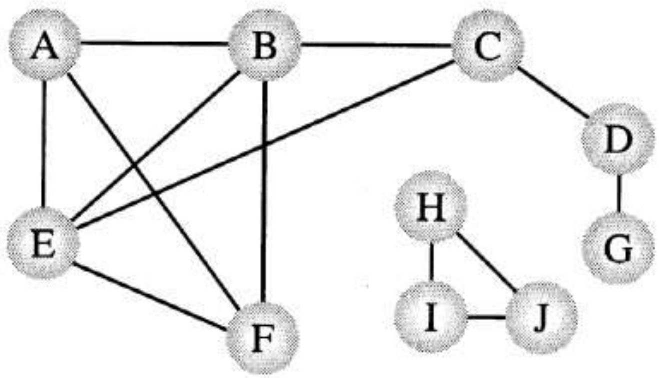
Biconnected Components

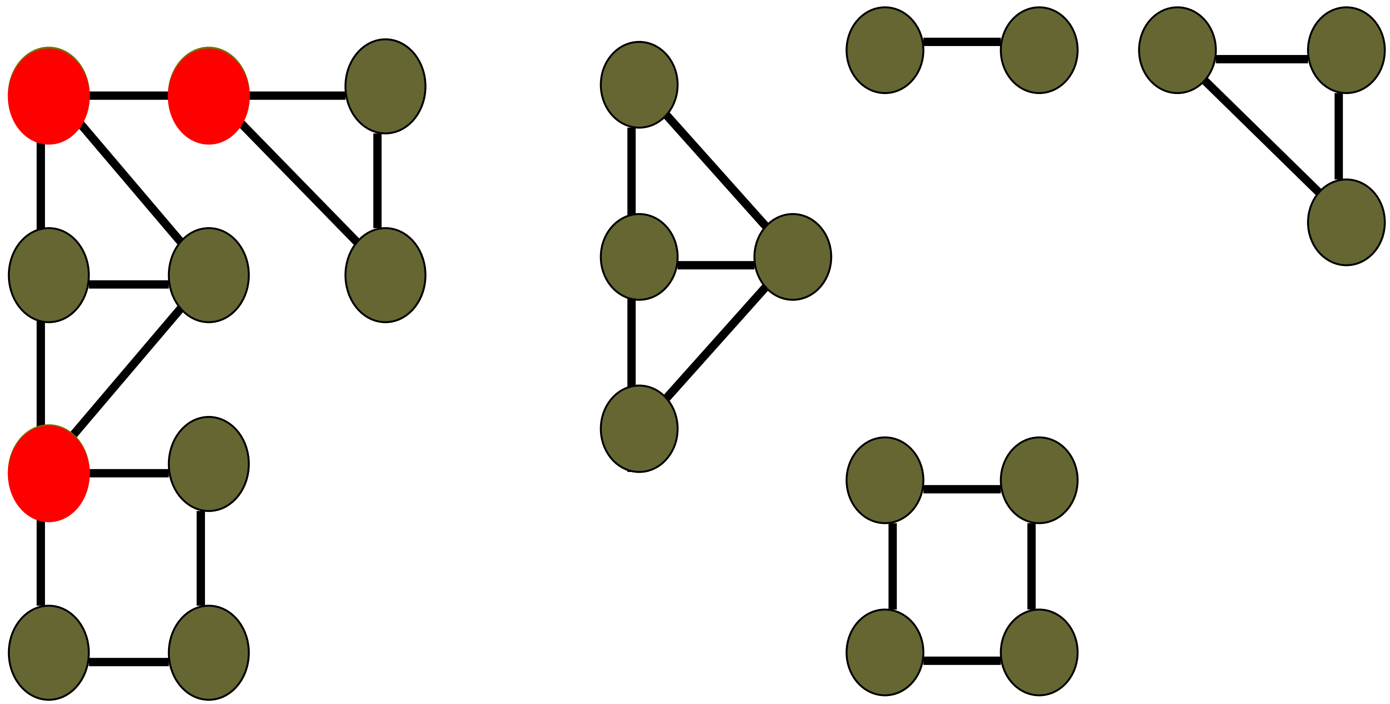
Connectivity/Biconnectivity

A node and all the nodes reachable from it compose a **connected component**. A graph is called **connected** if it has only one connected component.

Since the function **visit**() of DFS visits every node that is reachable and has not already been visited, the DFS can easily be modified to print out the connected components of a graph.

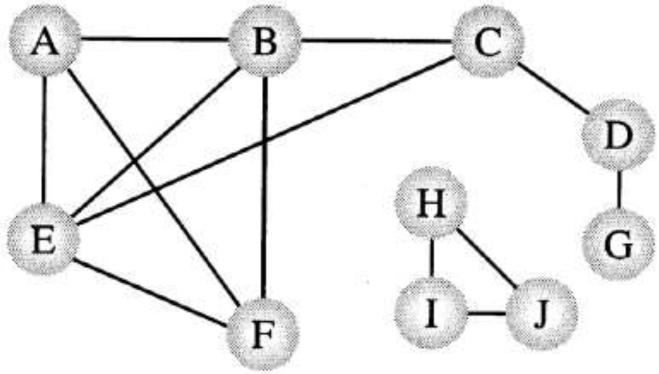


Two connected components



Connectivity/Biconnectivity

In actual uses of graphs, such as networks, we need to establish not only that every node is connected to every other node, but also there are **at least two independent paths between any two nodes**. A maximum set of nodes for which there are two different paths is called **biconnected**.



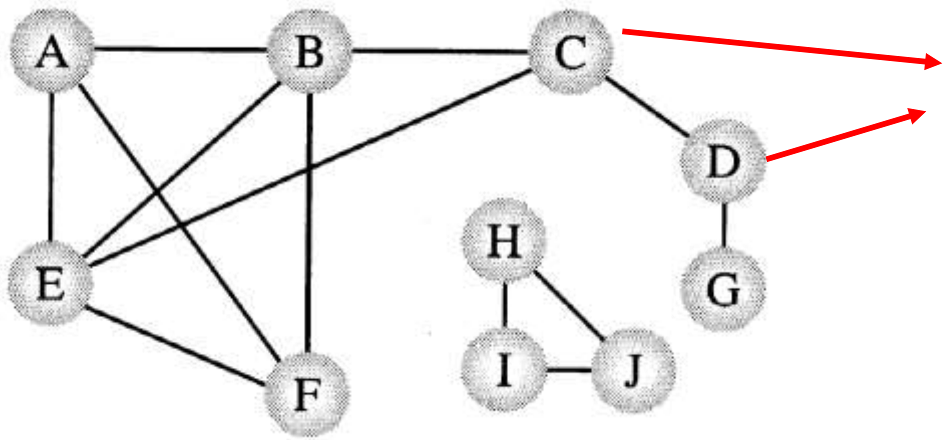
{H,I,J} and {A,B,C,E,F} are biconnected.

Connectivity/Biconnectivity

Another way to define this concept is that there are **no single points of failure**, no nodes that when deleted along with any adjoining arcs, would split the graph into two or more separate connected components. Such a node is called an **articulation point**.

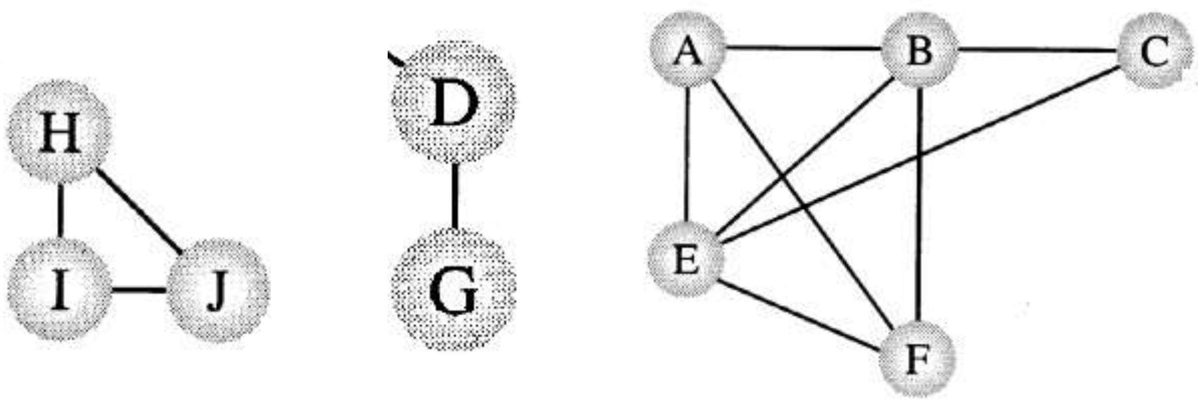
If a graph contains no articulation points, then it is biconnected. If a graph does contain articulation points, then it is useful to split the graph into the pieces where each piece is a maximal biconnected subgraph called a **biconnected component**.

Connectivity/Biconnectivity



Articulation points

Three biconnected components



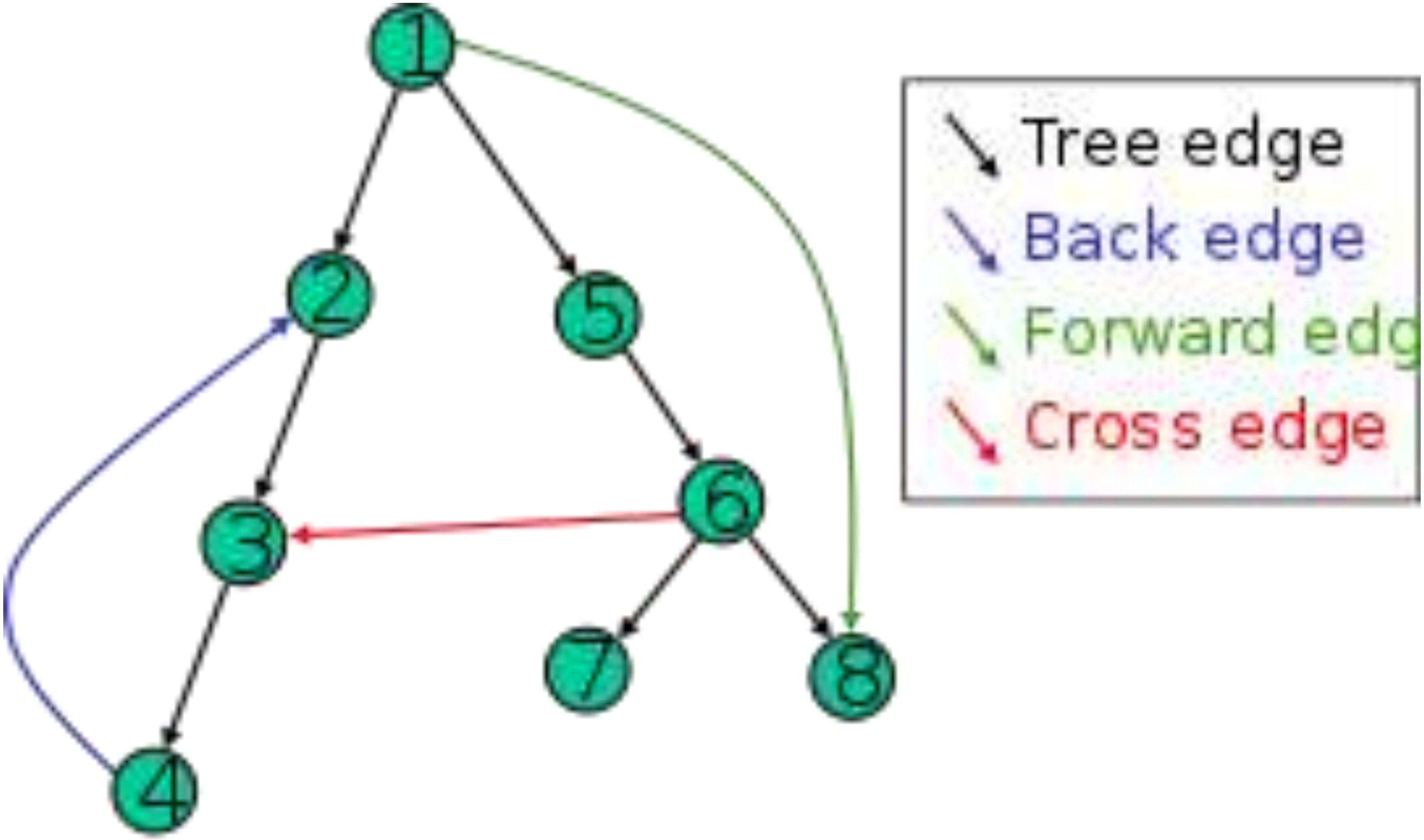
Finding Articulations

- Problem:
  - Given any graph $G = (V, E)$, find all the articulation points.
  - Possible strategy:
    - For all vertices v in V:

      Remove v and its incident edges

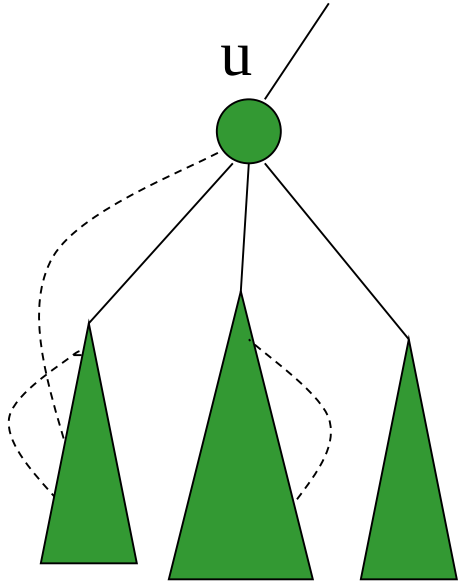
      Test connectivity using a DFS.
    - Execution time: $\Theta(n(n+m))$.
    - Can we do better?

Finding Articulations



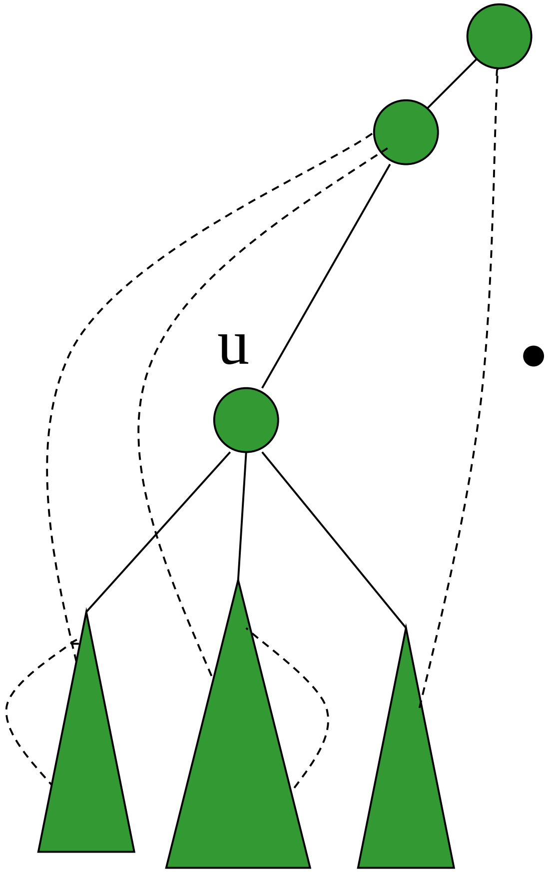
internal vertex u

- Consider an internal vertex u
  - Not a leaf,
  - Assume it is not the root
- Let v1, v2,…, vk denote the children of u
  - Each is the root of a subtree of DFS
  - If for some child, there is no back edge from any node in this subtree going to a proper ancestor of u, then u is an articulation point



Here u is an articulation point

internal vertex u



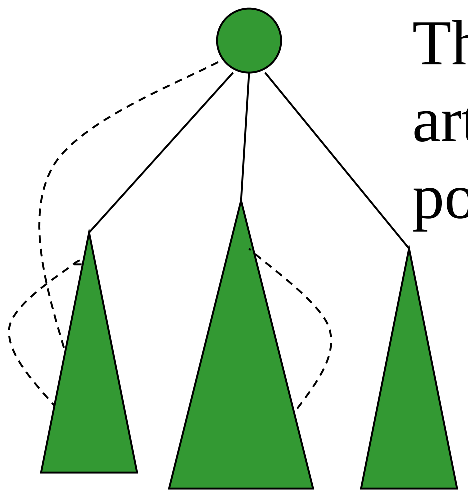
- Here u is not an articulation point
  - A back edge from every subtree of u to proper ancestors of u exists

What if u is a leaf

- A leaf is never an articulation point
- A leaf has no subtrees..

What about the root?

- the root is an articulation point if and only if it has two or more children.
  - Root has no proper ancestor
  - There are no cross edges between its subtrees



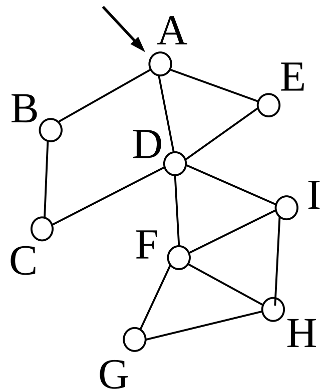
This root is an articulation point

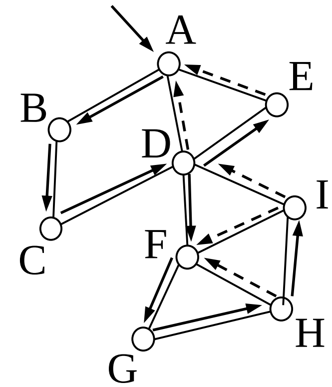
How to find articulation points?

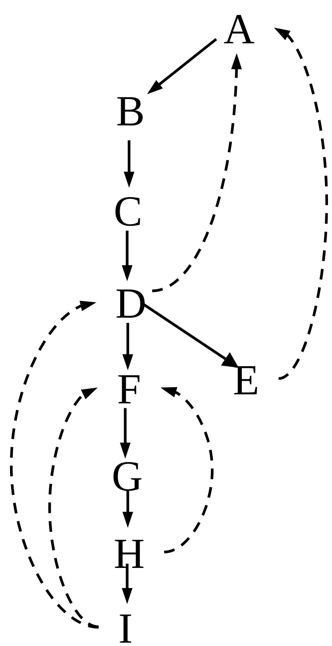
- Keep track of all back edges from each subtree?
  - Too expensive
- Keep track of the back edge that goes highest in the tree (closest to the root)
  - If any back edge goes to an ancestor of u, this one will.
- What is closest to root?
  - Smallest discovery time

Finding Articulation Points

- A DFS tree can be used to discover articulation points in $\Theta(n + m)$ time.

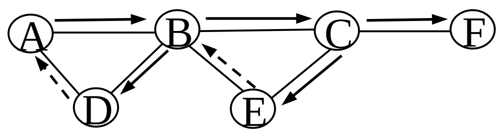




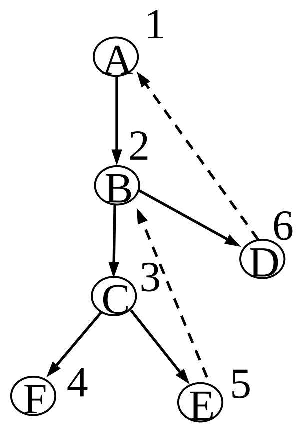


Can you characterize *D* ?

Depth First Search number

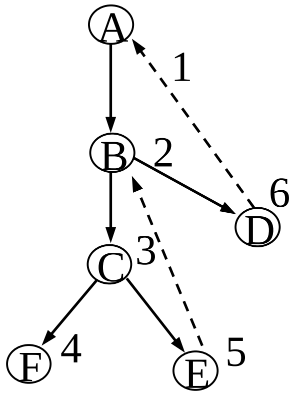


$G = (V, E)$



A	B	C	D	E	F
1	2	3	9	6	4

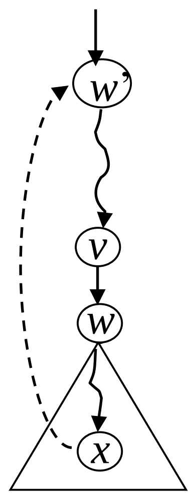
Any relation between Discovery time and articulation point ?



Assume that $(a,b) \Leftrightarrow a \rightarrow b$

Tree edge : (a,b) $a < b$

Back edge : (a,b) $a > b$



If there is a back edge from x to a proper ancestor of v, then v is reachable from x.

Finding Articulation Points

- A DFS tree can be used to discover articulation points in $\Theta(n + m)$ time.
  - We start with a program that computes a DFS tree labeling the vertices with their discovery times.
  - We also compute a function called low(v) that can be used to characterize each vertex as an articulation or non-articulation point.
  - The root of the DFS tree will be treated as a special case:
    - The root has a d[] value of 1.

Finding Articulation Points

- The root of the DFS tree is an articulation point if and only if it has two or more children.
  - Suppose the root has two or more children.
    - Recall that back edges never link vertices between two different subtrees.
    - So, the subtrees are only linked through the root vertex and its removal will cause two or more connected components (i.e. the root is an articulation point).
  - Suppose the root is an articulation point.
    - This means that its removal would produce two or more connected components each previously connected to this root vertex.
    - So, the root has two or more children.

Definition of low(v)

– Definition. The value of *low*(*v*) is the discovery time of the vertex closest to the root and reachable from *v* by following zero or more tree edges downward, and then at most one back edge.

– We can efficiently compute Low by performing a postorder traversal of the depth-first spanning tree.

low[v] = min{

$d[v]$,

lowest d[w] among all back edges (v,w)

lowest low[w] among all tree edges (v,w)

}

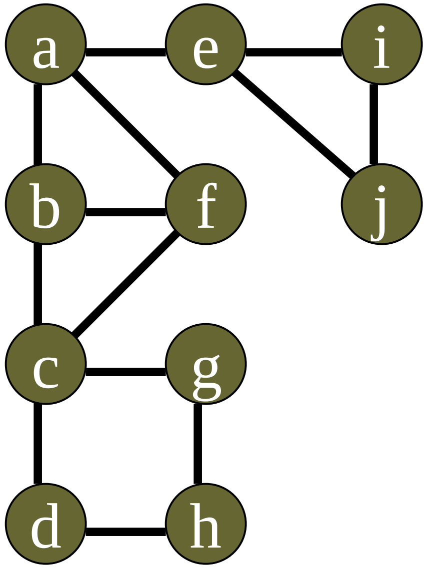
– In English: low(v) < $d[v]$ indicates if there is another way to reach v which is not via its parent

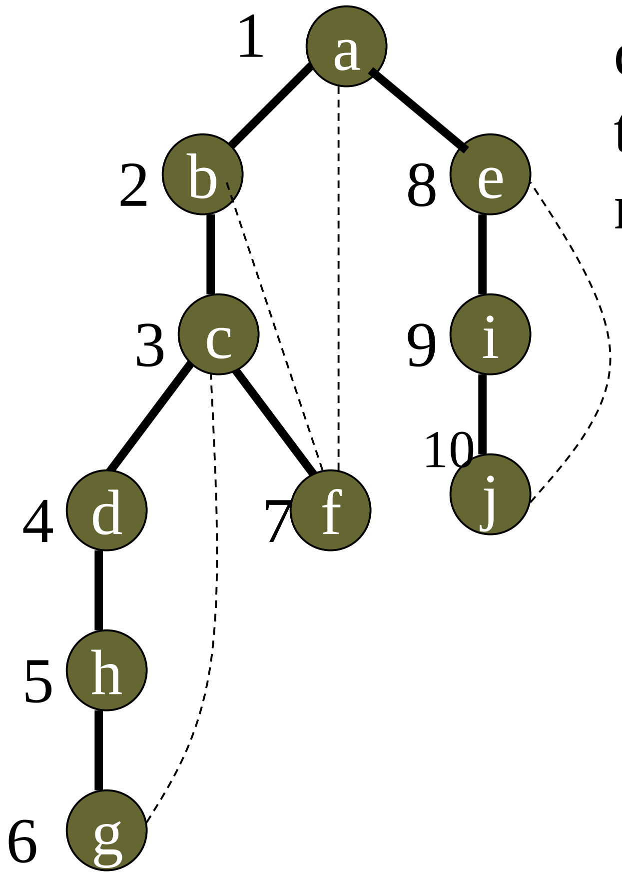
Definition of low(*v*)

- Once Low[u] is computed for all vertices u, we can test whether a nonroot vertex u is an articulation point

- **u is an articulation point iff it has a child v for which Low[v] >= d[u]**

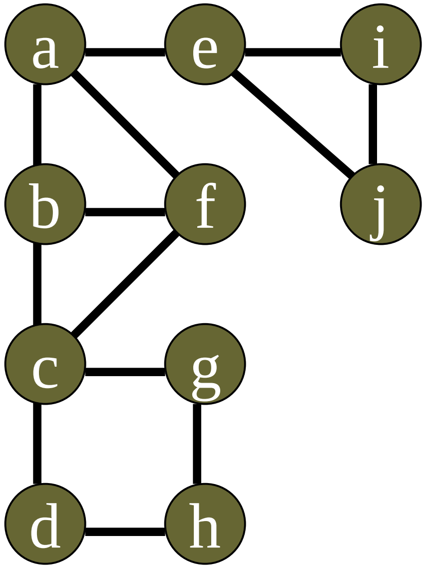
Example

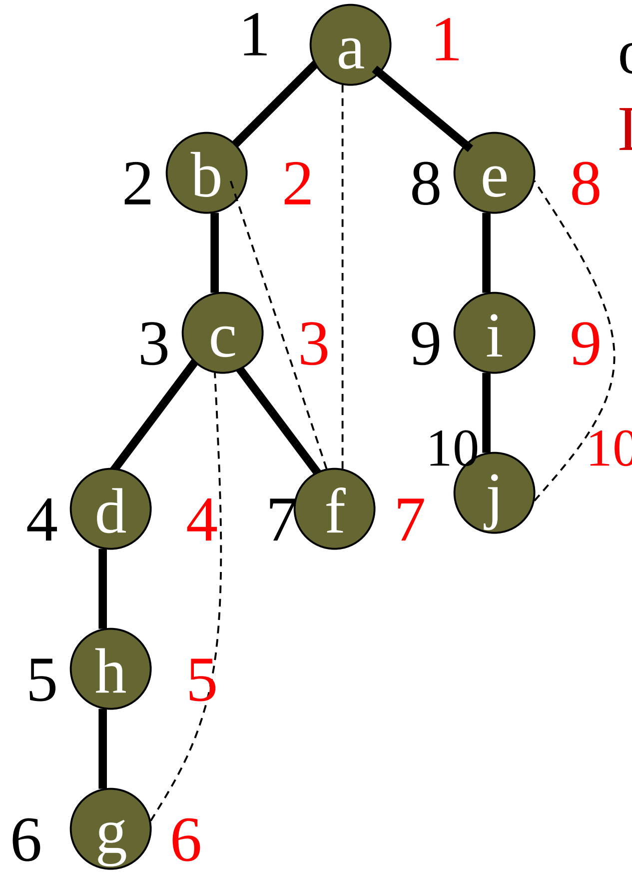




Calculate the discovering time of each node

Example





Initialize Low of each node

Low[u] = d[u]

low[v] = min{

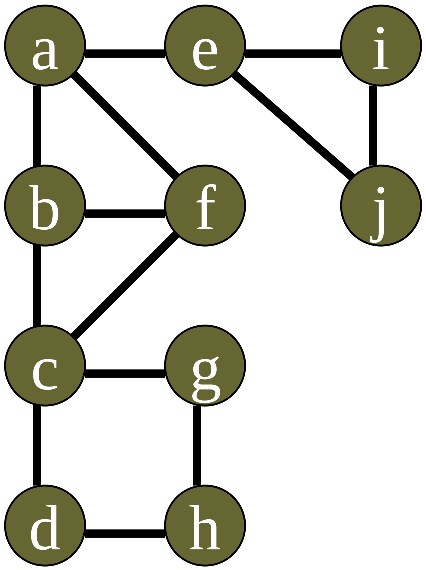
d[*v*],

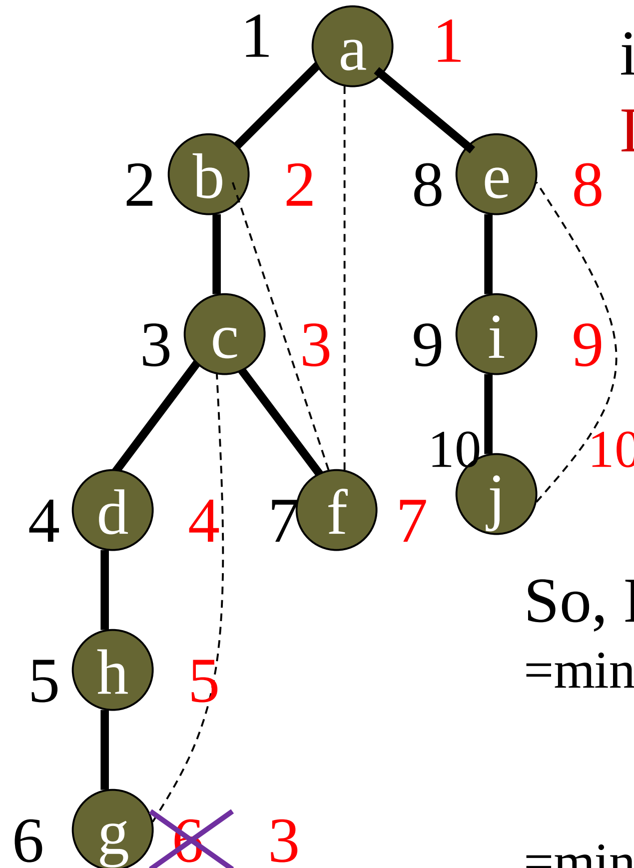
lowest d[w] among all back edges (v,w)

lowest low[w] among all tree edges (v,w)

}

Example



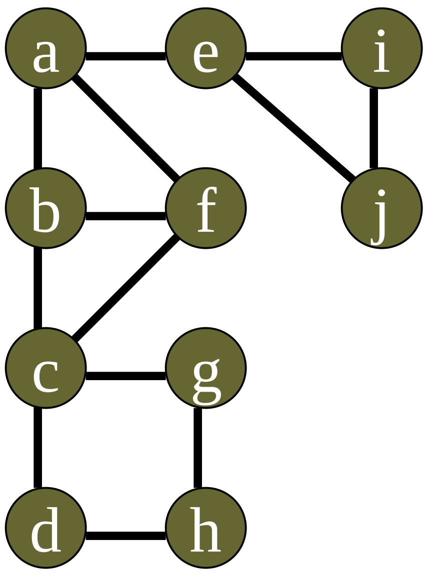


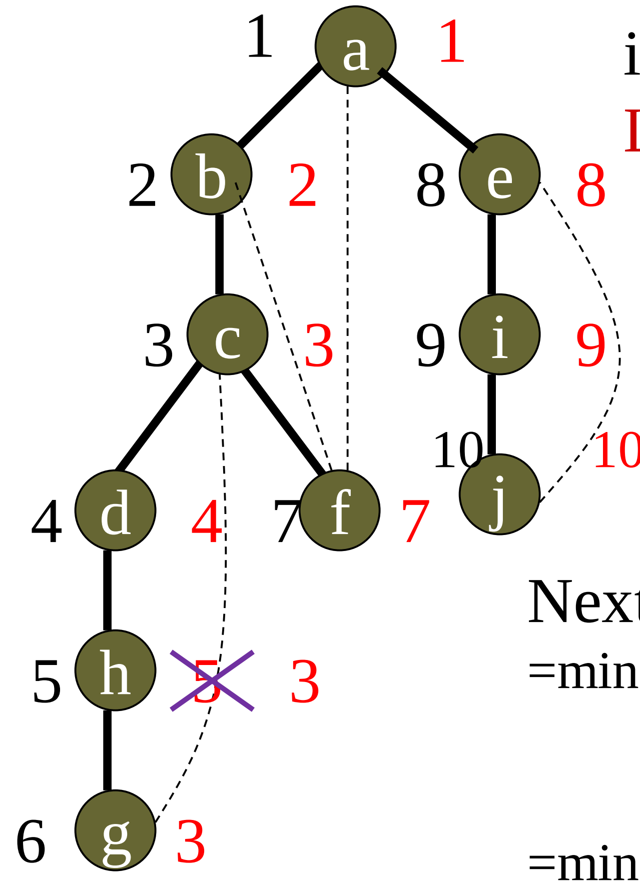
Travers the tree
in post-order
Left, Right, Root

So, Node Low[g]
=min{ d[g],
min all d[w]
min all low [w]}
=min { 6, 3, NA}
= 3

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example



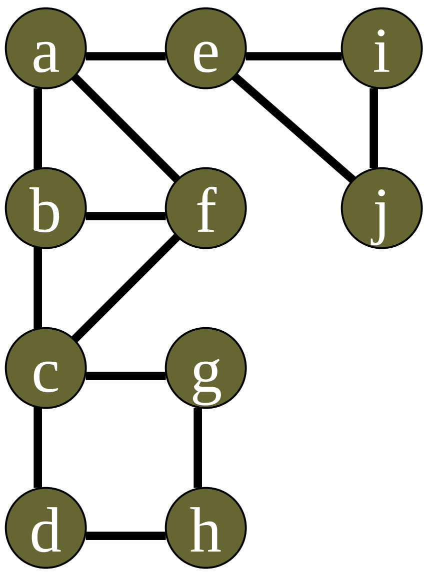


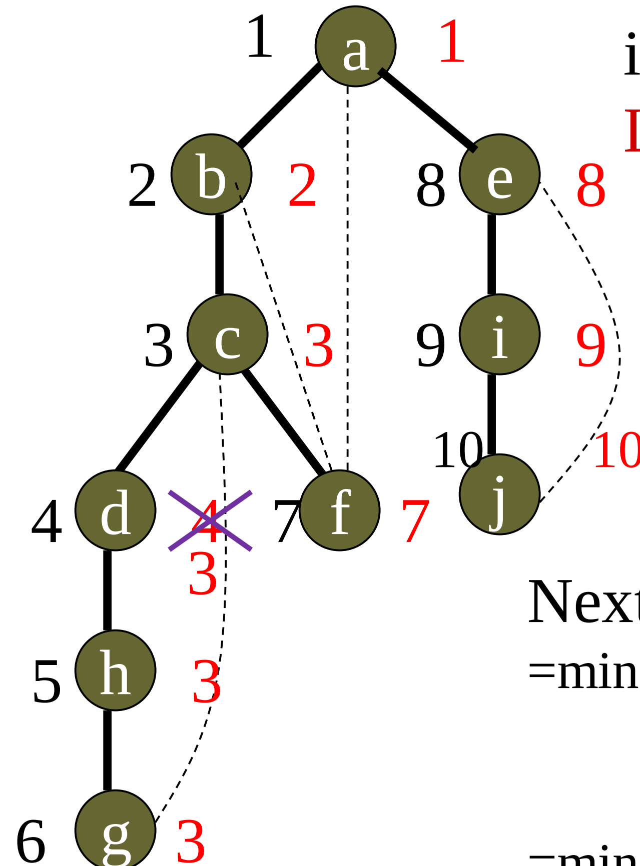
Travers the tree
in post-order
Left, Right, Root

Next, Node Low[h]
=min{ d[h],
min all d[w]
min all low [w]}
=min { 5, 3, 3}
= 3

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example



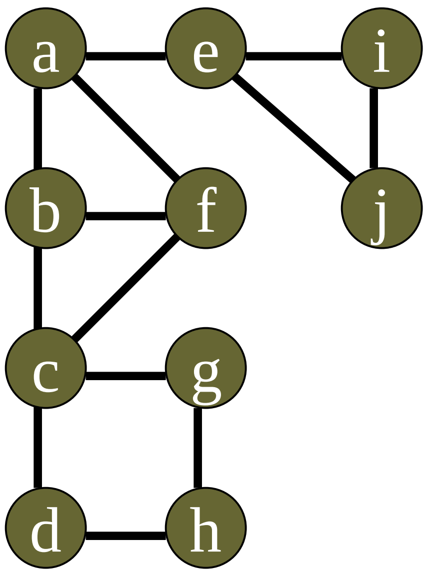


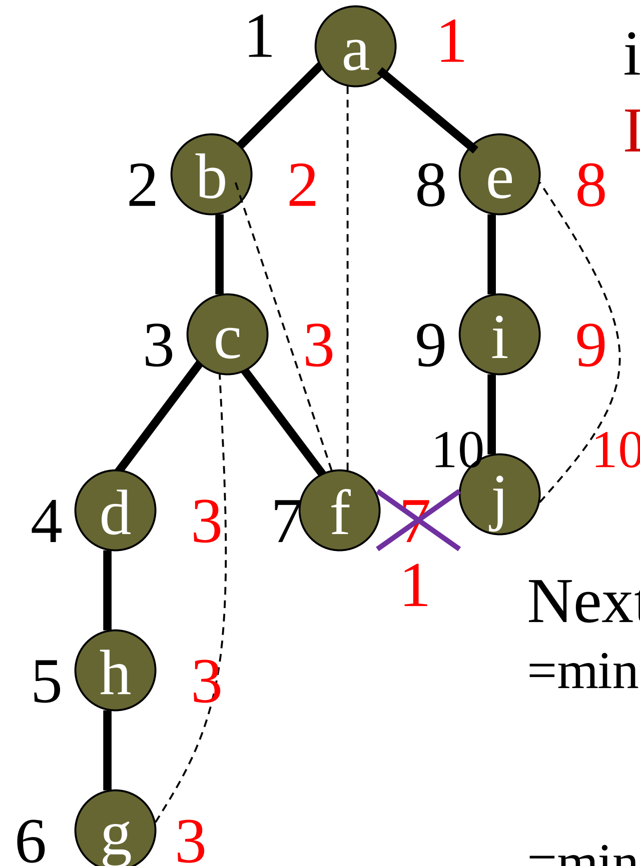
Travers the tree
in post-order
Left, Right, Root

Next, Node Low[d]
=min{ d[d],
min all d[w]
min all low [w]}
=min { 4, 3, 3}
= 3

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example



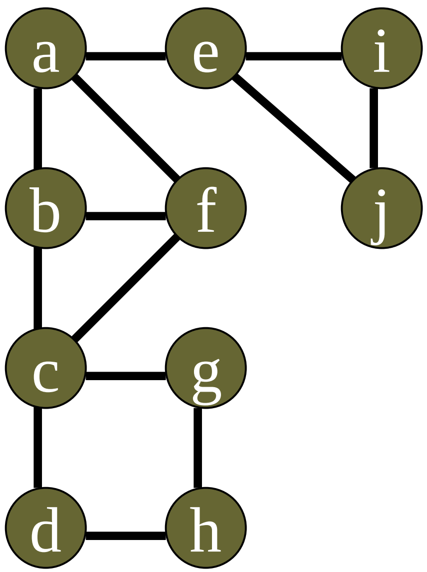


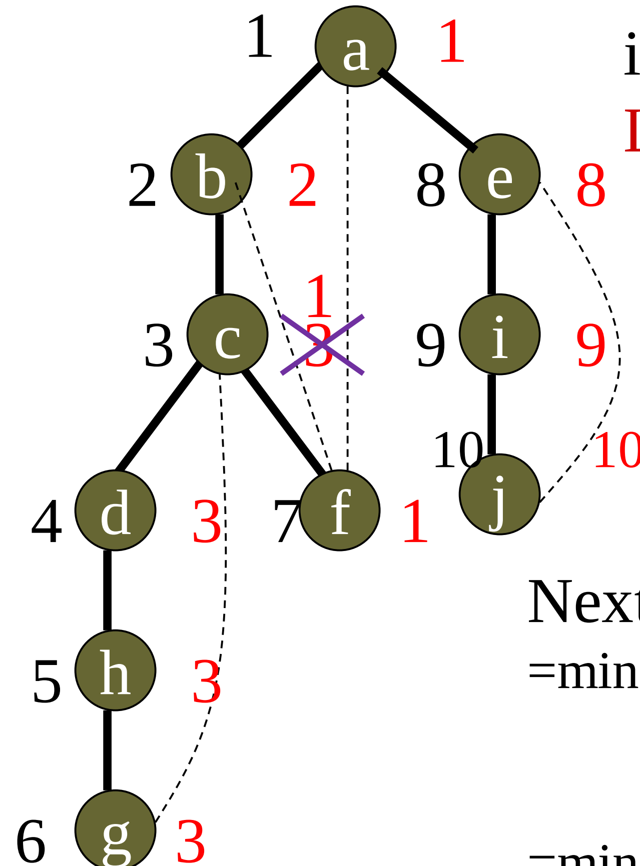
Travers the tree
in post-order
Left, Right, Root

Next, Node Low[f]
=min{ d[f],
min all d[w]
min all low [w]}
=min { 7, 1,NA}
= 1

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example



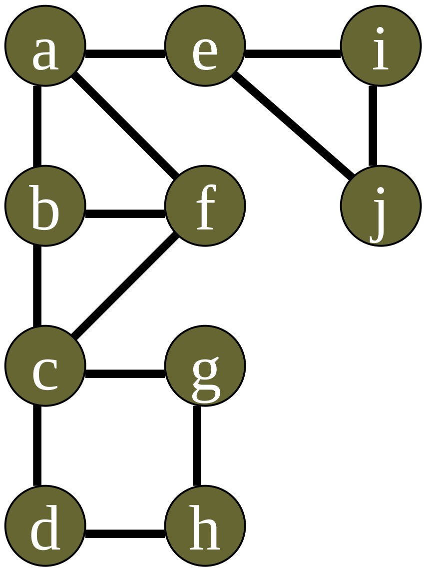


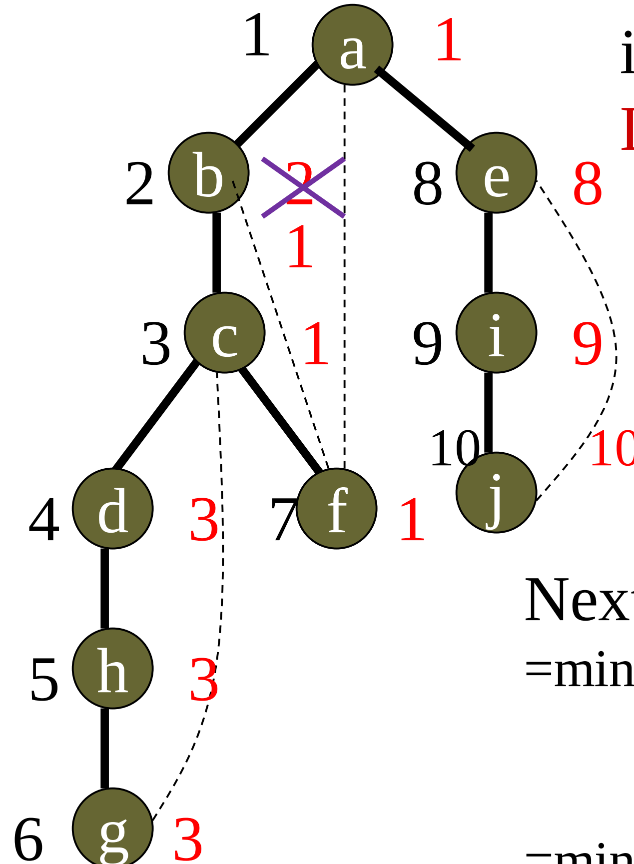
Travers the tree
in post-order
Left, Right, Root

Next, Node Low[c]
=min{ d[c],
min all d[w]
min all low [w]}
=min { 3, 1, 1}
= 1

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example



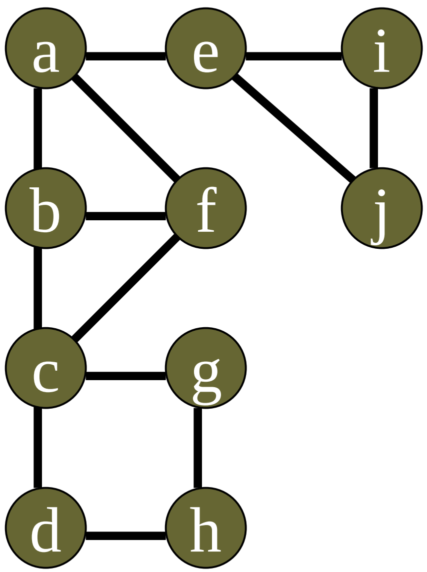


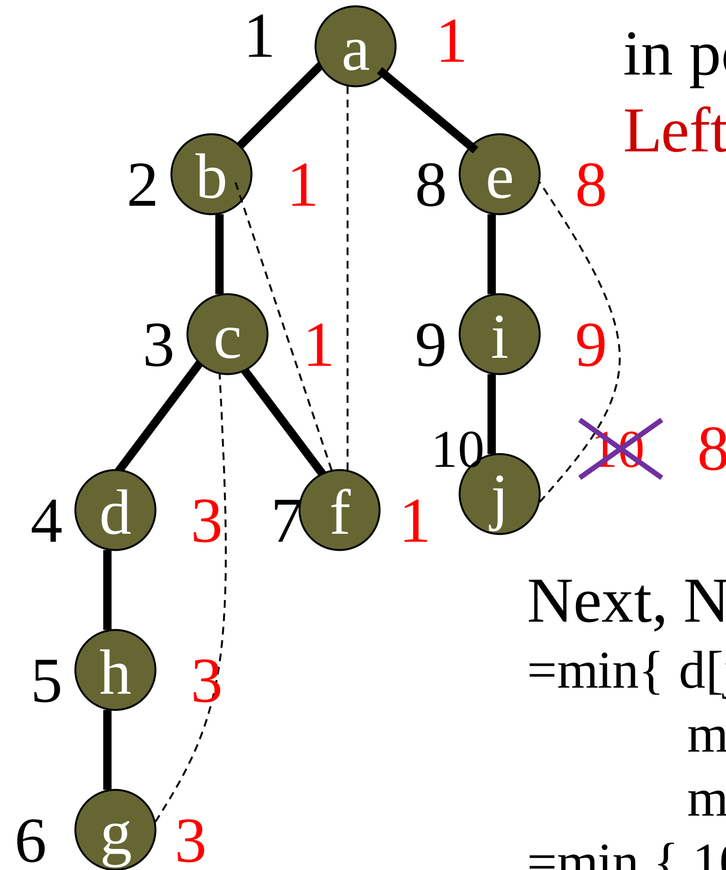
Travers the tree
in post-order
Left, Right, Root

Next, Node Low[b]
=min{ d[b],
min all d[w]
min all low [w]}
=min { 2, 1, 1}
= 1

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example



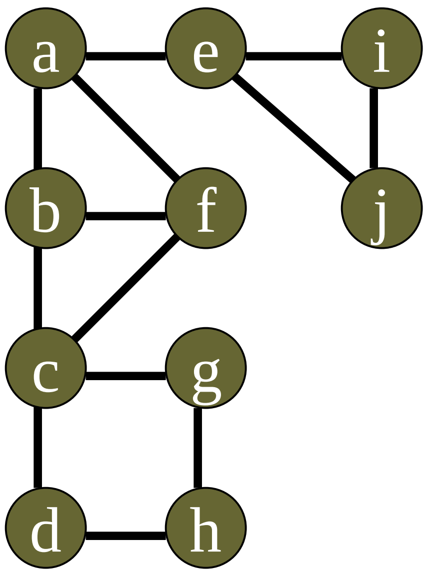


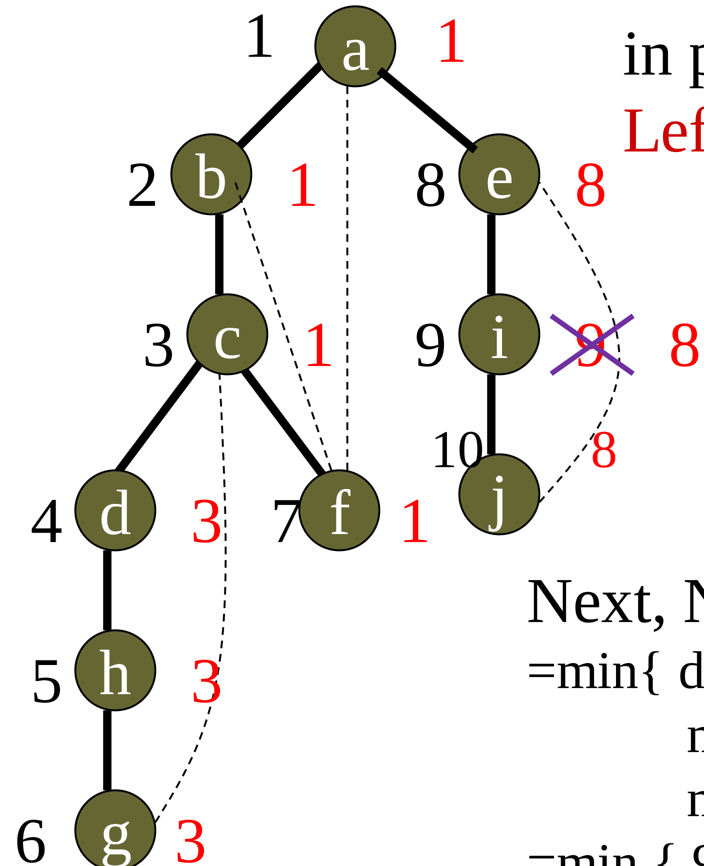
Travers the tree
in post-order
Left, Right, Root

Next, Node Low[j]
=min{ d[j],
min all d[w]
min all low [w]}
=min { 10, 8, NA}
= 8

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example



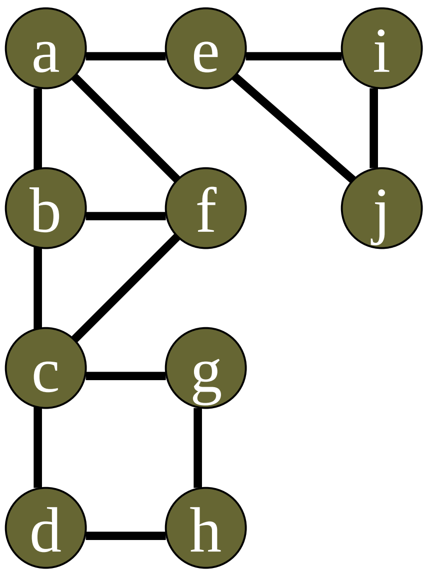


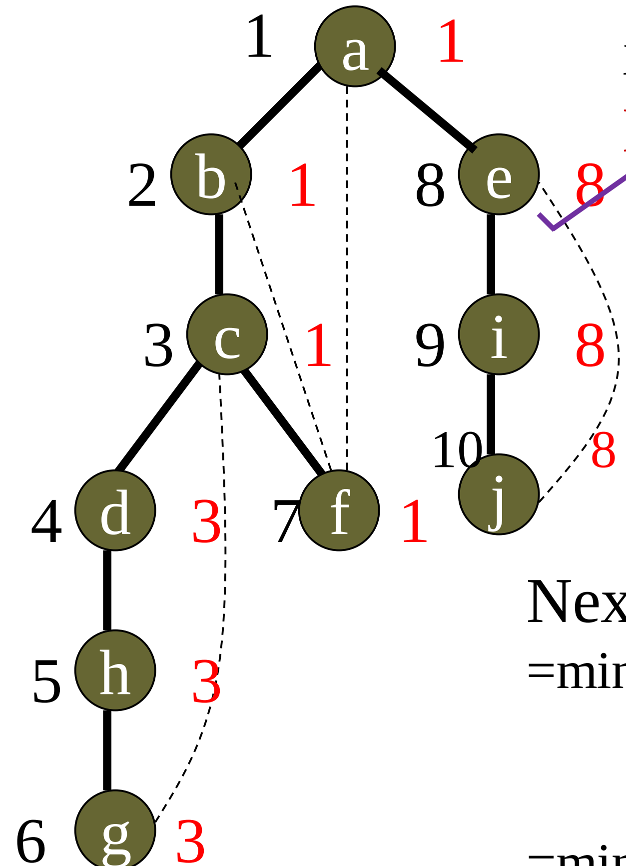
Travers the tree
in post-order
Left, Right, Root

Next, Node Low[i]
=min{ d[i],
min all d[w]
min all low [w]}
=min { 9, 8, 8}
= 8

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example





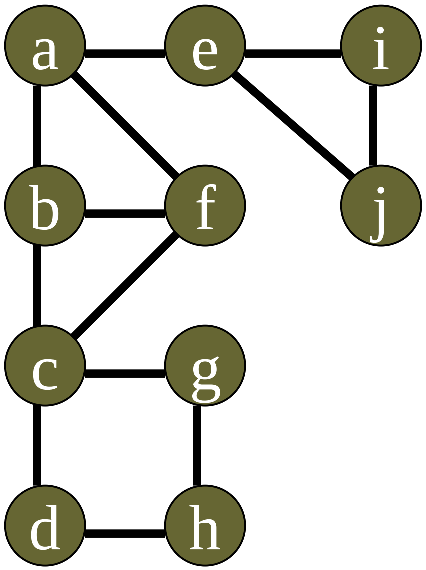
Travers the tree
in post-order
Left, Right, Root

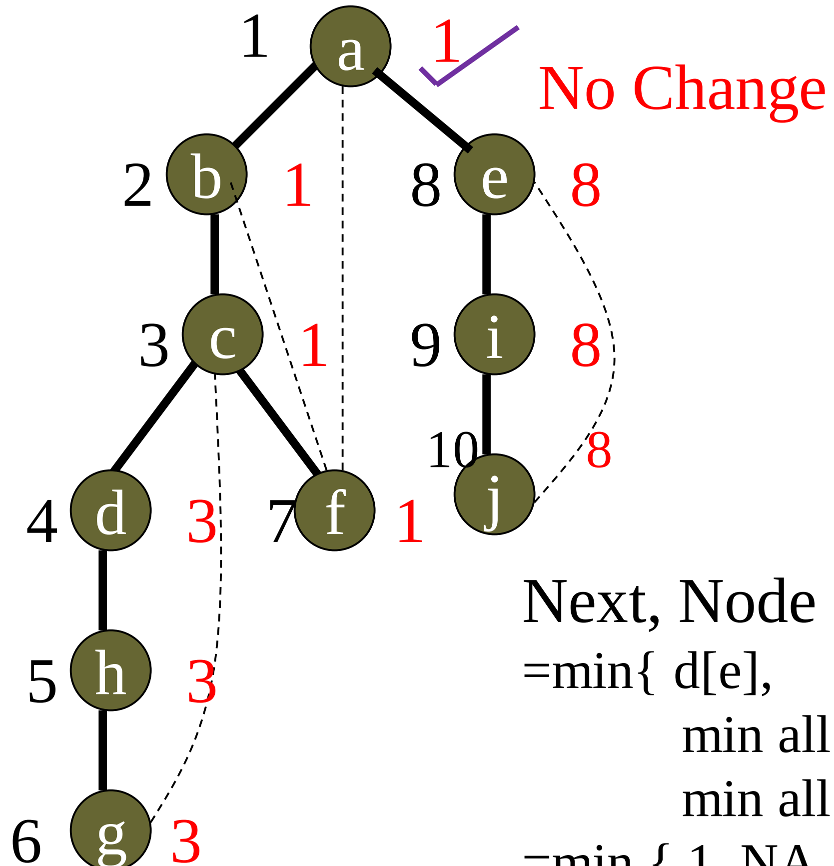
No Change

Next, Node Low[e]
=min{ d[e],
min all d[w]
min all low [w]}
=min { 8, 8, 8}
= 8

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example

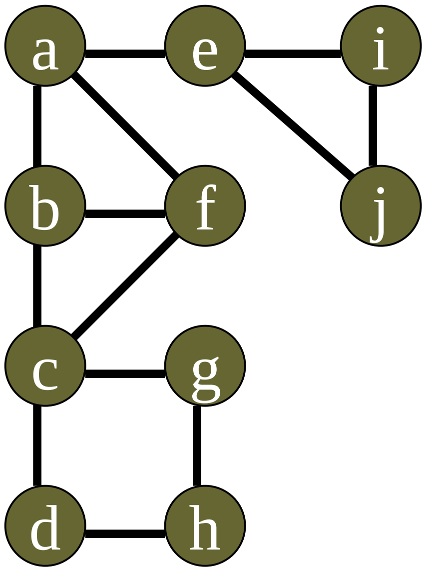


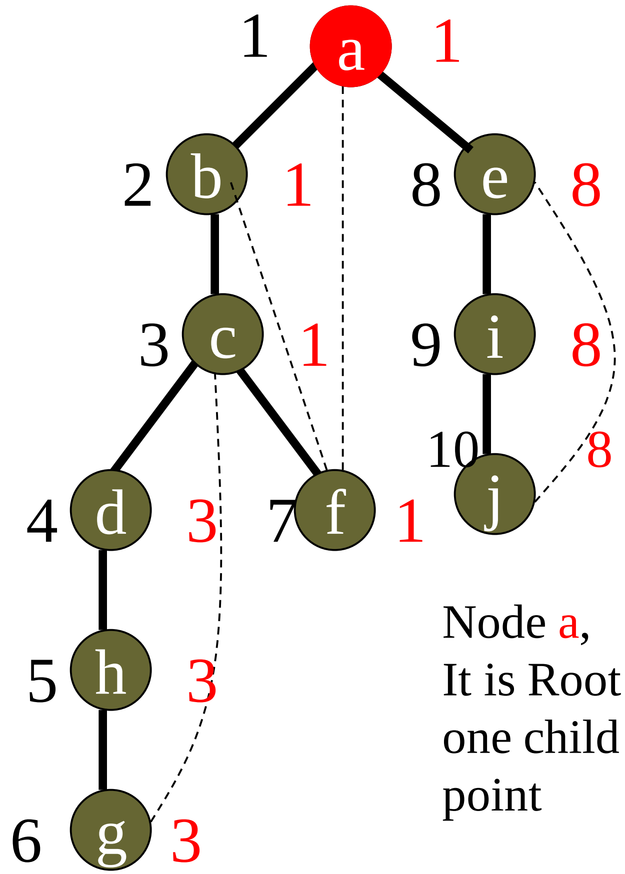


Next, Node Low[a]
=min{ d[e],
min all d[w]
min all low [w]}
=min { 1, NA,1}
= 1

low[v] = min{
d[*v*],
lowest d[w] among all back edges (v,w)
lowest low[w] among all tree edges (v,w)
}

Example

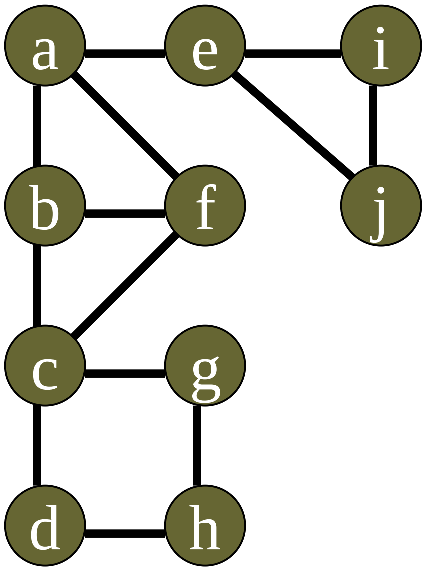


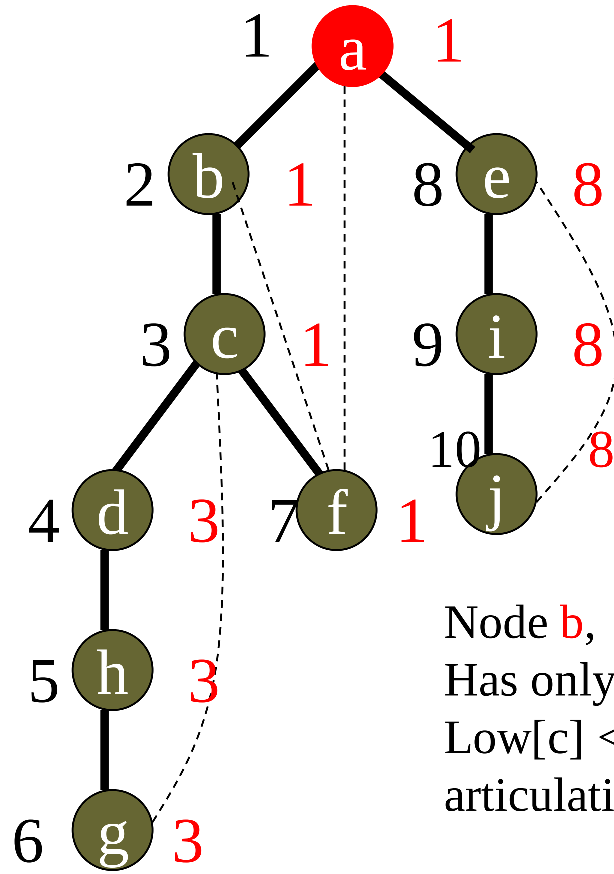


Node a,
It is Root & it has more than one child so it is an articulation point

u is an articulation point iff it has a child v for which Low[v] >= d[u]

Example

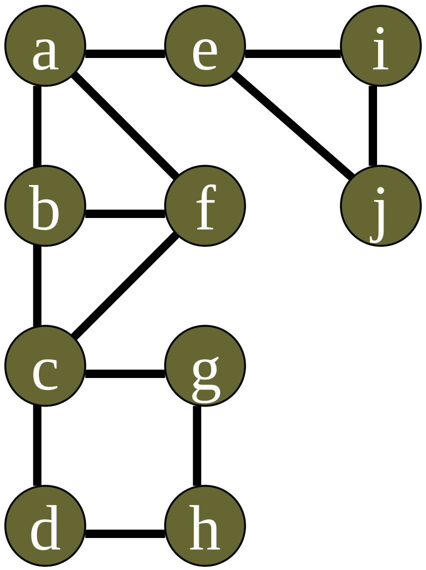


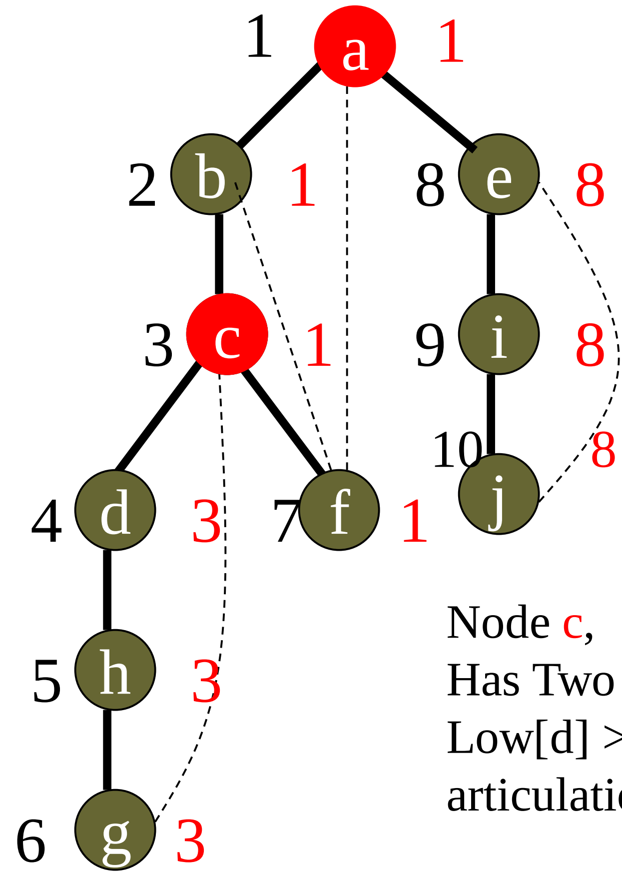


Node b,
Has only one child c &
Low[c] < d[b], so it is not an articulation point

u is an articulation point iff it has a child v for which Low[v] >= d[u]

Example

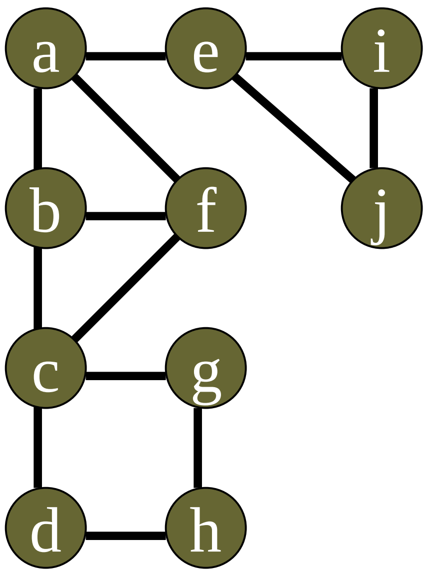


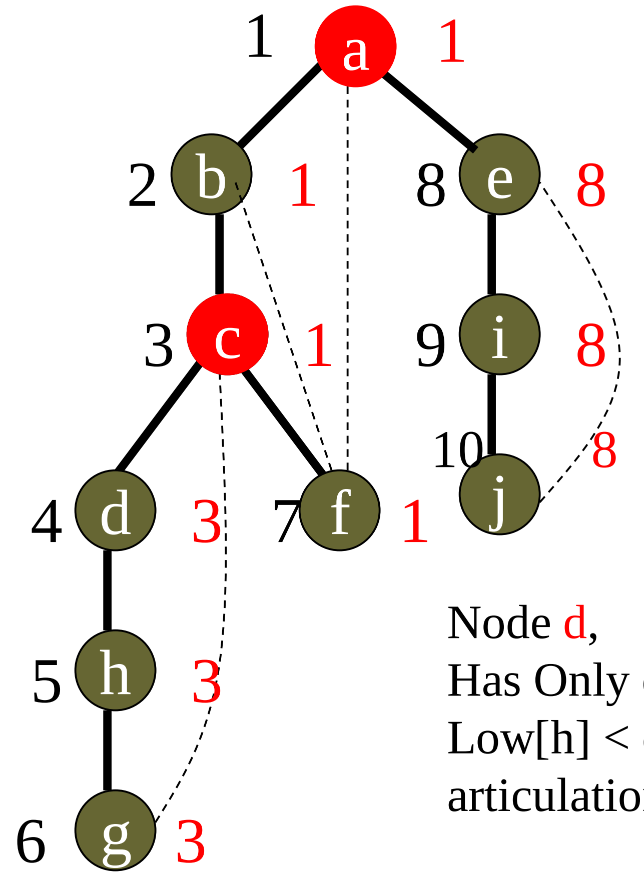


Node c,
Has Two child d & f, also
Low[d] >= d[c], so it is an
articulation point

u is an articulation point iff it has a child v for which Low[v] >= d[u]

Example

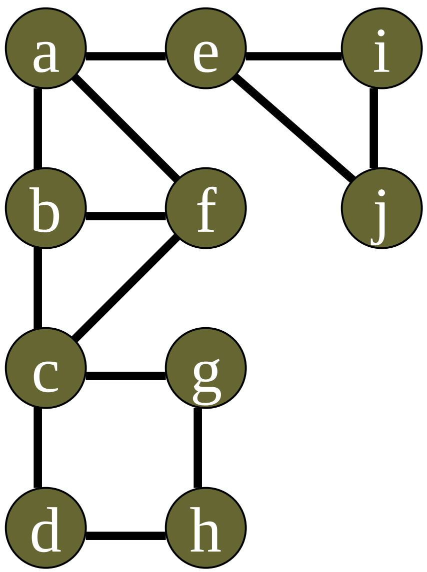


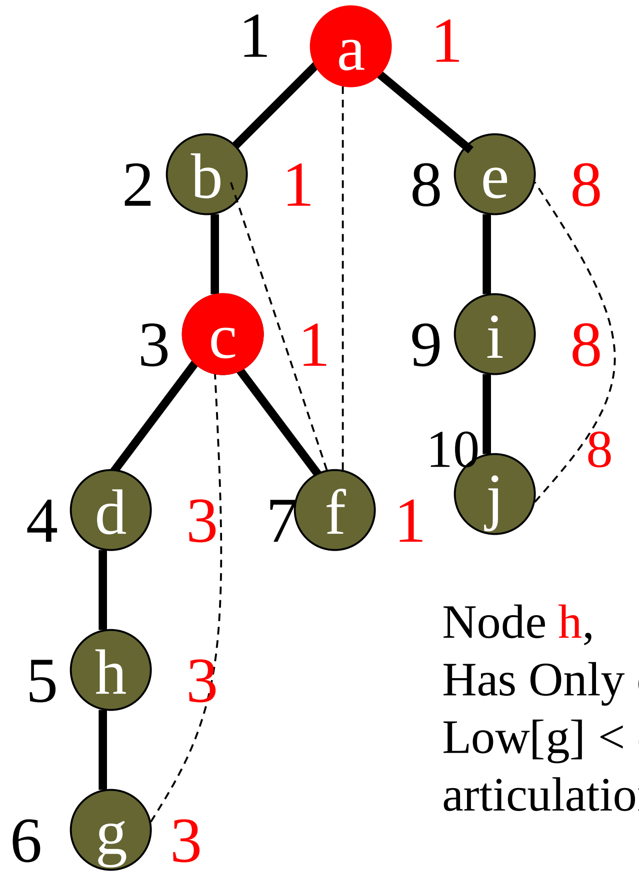


Node d,
Has Only one child h &
Low[h] < d[d], so it is not an articulation point

u is an articulation point iff it has a child v for which Low[v] >= d[u]

Example

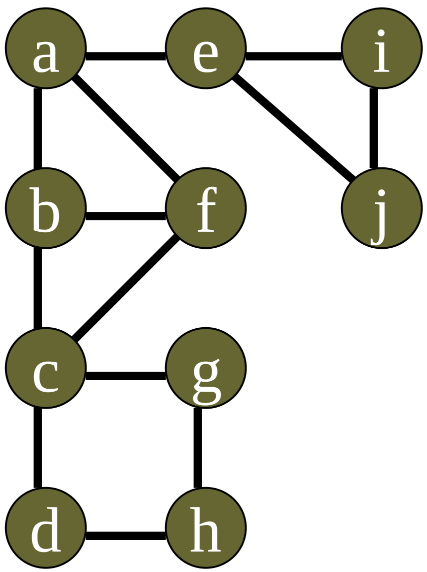


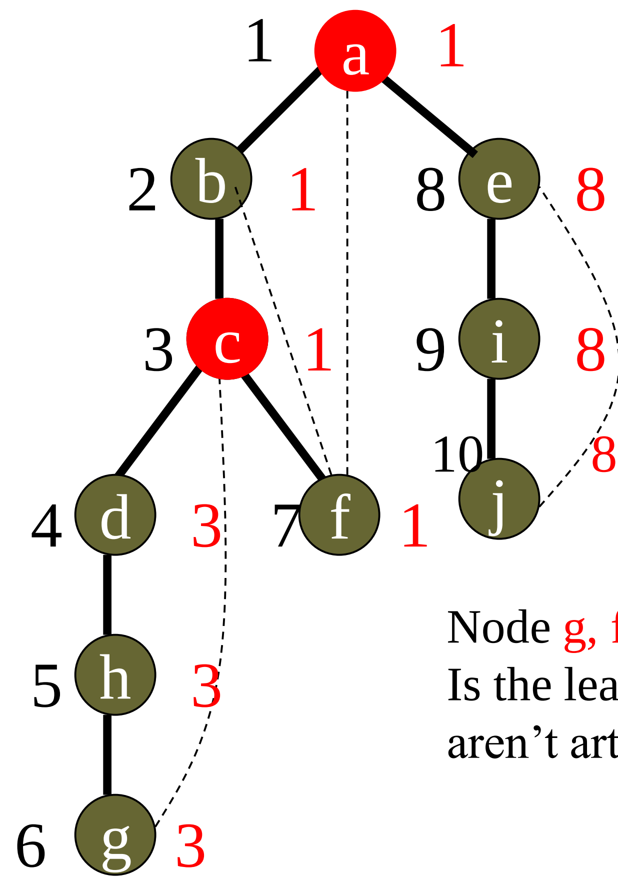


Node h,
Has Only one child g &
Low[g] < d[h], so it is not an articulation point

u is an articulation point iff it has a child v for which Low[v] >= d[u]

Example

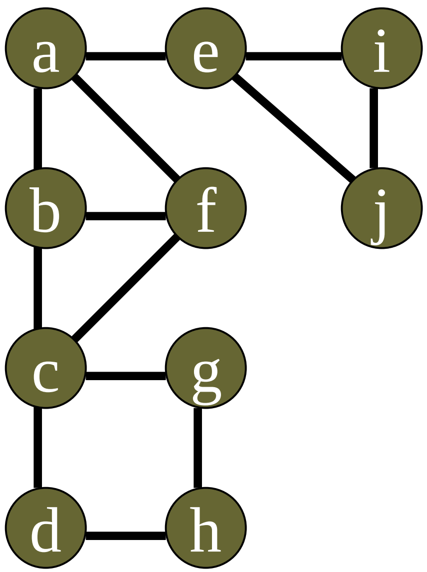


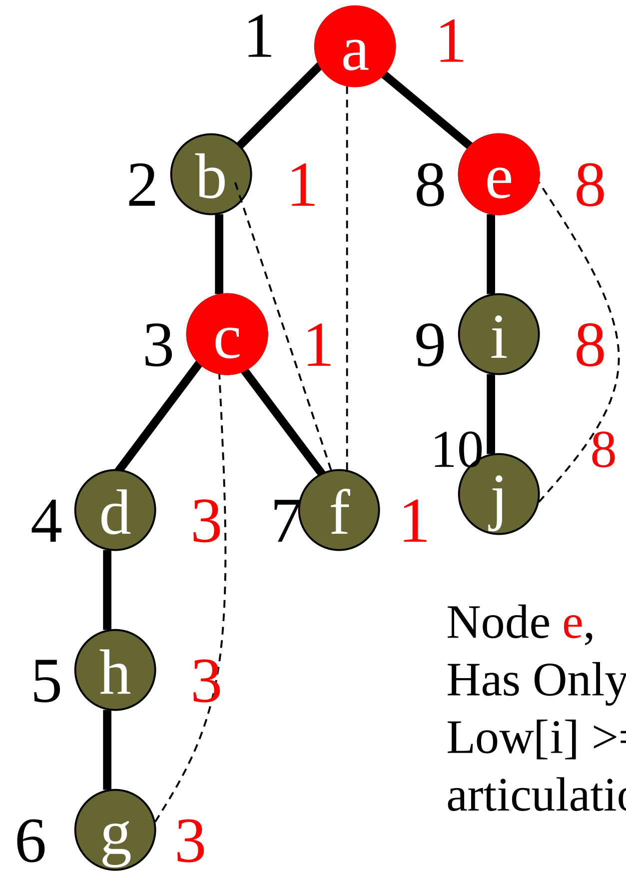


Node g, f & j,
Is the leaf node so these node aren't articulation point

u is an articulation point iff it has a child v for which Low[v] >= d[u]

Example

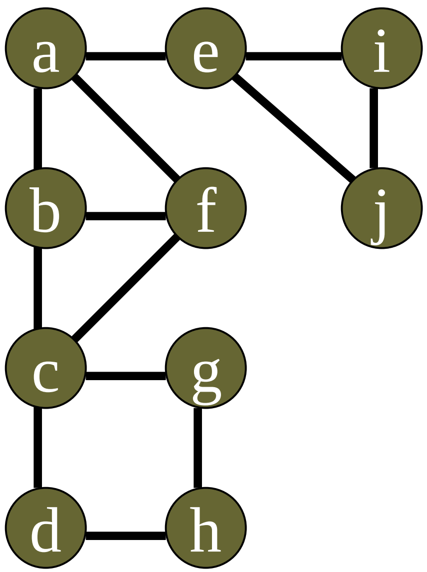


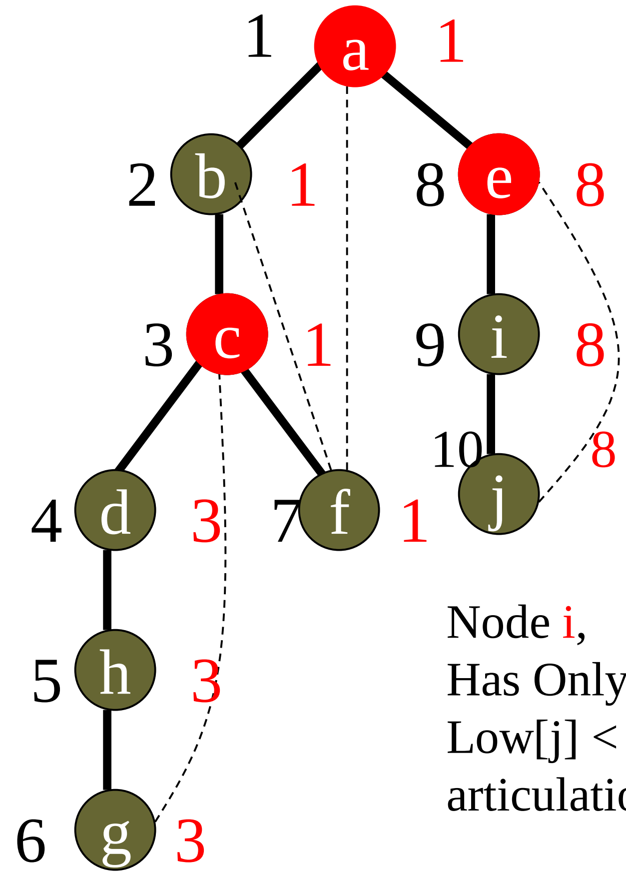


Node e,
Has Only one child i,
Low[i] >= d[e], so it is an articulation point

u is an articulation point iff it has a child v for which Low[v] >= d[u]

Example





Node i,
Has Only one child j,
Low[j] < d[i], so it is not an articulation point

u is an articulation point iff it has a child v for which Low[v] >= d[u]

Articulation Points: Pseudocode

```
Data: color[V], time, prev[V],d[V], f[V], low[V]

DFS(G) // where prog starts
{
    for each vertex u ∈ V
    {
        color[u] = WHITE;
        prev[u]=NIL;
        low[u]=inf;
        f[u]=inf; d[u]=inf;
    }
    time = 0;
    for each vertex u ∈ V
      if (color[u] == WHITE)
          DFS_Visit(u);
}
```

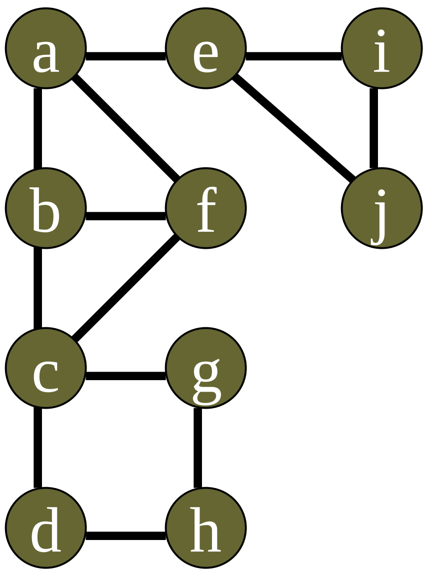
Articulation Points: Pseudocode

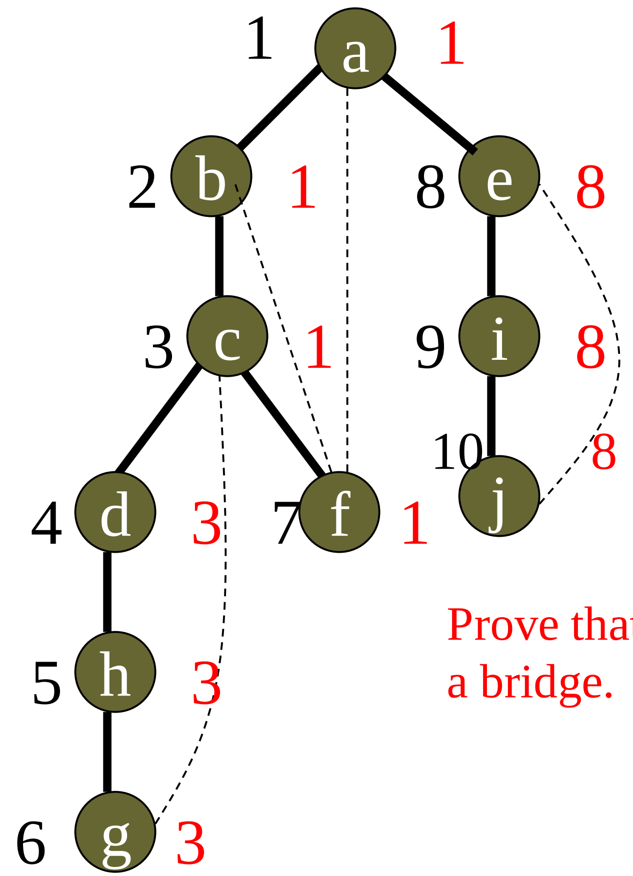
```
DFS_Visit(v)
{ color[v]=GREY;time=time+1;d[v] = time;
  low[v]= d[v];
  for each w ∈ Adj[v]{
    if(color[w] == WHITE){
       prev[w]=u;
        DFS_Visit(w);
        if low[w] >= d[v]
             record that vertex v is an articulation
        if (low[w] < low[v])
             low[v] := low[w];
    }
    else if w is not the parent of v then
         //--- (v,w) is a BACK edge
 `        if (d[w] < low[v])
                low[v] := d[w];
  }
  color[v] = BLACK;  time = time+1;   f[v] = time;
}
```

Source

- Mark Allen Weiss – Data Structure and Algorithm Analysis in C
  - Articulation Point

Example



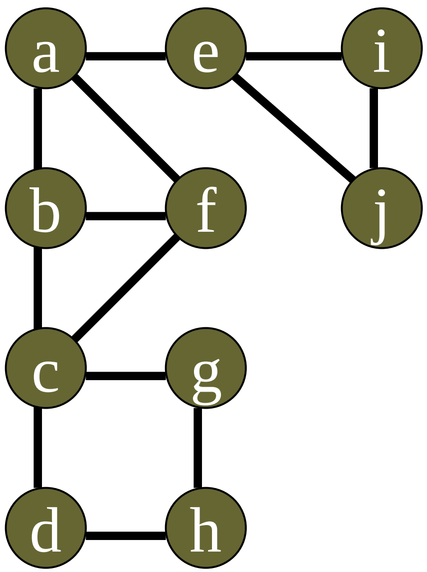


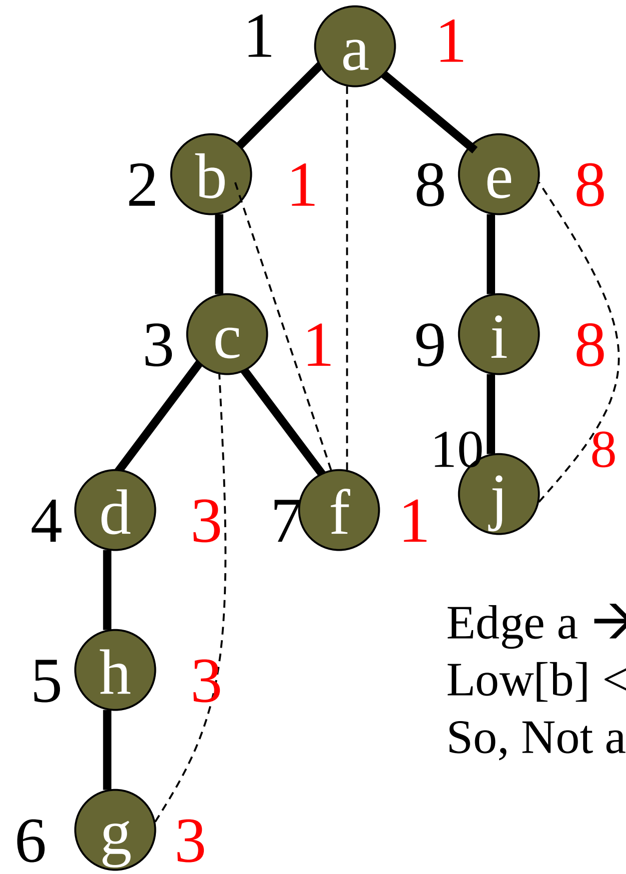
Prove that only tree edge can be a bridge.

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example



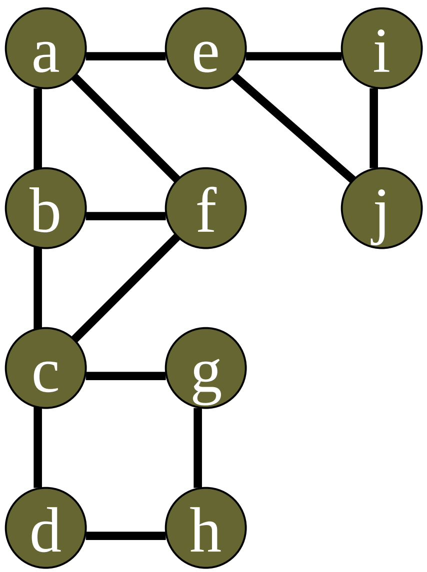


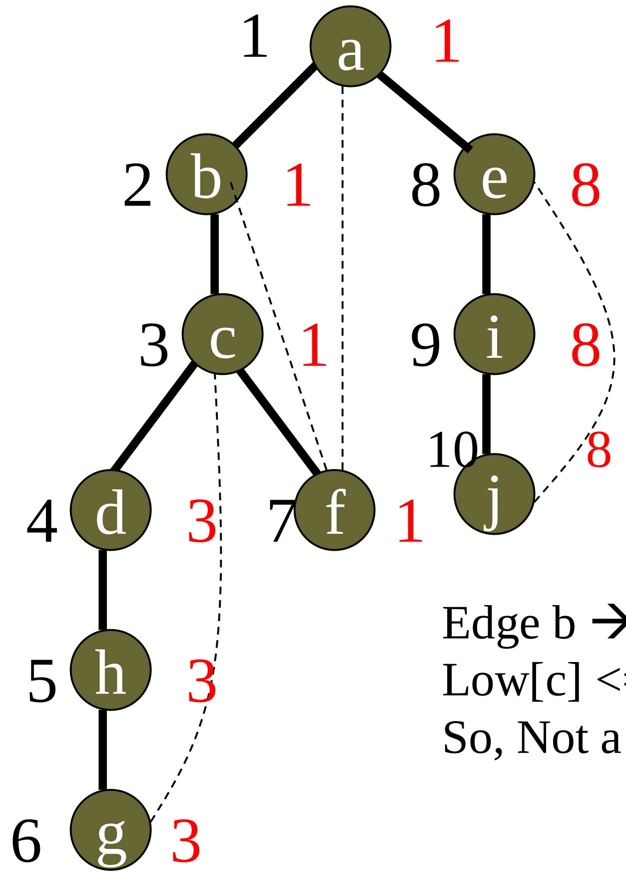
Edge a → b,
Low[b] <= d[a]
So, Not a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example



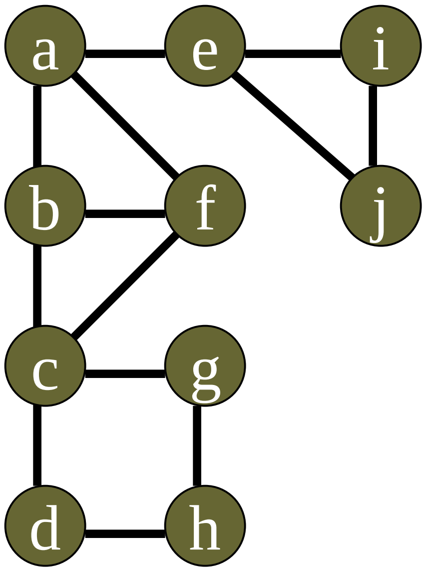


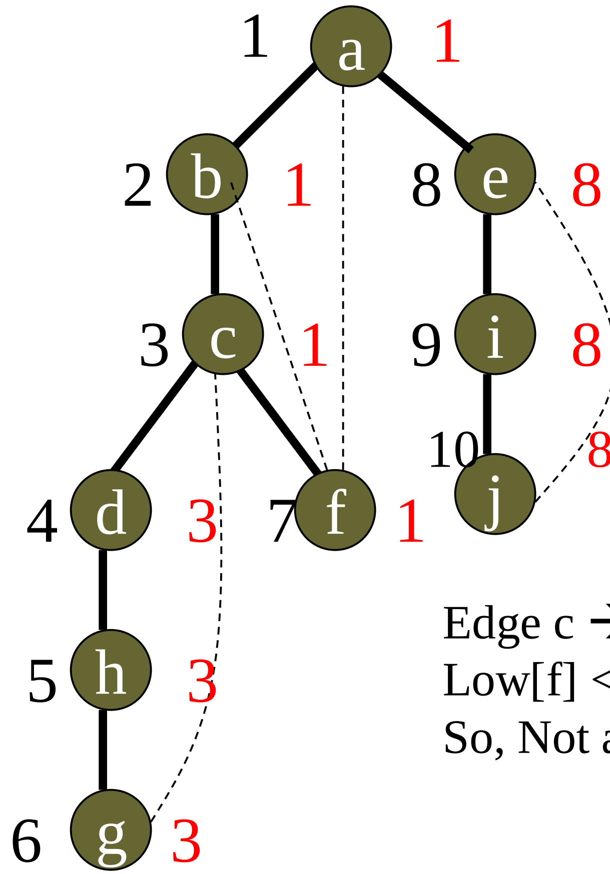
Edge b → c,
Low[c] <= d[b]
So, Not a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example



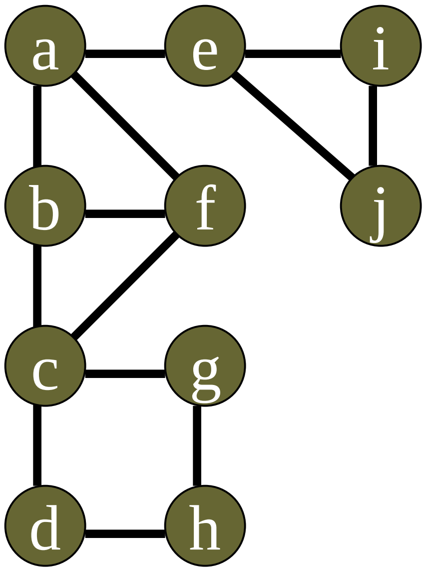


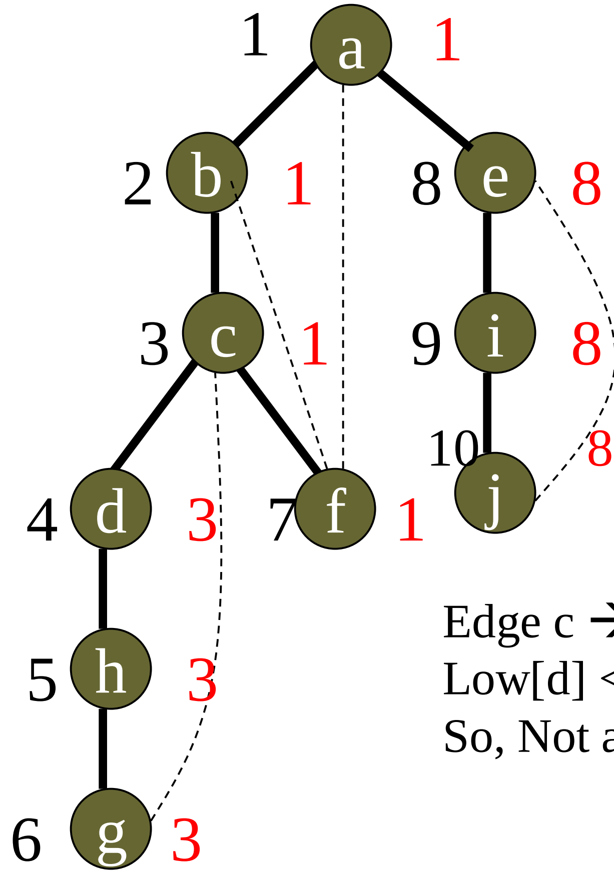
Edge c → f,
Low[f] <= d[c]
So, Not a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example



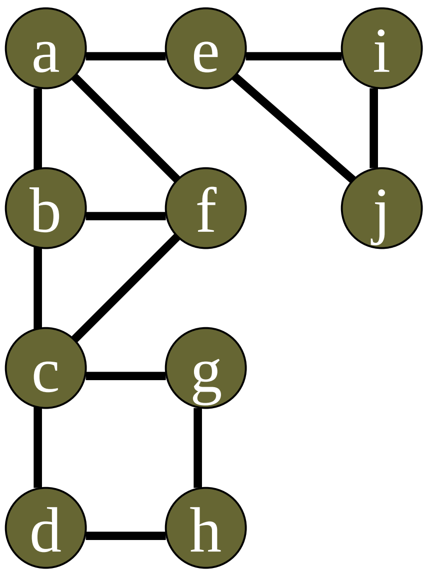


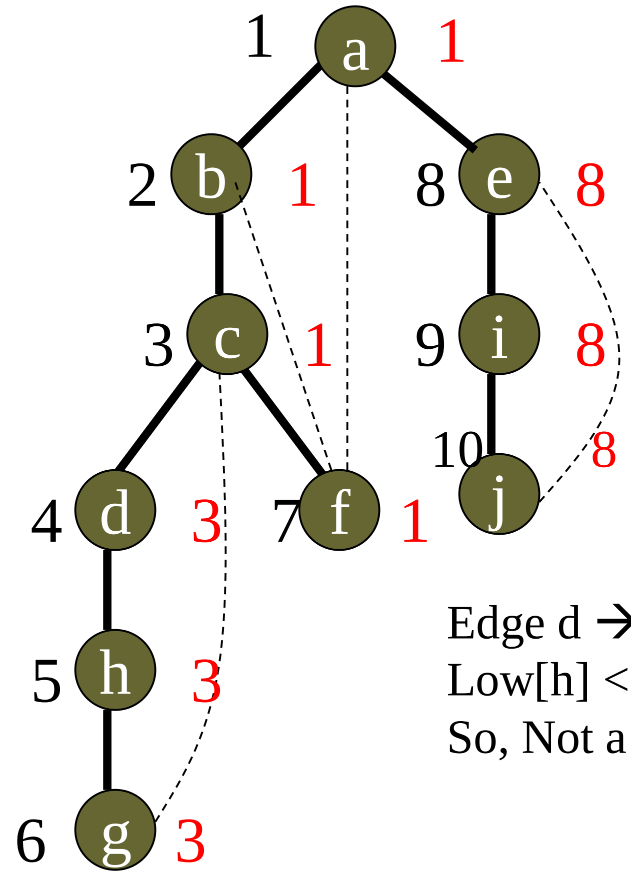
Edge c → d,
Low[d] <= d[c]
So, Not a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example



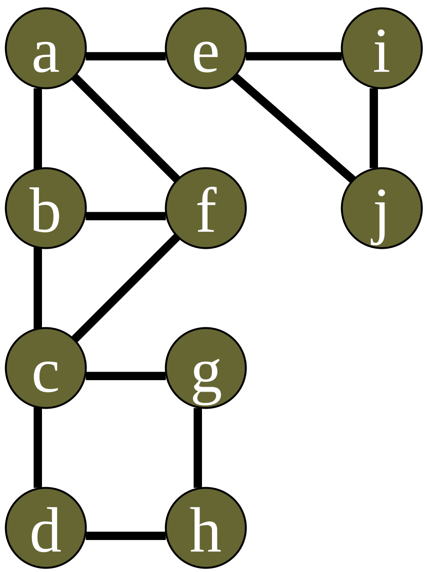


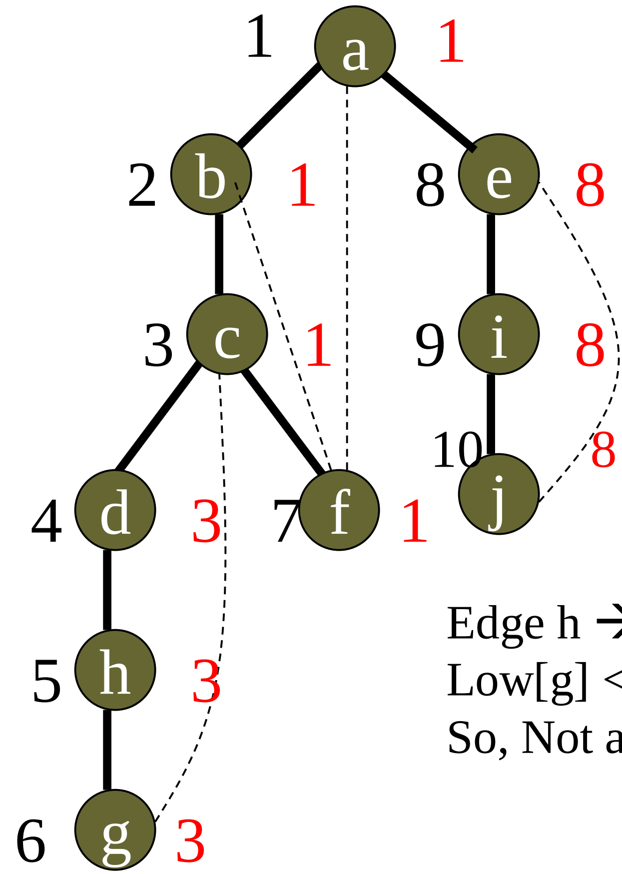
Edge d → h,
Low[h] <= d[d]
So, Not a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example



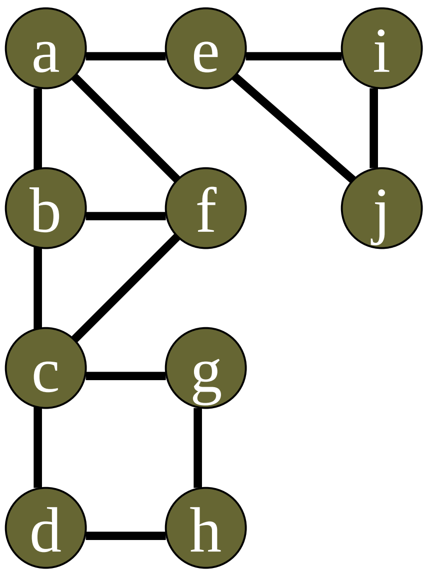


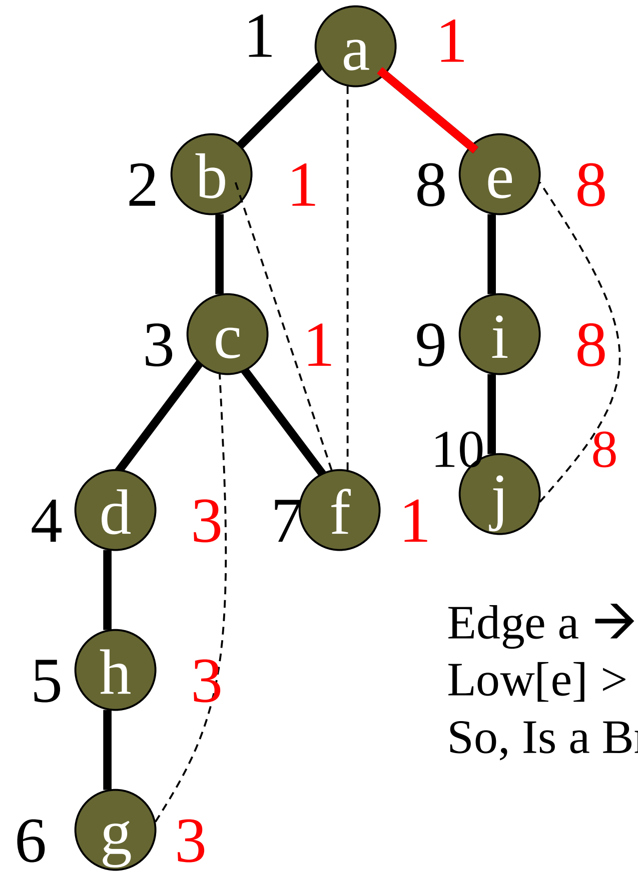
Edge h → g,
Low[g] <= d[h]
So, Not a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example



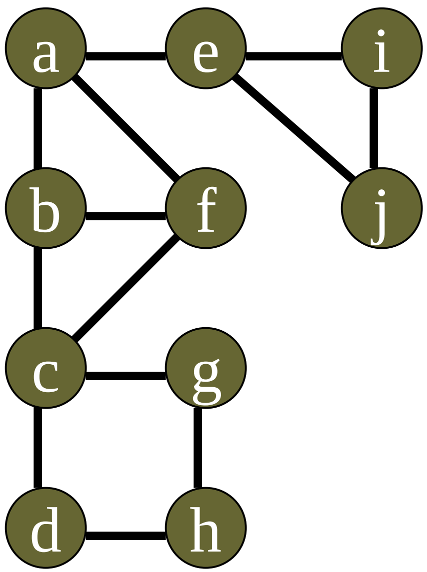


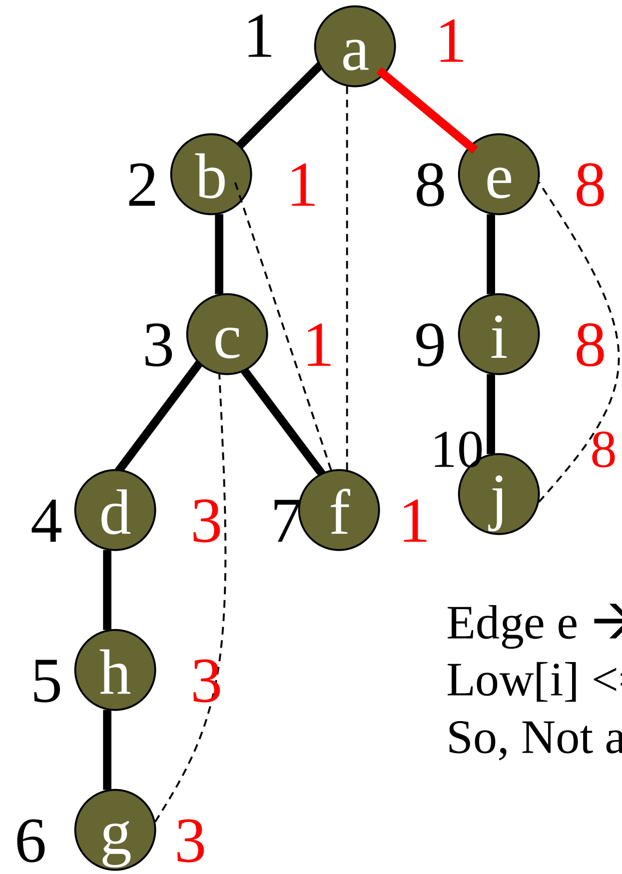
Edge a → e,
Low[e] > d[a]
So, Is a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example



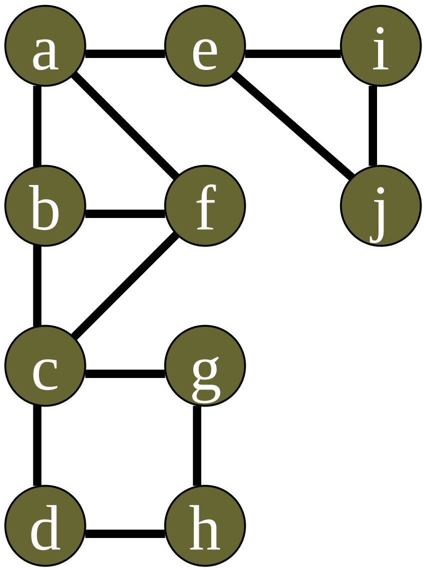


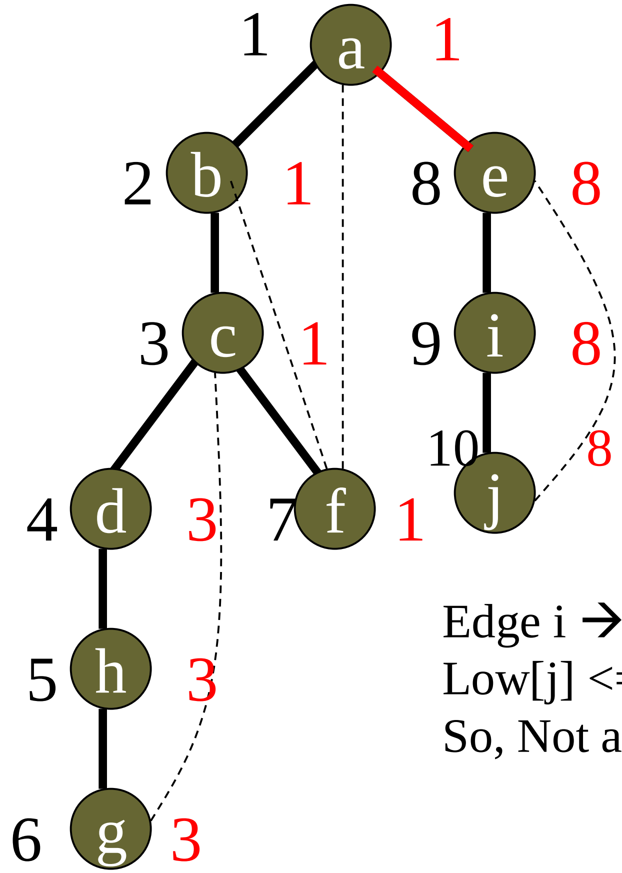
Edge e → i,
Low[i] <= d[e]
So, Not a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]

Example





Edge i → i,
Low[j] <= d[i]
So, Not a Bridge

A **bridge** is an edge in a connected graph whose removal makes it disconnected

(u,v) is a bridge if LOW(v) > d[u]