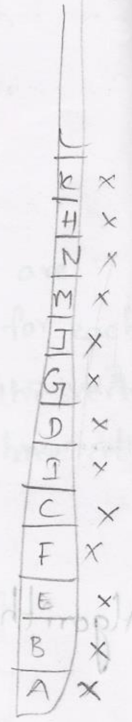
$S(n,e) = \theta(n)$

The space of BFS and DFS is given as ~~stable~~ stated as above.

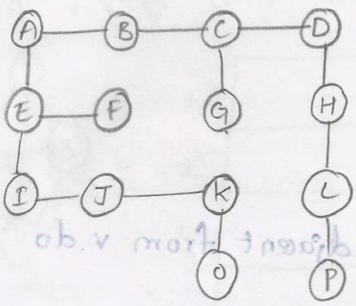


BFS





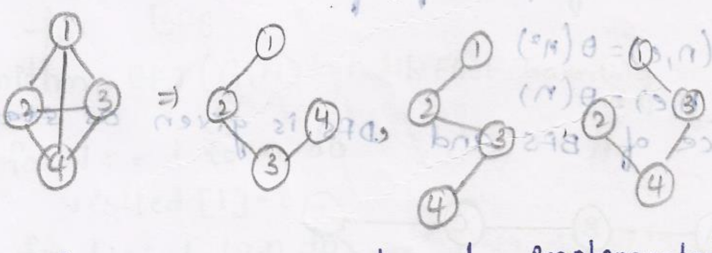
DFS



Spanning Tree :

For a graph $G = (V, E)$ is said to be as a spanning if it contains all vertices with $n-1$ edges. It should not contain a loops or cycles. It must not be a unique.

Ex:



Spanning trees can also be implemented using DFA and BFA.

Connected Graph :

Two vertices U and V are connected in undirected graph if there is a path from U to V and V to U. A connected component of undirected graph is a maximal connected sub graphs.

→ **STRONGLY CONNECTED GRAPH :**

A directed graph G is said to be strongly connected if for every pair of distinct vertices u and v in G there is a directly path from u to v and v to u. A strongly connected component is a maximal subgraph that is strongly connected.

→ **BI CONNECTED COMPONENTS :**

*. A vertex V(G) is an articulation point if and only if the deletion of B, together with deletion of all edges incident to v leaves behind a graph that has atleast two connected components.

*. A biconnected graph is a connected graph that has no articulation points.

*. A biconnected component of connected graph G is a maximal biconnected subgraph of G i,e., G contains no other subgraphs i,e., both connected & biconnected.

Note : i) Two biconnected components of same graph can have atmost one vertex in common.

ii) No edge can be into 2 or more biconnected components.

iii) The biconnected components of a connected, undirected graph can be found by using depth - first spanning tree. of G.

iv) A non tree edge u,v is an back edge with respect to a spanning tree T.

v) A non tree edge that is not back edge is called Cross Edge.

vi) No graph can have cross edges with respect to any depth-First spanning tree.

→ **FINDING THE ARTICULATION POINTS FOR GIVEN GRAPH :**

i) The root of depth-first spanning tree is an articulation point if and only if it has atleast 2 children.

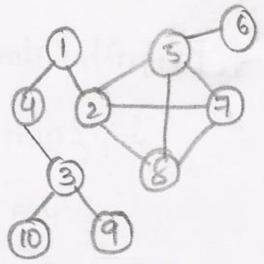
ii) u is an articulation point if and only if, u is either a spanning tree and has 2 or more childrens.

iii) u is not the root and u has a child w such that $low(w) \geq dfn(u)$ where dfn is depth first number.

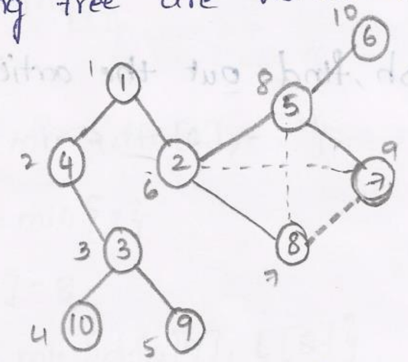
iv) $L(U) = \min\{ dfn[u], \min\{ L[w] \mid w \text{ is child of } u \}, \min\{ dfn[w] \mid (u,w) \text{ is a back edge} \} \}$

ALGORITHM FOR BICONNECTED COMPONENTS :

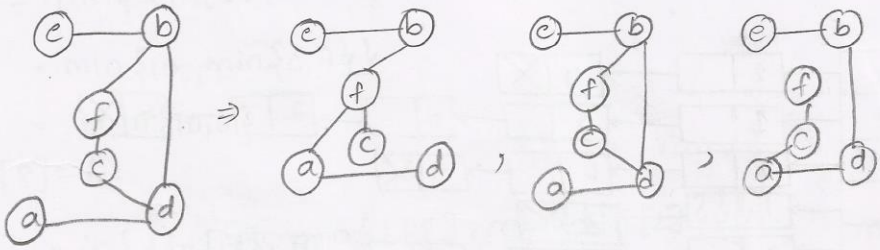
```
Algorithm Bicomp(u,v)
{
    dfn[u] := num;
    L[u] := num;
    num := num+1;
  for each vertex w adjacent from u do
  {
    if((v≠w)and(dfn[w]<dfn[u])) then
        add(u,w) to the top of the stack.
    if(dfn[w] = 0) then
    {
        if(L[w] ≥ dfn[u]) then
        {
            write("new bicomponent");
            repeat
            {
                Delete an edge from top of stacks;
                let this edge be(x,y);
                write(x,y);
            }
            until(((x,y)=(u,w)) or ((x,y)=(w,u)));
        }
        Bicomp(w,u); //ant w is unvisited
        L[u] := min[L[u],L[w]);
    }
    else if(w≠v)then
        L[u] := min[L[U], dfn[w]));
  }
}
```



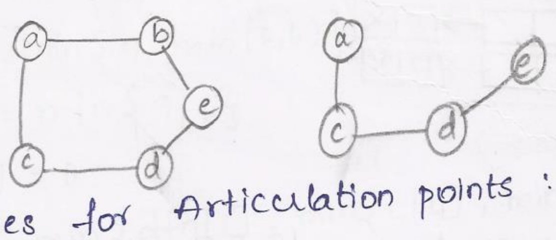
In order to find L[u] easily the vertices of depth-first spanning tree are visited in post order.



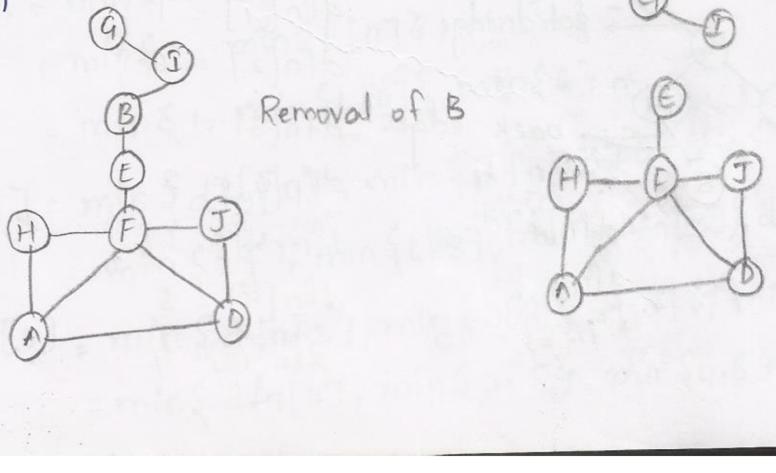
Example for connected components :



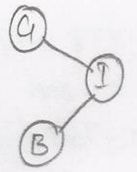
Examples for Biconnected Components :

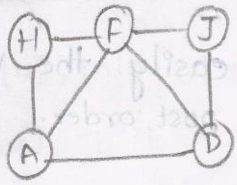


Examples for Articulation points :

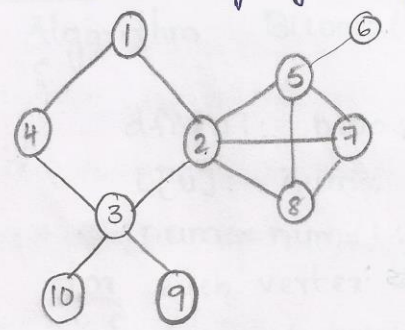


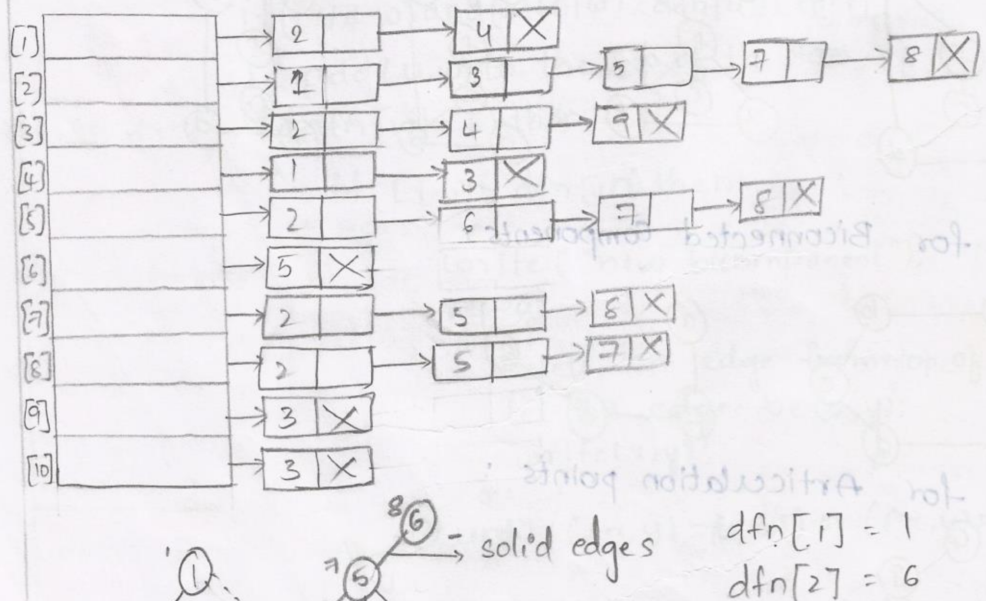
removal of E ⇒

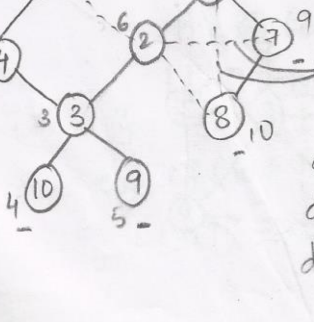




For the following given graph, find out the articulation points.







→ solid edges

- - → back edges

dfn[1] = 1
dfn[2] = 6
dfn[3] = 3
dfn[4] = 2
dfn[5] = 7
dfn[6] = 8
dfn[7] = 9
dfn[8] = 10
dfn[9] = 5
dfn[10] = 4

$L[10] = \min\{dfn[10], -, -\}$

$= \min\{4\}$

$L[10] = 4$

$L[9] = \min\{dfn[9], -, -\}$

$= \min\{5\}$

$L[9] = 5$

$L[6] = \min\{dfn[6], -, -\}$

$= \min\{8\}$

$L[6] = 8$

$L[7] = \min\{dfn[7], L[8]\}$

$L[8] = \min\{dfn[8], \min\{dfn[2], dfn[5]\}\}$

$= \min\{10, \min\{6, 7\}\}$

$= \min\{10, 6\}$

$L[8] = 6$

$L[7] = \min\{dfn[7], \min\{L[8], \min\{dfn[2]\}\}\}$

$= \min\{9, \min\{6\}, \min\{6\}\}$

$= \min\{9, \min(6, 6)\}$

$= \min\{9, 6\}$

$L[7] = 6$

$L[1] = \min\{dfn[1], \min\{L[4], \min\{dfn[2]\}\}\}$

$= \min\{1, \min\{L[4]\}, \min\{6\}$

$= \min\{1, \min\{L[4]\}, \min\{6\}$

$L[4] = \min\{dfn[4], \min\{dfn[3]\}, \min\{L[3]\}$

$= \min\{2\}, \min\{L[3], \min\{dfn[3]\}, dfn[1]\}$

$L[3] = \min\{dfn[3], \min\{L[9], L[10]\}, \min\{dfn[9], dfn[10]\}$

$= \min\{dfn[3], \min\{4, 5\}, \min\{4, 5\}\}$

$L[3] = \min\{3, \min\{4,5\}\}$

$= \min\{3, 4\}$

$L[3] = 3$

$L[4] = \min\{2, \min\{3\}, \min\{3,4\}\}$

$= \min\{2, \min\{3\}\}$

$= \min\{2, 3\}$ $L[1]$ $\min\{2, \min\{3,1\}\}$

$L[4] = \min\{2, 1\}$

$L[4] = 1$

$L[1] = \min\{1, \min\{1\}, \min\{6\}\}$

$= \min\{1, \min\{1,6\}\}$

$= \min\{1, 1\}$

$L[1] = 1$

$L[2] = \min\{dfn(2), \min\{L[5]\}, \min\{dfn[1]\}\}$

$= \min\{6, \min\{L[5]\}, \min\{1\}\}$

$L[5] = \min\{dfn(5), \min\{L[6], L[7]\}, \min\{dfn[8]\}\}$

$= \min\{7, \min\{6, 8\}, \min\{10\}\}$

$= \min\{7, 6, 10\}$

$L[5] = 6$

$L[2] = \min\{6, \min\{6\}, \min\{1\}\}$

$= \min\{6, 6, 1\}$

$L[2] = 1$

$L[w] \geq dfn[u]$

Vertex 4: $L[3] \geq dfn[4]$

$3 \geq 2$ ✓

Vertex 3: $L[10] \geq dfn[3]$

$4 \geq 3$ ✓

$L[9] \geq dfn[3]$

$5 \geq 3$ ✓

$L[2] \geq dfn[3]$

$1 \geq 3$ ×

Vertex 2

$L[5] \geq dfn[2]$

$6 \geq 6$ ✓

Vertex 5

$L[6] \geq dfn[5]$

$8 \geq 7$

$L[7] \geq dfn[5]$

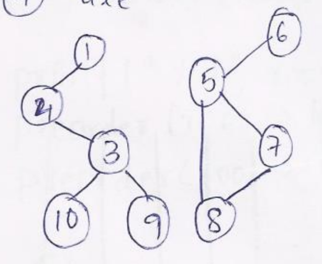
$6 \geq 7$

Vertex 7

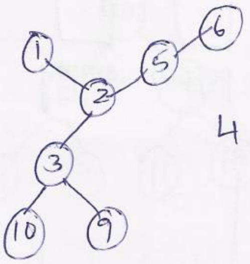
$L[8] \geq dfn[7]$

$6 \geq 9$ ×

2 & 4 are



2 is an articulation point.



4 is not an articulation point.

EFFICIENT NON RECURSIVE BINARY TREE TRAVERSAL :

The binary tree traversals are divided into three types.

* → Inorder

The Recursion algorithm of inorder is given as follows

⟹ void Inorder (Node *root)

{