Ch 6
Basic traversal and search

6.1 techniques for binary trees

Concept

- Traverse all vertices in a binary tree
- There are three kinds of traversal manners (assume that left child first)
  - In-order: complete all left child nodes → lookup content → go right child nodes

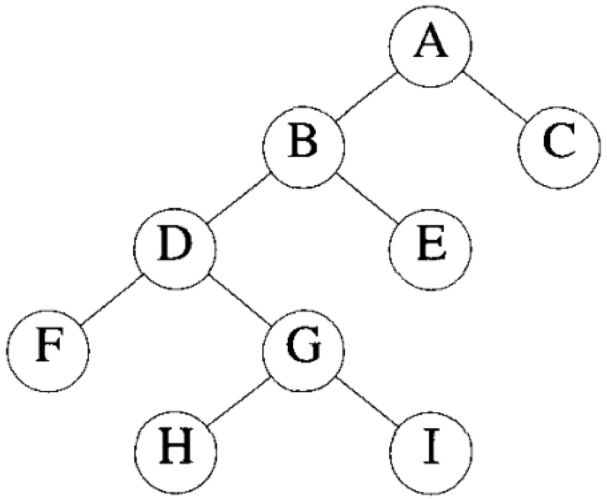
```
Algorithm InOrder(t)
// t is a binary tree. Each node of t has
// three fields: lchild, data, and rchild.
{
    if t ≠ 0 then
    {
        InOrder(t → lchild);
        Visit(t);
        InOrder(t → rchild);
    }
}
```

- Pre-order: lookup content → go left child nodes → go right child nodes
- Post-order: go left child nodes → go right child nodes → lookup content

```
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);
    }
}
```

```
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 → lchild);
        PostOrder(t → rchild);
        Visit(t);
    }
}
```

In-order example



call of InOrder	value in root	action
main	A	
1	B	
2	D	
3	F	
4	—	print ('F')
4	—	print ('D')
3	G	
4	H	
5	—	print ('H')
5	—	print ('G')
4	I	
5	—	print ('I')
5	—	print ('B')
2	E	
3	—	print ('E')
3	—	print ('A')
1	C	
2	—	print ('C')
2	—	

6.2 techniques for graphs

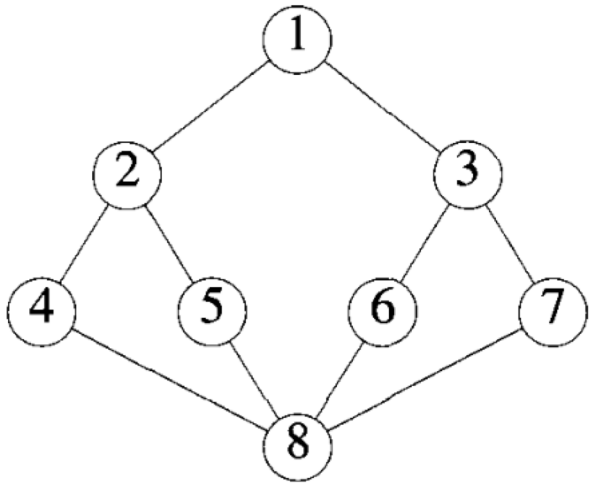
Concept

- Traverse all nodes in a graph

- Breadth first search and traversal (BFS)
- Depth first search and traversal (DFS)

BFS

- Use a queue to implement the idea



```
Algorithm BFS(v)
// A breadth first search of G is carried out beginning
// at vertex v. For any node i, visited[i] = 1 if i has
// already been visited. The graph G and array visited[ ]
// are global; visited[ ] is initialized to zero.
{
    u := v; // q is a queue of 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 first unexplored vertex.
    } until(false);
}
```

When the graph is disconnected

```
Algorithm BFT(G, n)
// Breadth first traversal of G
{
    for i := 1 to n do  // Mark all vertices unvisited.
        visited[i] := 0;
    for i := 1 to n do
        if (visited[i] = 0) then BFS(i);
}
```

DFS

- Implemented by the recursive

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

```python
graph = {
    'A': ["B", "D", "E"],
    'B': ["A", "C"],
    'C': ["B", "E"],
    'D': ["A", "E"],
    'E': ["A", "D", "F", "C"],
    'F': ["E"]
}
```

```python
def bfs(graph,start):
    queue = list()
    result = list()
    visited = list()

    queue.append(start)
    visited.append(start)

    while(len(queue)>0):
        current_vertex = queue.pop(0)
        result.append(current_vertex)
        for neighbor in graph[current_vertex]:
        if neighbor not in visited:
            queue.append(neighbor)
            visited.append(neighbor)

    return result
```

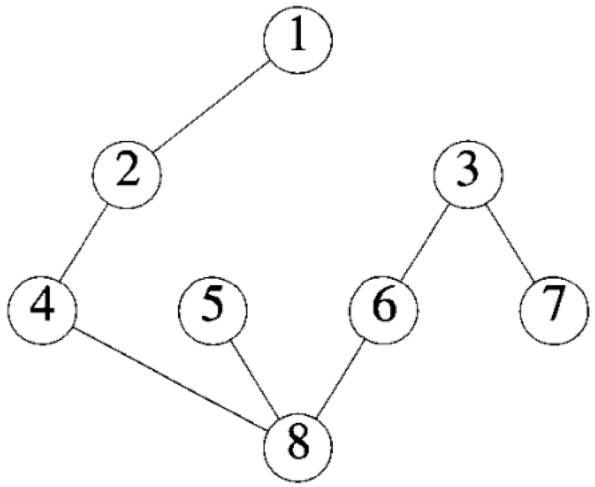
```python
def dfs(graph,start):
    stack = list()
    result = list()
    visited = list()

    stack.append(start)
    visited.append(start)

    while(_________):
        current_vertex = __________
        result.append(current_vertex)
        for neighbor in graph[current_vertex]:
        if neighbor not in visited:
            ______________________
            visited.append(neighbor)

    return result
```

6.3 connected components and spanning trees

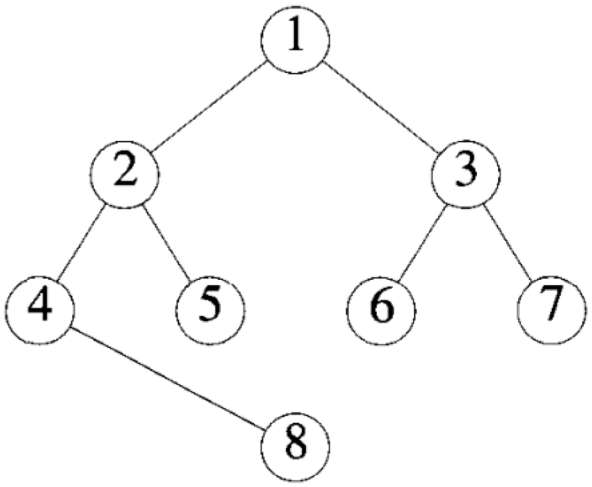
The application of BFS and DFS

- Identify the connected component in a graph
  - Check by if all nodes are visited

- Form spanning trees



(a) DFS(1) spanning tree

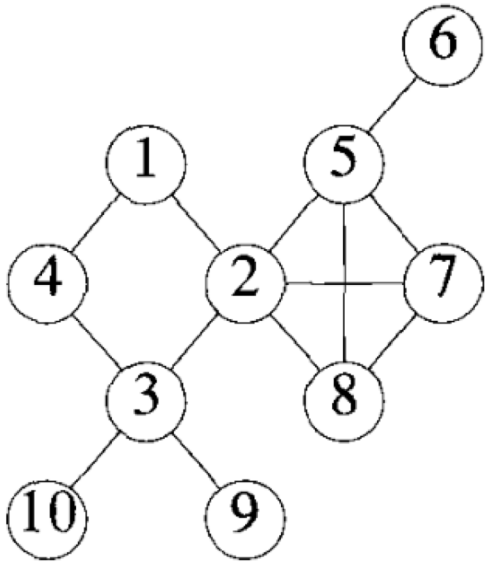


(b) BFS(1) spanning tree

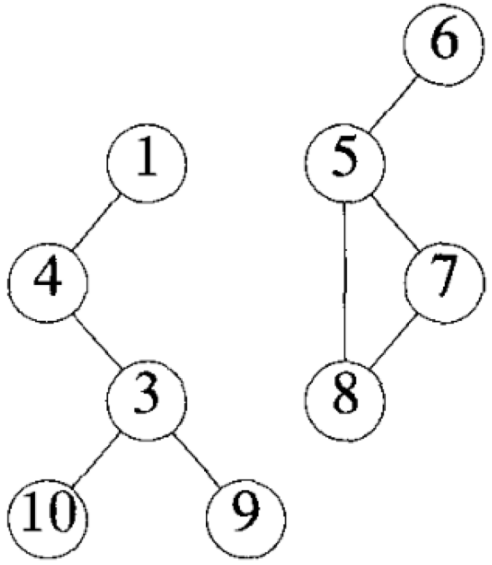
6.4 biconnected components and DFS

Introduction

- Articulation point:
  - In a connected graph, some nodes will be taken as articulation points if moving those nodes, the graph will be split



(a) Graph G

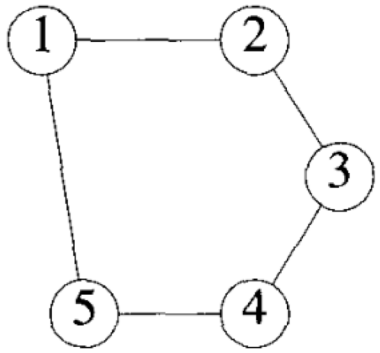


(b) Result of deleting vertex 2

Introduction

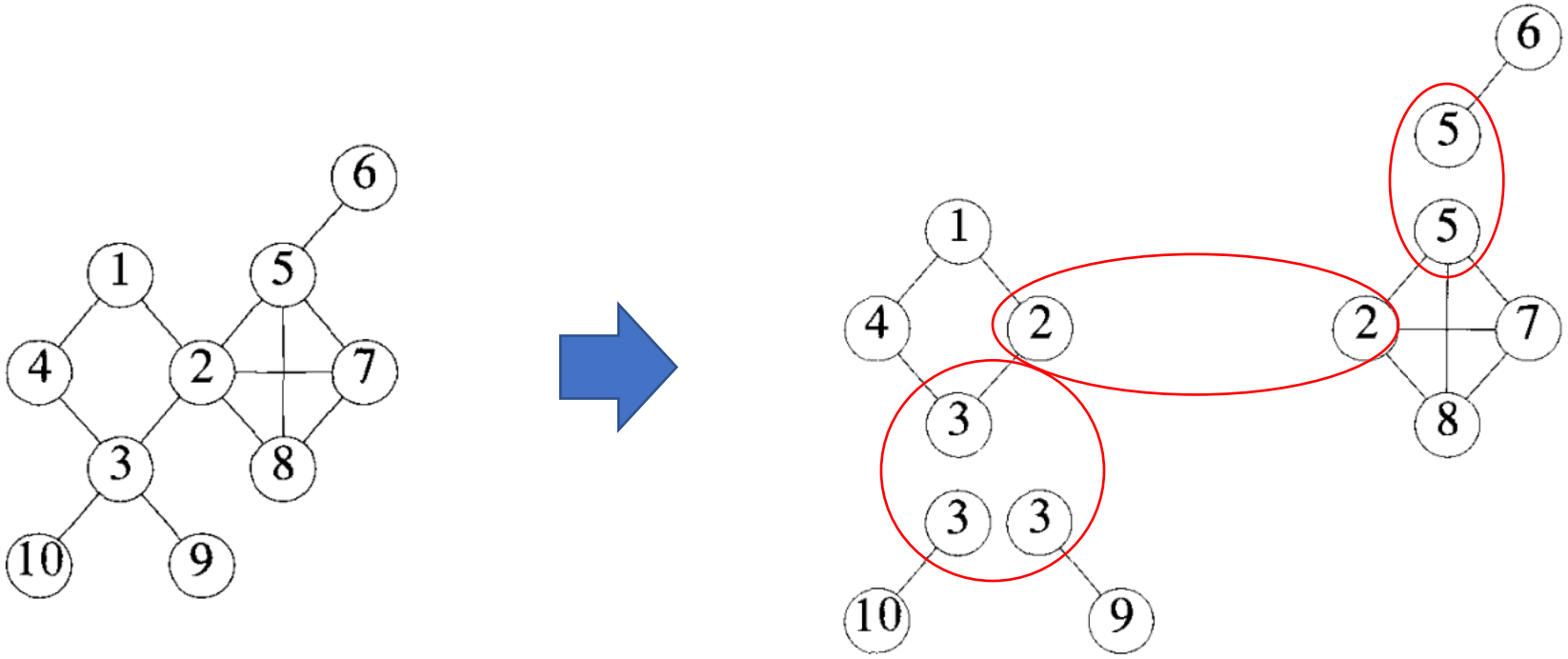
- A graph is biconnected if the graph has no articulation points
  - Can avoid single node failure

- Our target:
  - Find articulation points
  - Change the graph to be biconnected



The solution

- Concept
  - Find biconnected components



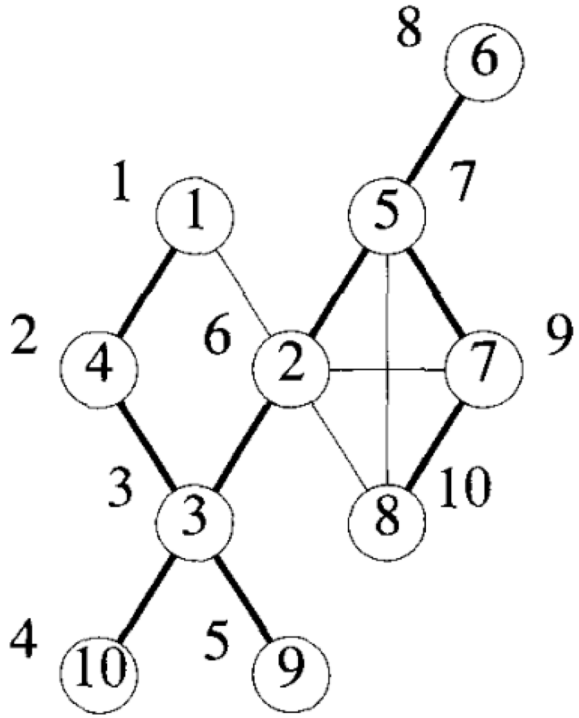
Construct a biconnected graph

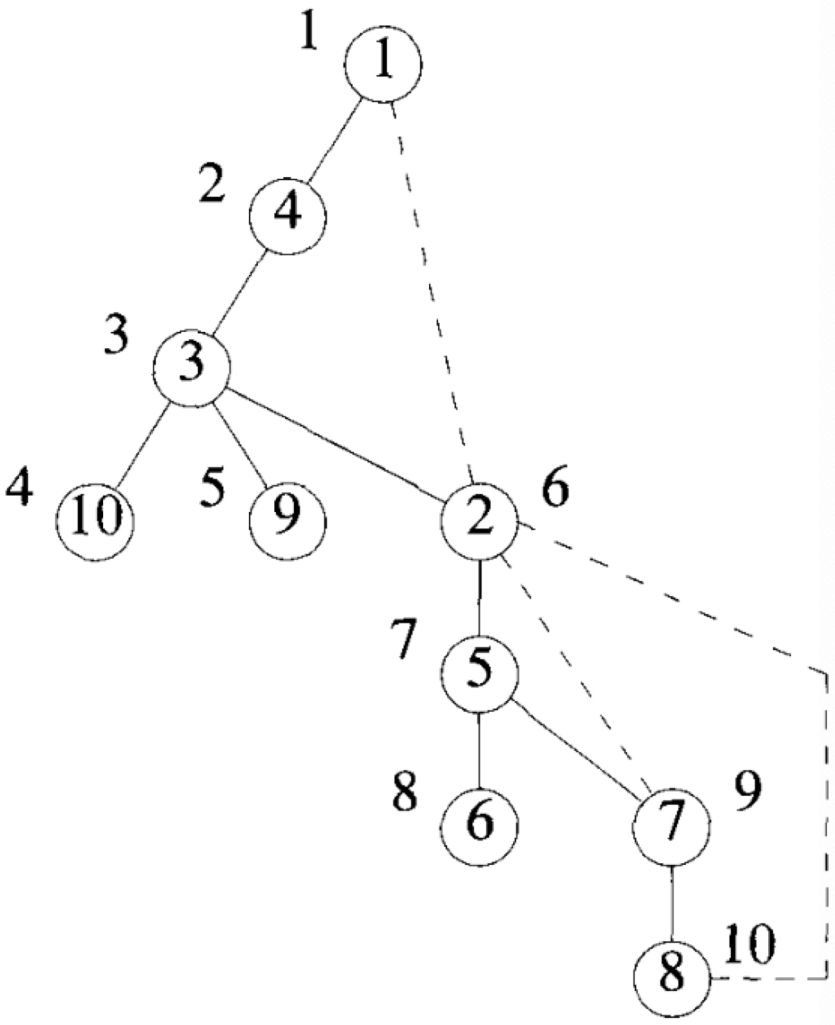
```
for each articulation point a do
{
    Let B_1, B_2, ..., B_k be the biconnected
    components containing vertex a;
    Let v_i, v_i ≠ a, be a vertex in B_i, 1 ≤ i ≤ k;
    Add to G the edges (v_i, v_{i+1}), 1 ≤ i < k;
}
```

- Find any two nodes that are not articulation points
- → then connects these two nodes

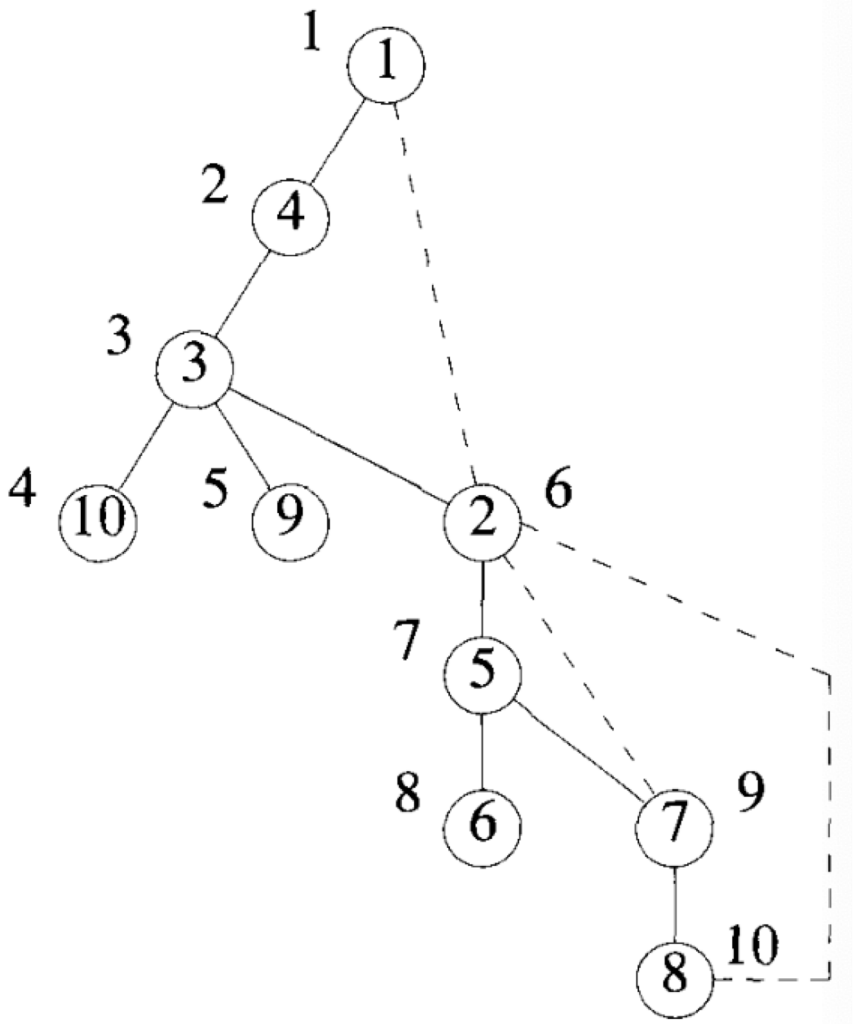
Find articulation point

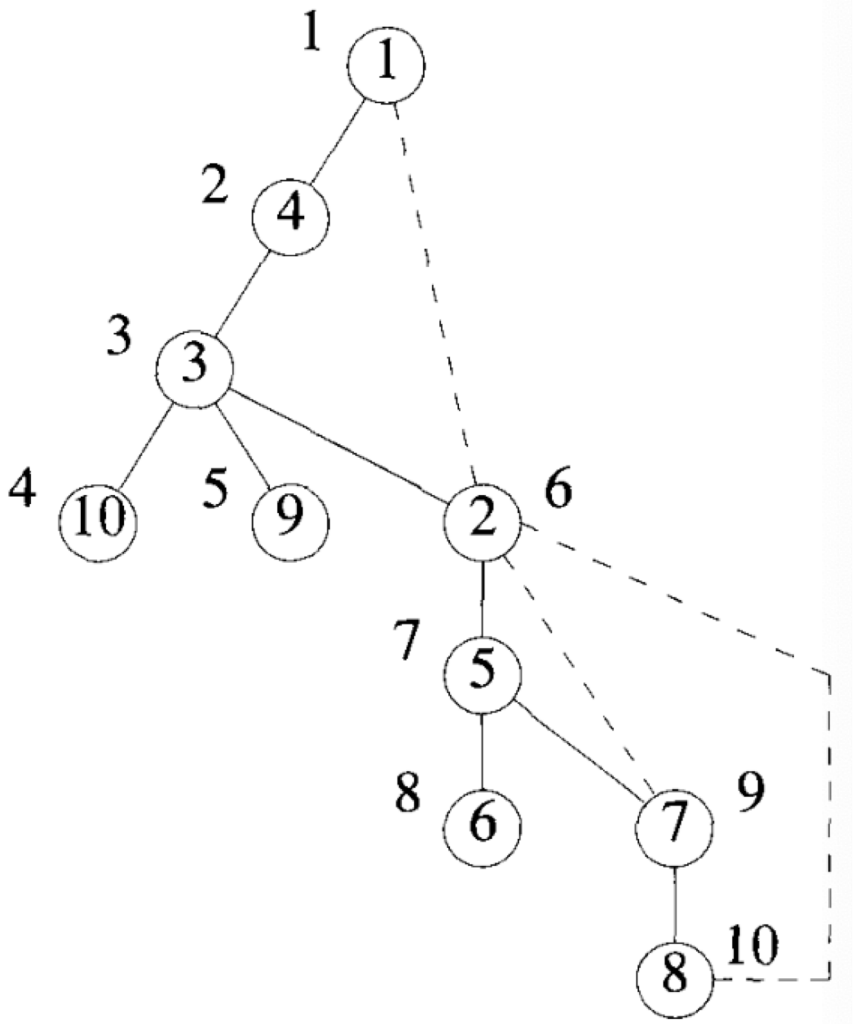
- Use DFS to traverse all nodes
  - Give visited node a sequence number. Ex: dfn[i] = j
    (the seq of node i is j)
- There are two kinds of edge on the DFS tree
  - Tree edge
  - Back edge (the shortcut to arrive ancestor)





- Calculate L values in a bottom-up fashion
  - The L[u] means that when taking u as a subtree root, the smallest dfn value of ancestor along the back edge
  - L[u] = min { dfn[u],
    min {L[w] | w is the child of u}
    min {dfn[w] | (u, w) is a back edge} }



A node u is an articulation point

- If u is root and u has at least two child nodes
  - By the DFS, the root is visited at least twice
- u is not root and one of its child cannot reach an ancestor through back edge
  - i.e., the child has only one path to ancestor