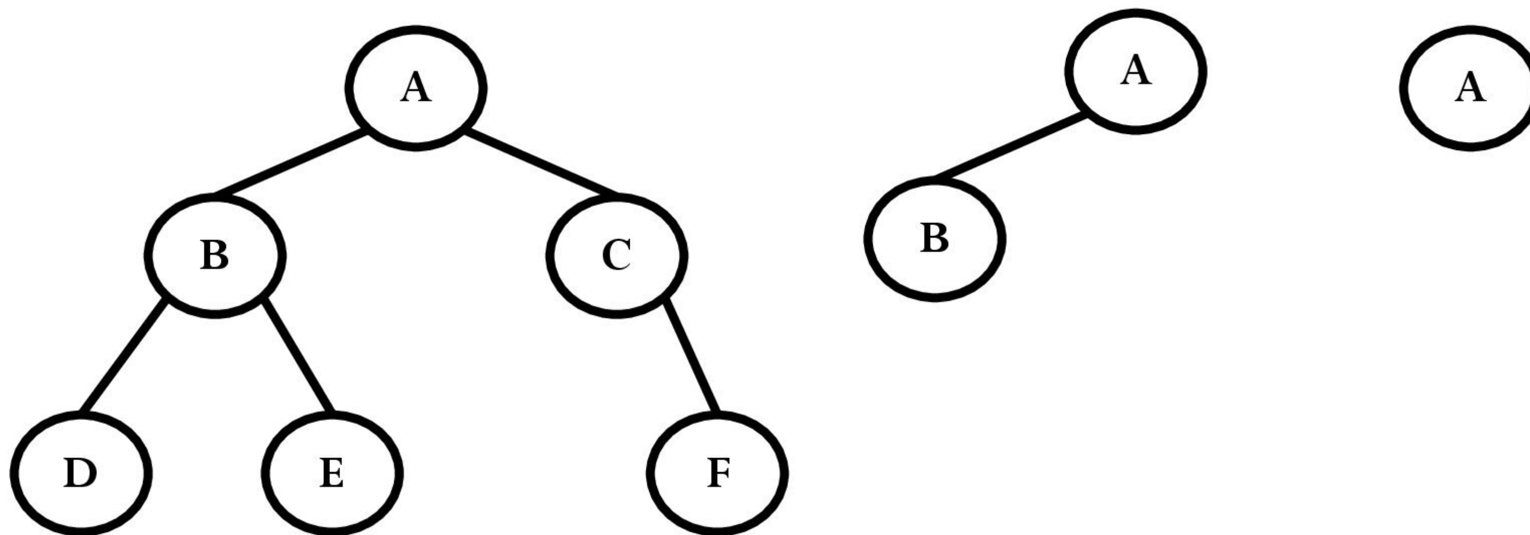
Data Structures – CST 201
Module ~ 4

Syllabus

- **Trees and Graphs**
  - Trees
    - **Binary Trees**
      - **Binary Tree Representation**
      - **Binary Tree Operations**
      - **Binary Tree Traversals**
    - Binary Search Trees
      - Binary Search Tree Operations
  - Graphs
    - Representation of Graphs
    - Depth First Search and Breadth First Search on Graphs
    - Applications of Graphs

Binary Trees

- A **Binary Tree** is a finite set of nodes that is either
  - Empty or
  - Consists of root node and two disjoint binary trees, called, left subtree and right subtree



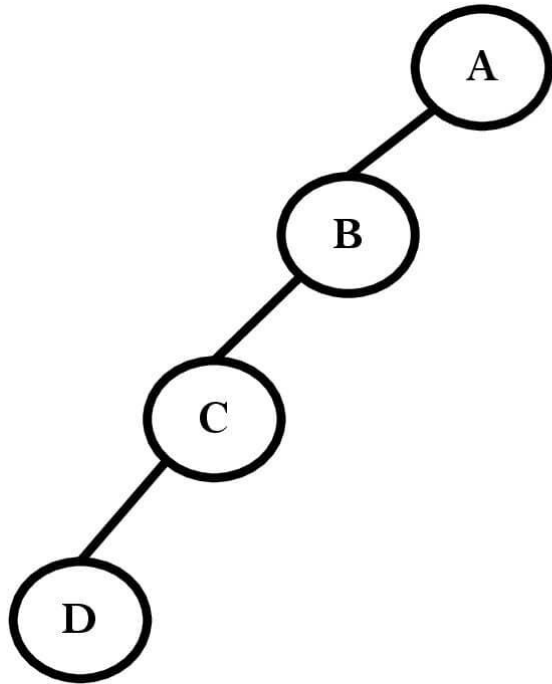
Binary Trees

- Any tree can be transformed into binary tree by left child-right sibling representation
- A tree can never be empty but binary tree can be
- In binary tree a node can have atmost 2 children, whereas in case of a tree, a node may have any number of children

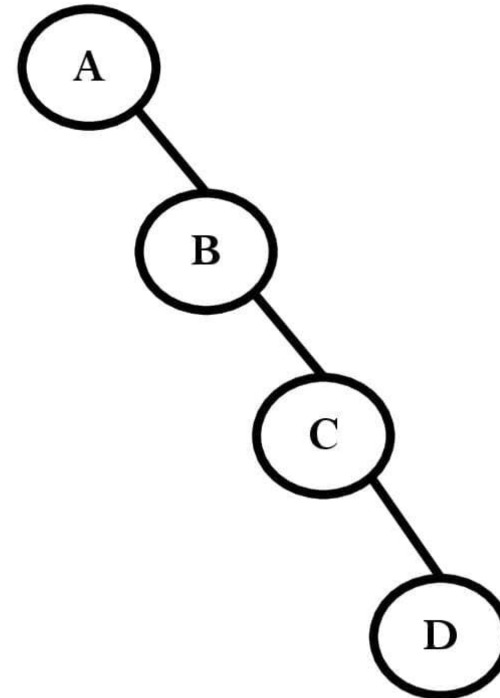
Different Binary Trees

- Skewed Binary Tree
  - Left Skewed Binary Tree
  - Right Skewed Binary Tree
- Complete Binary Tree
- Full/Strictly Binary Tree

Skewed Binary Trees



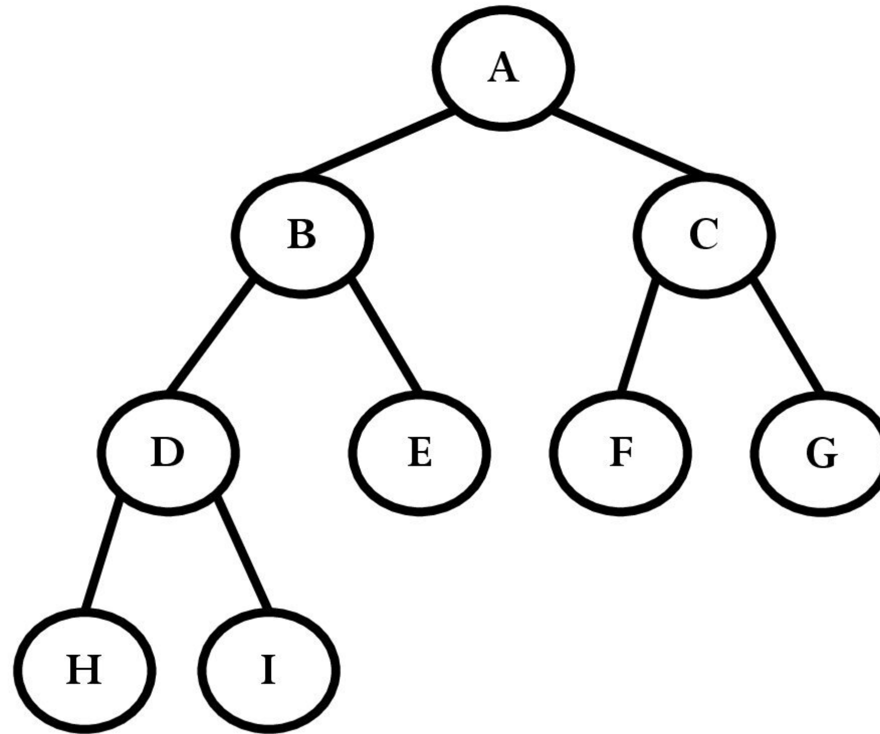
Left Skewed Binary Tree



Right Skewed Binary Tree

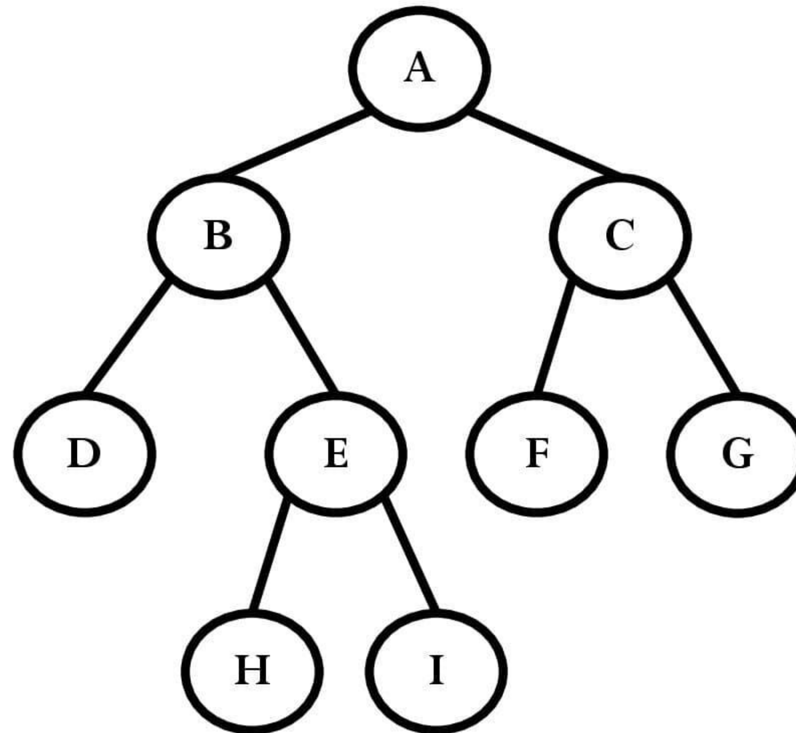
Complete Binary Trees

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and the last level has all its nodes to the left side



Full/Strictly Binary Trees

- Every node other than the leaves has two children



Binary Tree Representations

1. Array/Sequential/Linear Representation
2. Linked Representation

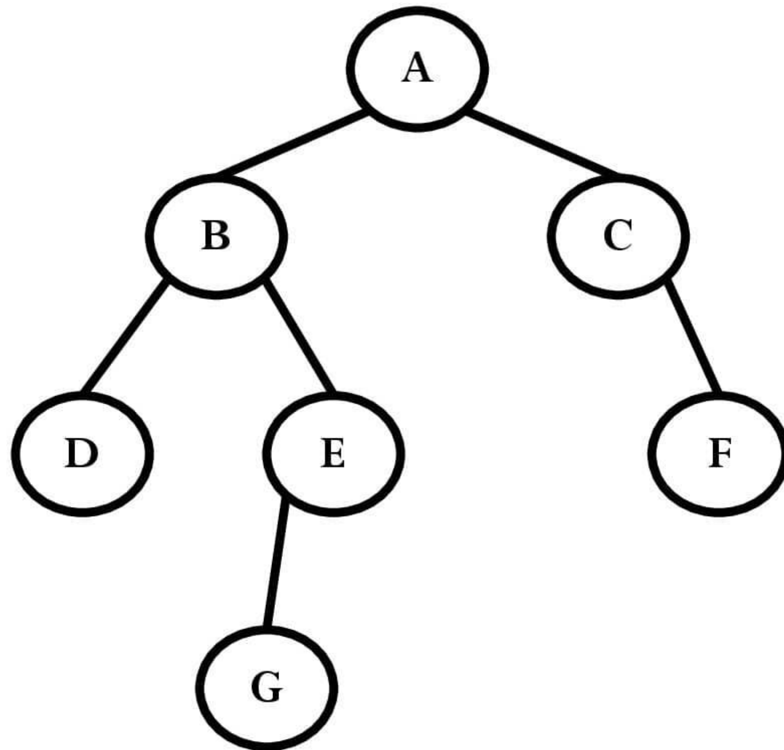
Binary Tree -Array Representation

- It is a sequential representation
- Use a 1-D array to store the nodes
- Block of memory for an array is allocated before storing the actual tree in it.
- Once the memory is allocated, the size of tree is restricted as permitted by the memory.
- Nodes are stored level by level, starting from zero level
- The root node stored in first memory location

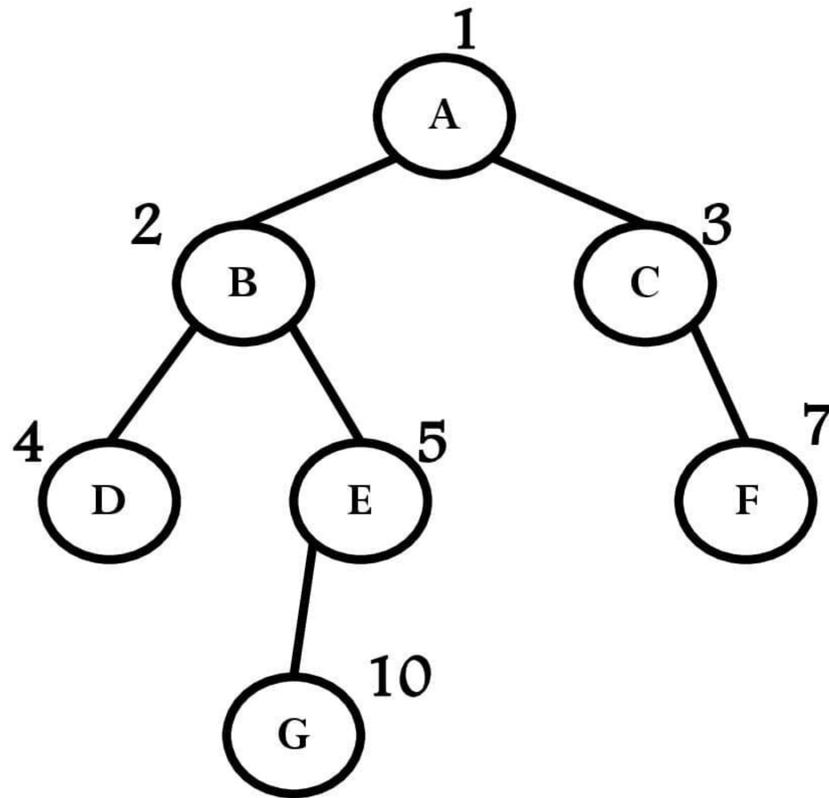
Binary Tree ~Array Representation

- Determine the locations of the parent, left child, and right child of any node i in the binary tree($1 \le i \le n$)
  - **parent(i)**
    - If i !=1, then parent(i)= floor(i/2)
    - if i =1, i is at root and has no parent
  - **left_child(i)**
    - left_child(i) = 2i   If $2i \le n$
    - No left child   Otherwise
  - **right_child(i)**
    - right_child(i) = 2i+1   If $2i+1 \le n$
    - No right child   Otherwise

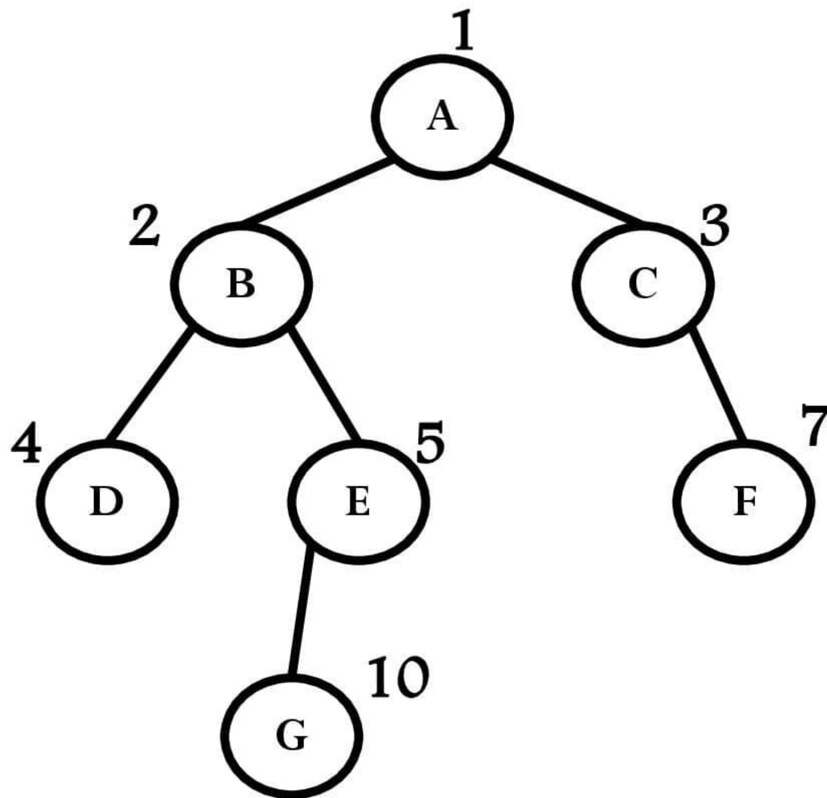
Binary Tree ~Array Representations



Binary Tree ~Array Representations

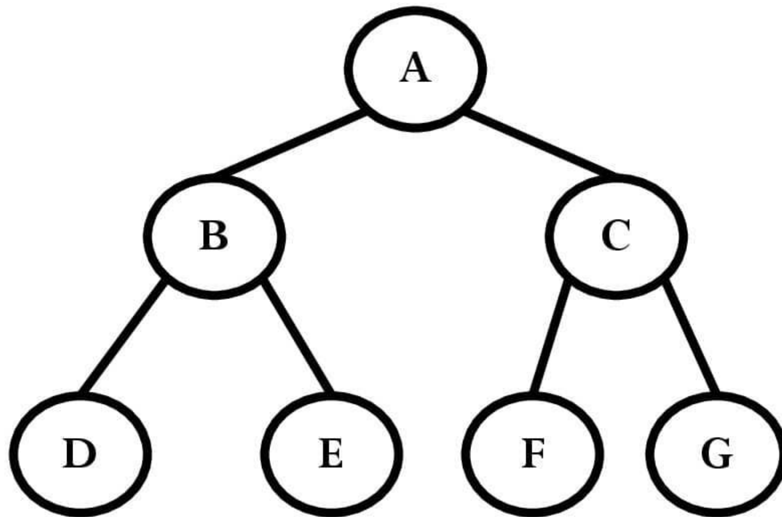


Binary Tree ~Array Representations

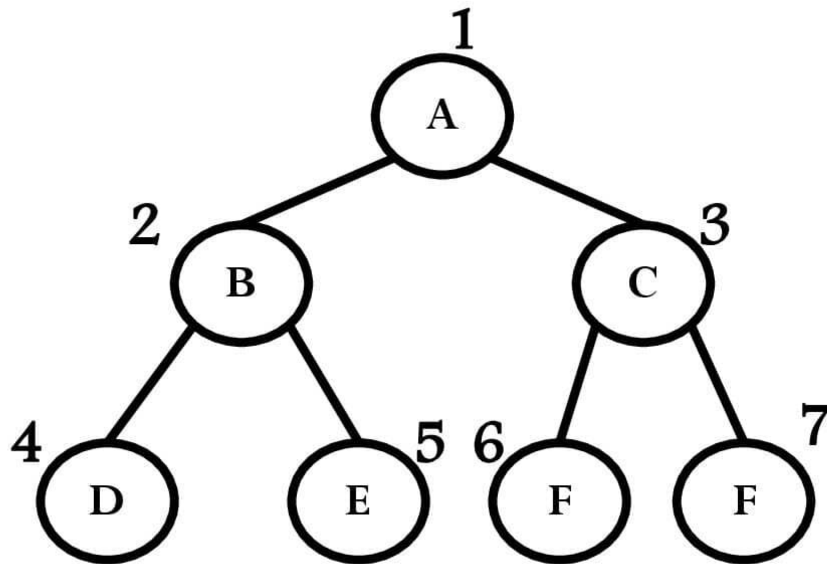


Value	Index
A	1
B	2
C	3
D	4
E	5
	6
F	7
	8
	9
G	10

Binary Tree ~Array Representations

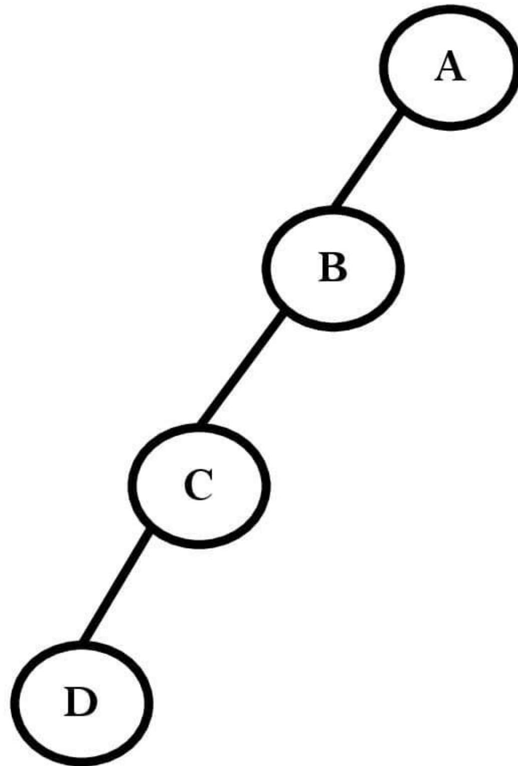


Binary Tree ~Array Representations

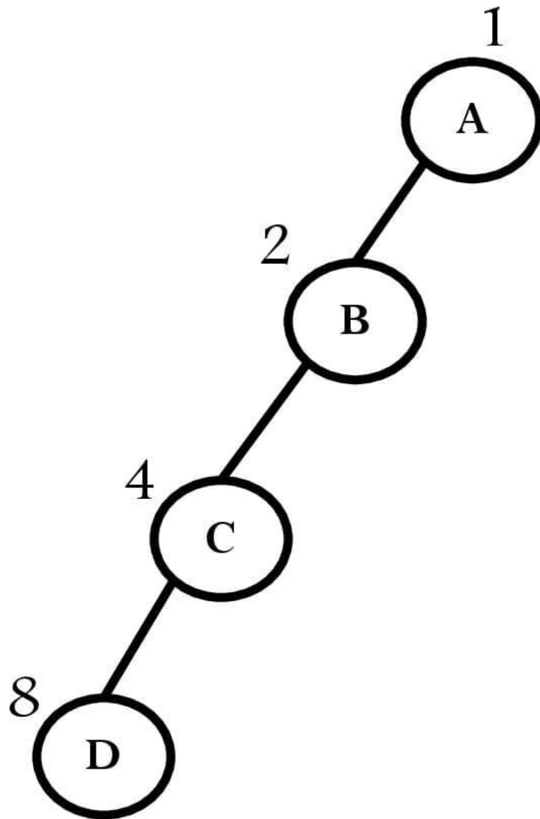


A	1
B	2
C	3
D	4
E	5
F	6
G	7

Binary Tree ~Array Representations



Binary Tree ~Array Representations



Value	Index
A	1
B	2
	3
C	4
	5
	6
	7
D	8

CS KTU LECTURES

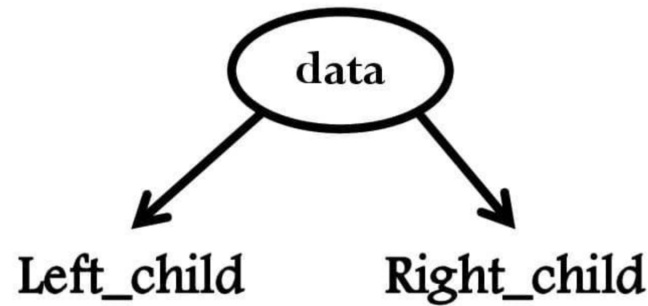
Binary Tree ~Array Representations

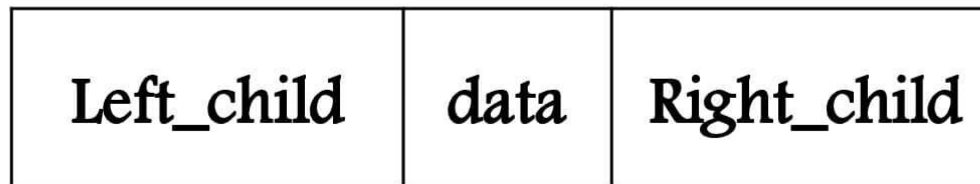
- **Advantages**
  - Any node can be accessed from any other node by calculating the index.
  - Data are stored without any pointer
  - Programming languages, where dynamic memory allocation is not possible(like BASIC, FROTRAN), array representation is only possible
  - Good for Complete Binary tree.
  - Searching is fast

Binary Tree ~Array Representations

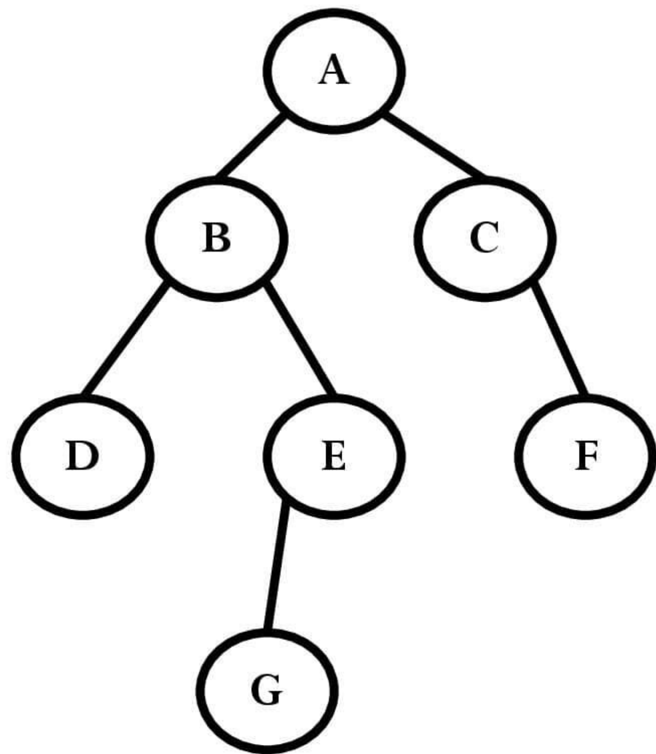
- **Disadvantages**
  - Good for Complete Binary tree. If it is not a complete binary tree then, a lot of memory is wasted
  - Dynamic memory allocation is not possible

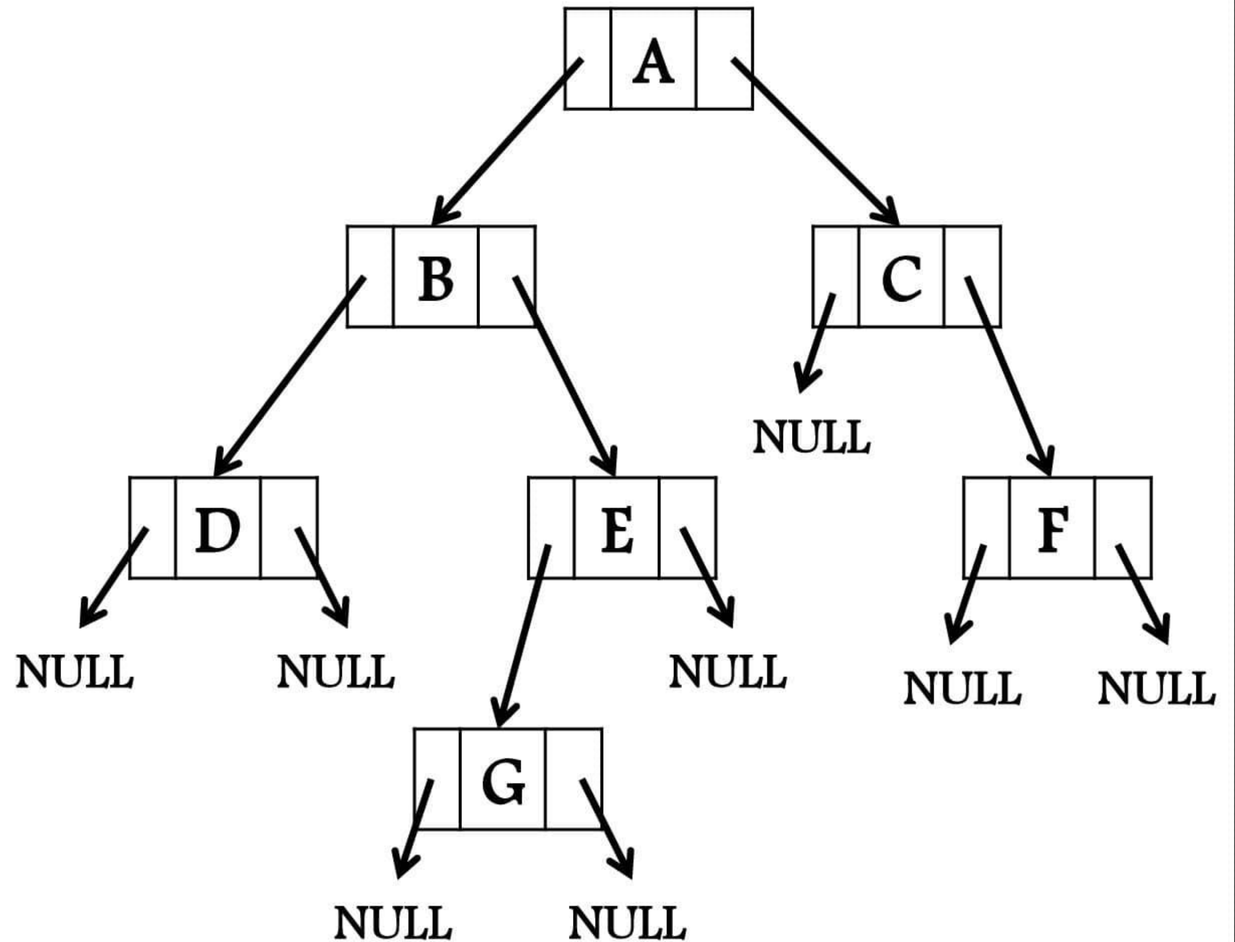
Binary Tree ~Linked Representations



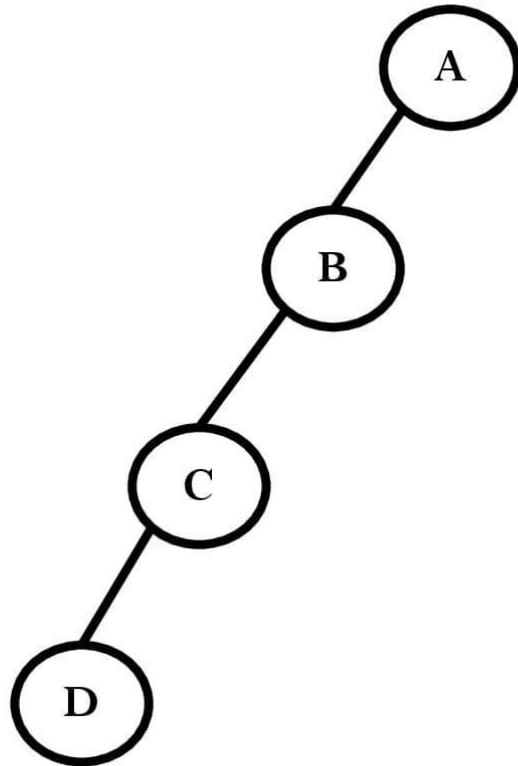


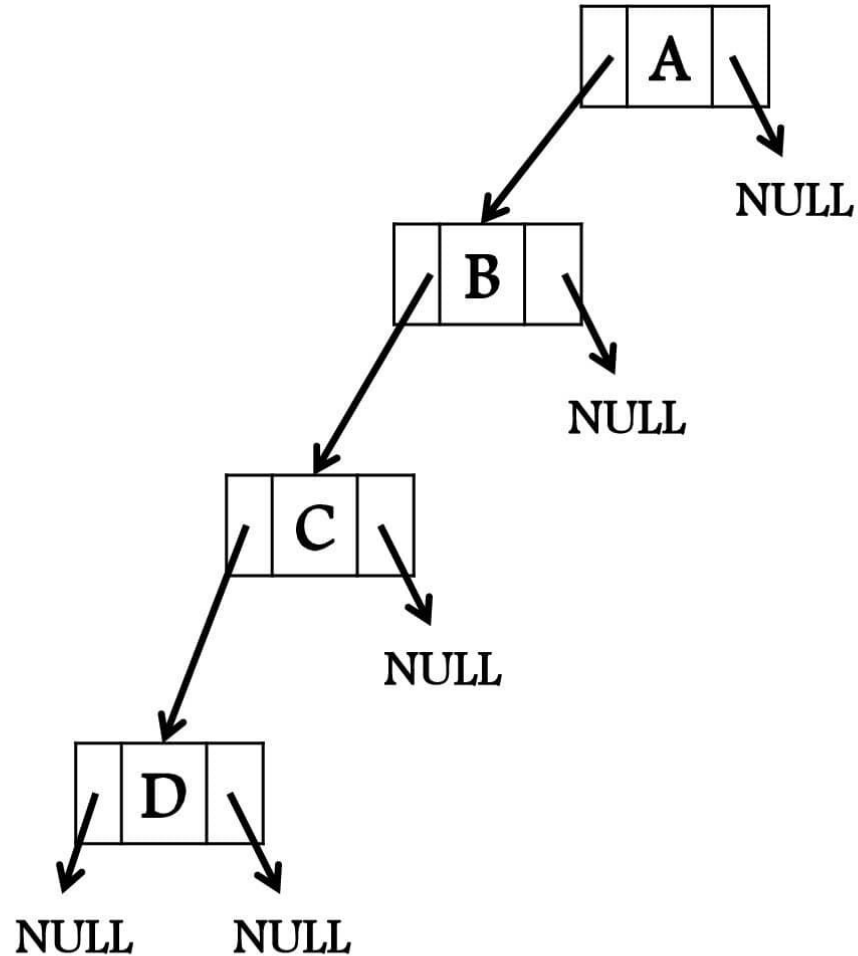
Binary Tree ~Linked Representations





Binary Tree –Linked Representations



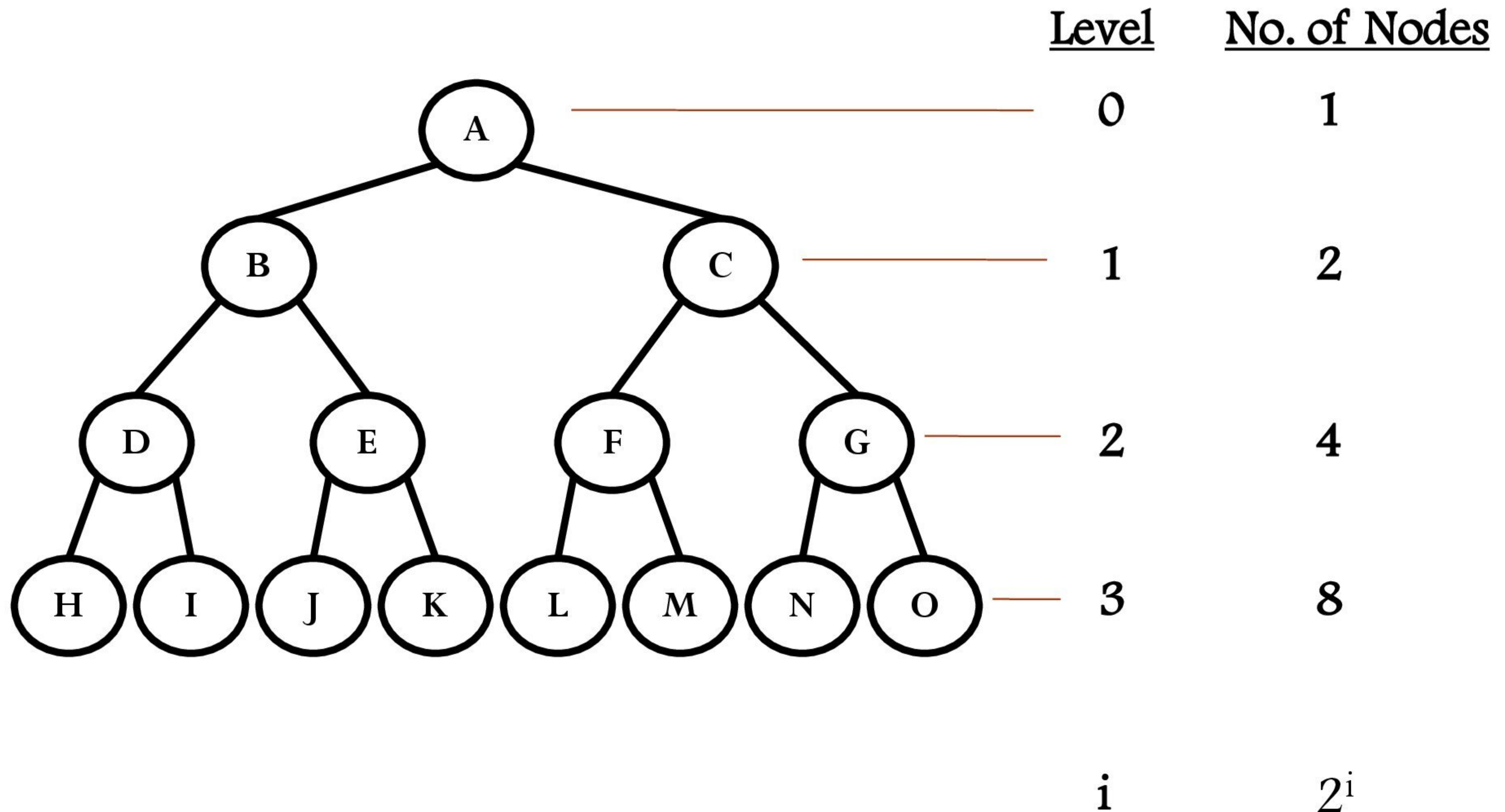


Binary Tree -Linked Representations

- **Advantages**
  - Dynamic memory allocation is possible

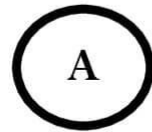
Binary Tree ~Properties

- The maximum number of nodes on level i of a binary tree is 2^i, i>=0



Binary Tree ~Properties

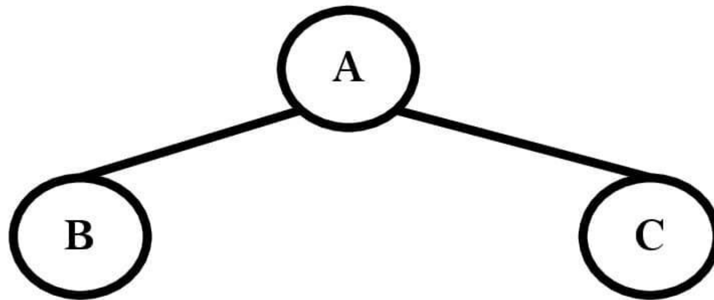
- The maximum number of nodes in a binary tree of height h is $2^{h+1}-1$, $h \geq 0$



Height	No. of Nodes
0	1

Binary Tree -Properties

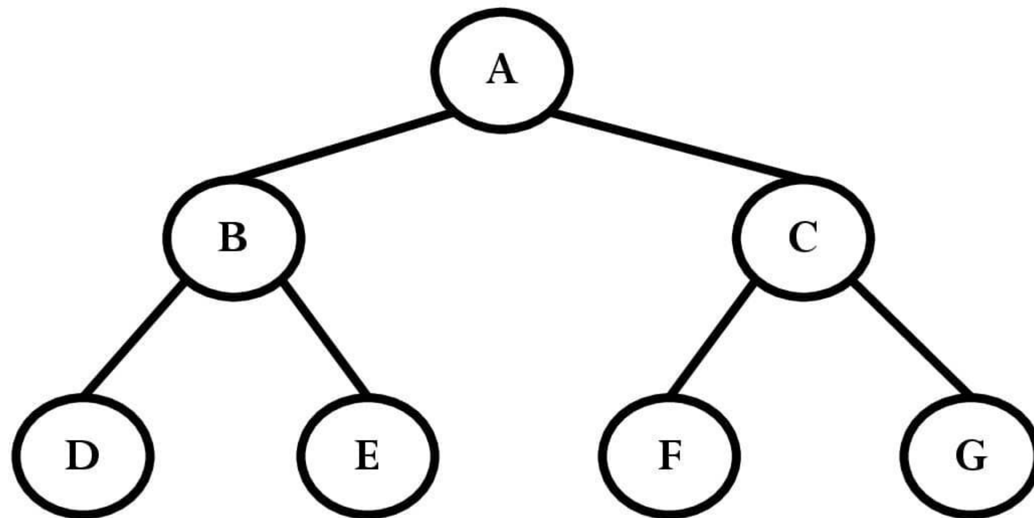
- The maximum number of nodes in a binary tree of height h is $2^{h+1}-1$, $h \geq 0$



Height	No. of Nodes
0	1
1	3

Binary Tree ~Properties

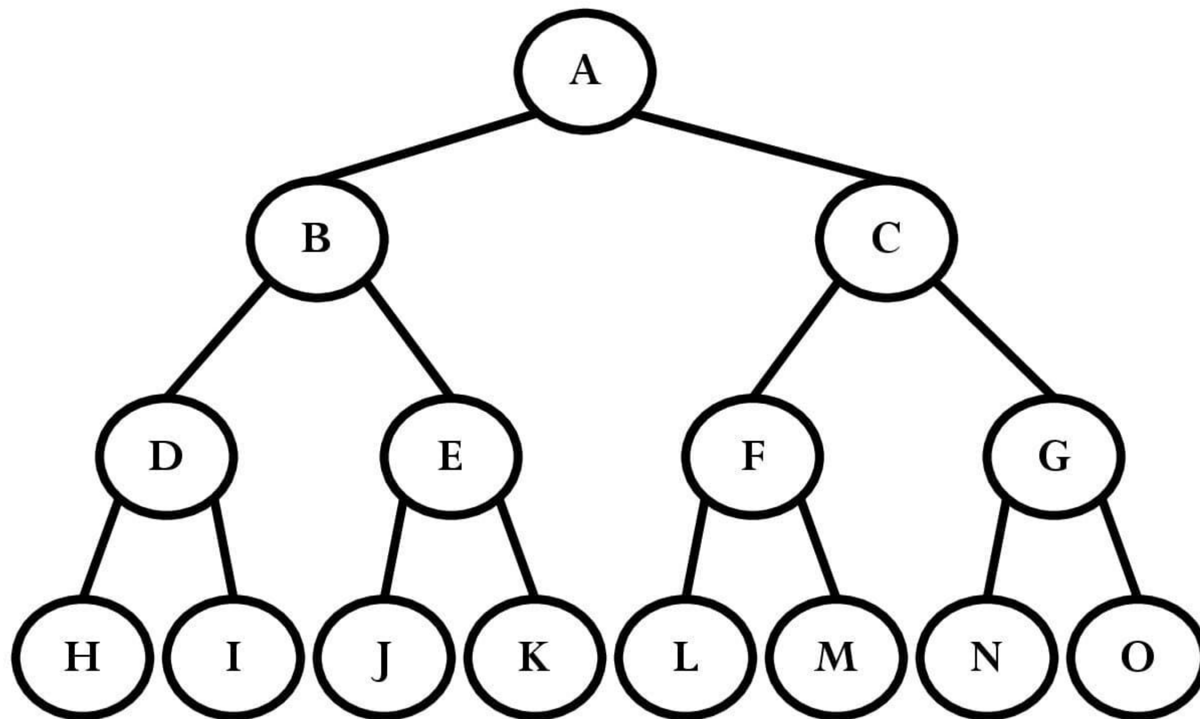
- The maximum number of nodes in a binary tree of height h is $2^{h+1}-1$, $h \geq 0$



Height	No. of Nodes
0	1
1	3
2	7

Binary Tree ~Properties

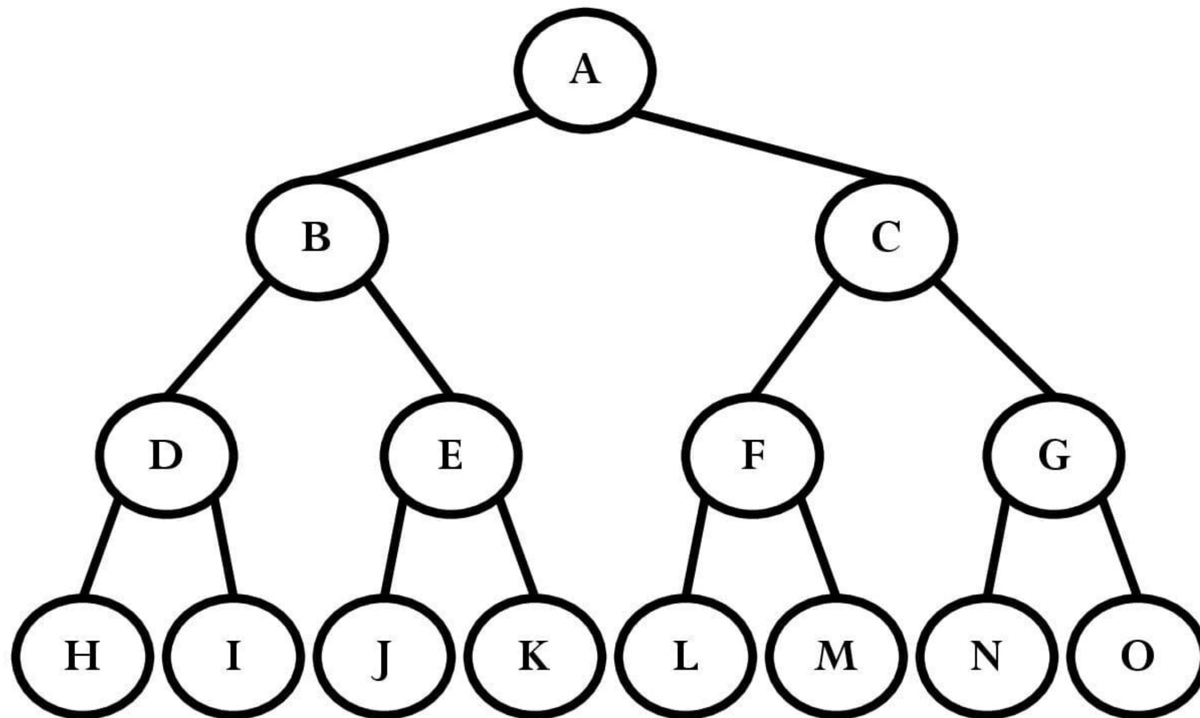
- The maximum number of nodes in a binary tree of height h is $2^{h+1}-1$, $h \geq 0$



Height	No. of Nodes
0	1
1	3
2	7
3	15

Binary Tree ~Properties

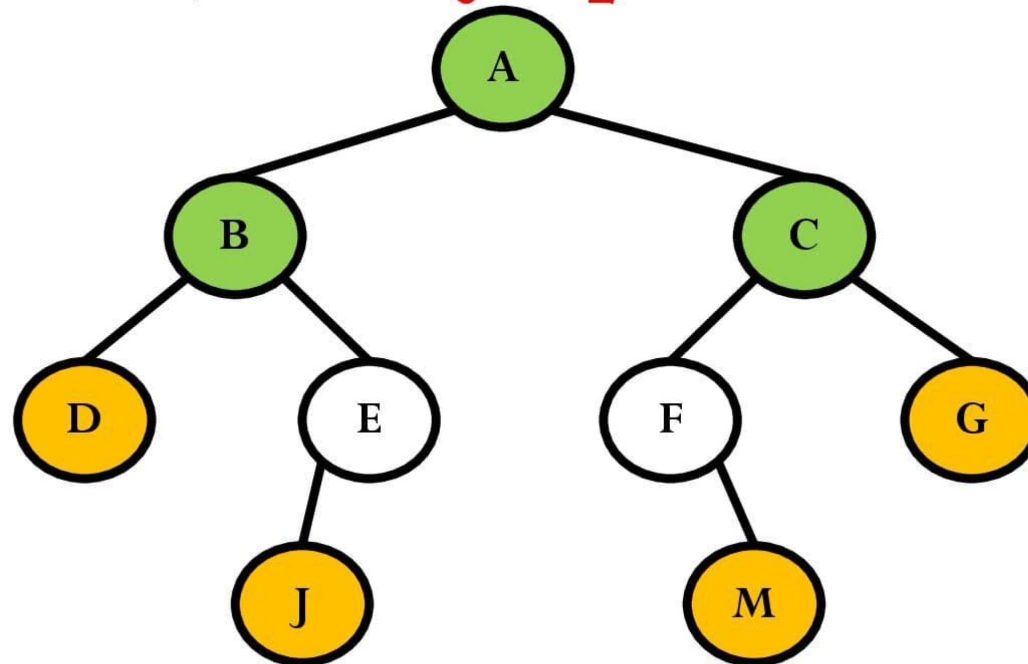
- The maximum number of nodes in a binary tree of height h is $2^{h+1}-1$, $h \geq 0$



Height	No. of Nodes
0	1
1	3
2	7
3	15
h	$2^{h+1}-1$

Binary Tree -Properties

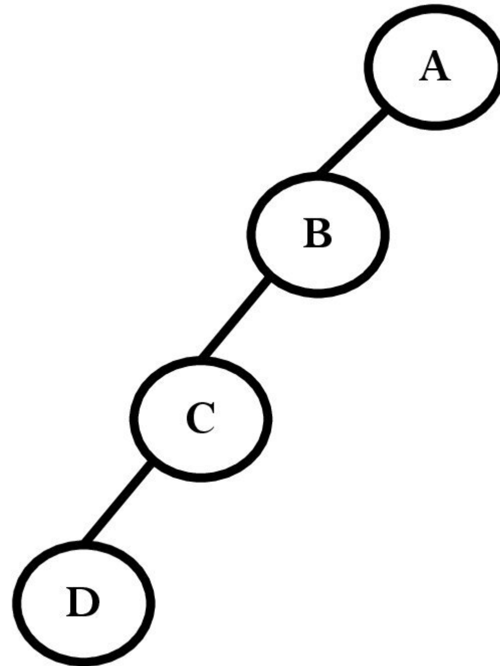
- For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0=n_2+1$



- Number of nodes with degree 2 = n_2 = 3
- Number of leaf nodes = n_0 = 4

Binary Tree ~Properties

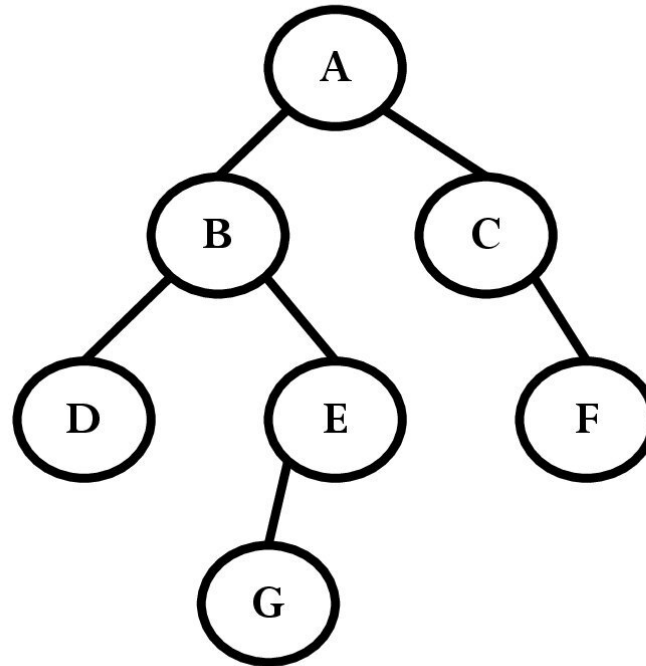
- Minimum number of nodes possible in a binary tree of height h is h+1



- Consider this skewed binary tree
- Here, height of the tree = 3
- Minimum number of nodes possible = 4

Binary Tree -Properties

- For any non empty binary tree, if **n** is the number of nodes and **e** is the number of edges then **n=e+1**

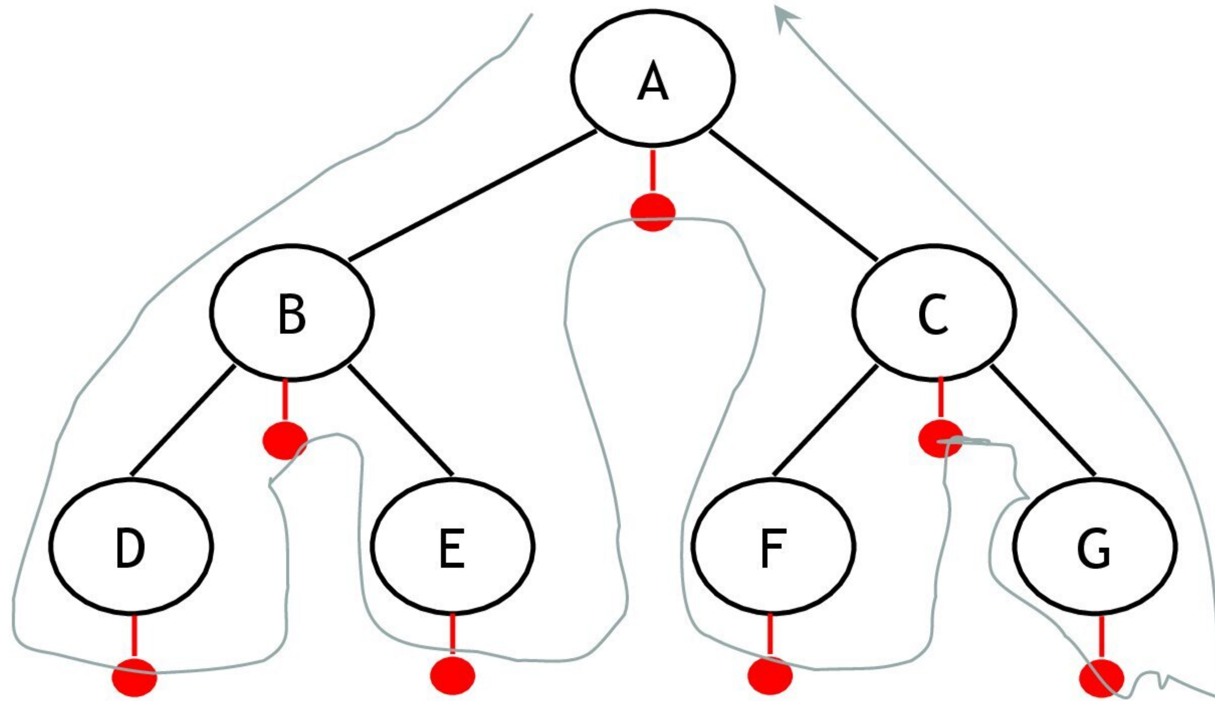


- Number of nodes = n = 7
- Number of edges = e = 6

Binary Tree Traversals

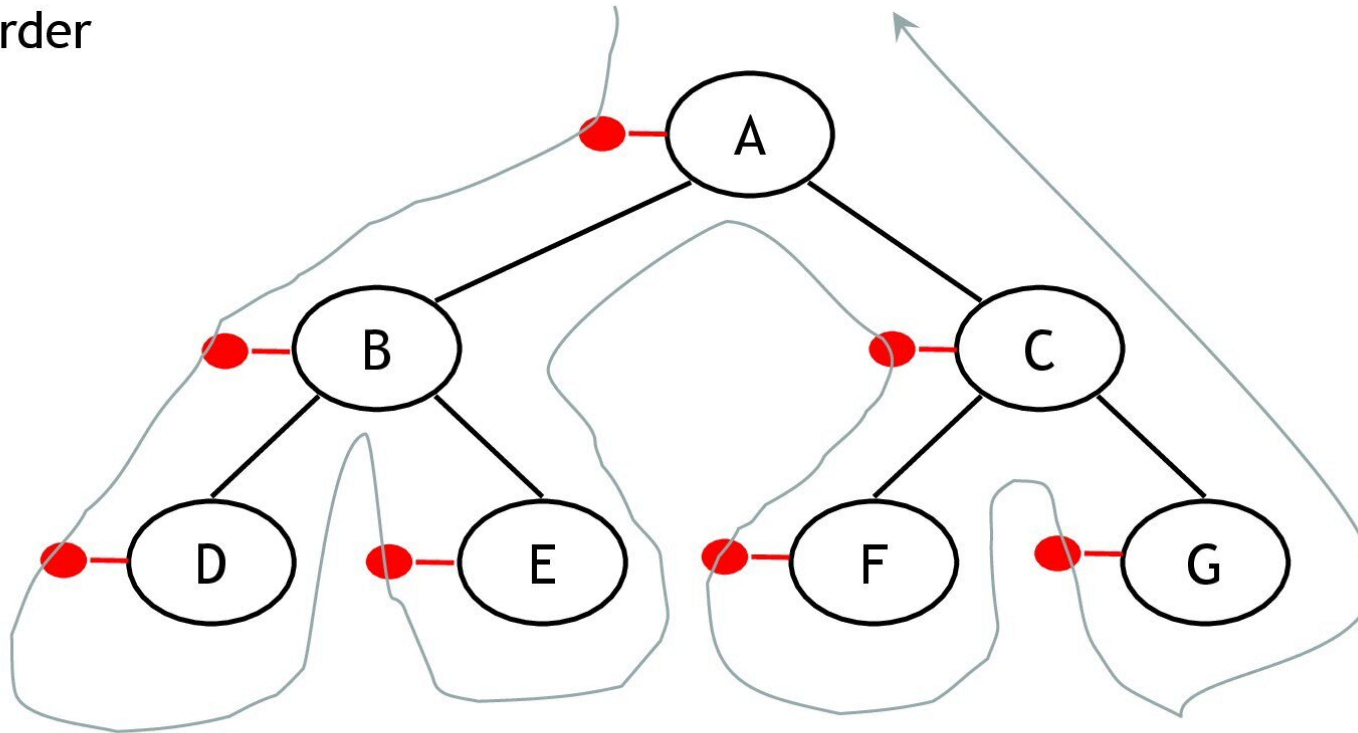
- Traversal means visit each node in the binary tree exactly once
- There are three commonly used tree traversals
  - Inoder Traversal
  - Preorder Traversal
  - Postorder Traversal

Inorder Traversal



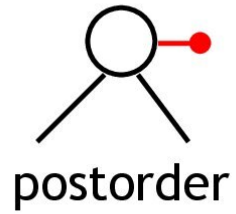
D B E A F C G

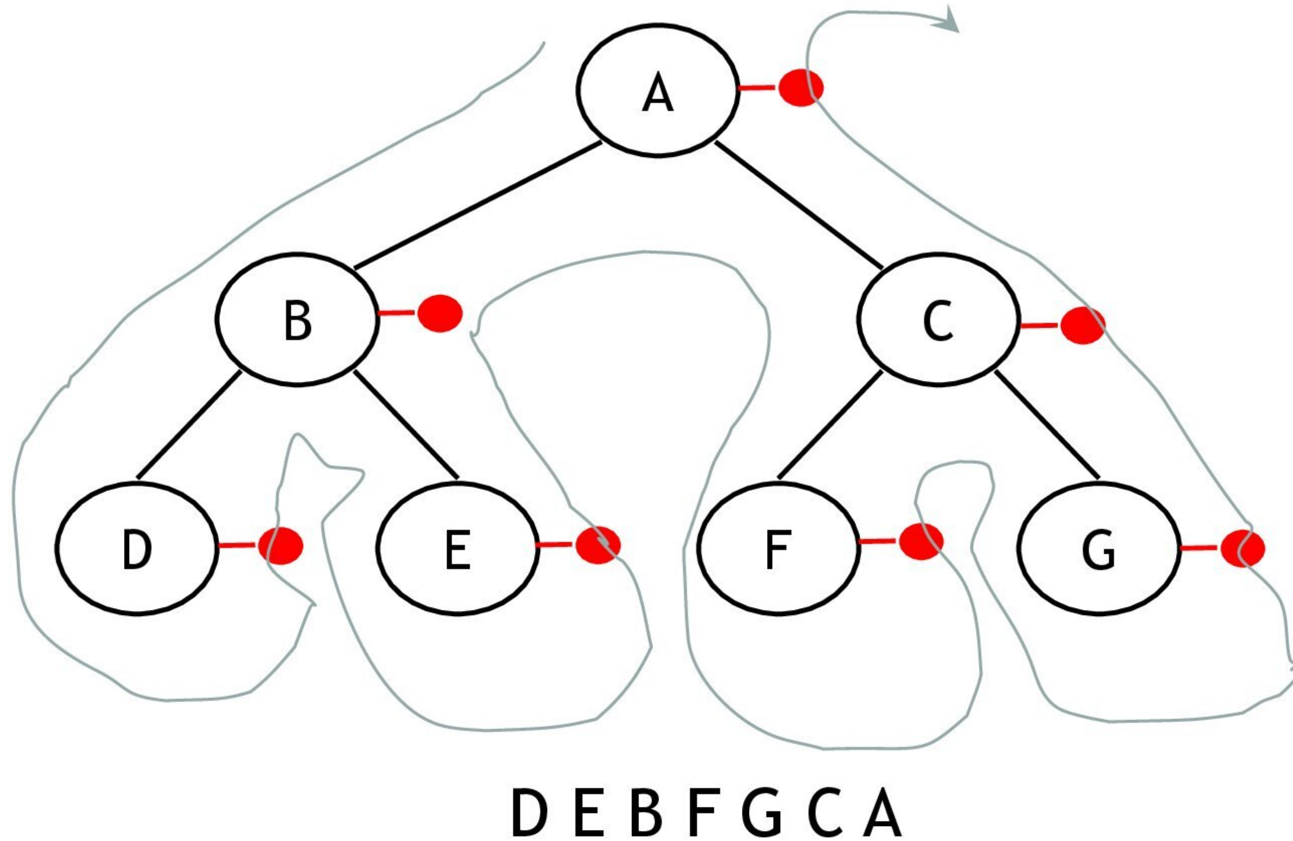
Preorder Traversal



A B D E C F G

Postorder Traversal





D E B F G C A

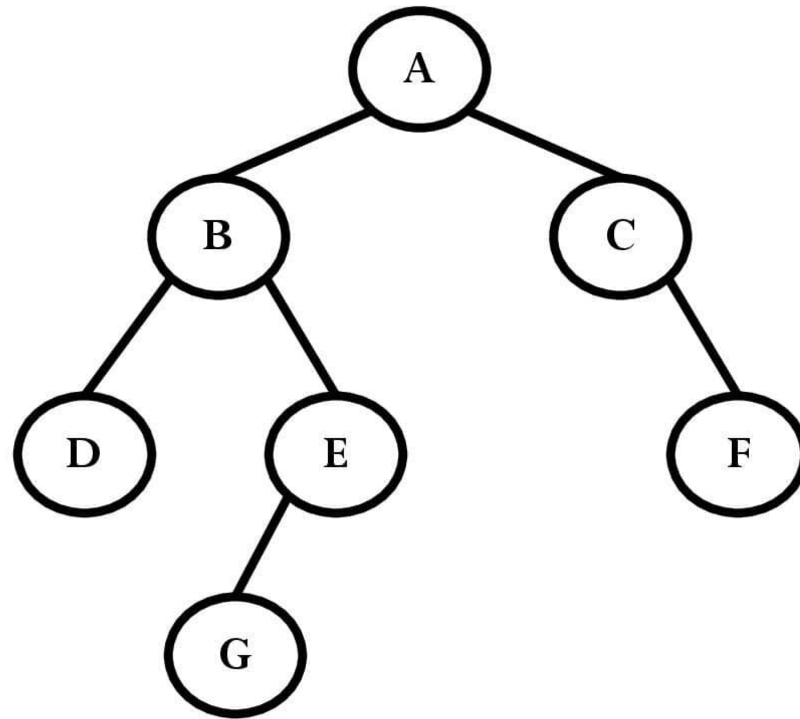
Inorder Traversal

Left → Root → Right

Algorithm Inorder(tree)

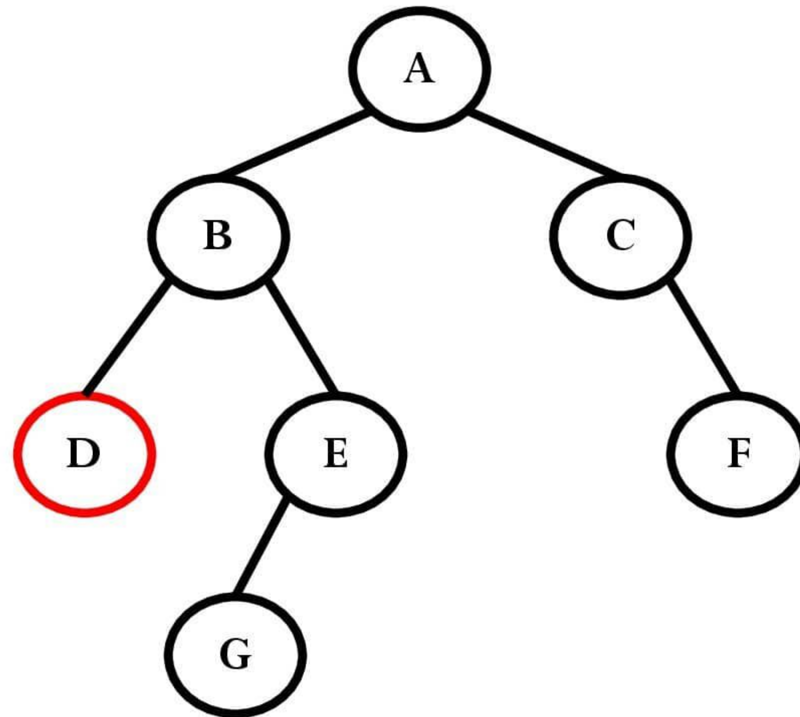
1. Traverse the left subtree. ie, call Inorder(tree→lChild)
2. Visit the tree
3. Traverse the right subtree. ie, call Inorder(tree→rChild)

Inorder Traversal



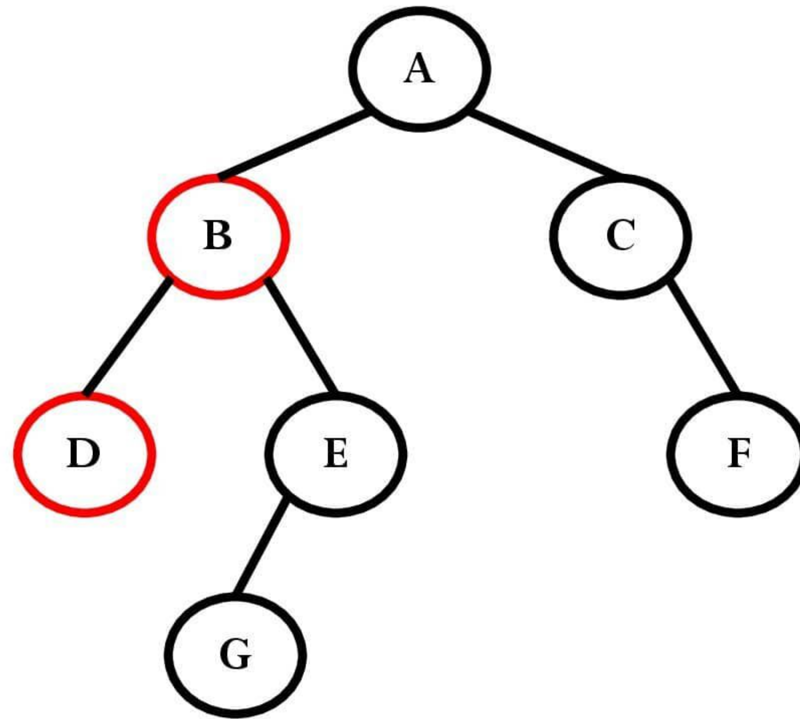
Inorder Traversal:

Inorder Traversal



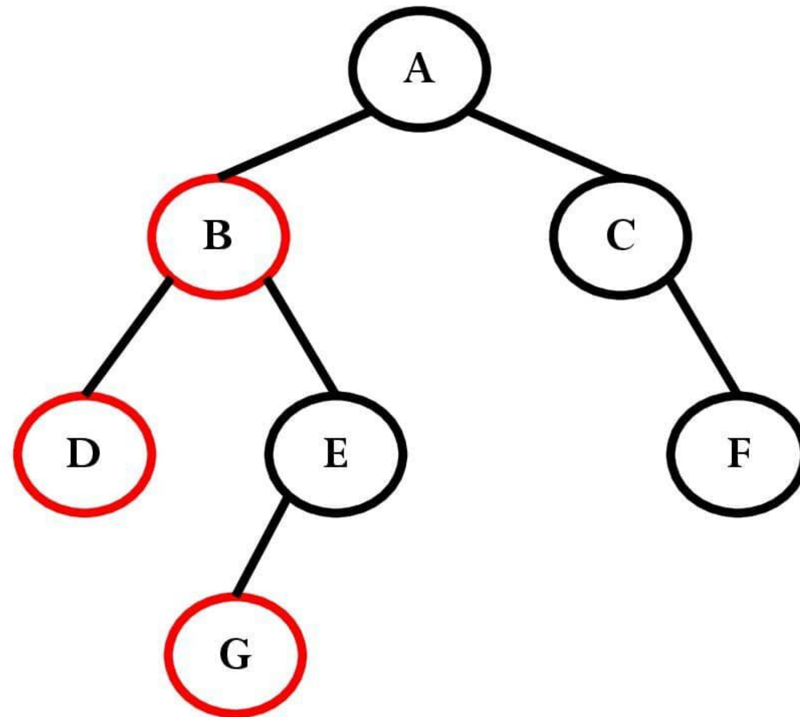
Inorder Traversal: D

Inorder Traversal



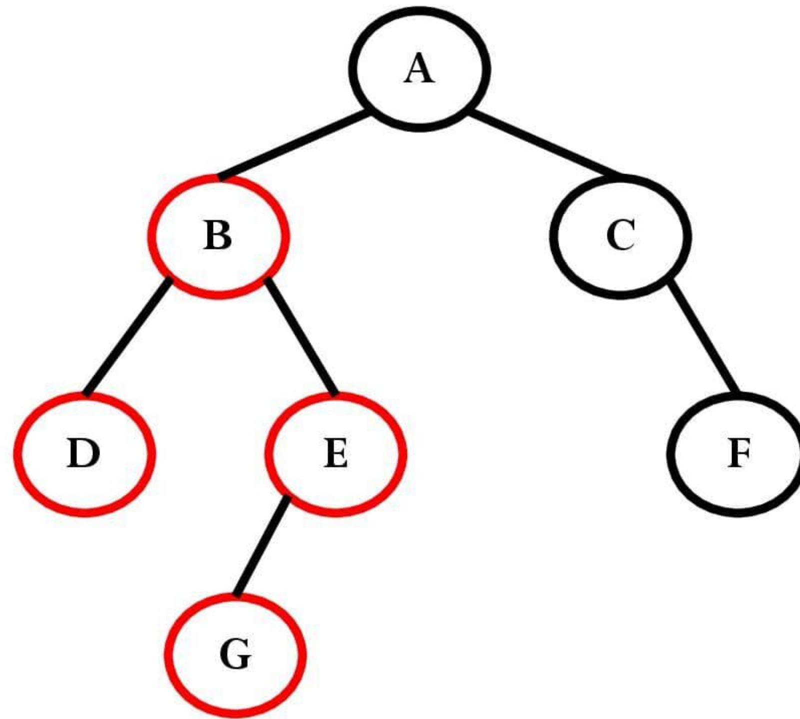
Inorder Traversal: D B

Inorder Traversal



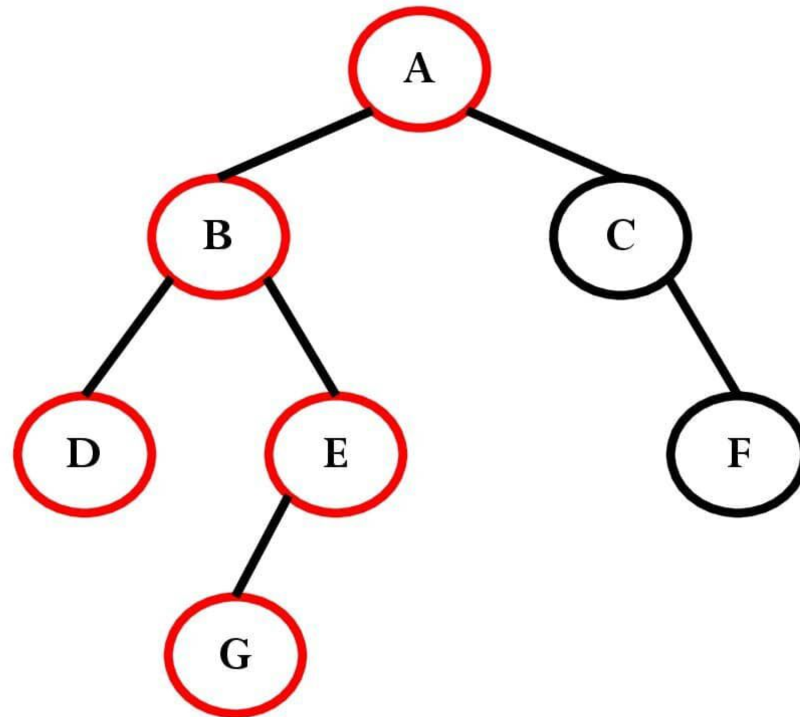
Inorder Traversal: D B G

Inorder Traversal



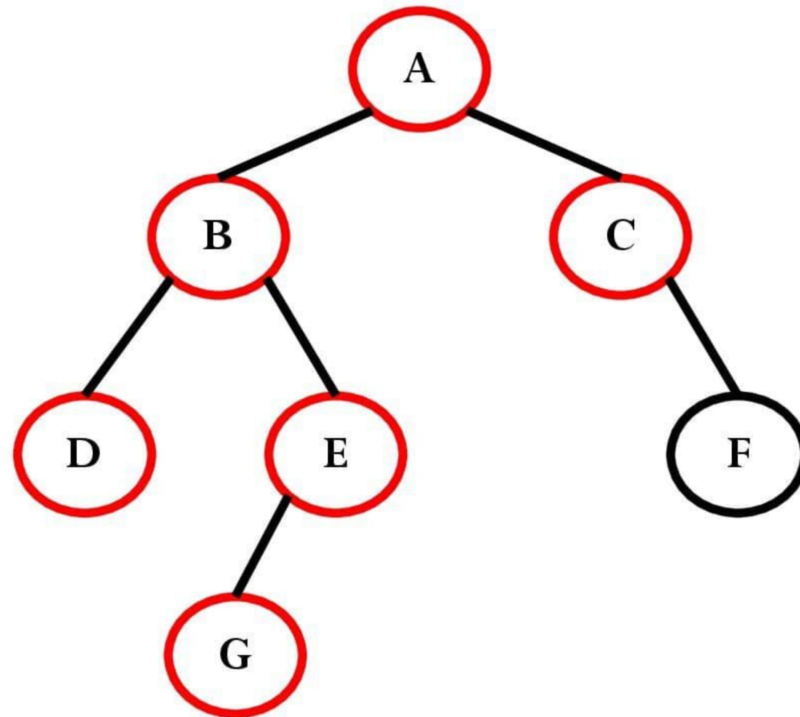
Inorder Traversal: D B G E

Inorder Traversal



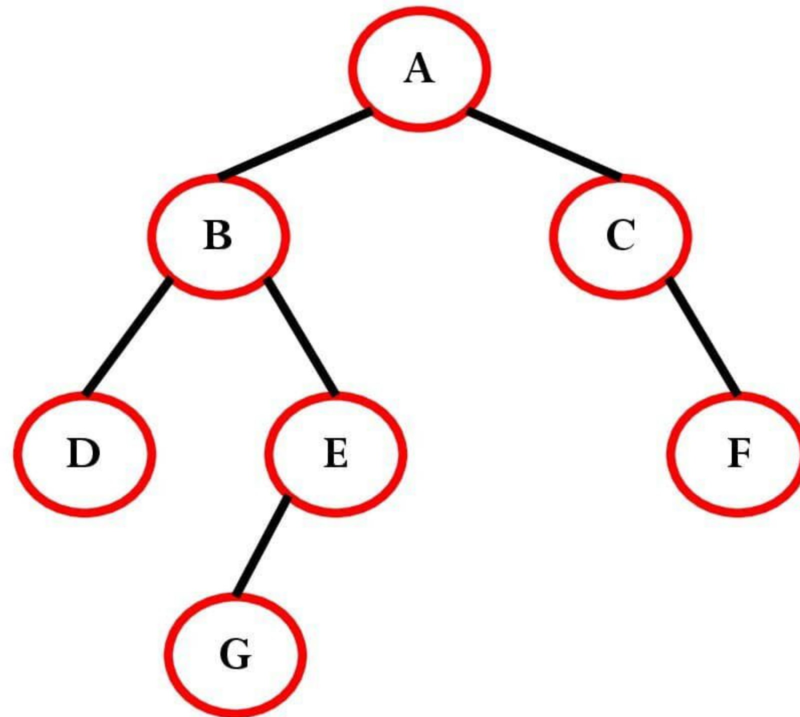
Inorder Traversal: D B G E A

Inorder Traversal



Inorder Traversal: D B G E A C

Inorder Traversal



Inorder Traversal: D B G E A C F

Inorder Traversal

```
struct node
{
        int data;
        struct node *lchild, *rchild;
}
void Inorder ( struct node * tree)
{
    if (tree != NULL)
    {
            Inorder(tree->lchild);
            printf("%d",tree->data);
            Inorder(tree->rchild);
    }
}
```

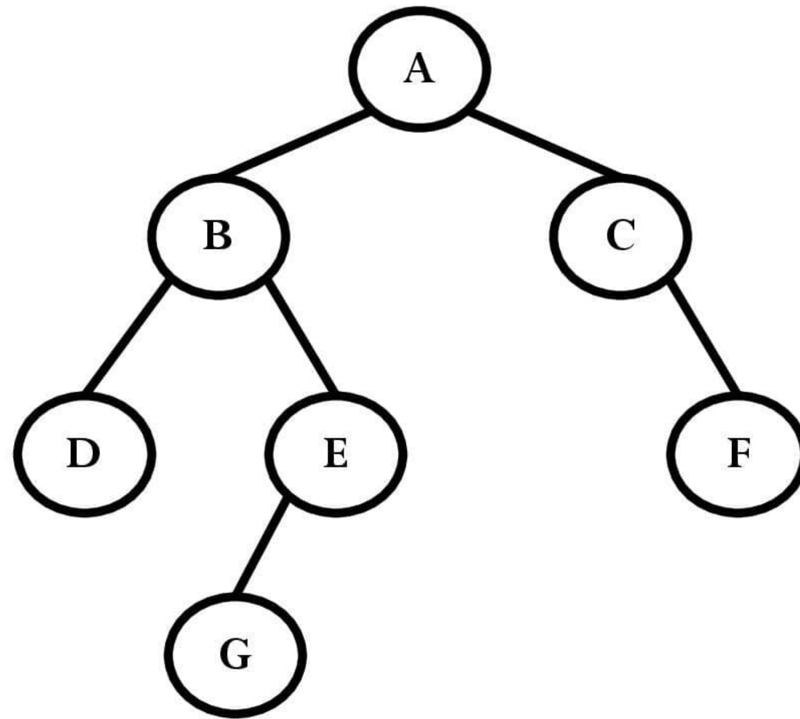
Preorder Traversal

Root → Left → Right

Algorithm Preorder(tree)

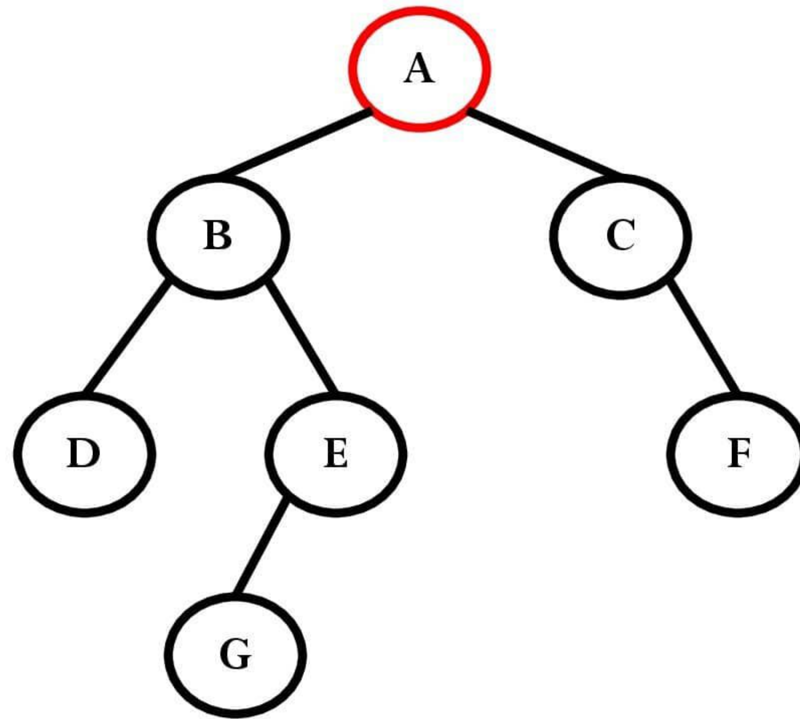
1. Visit the tree
2. Traverse the left subtree. ie, call Preorder(tree→lChild)
3. Traverse the right subtree. ie, call Preorder(tree→rChild)

Preorder Traversal



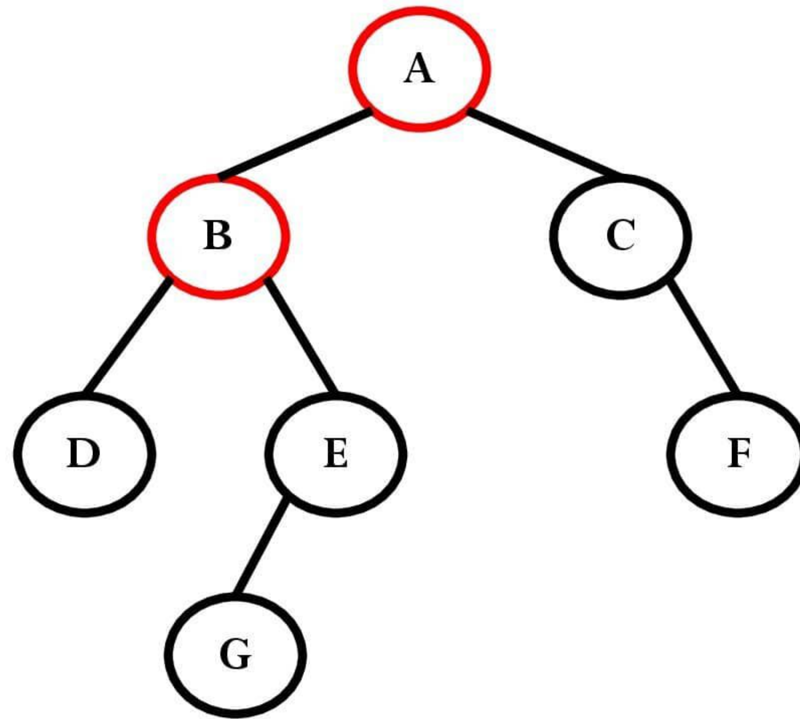
Preorder Traversal:

Preorder Traversal



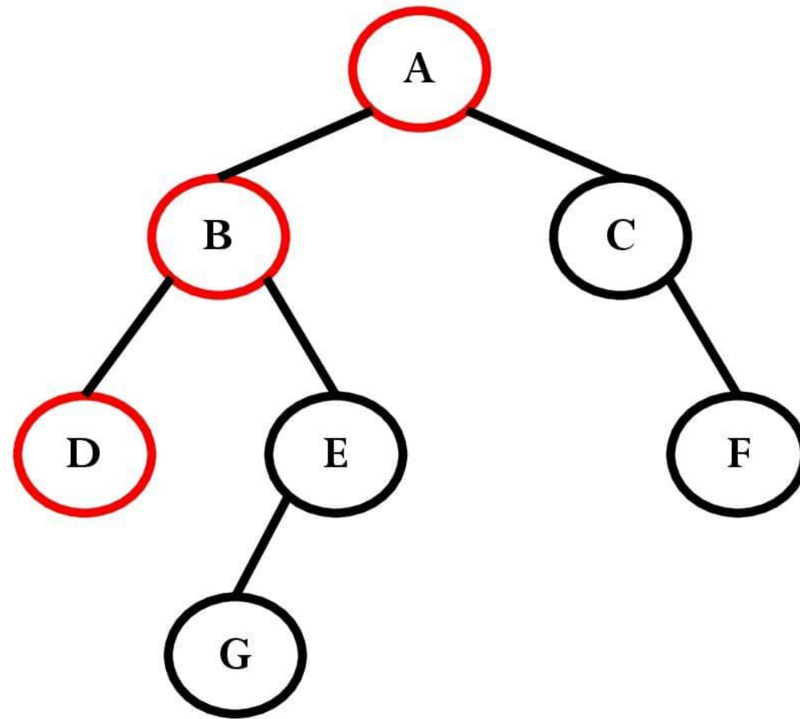
Preorder Traversal: A

Preorder Traversal



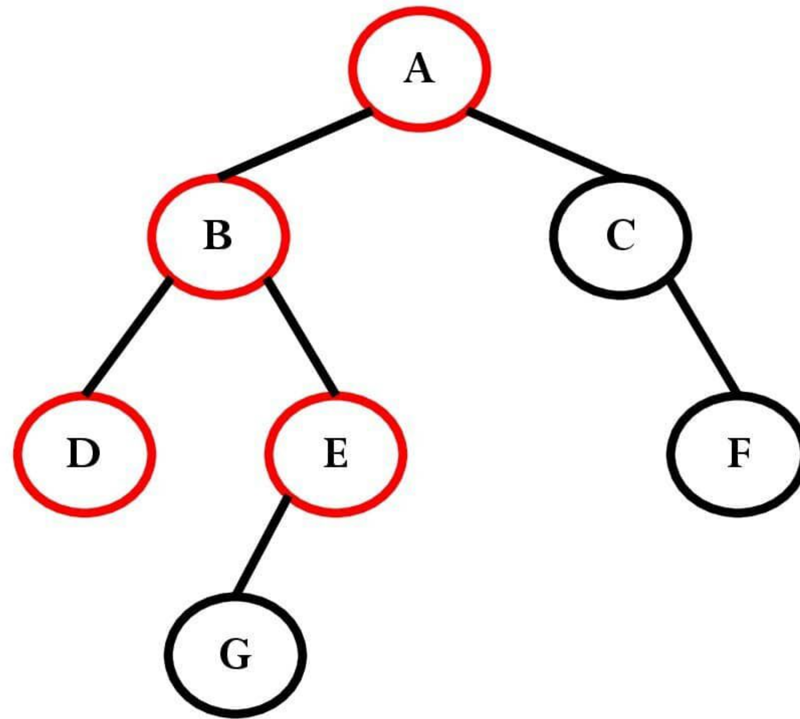
Preorder Traversal: A B

Preorder Traversal



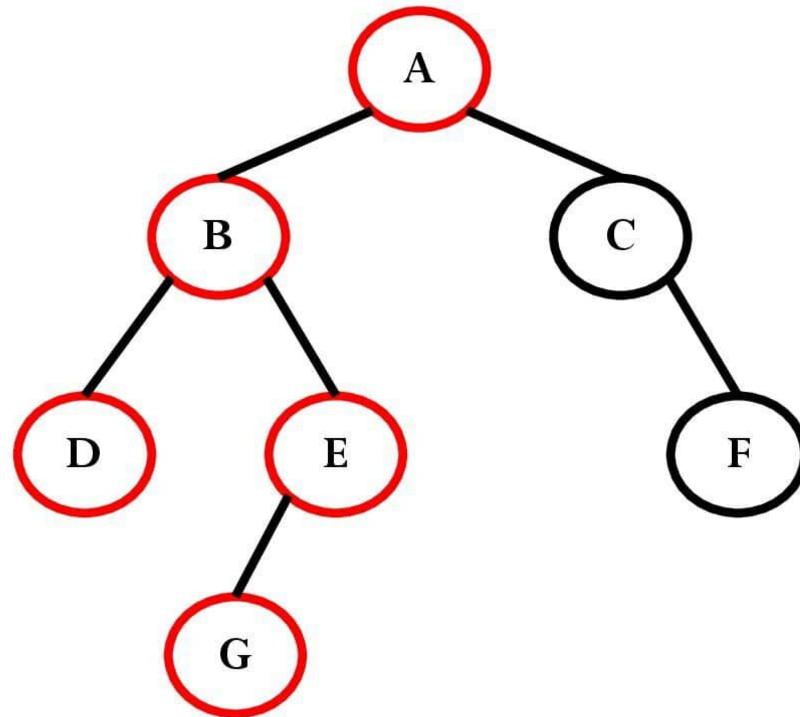
Preorder Traversal: A B D

Preorder Traversal



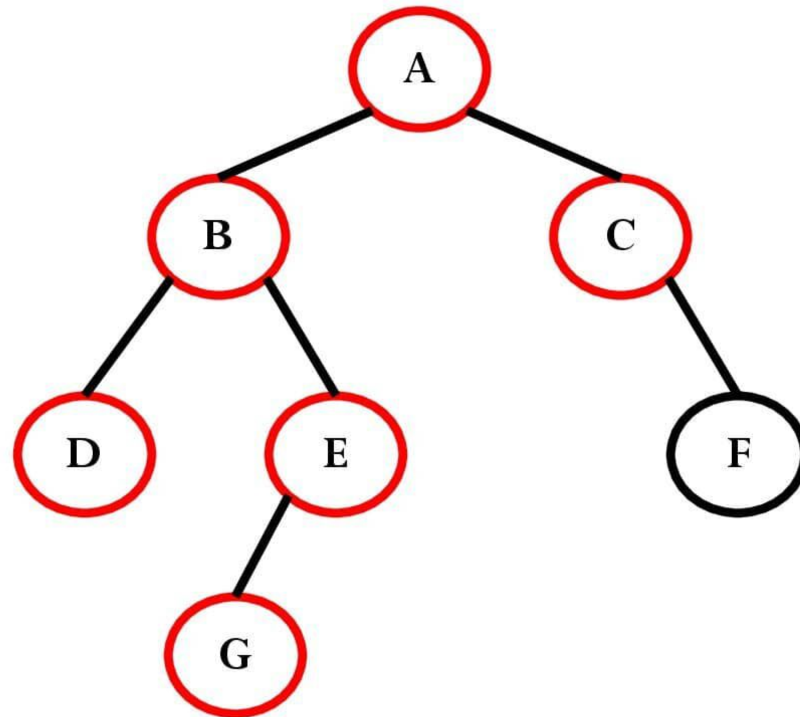
Preorder Traversal: A B D E

Preorder Traversal



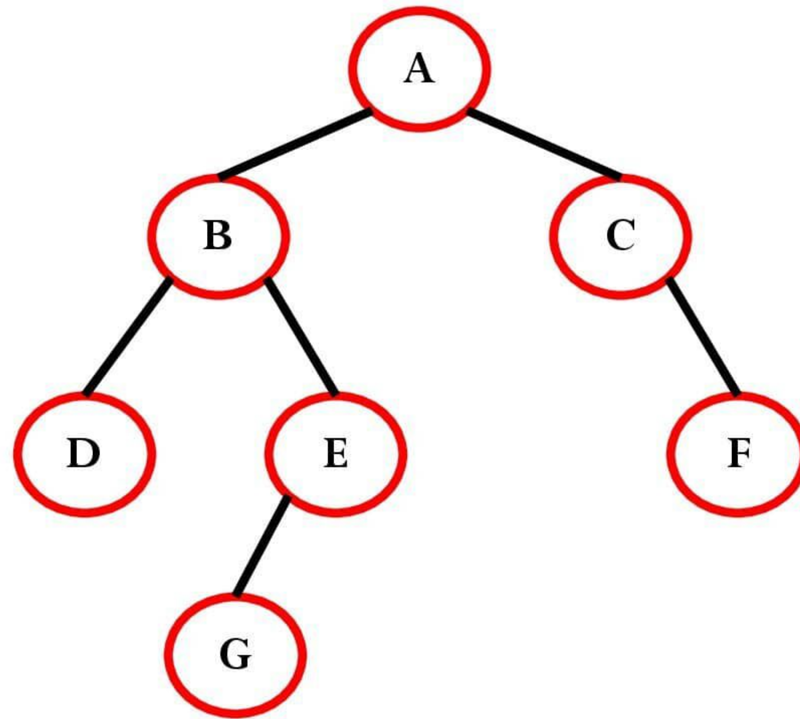
Preorder Traversal: A B D E G

Preorder Traversal



Preorder Traversal: A B D E G C

Preorder Traversal



Preorder Traversal: A B D E G C F

Preorder Traversal

```
struct node
{
        int data;
        struct node *lchild, *rchild;
}
void Preorder ( struct node * tree)
{
    if (tree != NULL)
    {
            printf("%d",tree->data);
            Preorder(tree->lchild);
            Preorder(tree->rchild);
    }
}
```

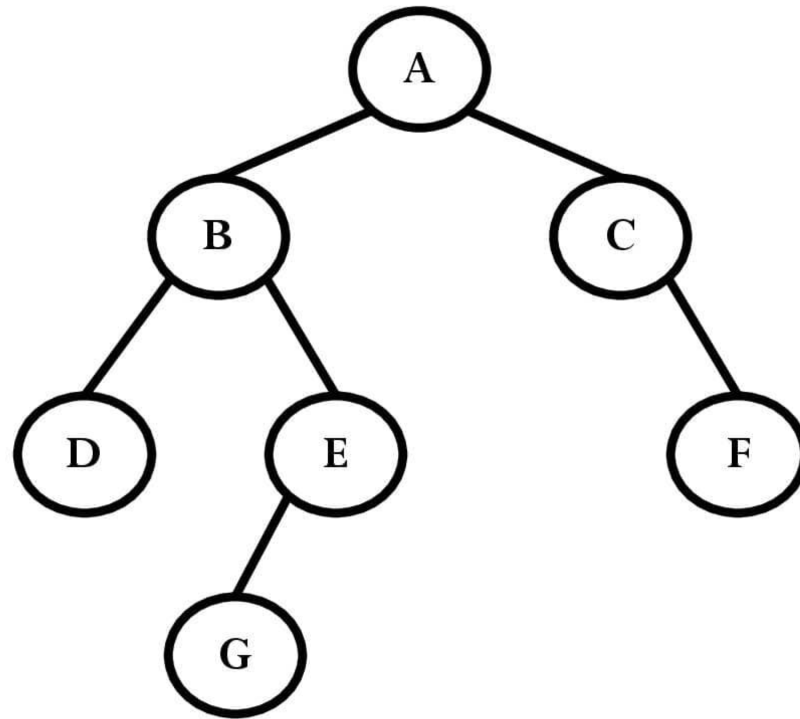
Postorder Traversal

Left → Right → Root

Algorithm Postorder()

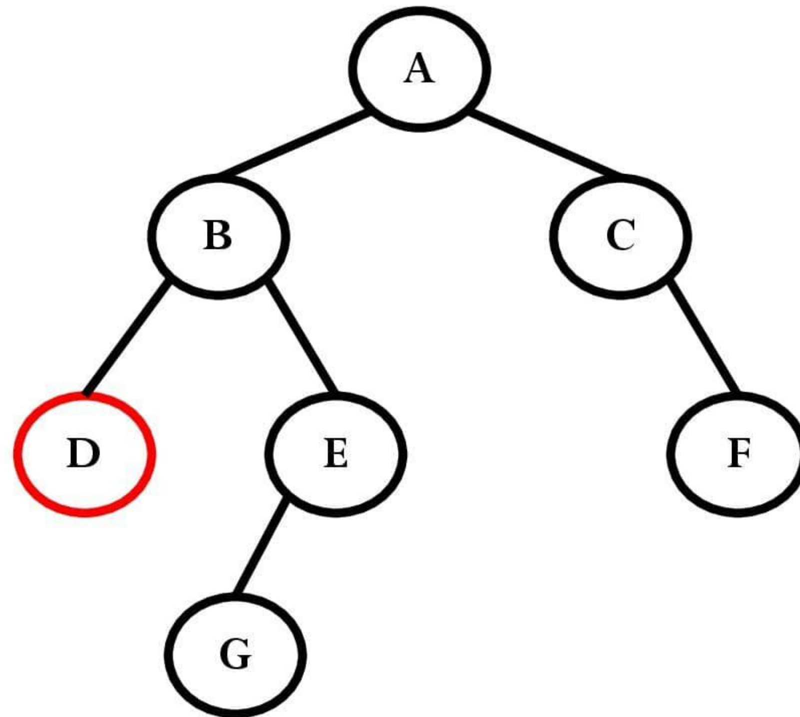
1. Traverse the left subtree. ie, call Inorder(tree→lChild)
2. Traverse the right subtree. ie, call Inorder(tree→rChild)
3. Visit the tree

Postorder Traversal



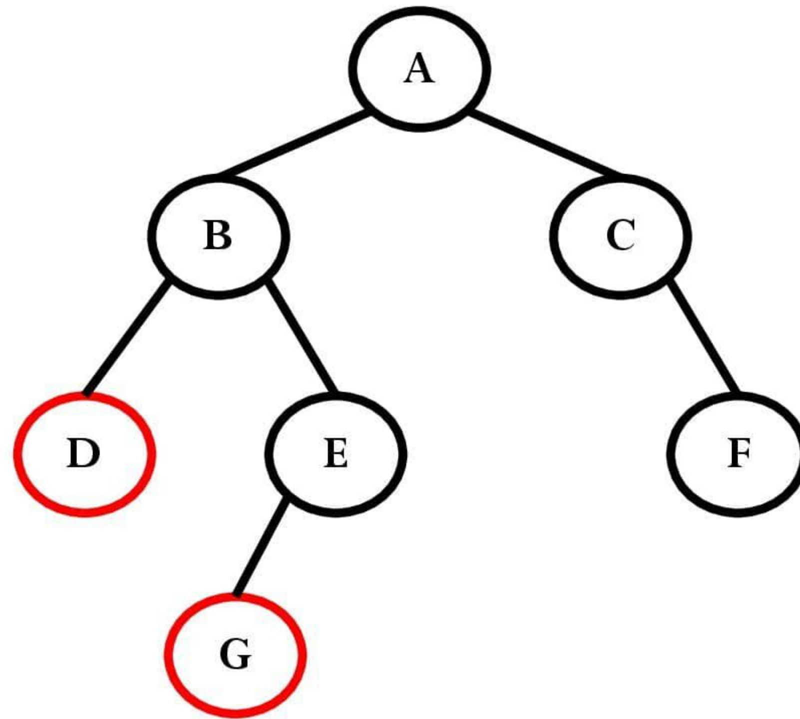
Postorder Traversal:

Postorder Traversal



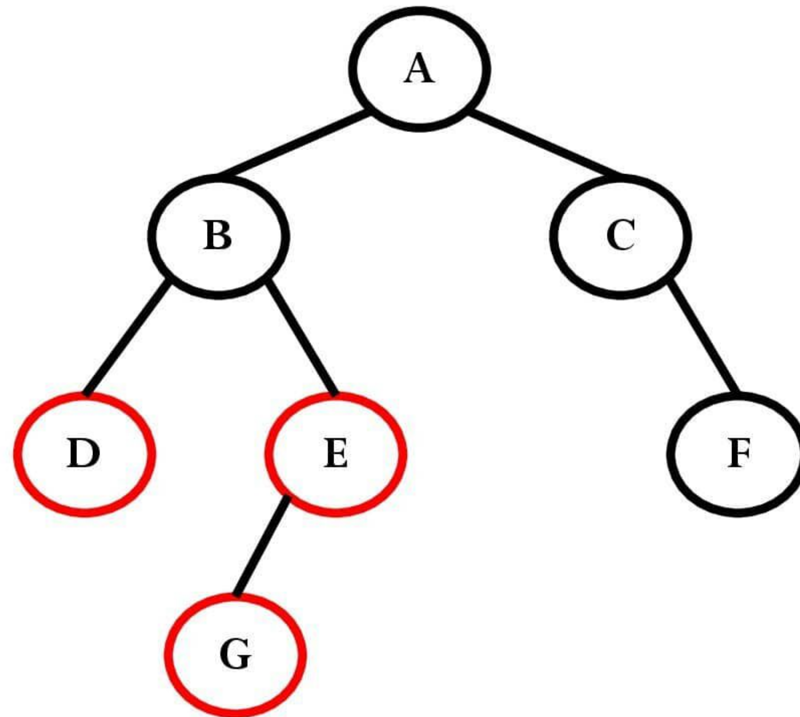
Postorder Traversal: D

Postorder Traversal



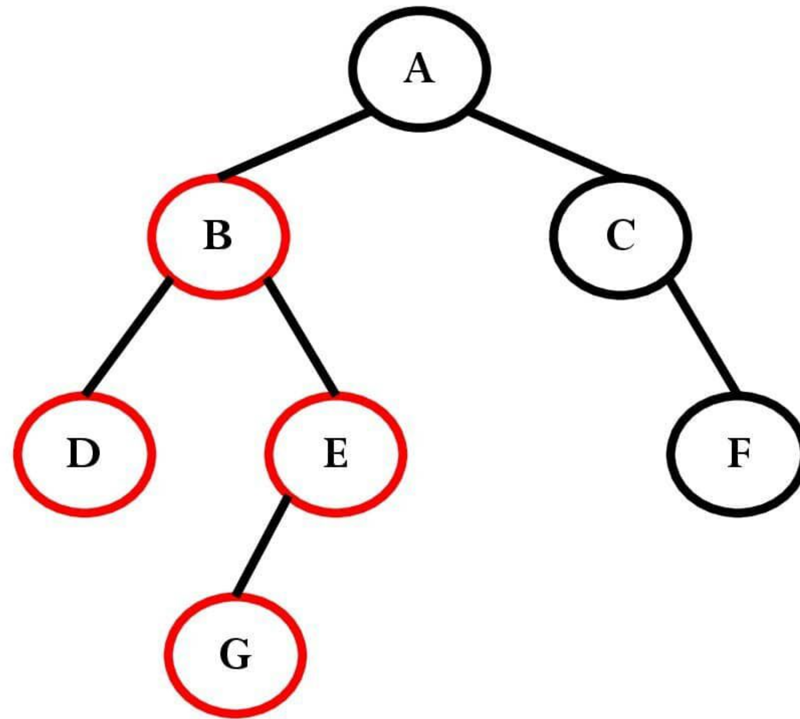
Postorder Traversal: D G

Postorder Traversal



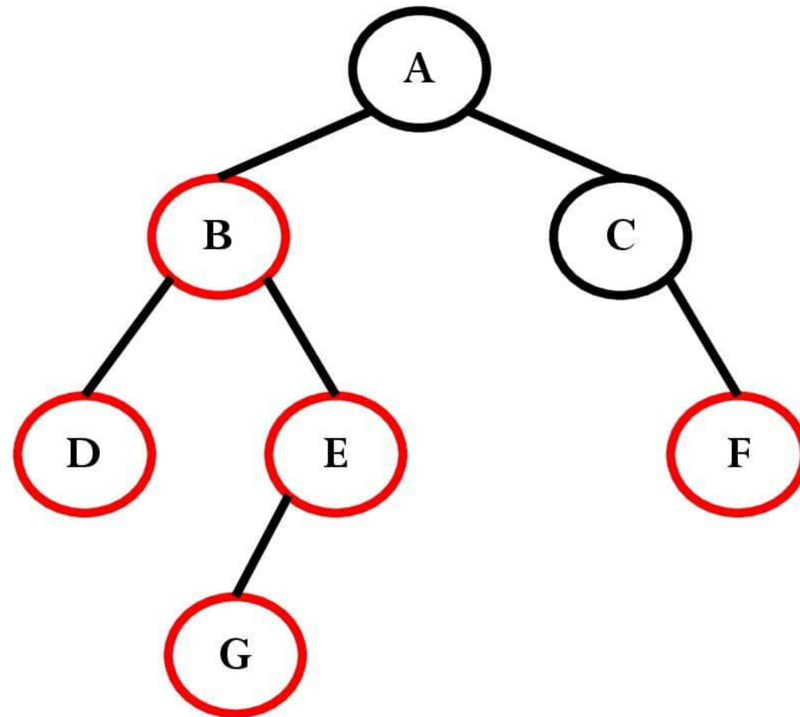
Postorder Traversal: D G E

Postorder Traversal



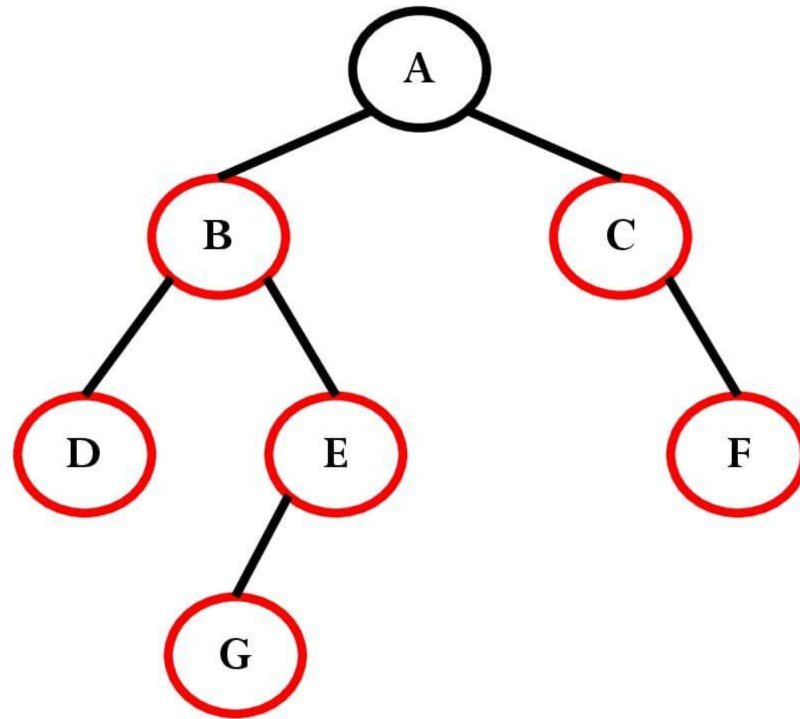
Postorder Traversal: D G E B

Postorder Traversal



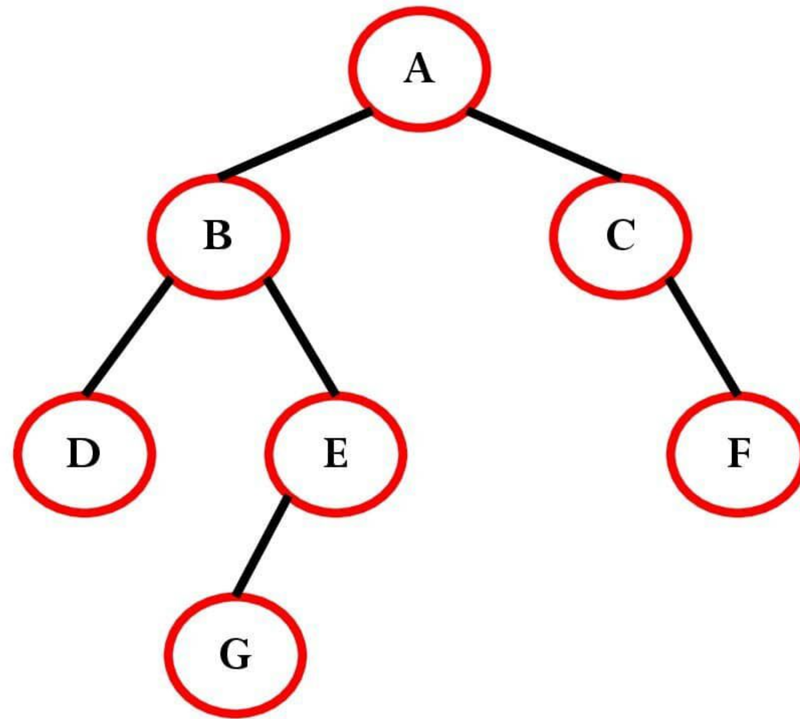
Postorder Traversal: D G E B F

Postorder Traversal



Postorder Traversal: D G E B F C

Postorder Traversal



Postorder Traversal: D G E B F C A

Postorder Traversal

```
struct node
{
        int data;
        struct node *lchild, *rchild;
}
void Postorder ( struct node * tree)
{
    if (tree != NULL)
    {
            Postorder(tree->lchild);
            Postorder(tree->rchild);
            printf("%d",tree->data);
    }
}
```