Trees

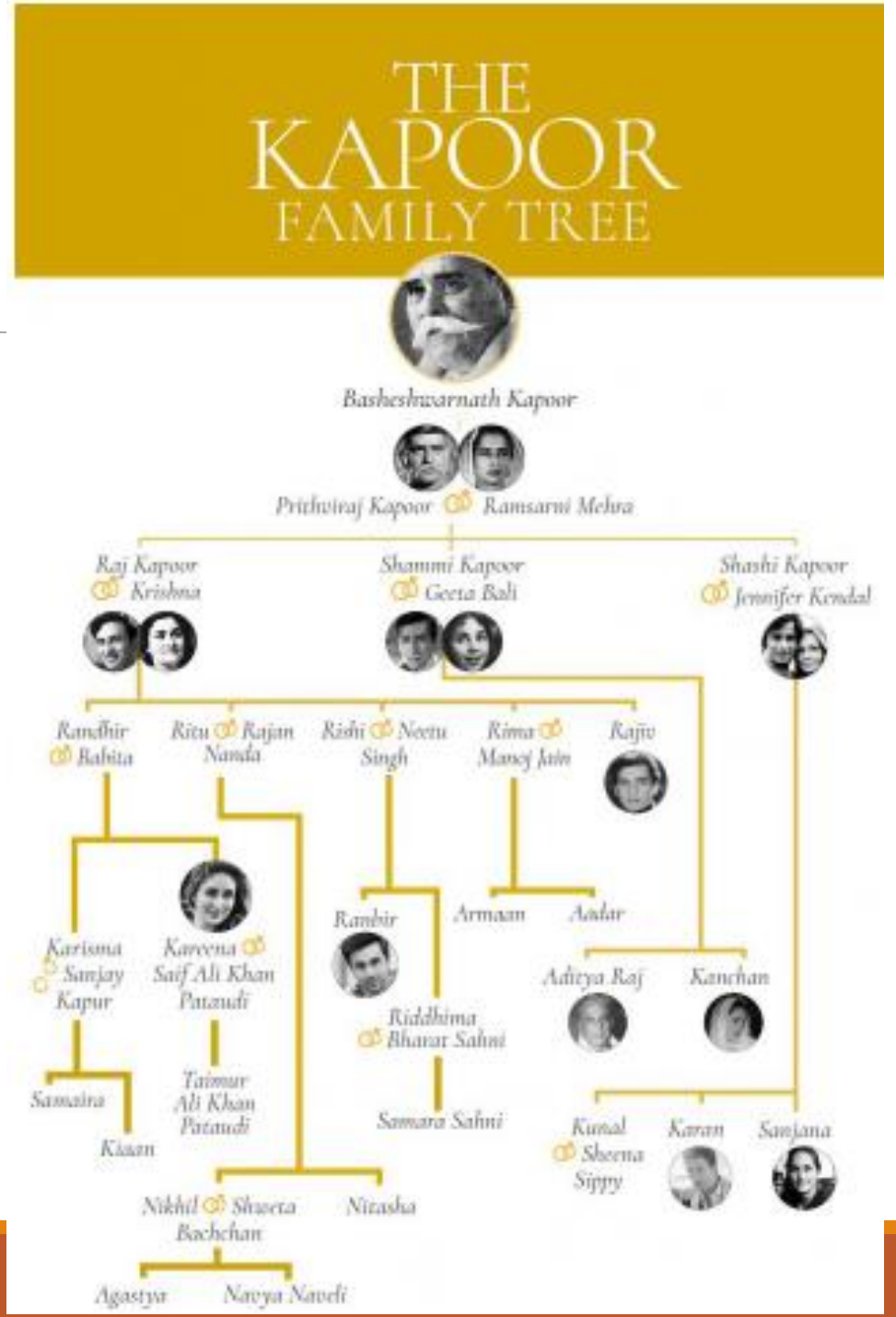
DR. SANGA CHAKI

Contents

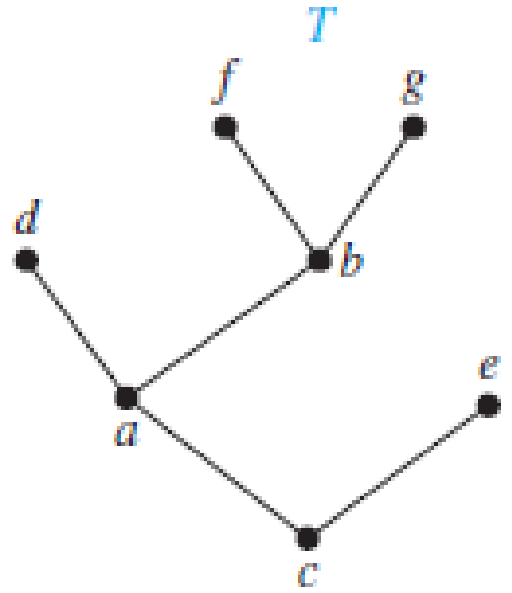
1. Tree Introduction
2. Tree Terminology
3. Types of trees
4. Implementation using Arrays and LL
5. Tree traversal
6. Construction of trees from traversals
7. BST

Trees

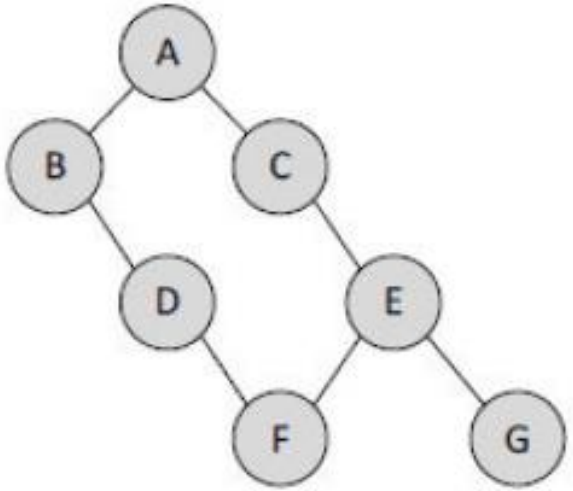
1. A particular type of graph containing vertices and edges
2. So named because such graphs resemble inverted trees.
3. Example: family trees are graphs that represent genealogical charts.
4. Family trees use vertices to represent the members of a family and edges to represent parent–child relationships.



Trees



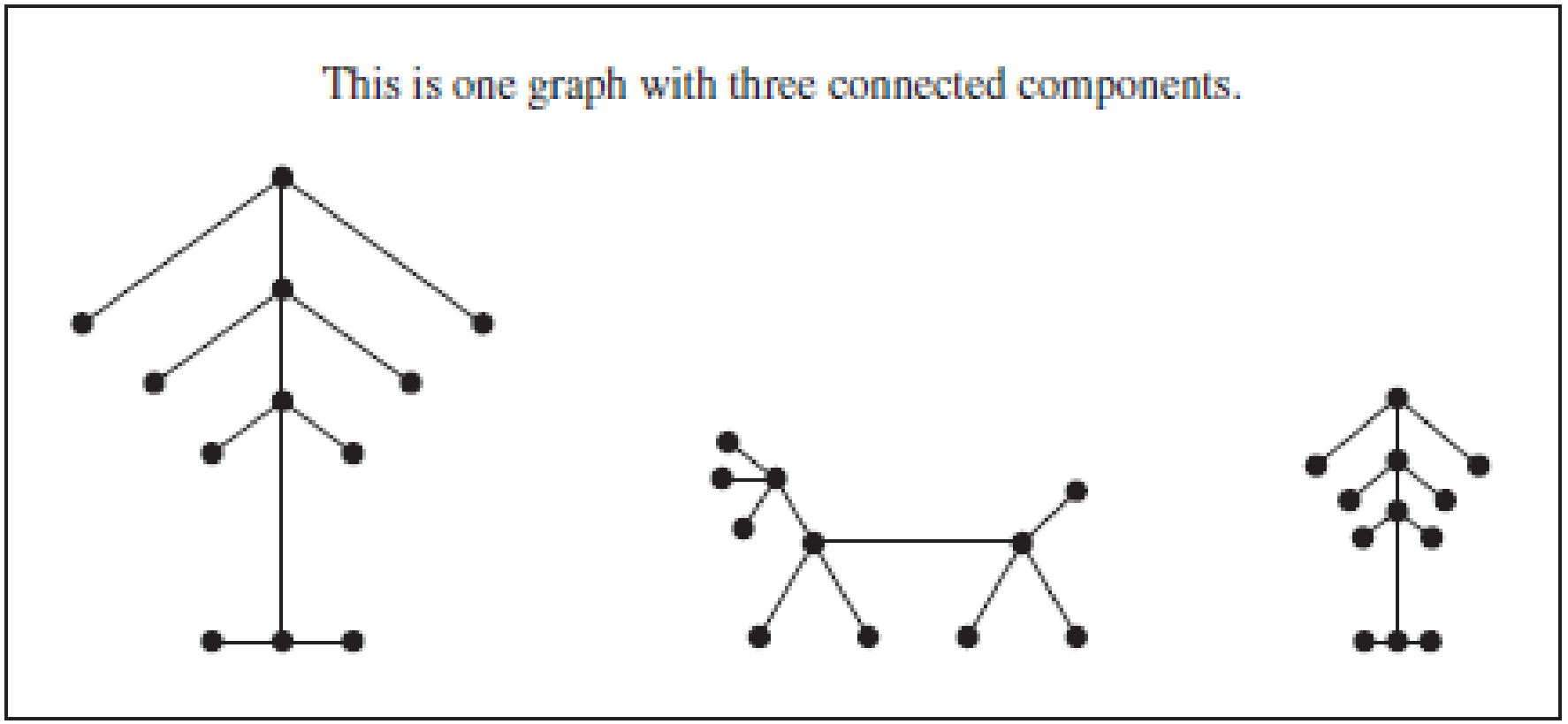
1. **Definition: A tree is a <u>connected</u> undirected graph with no <u>simple circuits</u>.**
2. Connected graph: A graph is a connected graph if, for each pair of vertices, there exists at least one single path which joins them
3. Circuits: A circuit is path that begins and ends at the same vertex.
4. Simple Circuit: Circuit with no repeated vertex
5. A tree should not have any loops or multiple edges between vertices



(b) Invalid Binary Tree

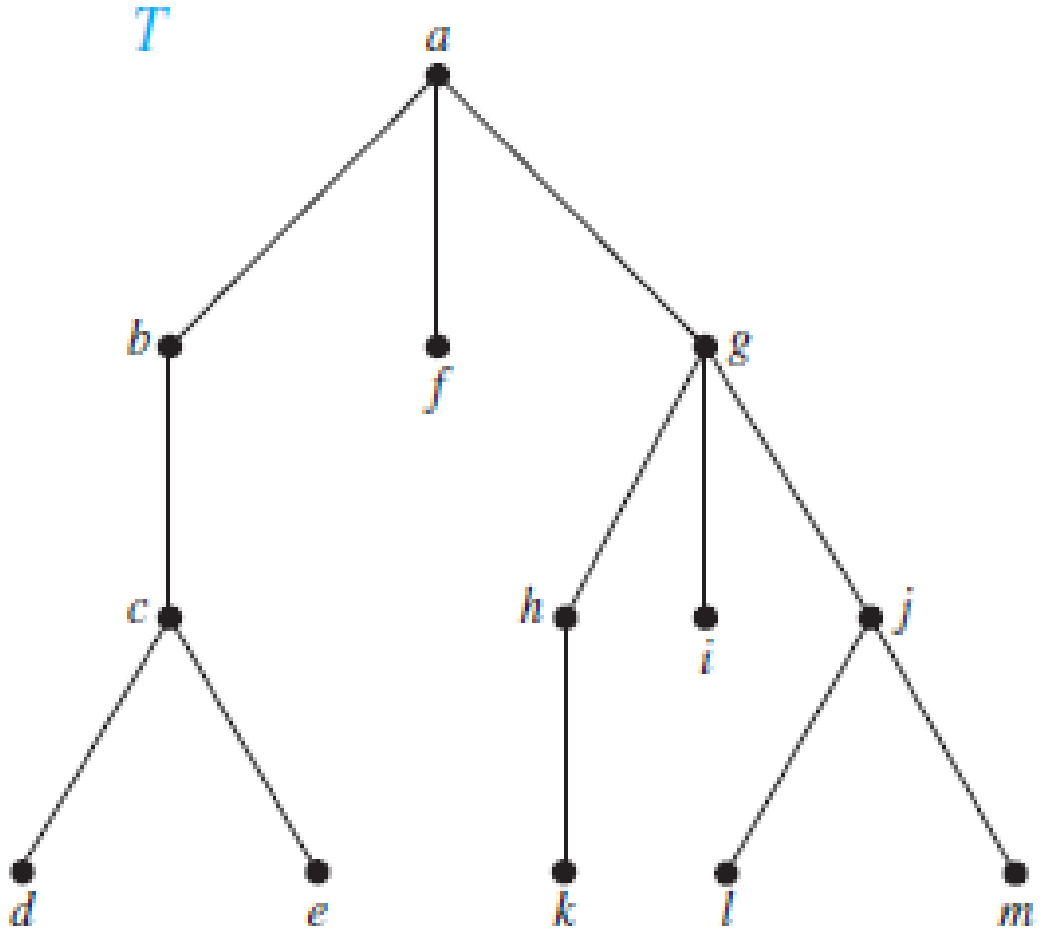
Forests

1. Graph containing more than one tree is called a forest



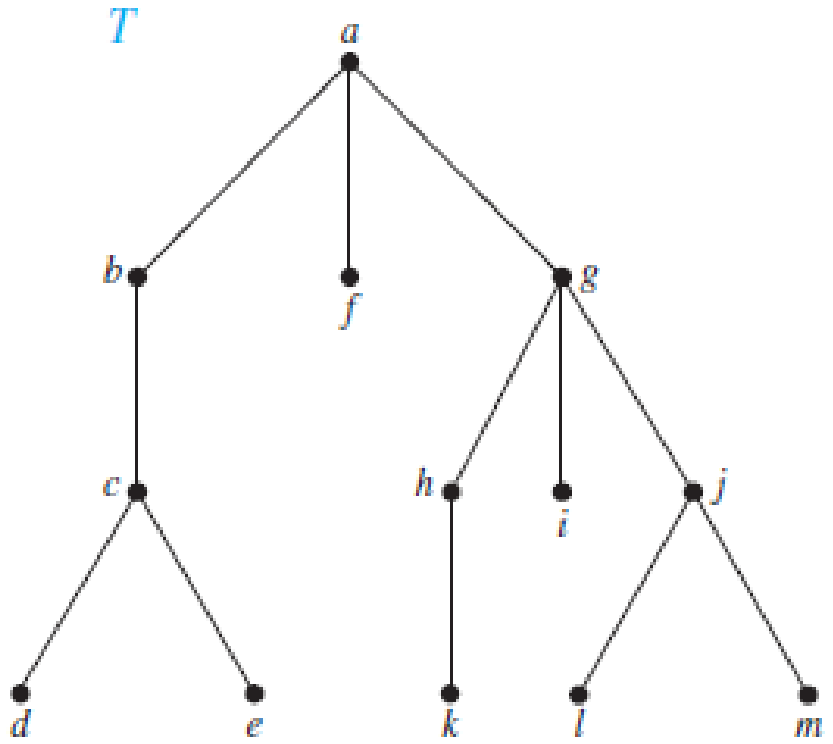
Rooted Trees

1. **A rooted tree is a tree in which one vertex has been designated as the root**
2. **Every edge is directed away from the root.**
3. Different choices of the root produce different rooted trees.
4. We usually draw a rooted tree with its root at the top of the graph.
5. Example: Root = a



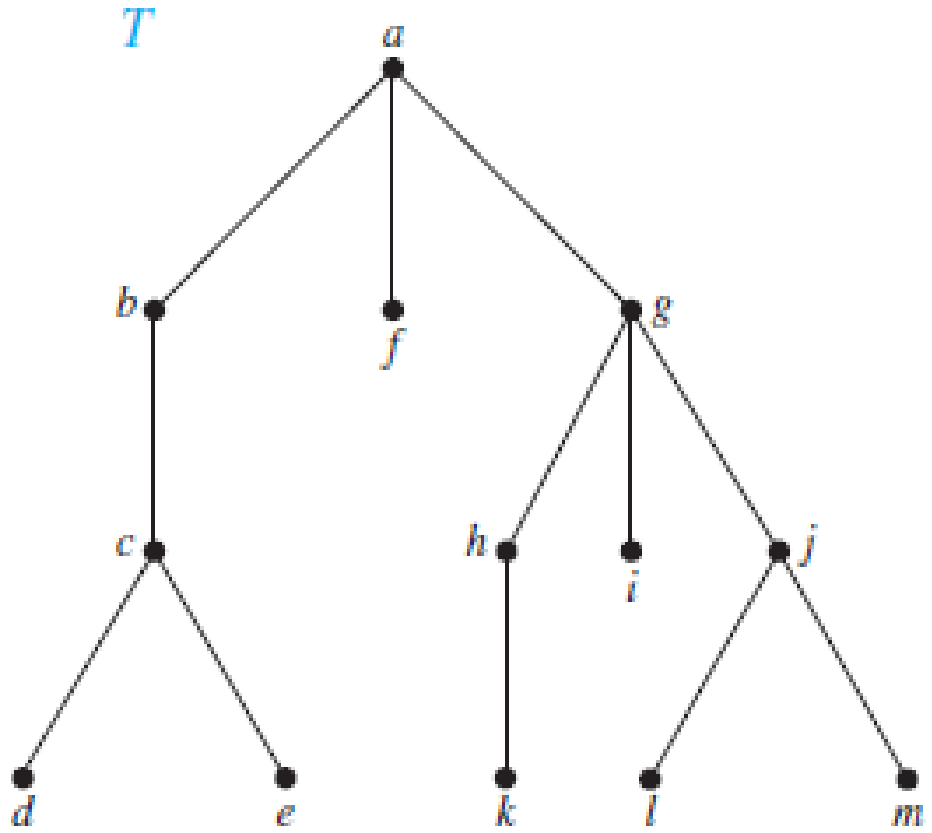
Rooted Tree Terminology

1. Suppose that T is a rooted tree.
2. If v is a vertex in T other than the root, the **parent** of v is the unique vertex u such that there is a direct edge from u to v, and u is closer to the root that v.
3. When u is the parent of v, v is called a **child** of u.
4. Vertices with the same parent are called **siblings**.
5. Eg: a is parent of b, f, g.
6. b, f, g are siblings
7. c is child of b

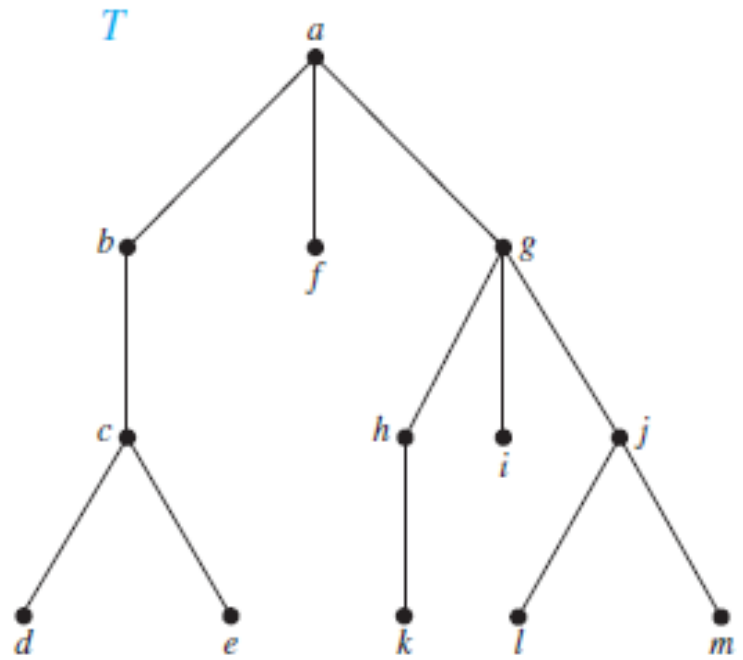


Rooted Tree Terminology

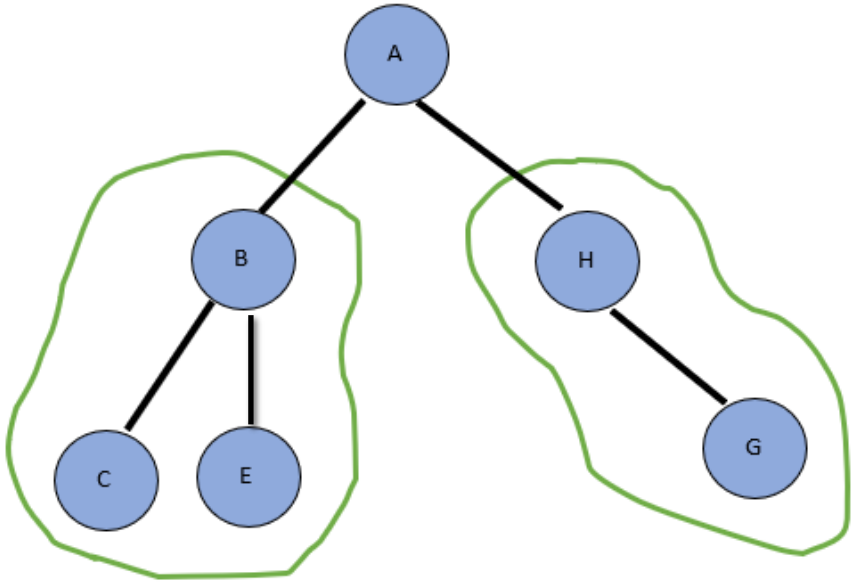
1. The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root
   - its parent, its parent's parent, and so on, until the root is reached
   - Ancestors of m are a, g and j
2. The **descendants** of a vertex v are those vertices that have v as an ancestor.
   - Descendants of g are h, k, i, j, l, m



Rooted Tree Terminology

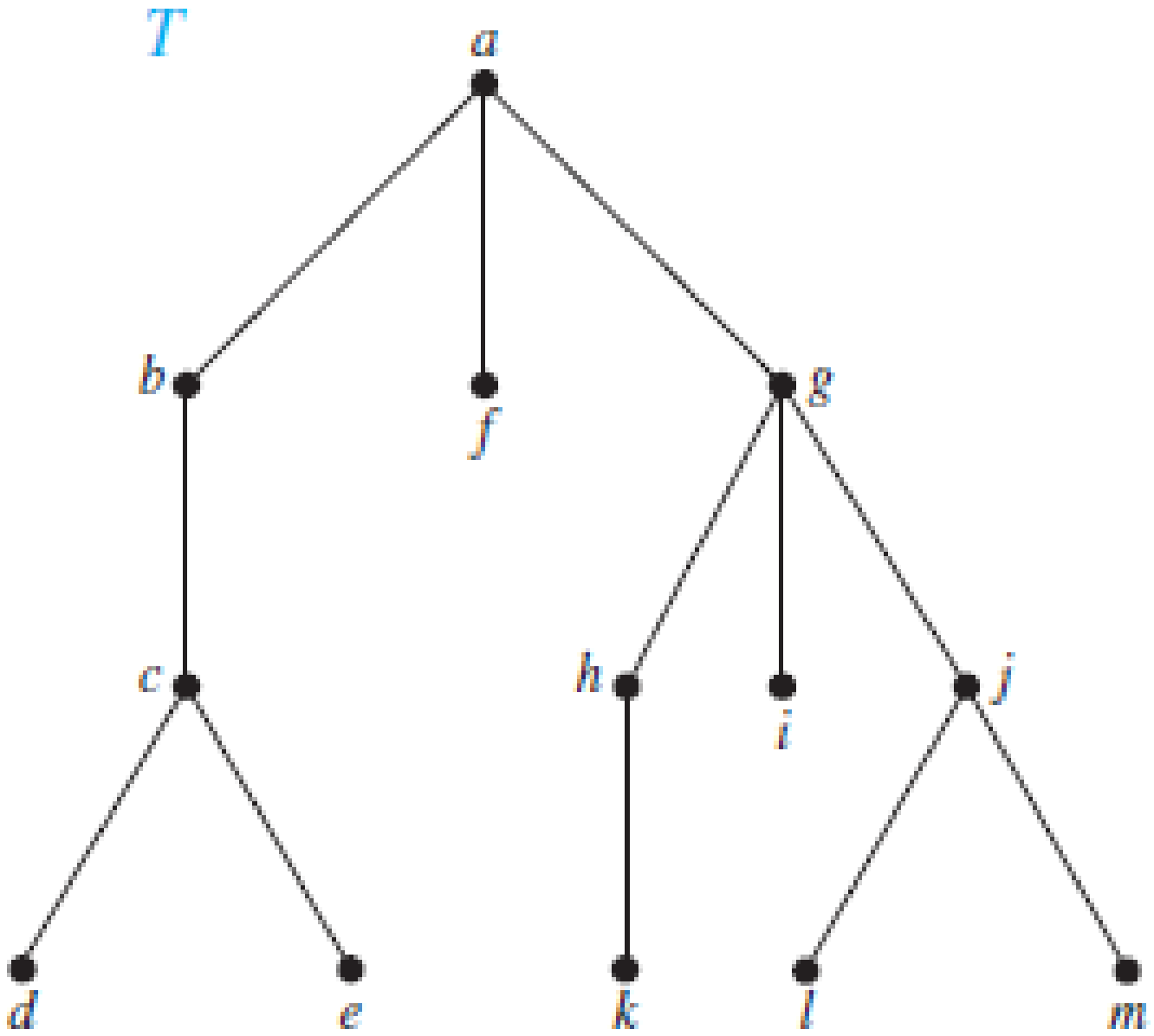


1. A vertex of a rooted tree is called a **leaf** if it has no children. Eg. d, e etc
2. Vertices that have children are called **internal vertices**. Eg. b, c etc
3. If ***a*** is a vertex in a tree, the **subtree** with ***a*** as its root is the subgraph of the tree consisting of ***a*** and its descendants and all edges incident to these descendants.
4. Eg. Subtrees with B and H as roots



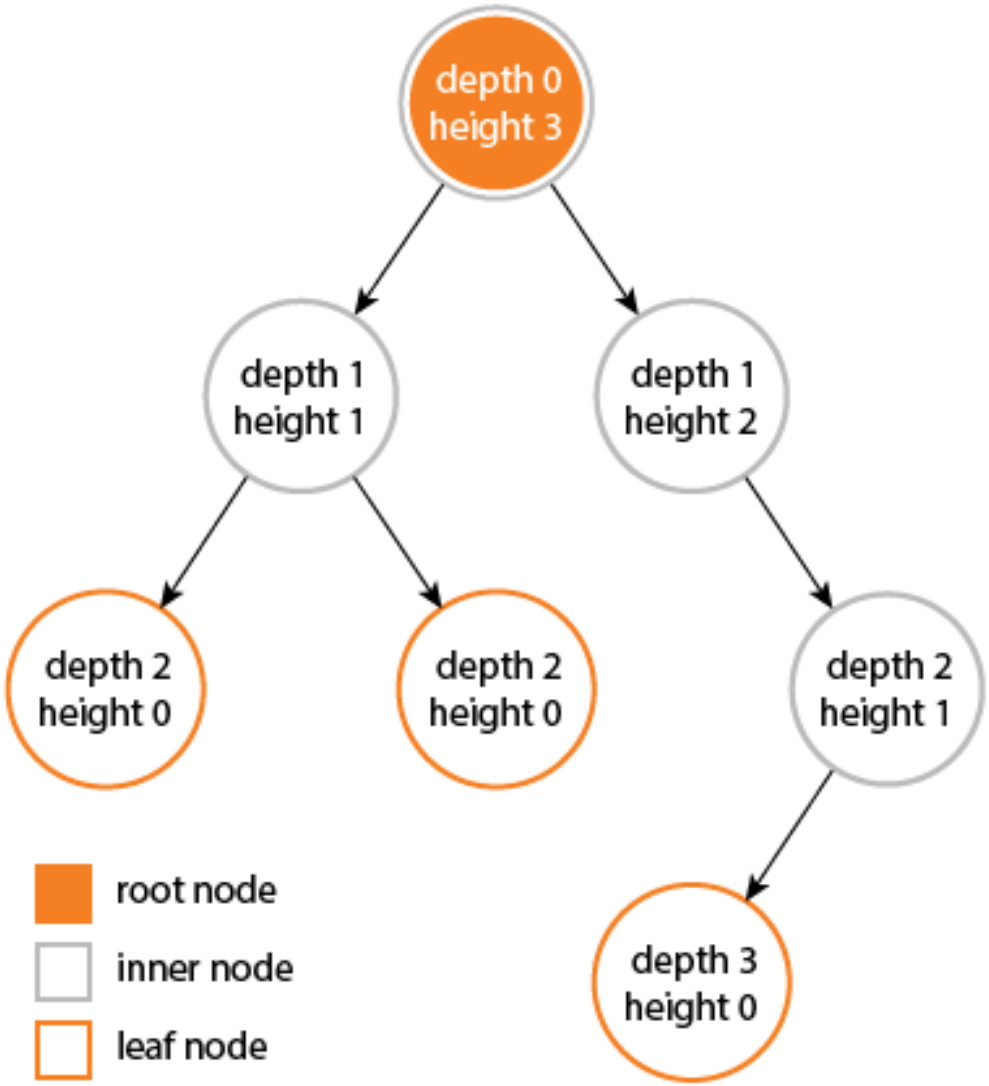
Rooted Tree

1. **Example**: In the below rooted tree T (with root a), find
   - the parent of c,
   - the children of g,
   - The siblings of h,
   - all ancestors of e,
   - all descendants of b,
   - all internal vertices, and
   - all leaves.
   - What is the subtree rooted at g?



Rooted Tree: Height, Depth and Level of a Node

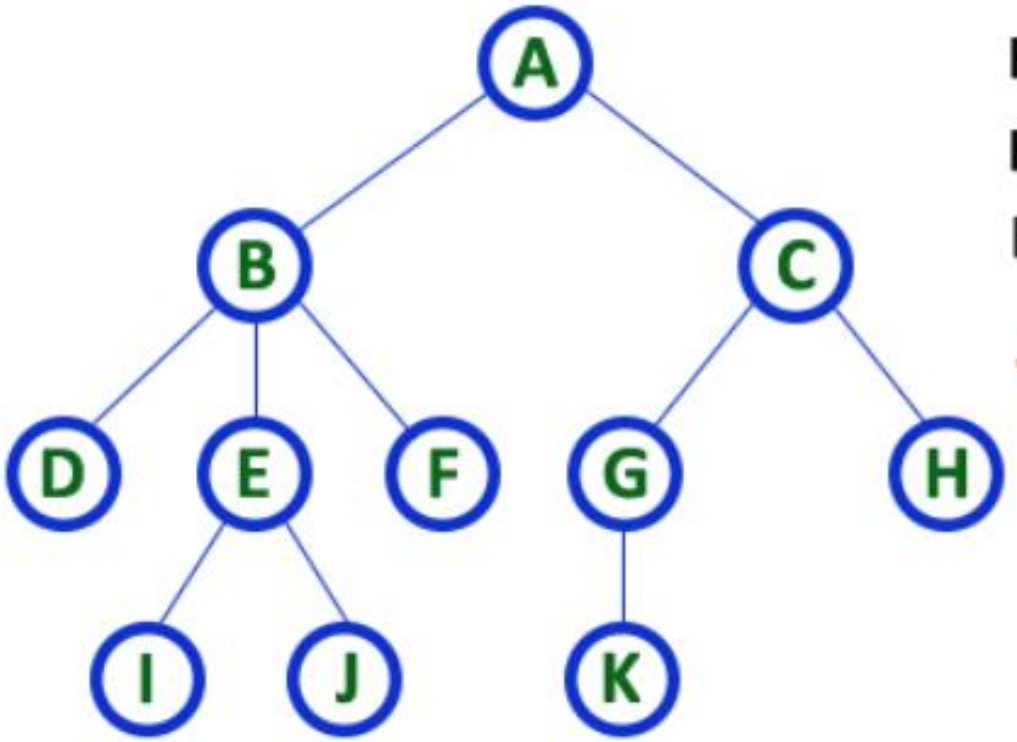
1. The **depth of a node** is the number of edges from the node to the tree's root node.
   - A root node will have a depth of 0.
2. **Level of a node** = depth + 1
3. The **height of a node** is the number of edges on the longest path from the node to a leaf.
   - A leaf node will have a height of 0.
4. The **height of a tree** = height of its root node = the **depth of the tree**



Rooted Tree: Degree

1. Degree of a Node: In the tree data structure, the total number of children of a node is called the degree of the node.
2. Degree of Tree: The highest degree of the node among all the nodes in a tree is called the Degree of Tree.
3. So, degree of tree = Maximum number of children that a node can have in that tree.

Rooted Tree: Degree



Here Degree of B is 3

Here Degree of A is 2

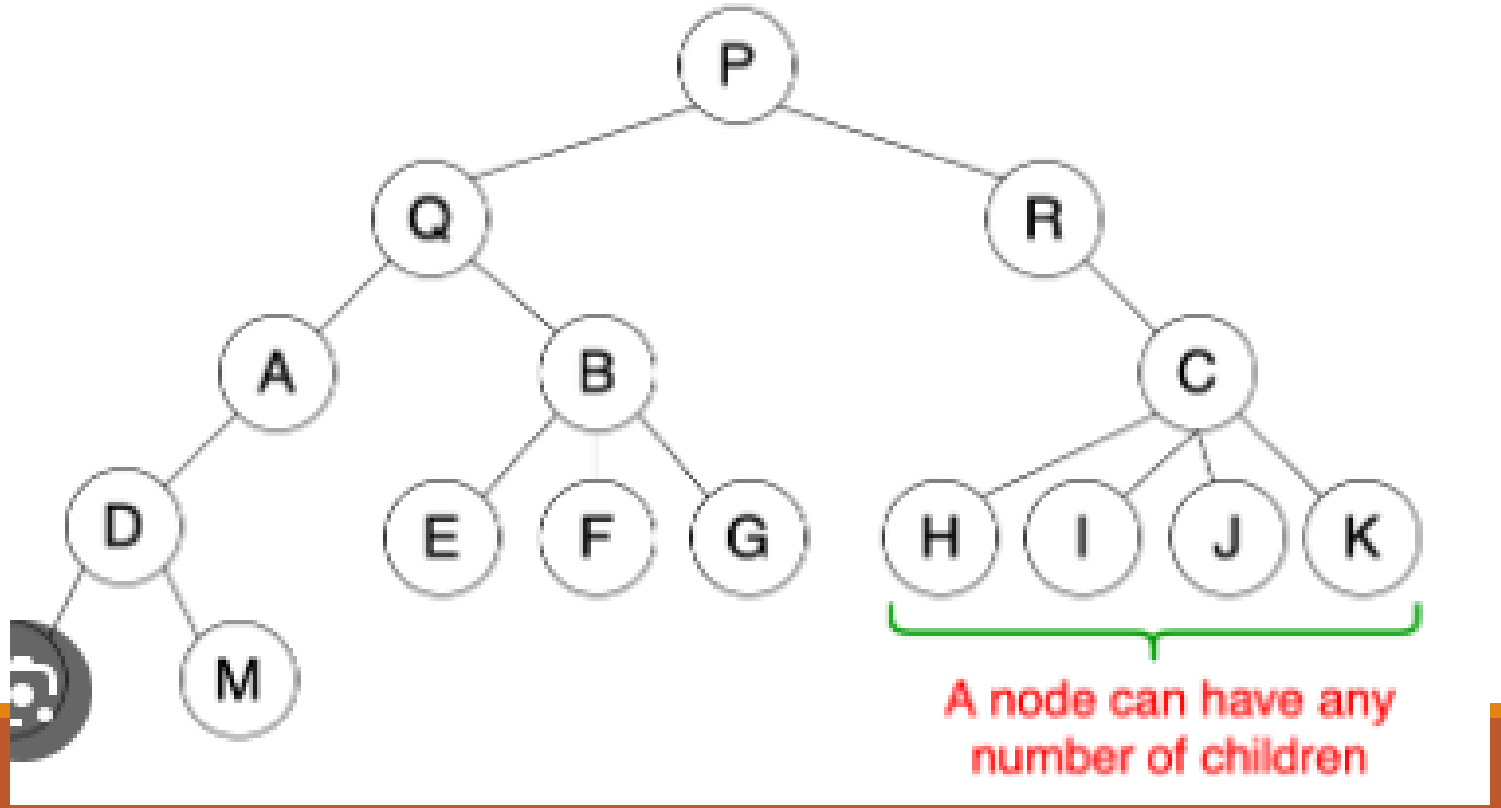
Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

Types of Trees

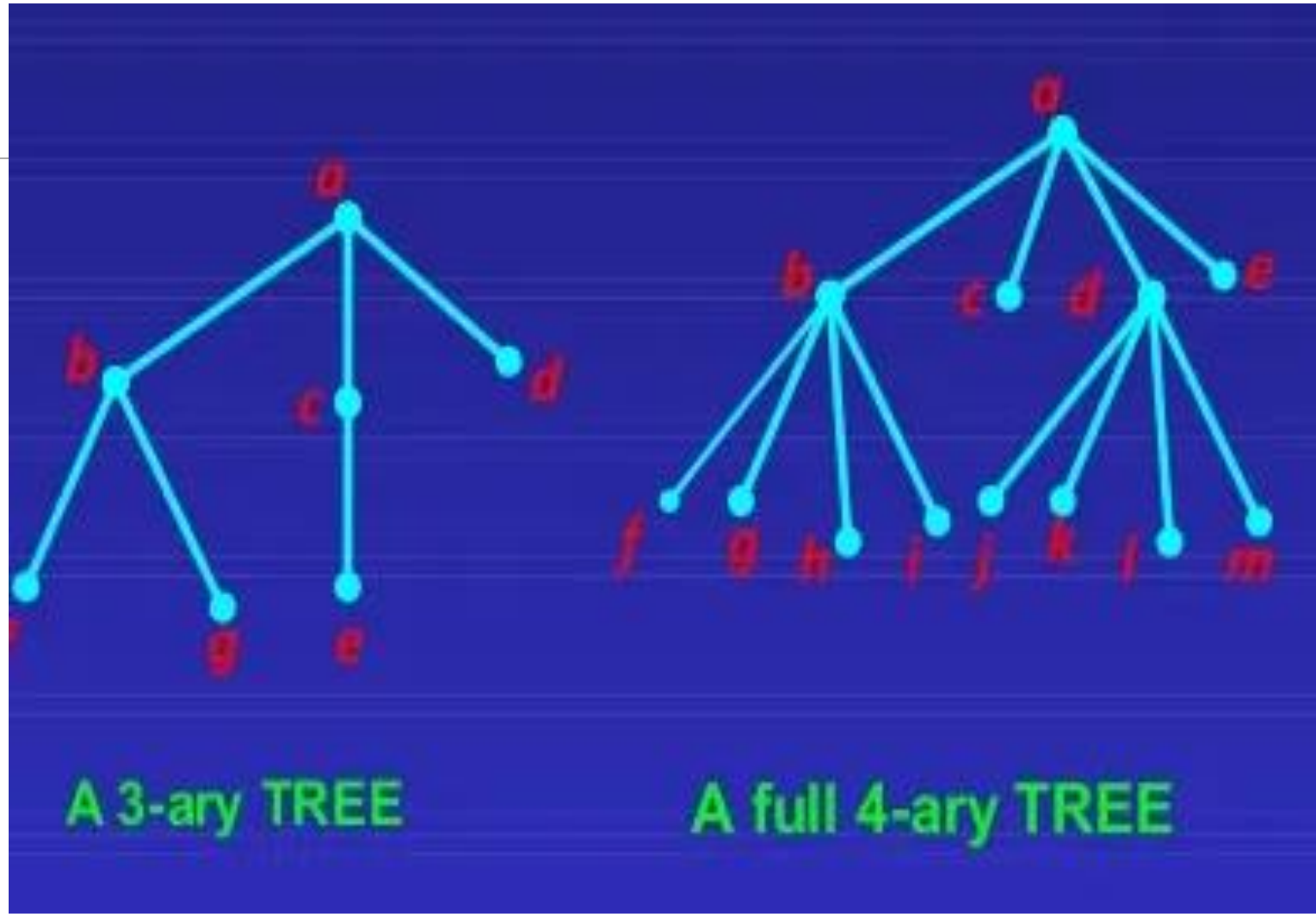
General Tree

1. **General tree** is a tree in which each node can have either zero or many child nodes. There is no limitation on the degree of a node.



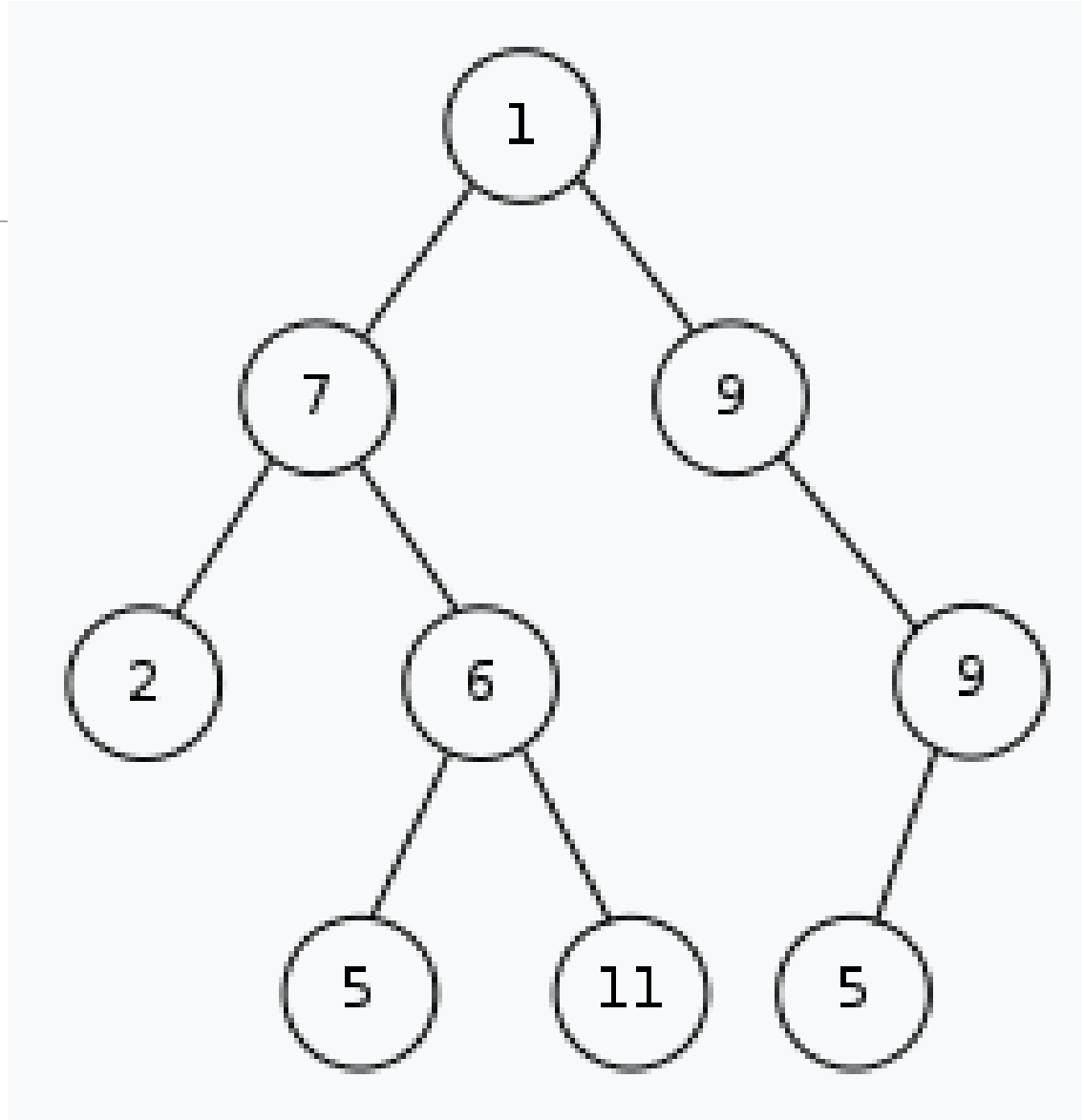
M-ary Tree

1. A tree is called an **m-ary tree** if every internal vertex has no more than **m children**.
2. The tree is called a **full m-ary tree** if every internal vertex has exactly m children.



Binary Tree

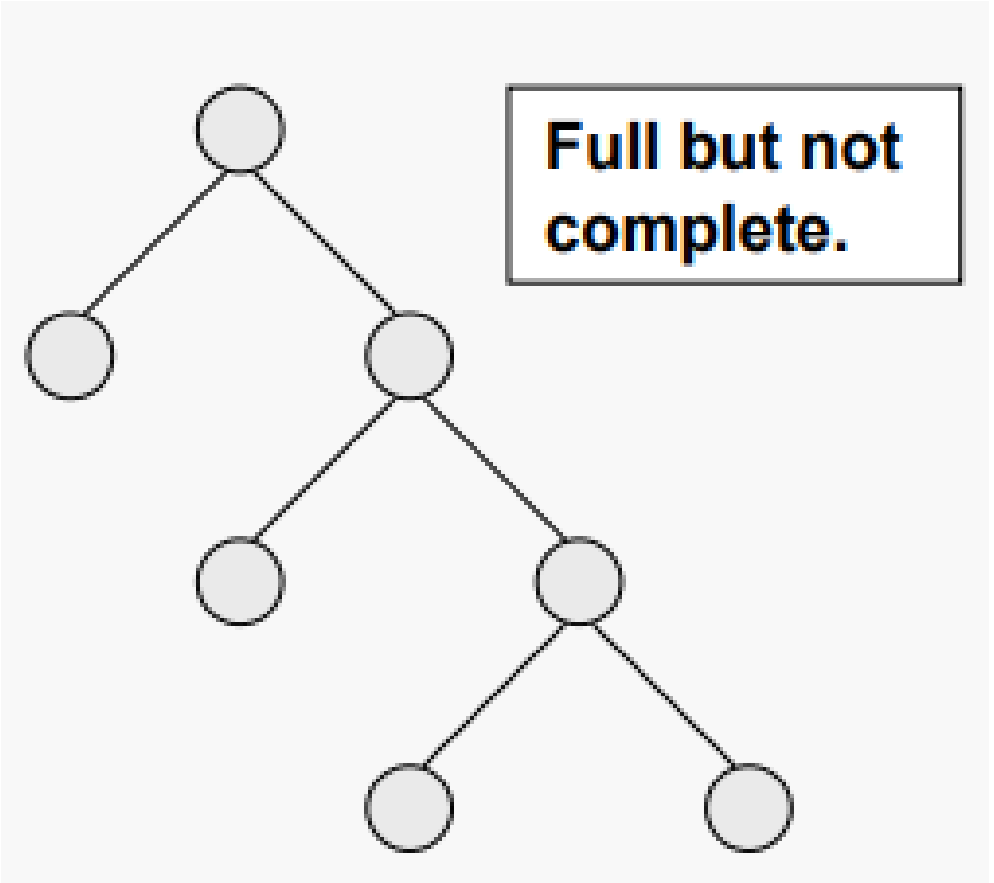
1. An m-ary tree with m = 2 is called a **binary tree**.
2. So in a binary tree, each internal vertex can have a maximum of 2 children.
3. Any node in a BT can have 0, 1 or 2 child.



Binary Tree Terminology

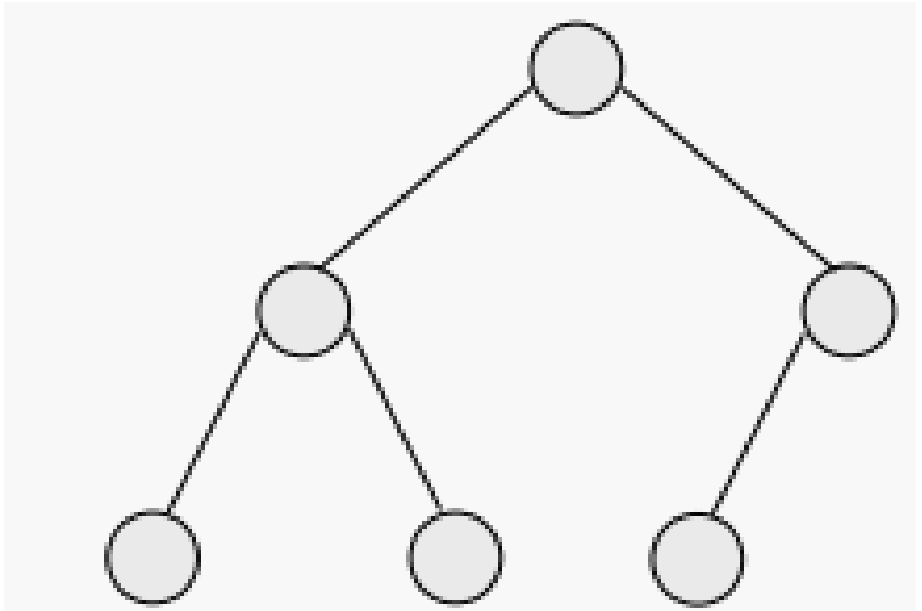
Binary Trees - Terminology

1. **Strictly/Full binary tree** : a binary tree T is full if each node is either a leaf or possesses exactly two child nodes.

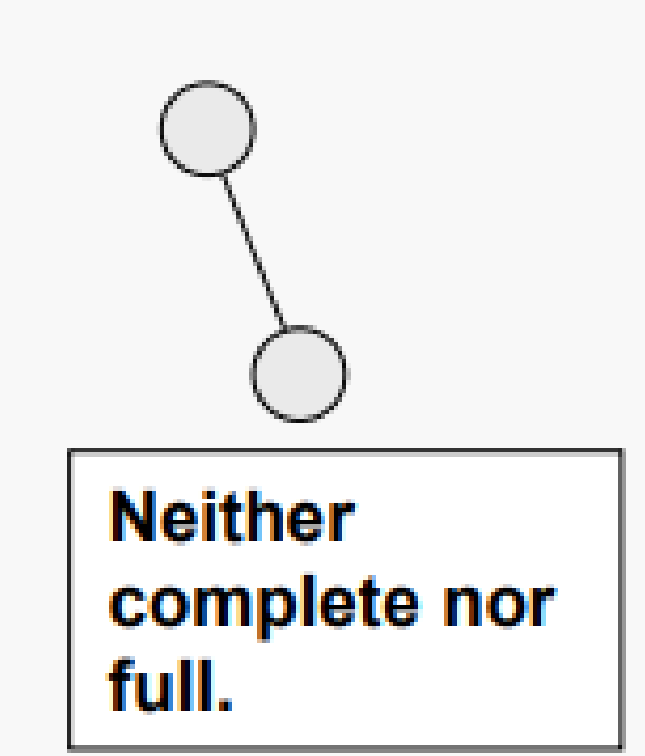


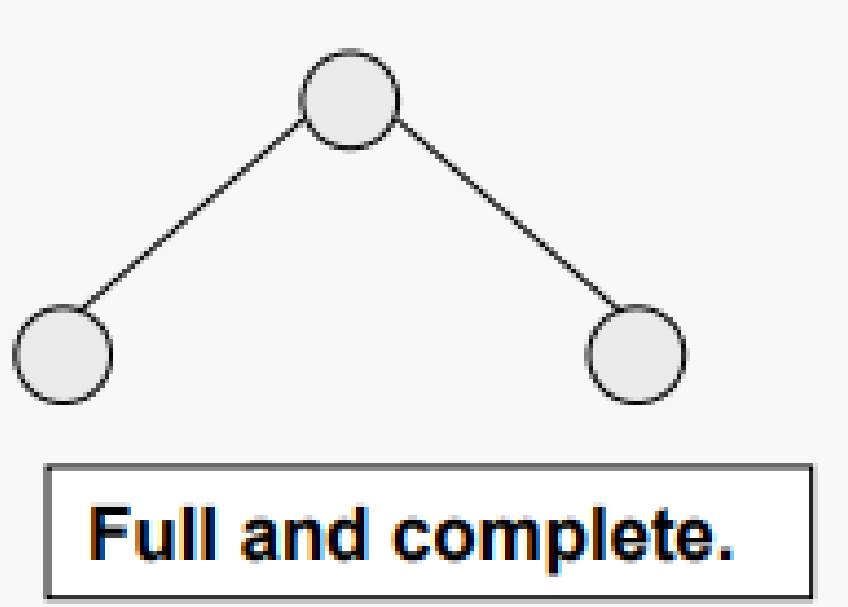
Binary Trees - Terminology

1. **Complete binary tree** : A complete binary tree has maximum number of possible nodes at all levels except the last level, and all the nodes of the last level appear as far left as possible.



Binary Trees - Terminology





Full Binary Tree - Theorems

Theorem: Let T be a nonempty, full binary tree Then:

(a) If T has I internal nodes, the number of leaves is $L = I + 1$.

(b) If T has I internal nodes, the total number of nodes is $N = 2I + 1$.

(c) If T has a total of N nodes, the number of internal nodes is $I = (N - 1)/2$.

(d) If T has a total of N nodes, the number of leaves is $L = (N + 1)/2$.

(e) If T has L leaves, the total number of nodes is $N = 2L - 1$.

(f) If T has L leaves, the number of internal nodes is $I = L - 1$.

Binary Trees – More Theorems

1. Let T be a binary tree with L levels. Then the number of leaves is at most 2^{L-1}
2. Let T be a binary tree. For every $k \geq 0$, there are no more than 2^k nodes in level k.
3. Let T be a binary tree with L levels. Then T has no more than $2^L - 1$ nodes.
4. Let T be a binary tree with N nodes. Then the number of levels is at least $\lceil \log (N + 1) \rceil$.
5. Let T be a binary tree with L leaves. Then the number of levels is at least $\lceil \log L \rceil + 1$.

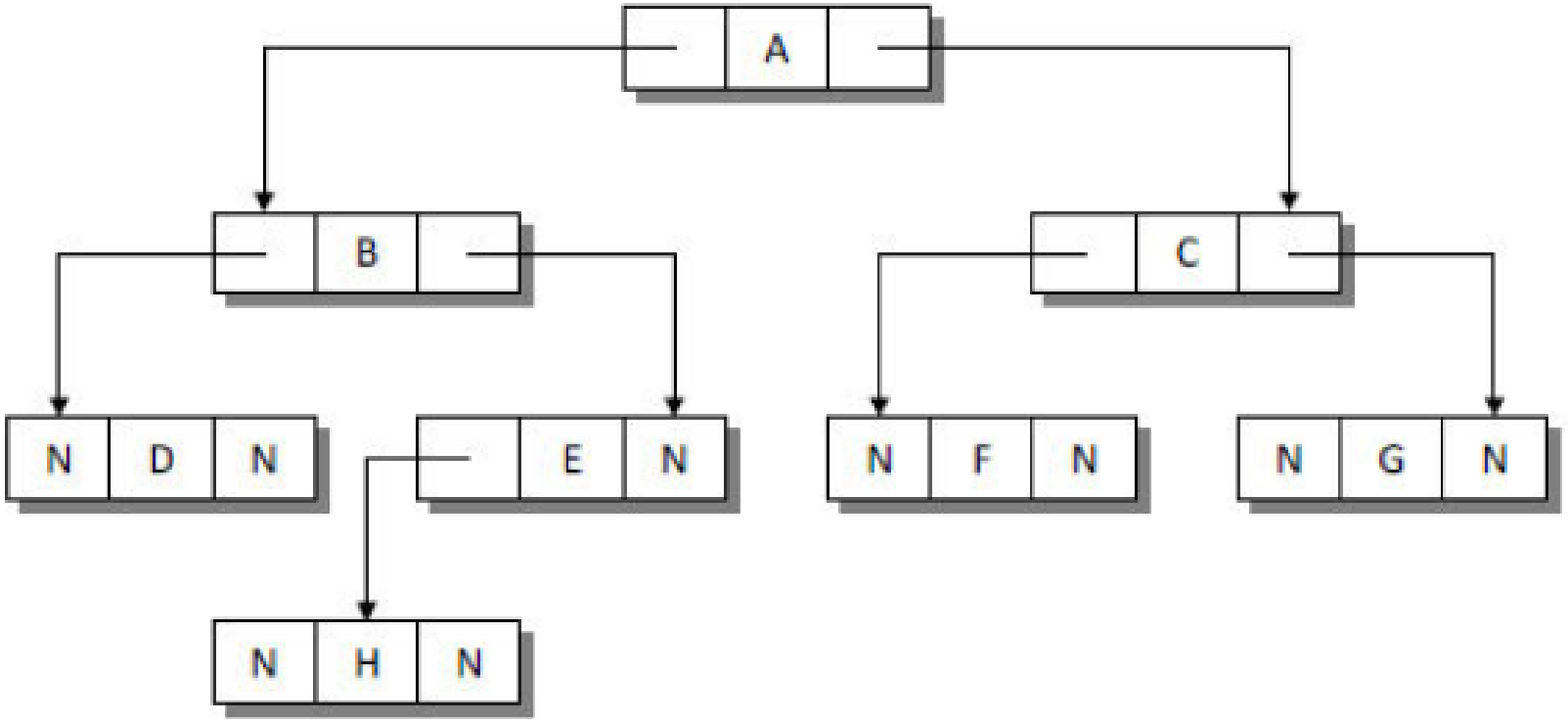
BT Representation

Binary Trees - Representation

1. There are two ways by which we can represent a binary tree—
    1. Linked representation and
    2. Array representation

Linked Representation of Binary Trees

1. In linked representation each node contains addresses of its left child and right child, along with data value.
2. If a child is absent, the link contains a NULL value.



1. Root = A
2. Leaves = D, H, F, G
3. Node E has only left child.

Array Representation of Binary Trees

1. When a binary tree is represented by arrays three separate arrays are required.
2. One array **arr** stores the data fields of the nodes.
3. The other two arrays **lc** and **rc** represents the left child and right child of the nodes
4. Example array representation of the previous tree.

	0	1	2	3	4	5	6	7	8	9
arr	A	B	C	D	E	F	G	'\0'	'\0'	H
lc	1	3	5	-1	9	-1	-1	-1	-1	-1
rc	2	4	6	-1	-1	-1	-1	-1	-1	-1

Array Representation of Binary Trees

1. The array lc and rc contains the index of the array arr where the data is present.
2. If the node does not have any left child or right child then the element of the array lc or rc contains a value -1.
3. The element of the array arr contains the root node data.
4. Some elements of the array arr contain '\0' which represents an empty child.

arr	A	B	C	D	E	F	G	'\0'	'\0'	H
lc	1	3	5	-1	9	-1	-1	-1	-1	-1
rc	2	4	6	-1	-1	-1	-1	-1	-1	-1

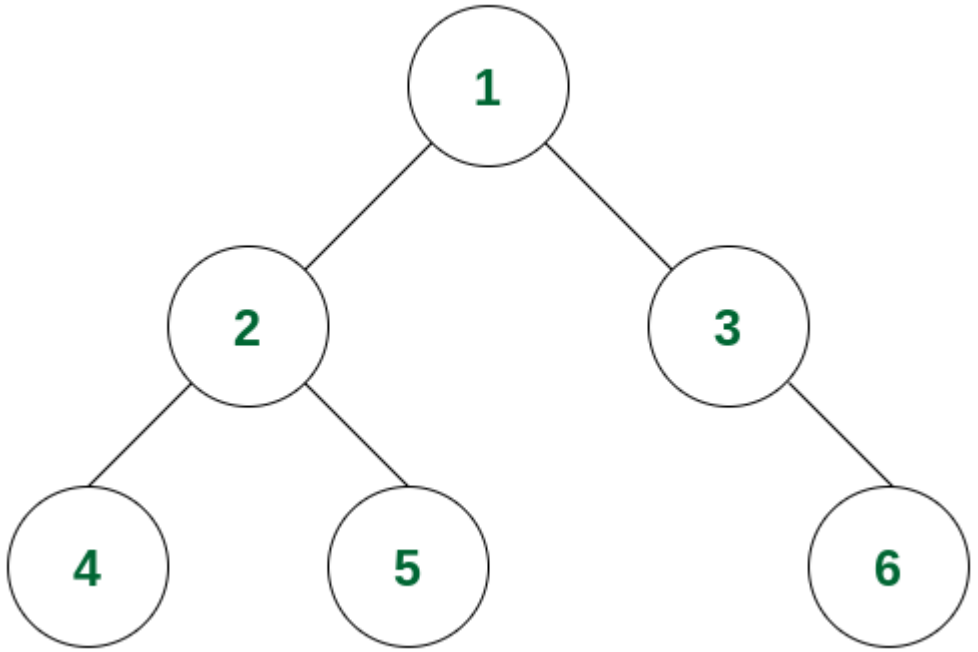
BT Traversals

Tree Traversals

1. The goal: to visit each node in Binary Tree
   - all the nodes in the left subtree,
   - visit the root node and
   - visit all the nodes in the right subtree as well.
2. Depending on the order in which we do this, there can be three types of traversals.
   a) **Inorder traversal**
   b) **Preorder traversal and**
   c) **Postorder traversal**

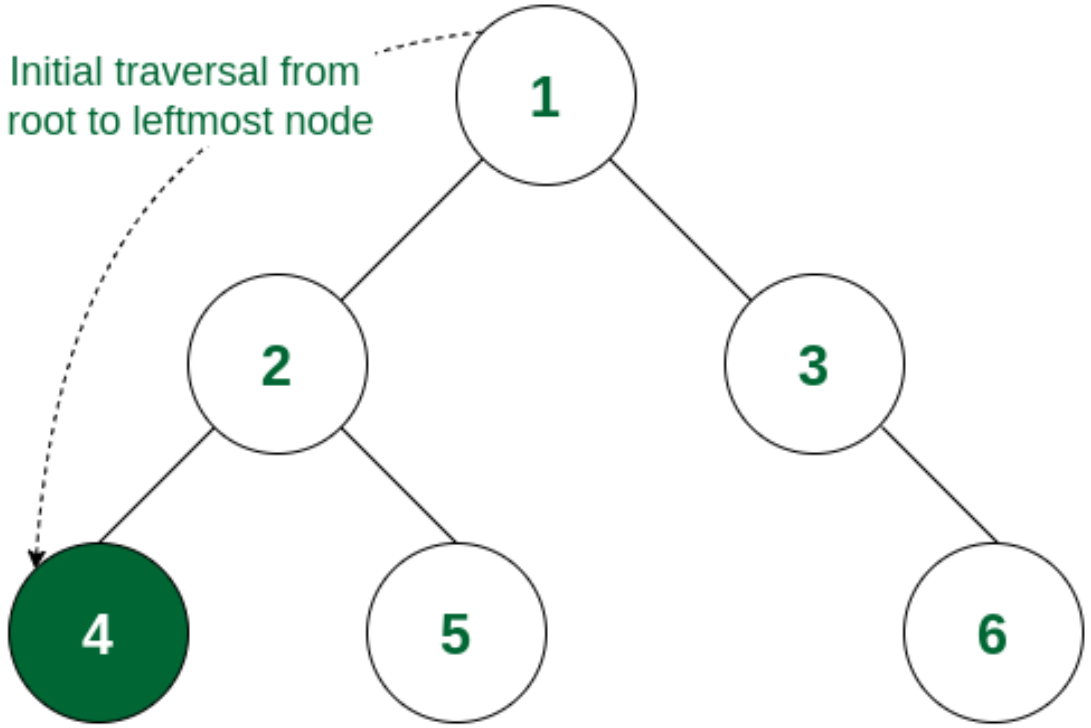
In-order Traversal

1. Follow the below steps to implement the idea (**left-root-right**):
   a) Start from root of BT
   b) Traverse left subtree
   c) Visit the root and print the data.
   d) Traverse the right subtree
2. Performed recursively



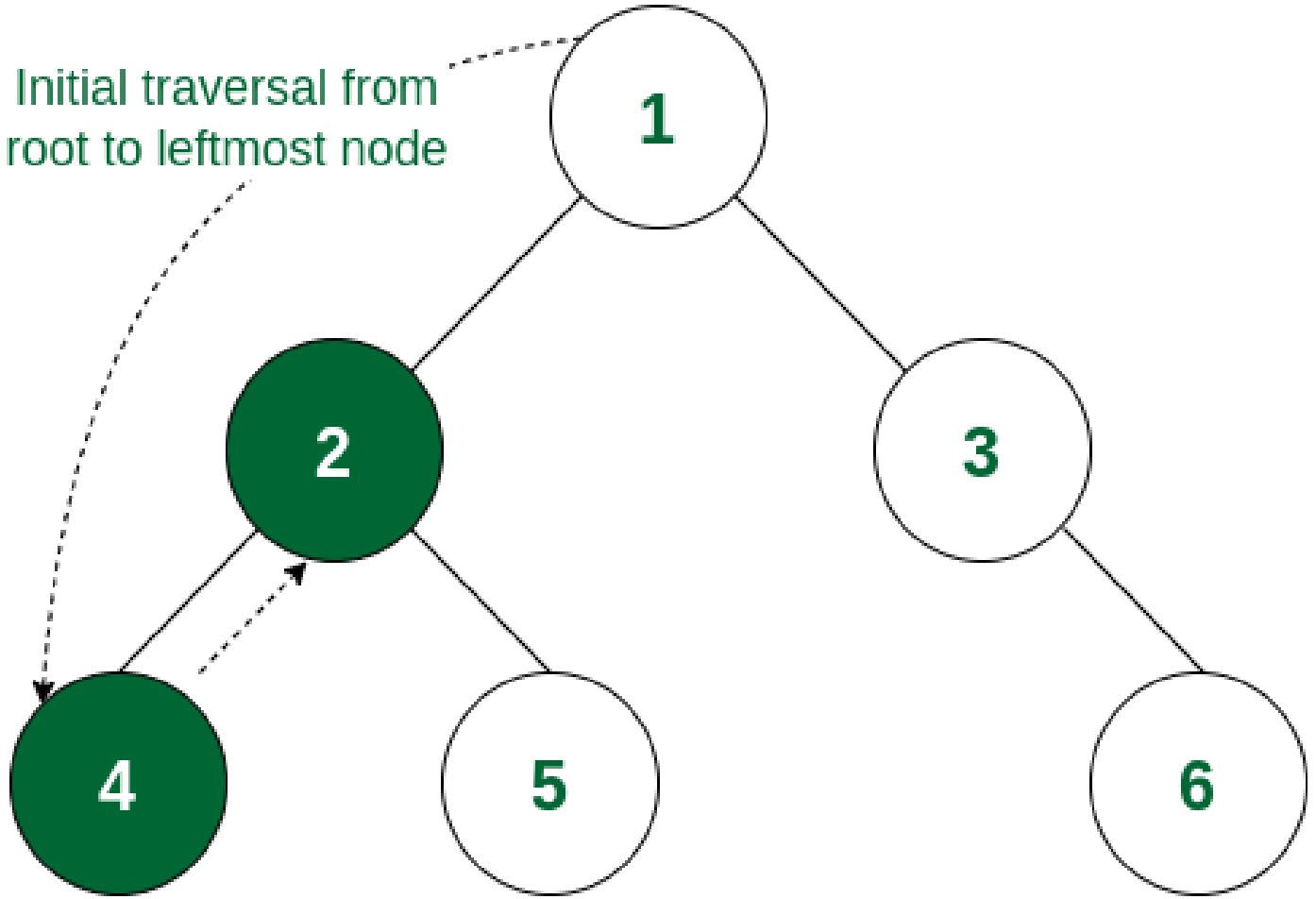
In-order Traversal

1. Step 1: The traversal will go from 1 to its left subtree i.e., 2,
2. then from 2 to its left subtree root, i.e., 4.
3. Now 4 has no left subtree, so it will be visited.
4. It also does not have any right subtree. So no more traversal from 4
5. Print 4



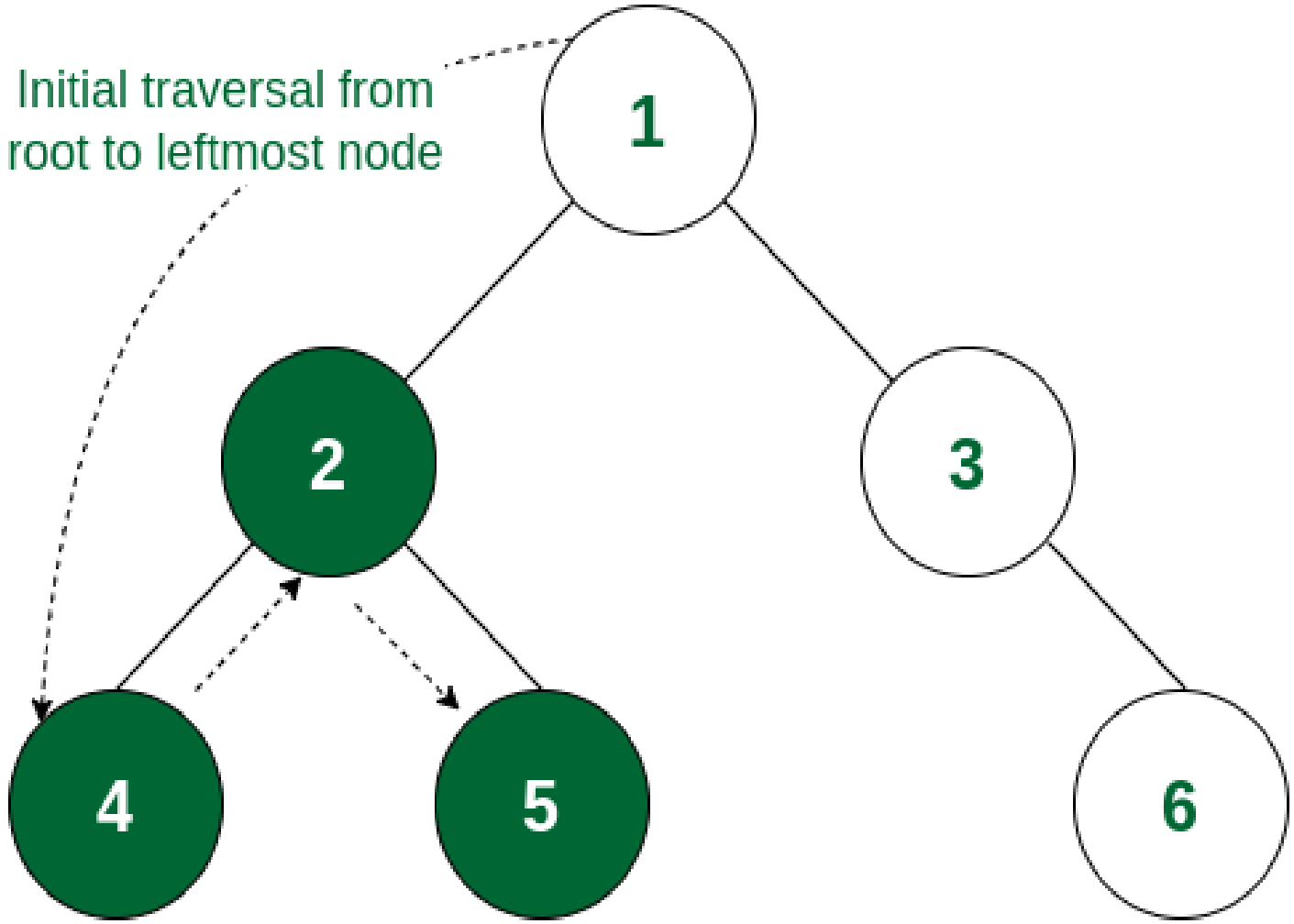
In-order Traversal

1. Step 2: As the left subtree of 2 is visited completely,
2. now it visits and prints data of node 2 before moving to its right subtree.



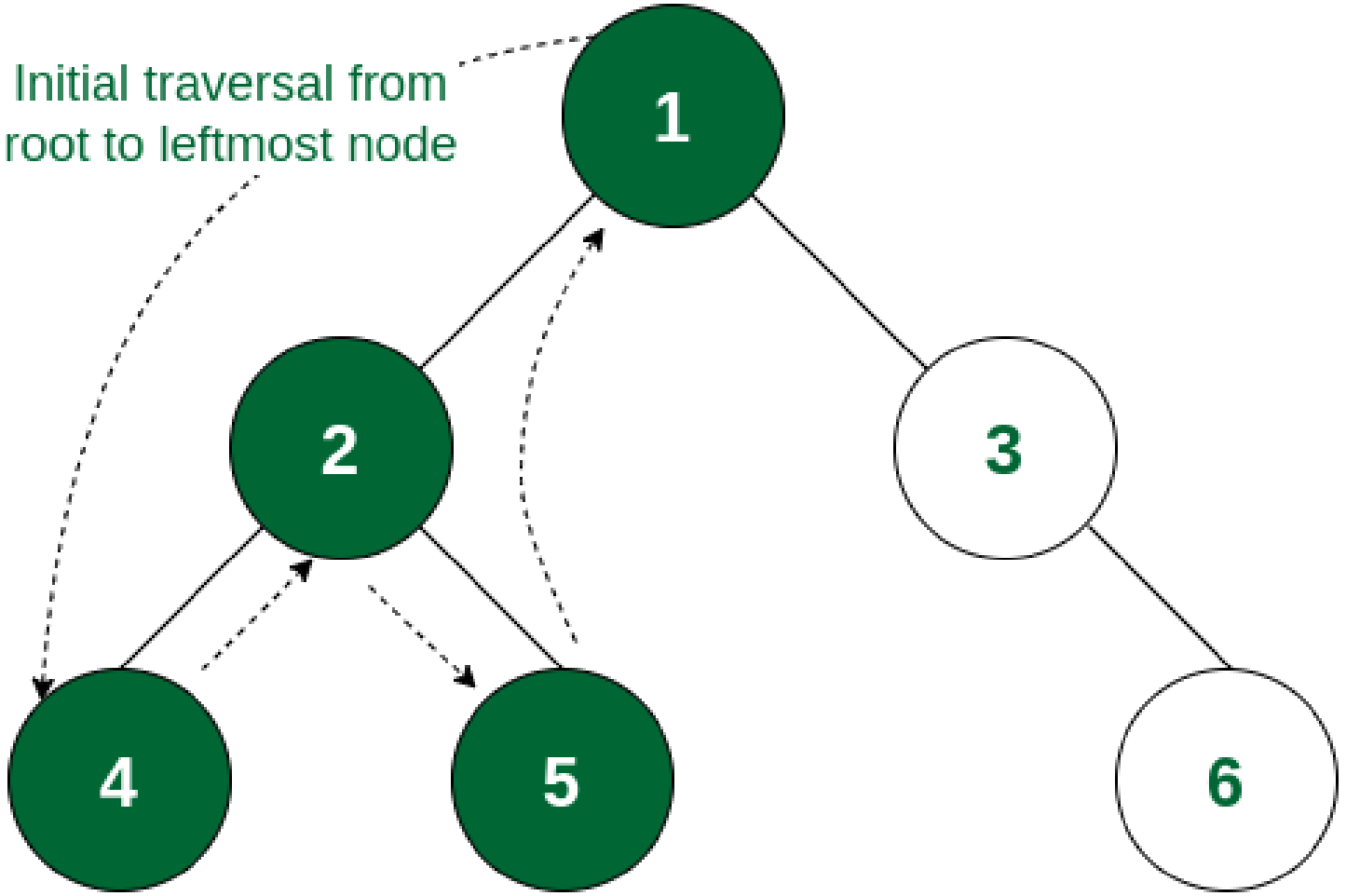
In-order Traversal

1. Step 3: Now the right subtree of 2 will be traversed i.e., move to node 5.
2. For node 5 there is no left subtree, so it gets printed
3. after that, the traversal comes back because there is no right subtree of node 5.



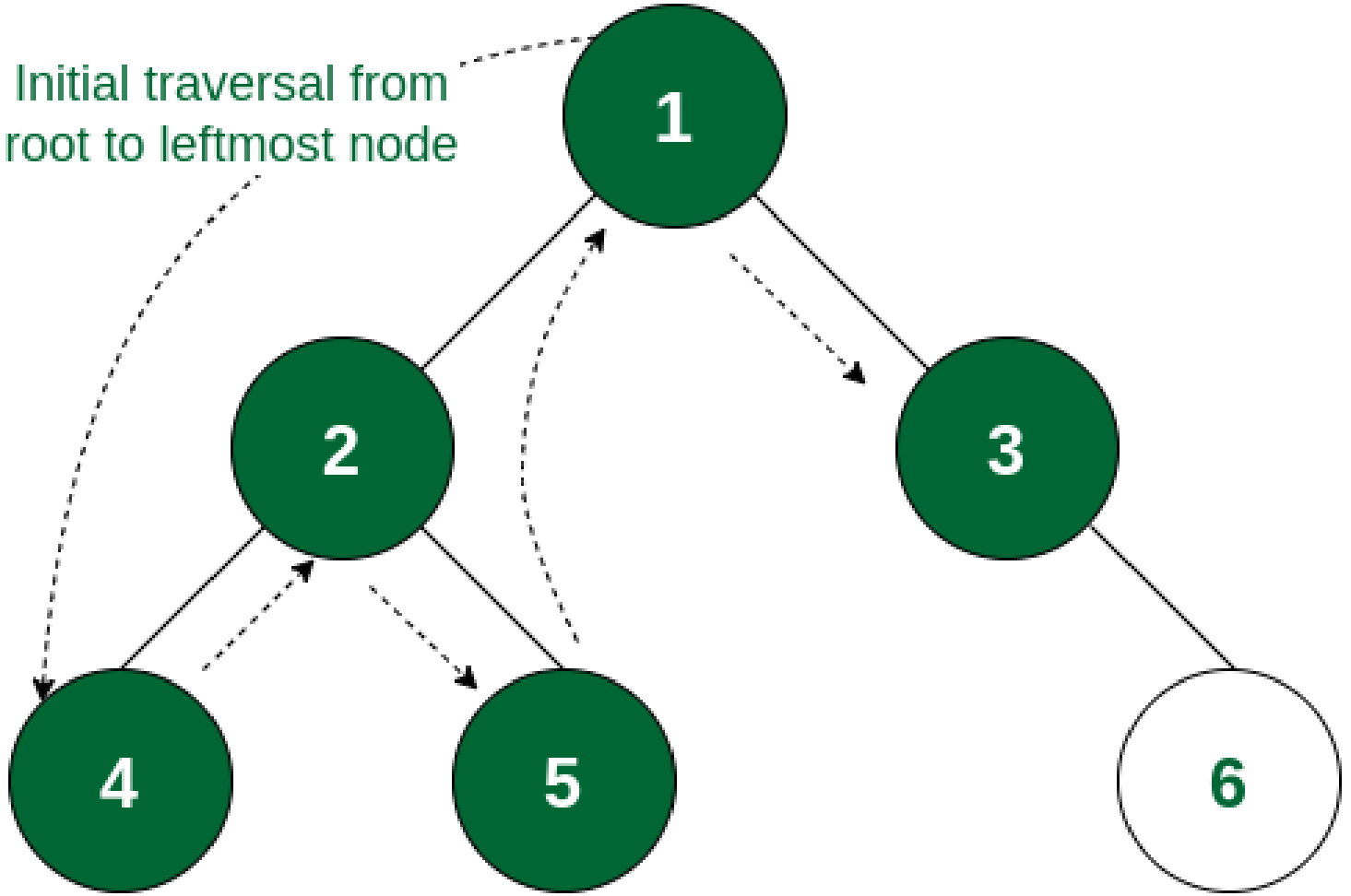
In-order Traversal

1. Step 4: As the left subtree of node 1 is completely traversed, the root itself, i.e., node 1 will be visited..



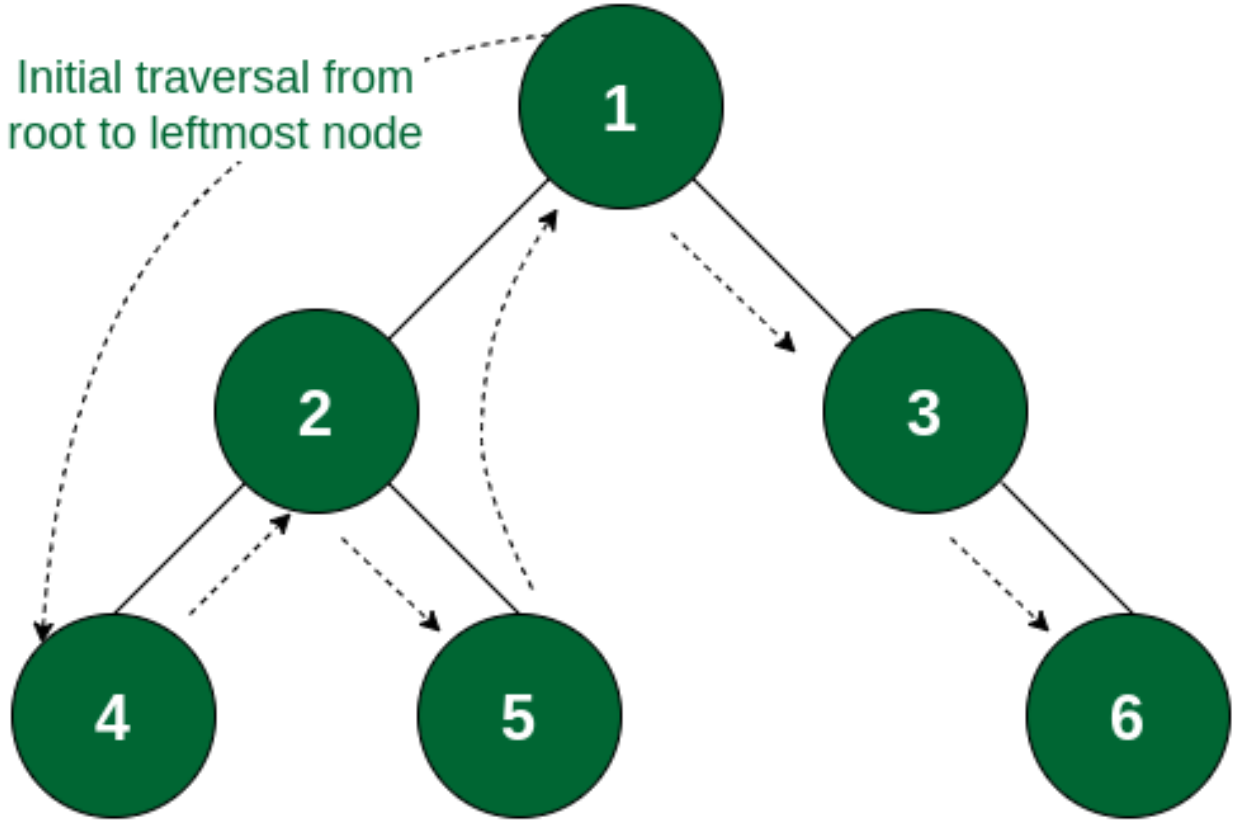
In-order Traversal

1. Step 5: Left subtree of node 1 and the node itself is visited. So now the right subtree of 1 will be traversed i.e., move to node 3. As node 3 has no left subtree so it gets visited.



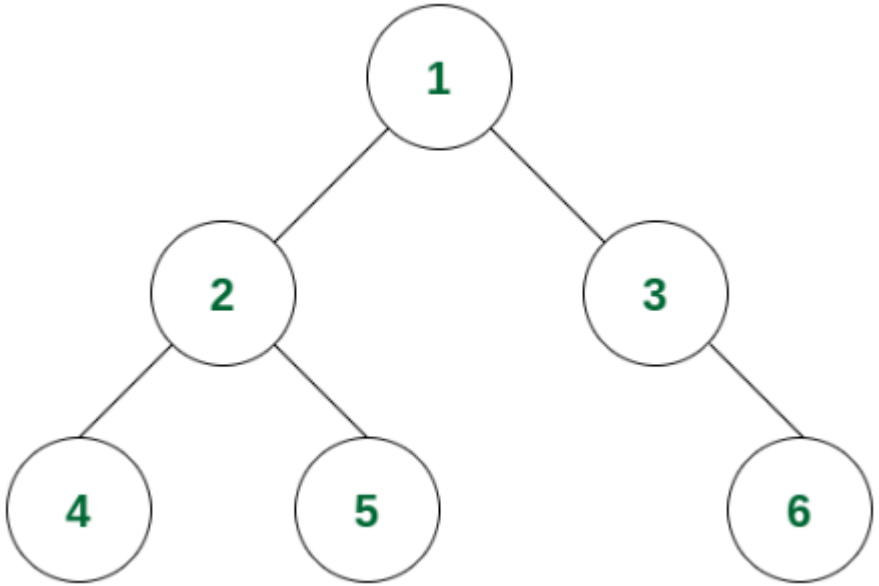
In-order Traversal

1. Step 6: The left subtree of node 3 and the node itself is visited. So traverse to the right subtree and visit node 6. Now the traversal ends as all the nodes are traversed.
2. So the order of traversal of nodes is 4 -> 2 -> 5 -> 1 -> 3 -> 6.



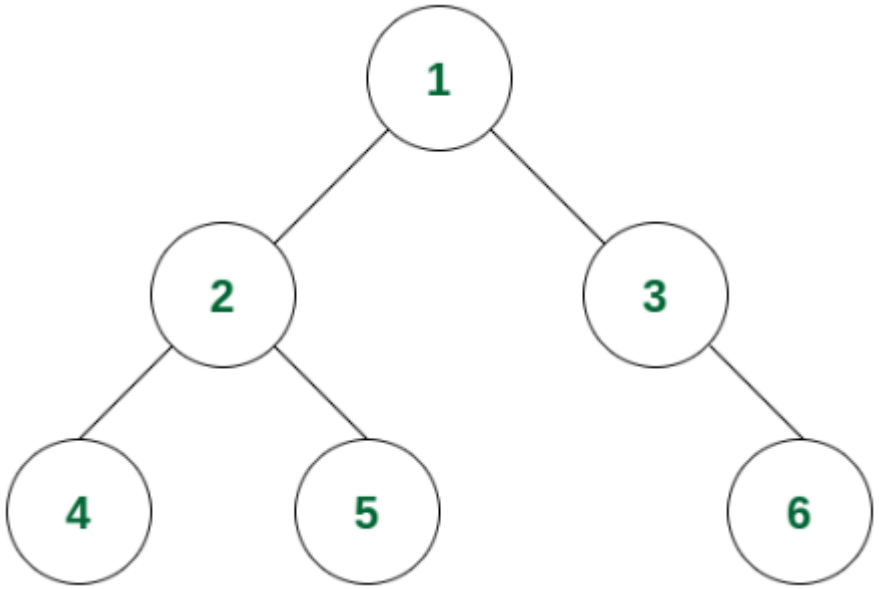
Preorder Traversal

1. Follows the **Root-Left-Right** policy where:
   - The root node of the subtree is visited first.
   - Then the left subtree is traversed.
   - At last, the right subtree is traversed.
2. Step 1: At first the root will be visited, i.e. node 1
3. Step 2: After this, traverse in the left subtree. Now the root of the left subtree is visited i.e., node 2 is visited.



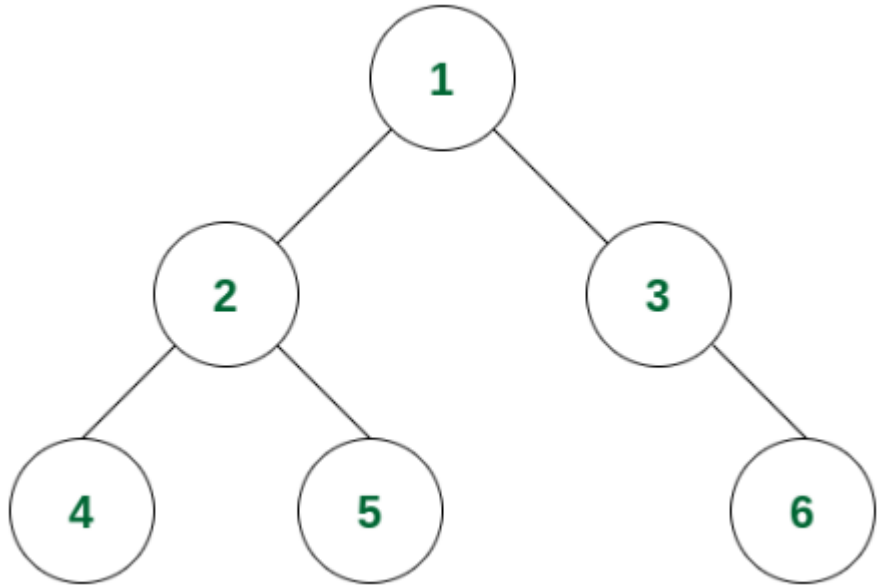
Preorder Traversal

1. Step 3: The left subtree of node 2 is traversed and the root of that subtree i.e., node 4 is visited.
2. Step 4: There is no subtree of 4 and the left subtree of node 2 is visited. So now the right subtree of node 2 will be traversed and the root of that subtree i.e., node 5 will be visited.
3. Step 5: The left subtree of node 1 is visited. So now the right subtree of node 1 will be traversed and the root node i.e., node 3 is visited.



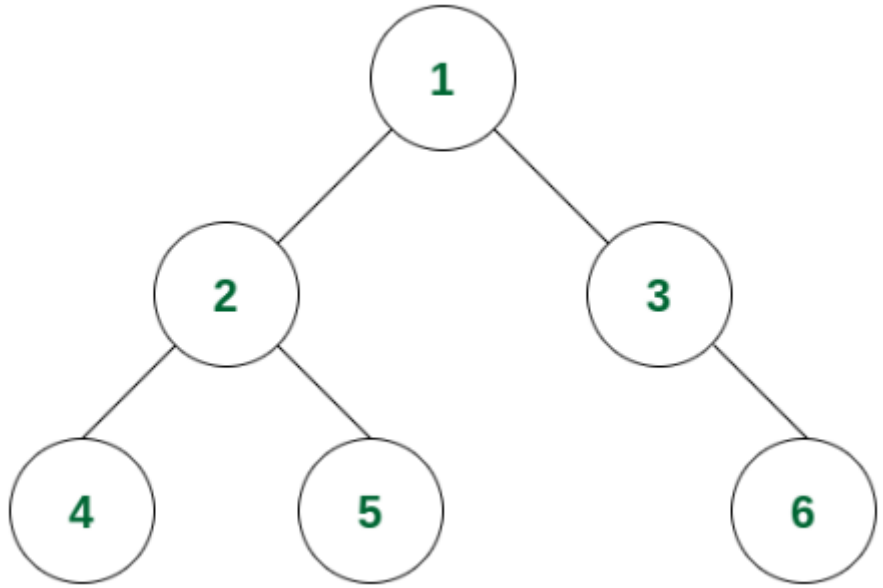
Preorder Traversal

1. Step 6: Node 3 has no left subtree. So the right subtree will be traversed and the root of the subtree i.e., node 6 will be visited. After that there is no node that is not yet traversed. So the traversa ends.
2. So the order of traversal of nodes is
   - 1 -> 2 -> 4 -> 5 -> 3 -> 6.



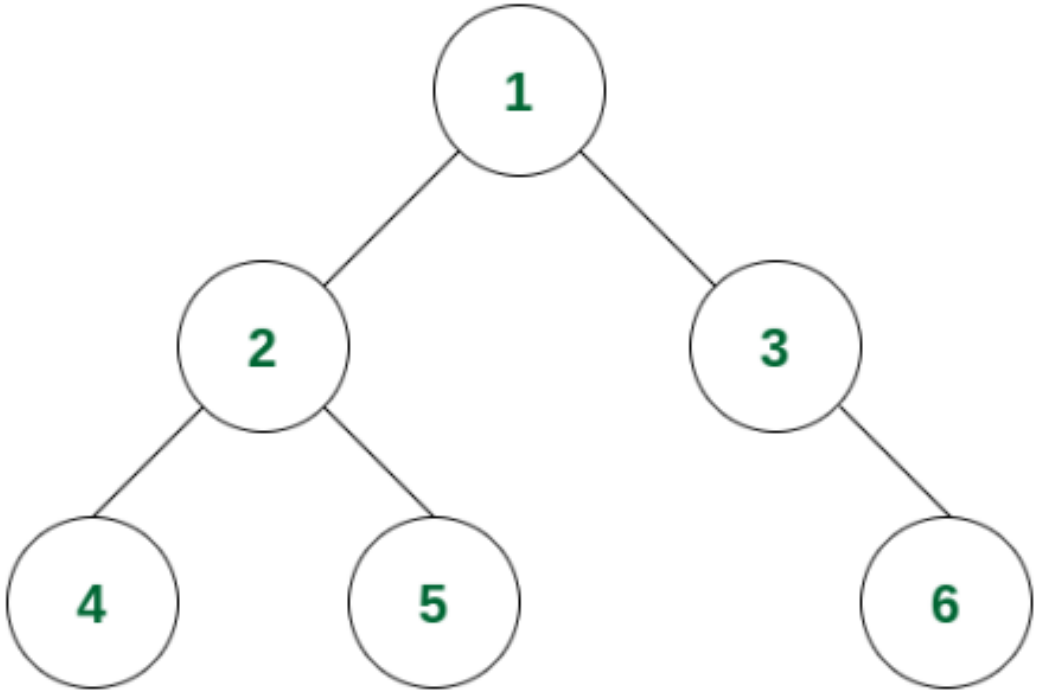
Postorder Traversal

1. Follows the **Left-Right-Root** policy such that for each node:
   - The left subtree is traversed first
   - Then the right subtree is traversed
   - Finally, the root node of the subtree is traversed



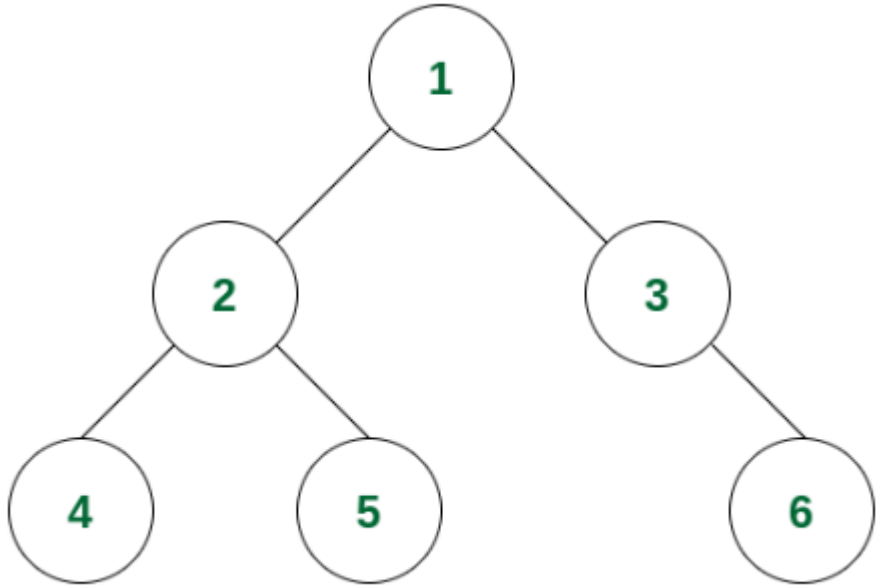
Postorder Traversal

1. Step 1: The traversal will go from 1 to its left subtree i.e., 2, then from 2 to its left subtree root, i.e., 4. Now 4 has no subtree, so it will be visited.
2. Step 2: As the left subtree of 2 is visited completely, now it will traverse the right subtree of 2 i.e., it will move to 5. As there is no subtree of 5, it will be visited.



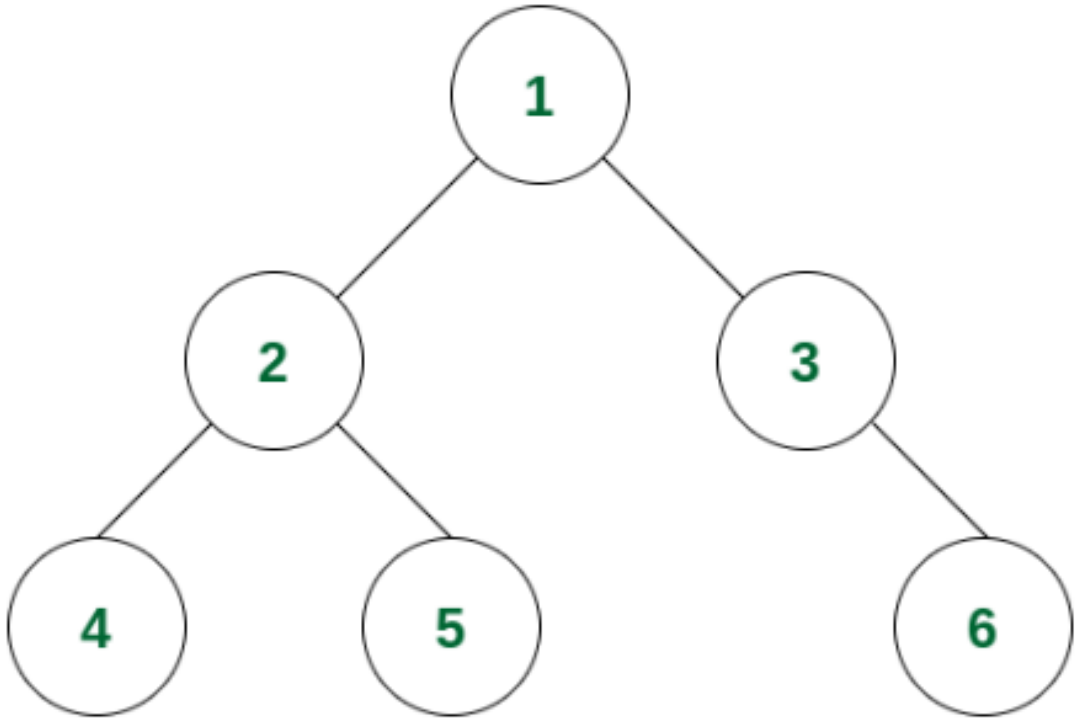
Postorder Traversal

1. Step 3: Now both the left and right subtrees of node 2 are visited. So now visit node 2 itself.
2. Step 4: As the left subtree of node 1 is traversed, it will now move to the right subtree root, i.e., 3. Node 3 does not have any left subtree, so it will traverse the right subtree i.e., 6. Node 6 has no subtree and so it is visited.



Postorder Traversal

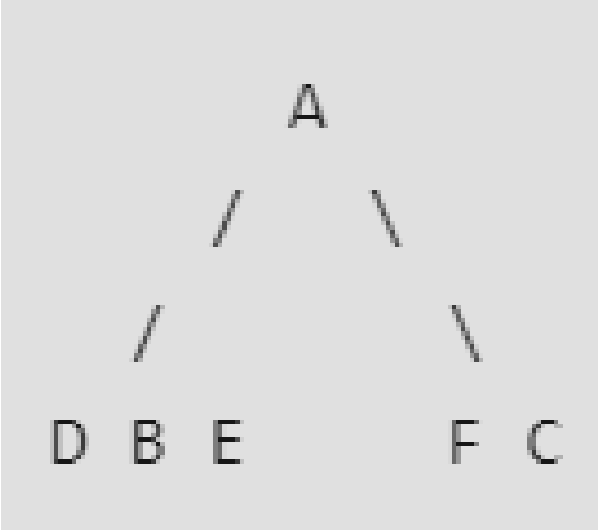
1. Step 5: All the subtrees of node 3 are traversed. So now node 3 is visited.
2. Step 6: As all the subtrees of node 1 are traversed, now it is time for node 1 to be visited and the traversal ends after that as the whole tree is traversed.
3. So the order of traversal of nodes is
   4 -> 5 -> 2 -> 6 -> 3 -> 1.



BT Construction

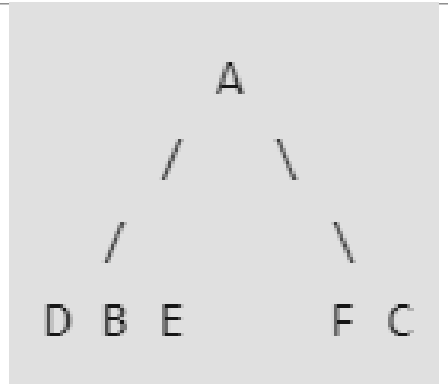
BT from Inorder and Preorder Traversals

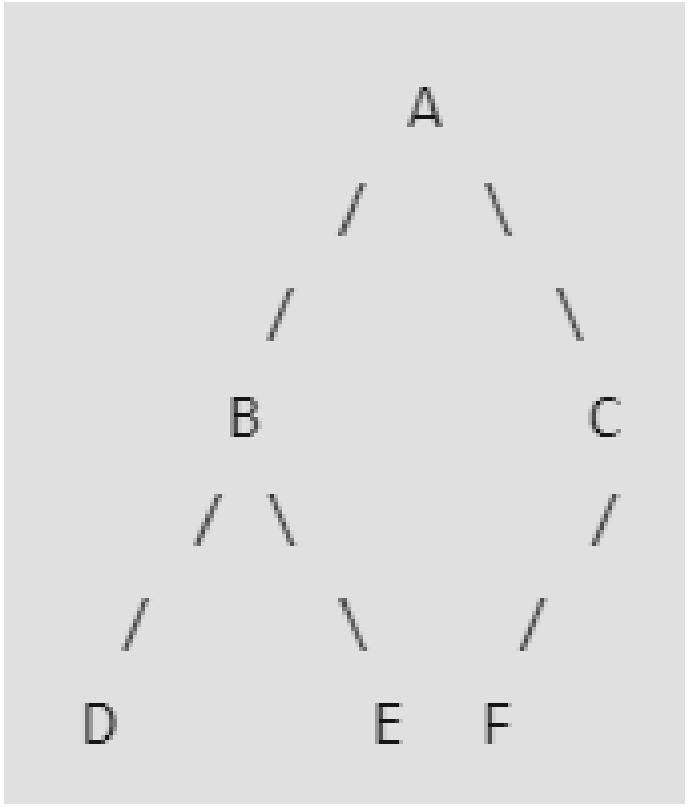
1. Inorder: Left Root Right
2. Preorder: Root Left Right
3. Let us consider the below traversals:
   - Inorder sequence: D B E A F C
   - Preorder sequence: A B D E C F
4. In a Preorder sequence, the leftmost element is the root of the tree.
5. So we know ‘A’ is the root for given sequences.
6. By searching ‘A’ in the Inorder sequence, we can find out all elements on the left side of ‘A’ is in the left subtree and elements on right in the right subtree.



BT from Inorder and Preorder Traversals

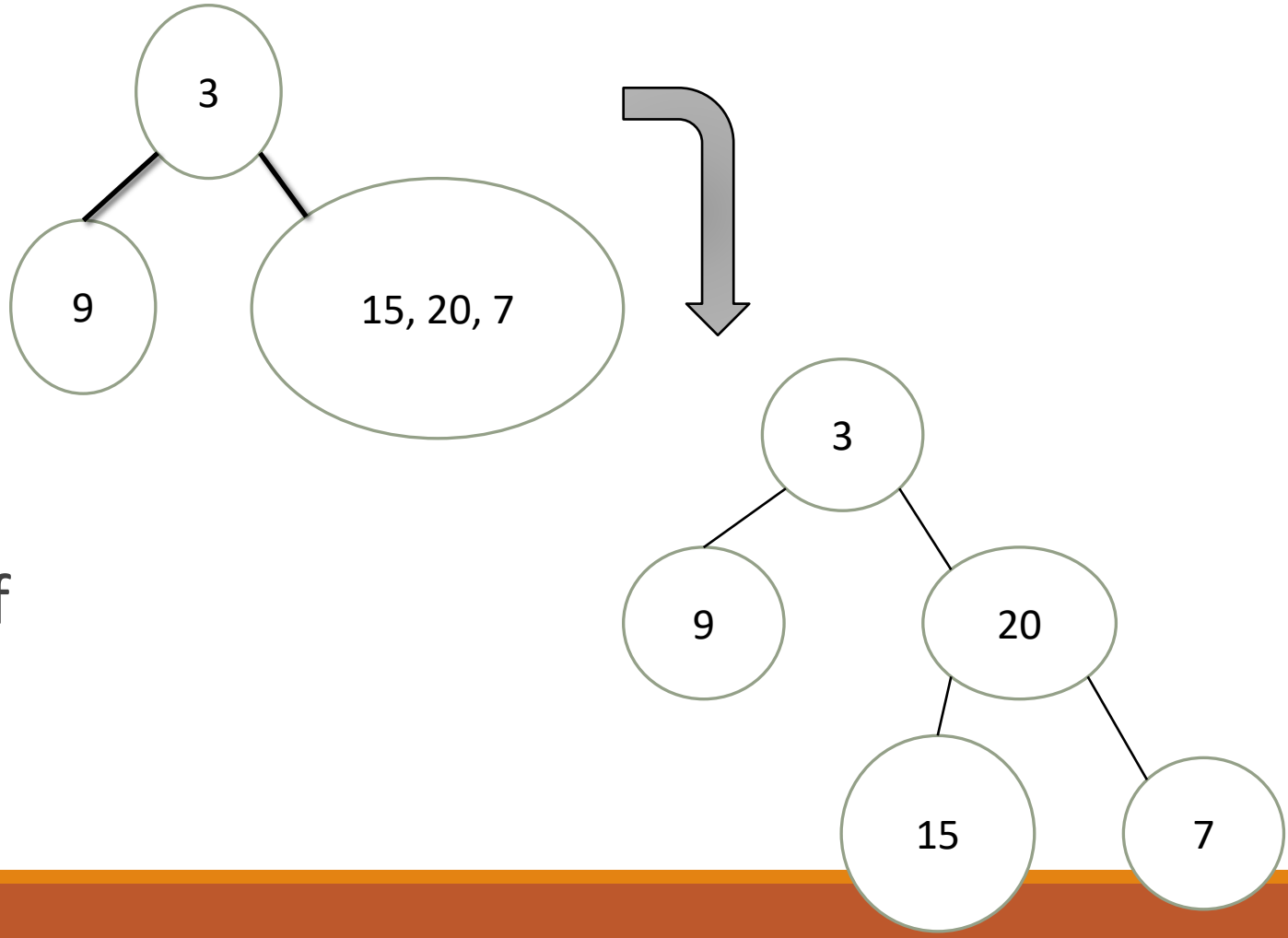
1. Inorder: Left Root Right
2. Preorder: Root Left Right
3. Let us consider the below traversals:
   - Inorder sequence: D B E A F C
   - Preorder sequence: A B D E C F
4. Perform same operation recursively on the left and right subtrees of the previous BT





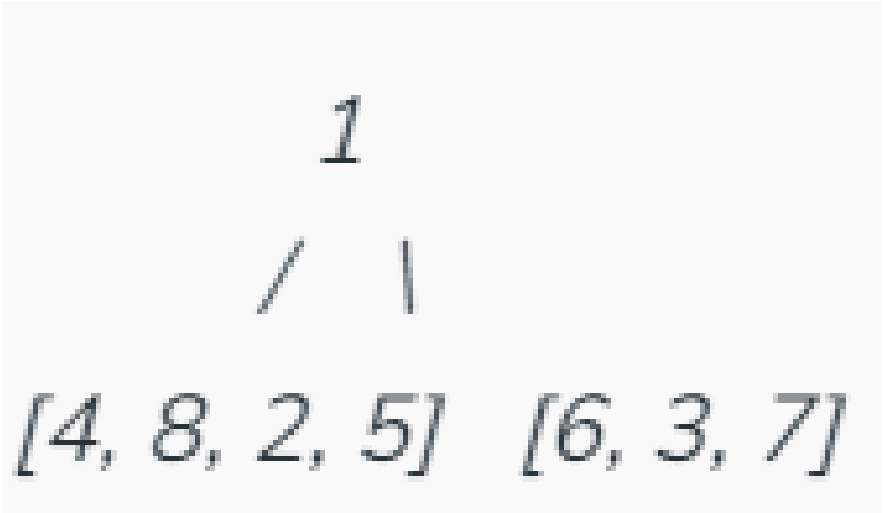
BT from Inorder and Preorder Traversals

1. Inorder: Left Root Right
2. Preorder: Root Left Right
3. Let us consider the below traversals:
   - Inorder sequence: 9,3,15,20,7
   - Preorder sequence: 3,9,20,15,7
4. In a Preorder sequence, the leftmost element is the root of the tree.



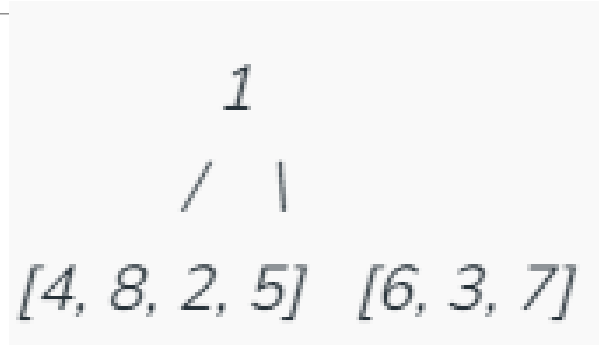
BT from Inorder and Postorder Traversals

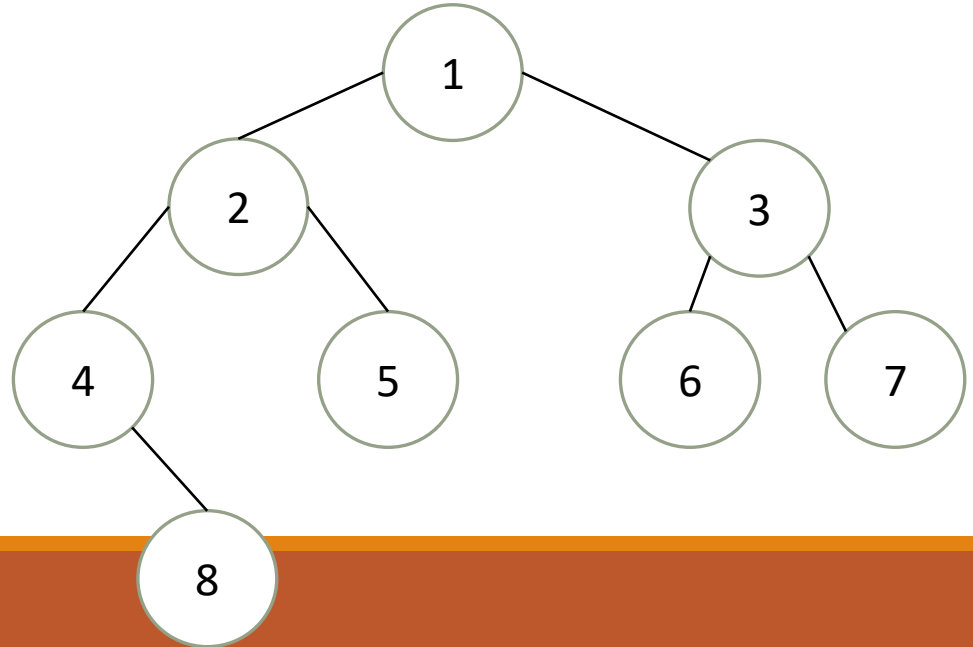
1. Inorder: Left Root Right
2. Postorder: Left Right Root
3. Let us consider the below traversals:
4. Inorder sequence: 4, 8, 2, 5, 1, 6, 3, 7
5. Postorder sequence: 8, 4, 5, 2, 6, 7, 3, 1
6. In a Postorder sequence, the rightmost element is the root of the tree – “1”
7. We search “1” in inorder sequence to find the left and right subtrees of the root.
8. Repeat recursively



BT from Inorder and Postorder Traversals

1. Inorder: Left Root Right
2. Postorder: Left Right Root
3. Let us consider the below traversals:
4. Inorder sequence: 4, 8, 2, 5, 1, 6, 3, 7
5. Postorder sequence: 8, 4, 5, 2, 6, 7, 3, 1
6. In a Postorder sequence, the rightmost element is the root of the tree – “1”
7. Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3} Make the created tree as right child of root.





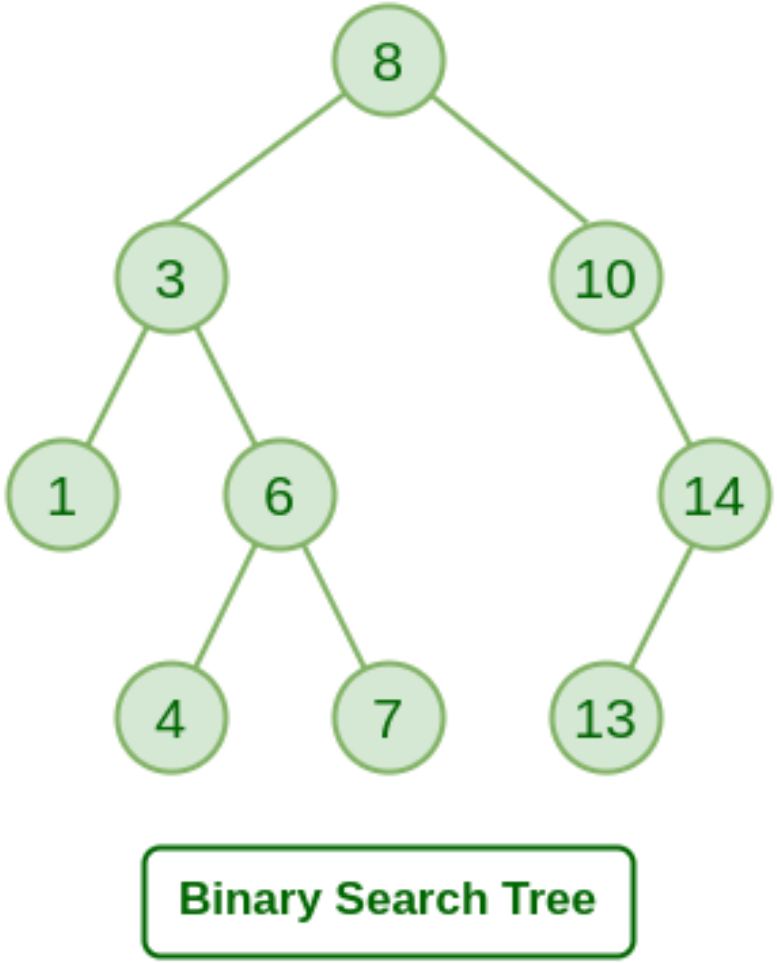
Binary Search Trees (BST)

BST

1. Deals with the problem: How should items in a list be stored so that an item can be easily located?
2. Primary goal: To implement a searching algorithm that finds items efficiently when the items are totally ordered.
3. We use Binary Search Trees (BST) for this purpose.
4. BSTs are a variant of Binary Trees

Binary Search Tree

1. A binary search tree (BST) is a binary tree where each node/vertex has a comparable key (associated value) and satisfies the restriction that
   a) the key in any node is larger than the keys in all nodes in that node's left subtree and
   b) the key in any node is smaller than the keys in all nodes in that node's right subtree.
   c) The left and right subtree each must also be a binary search tree.



Binary Search Tree – Operations

1. Possible operations:
   - Construction of a BST by Insertion of nodes
   - Insertion of a node in a previously created BST
   - Traversal, and Searching
   - Deletion of a node

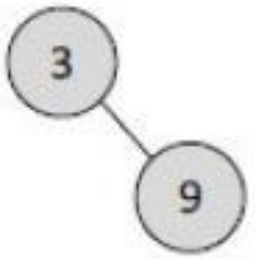
Insertion of a Node in BST

1. Find where the node can be inserted so as to make the resulting tree a BST.
2. While inserting a node in a BST the value being inserted is compared with the root node.
3. A left sub-tree is taken if the value is smaller than the root node
4. A right sub-tree if it is greater or equal to the node.
5. This operation is repeated at each level till a node is found whose left or right sub-tree is empty.
6. Finally, the new node is appropriately made the left or right child of this node.

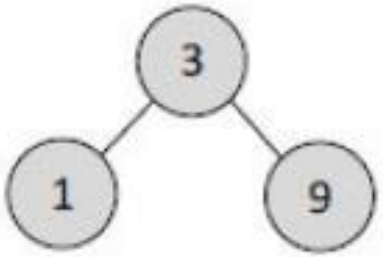
Insertion of a Node in BST

1. Example: Insert these nodes and create a BST stepwise: 3, 9, 1, 4, 7, 11

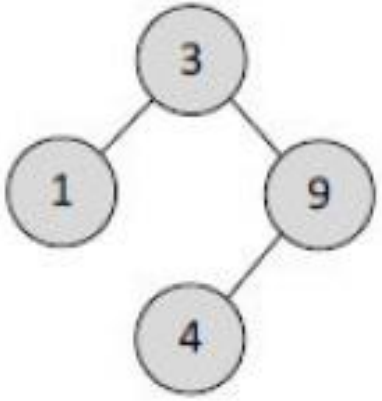
Step 1



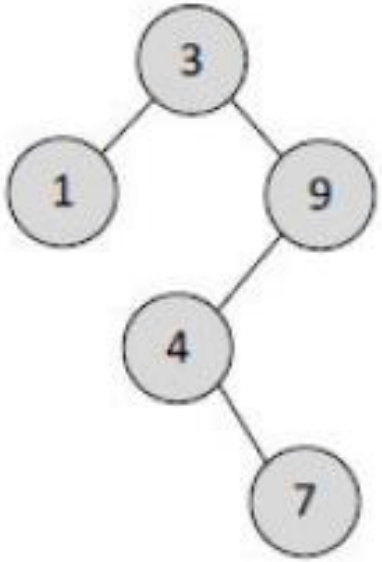
Step 2



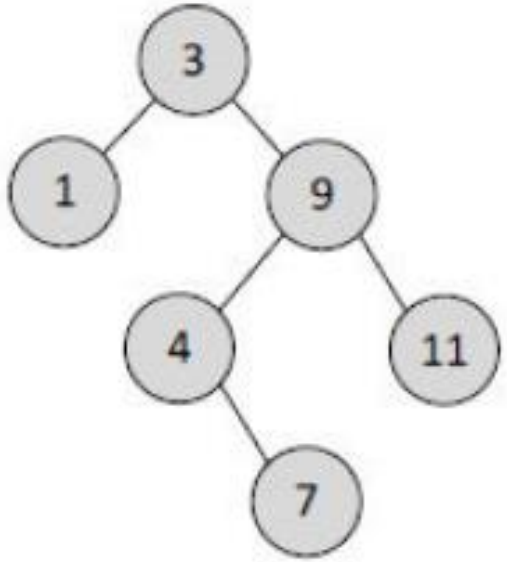
Step 3



Step 4



Step 5



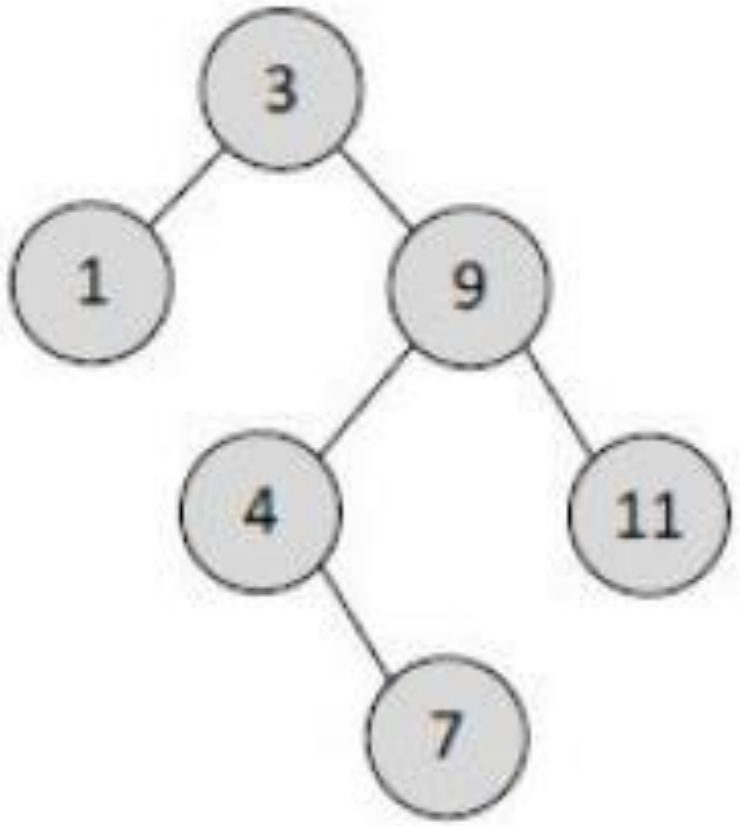
Step 6

Searching of a Node in BST

1. To search any node in a binary tree, initially the value to be searched (key) is compared with the root node.
2. If they match then the search is successful.
3. If the value is greater than the root node then searching process proceeds in the right sub-tree of the root node,
4. Otherwise, it proceeds in the left sub-tree of the root node.

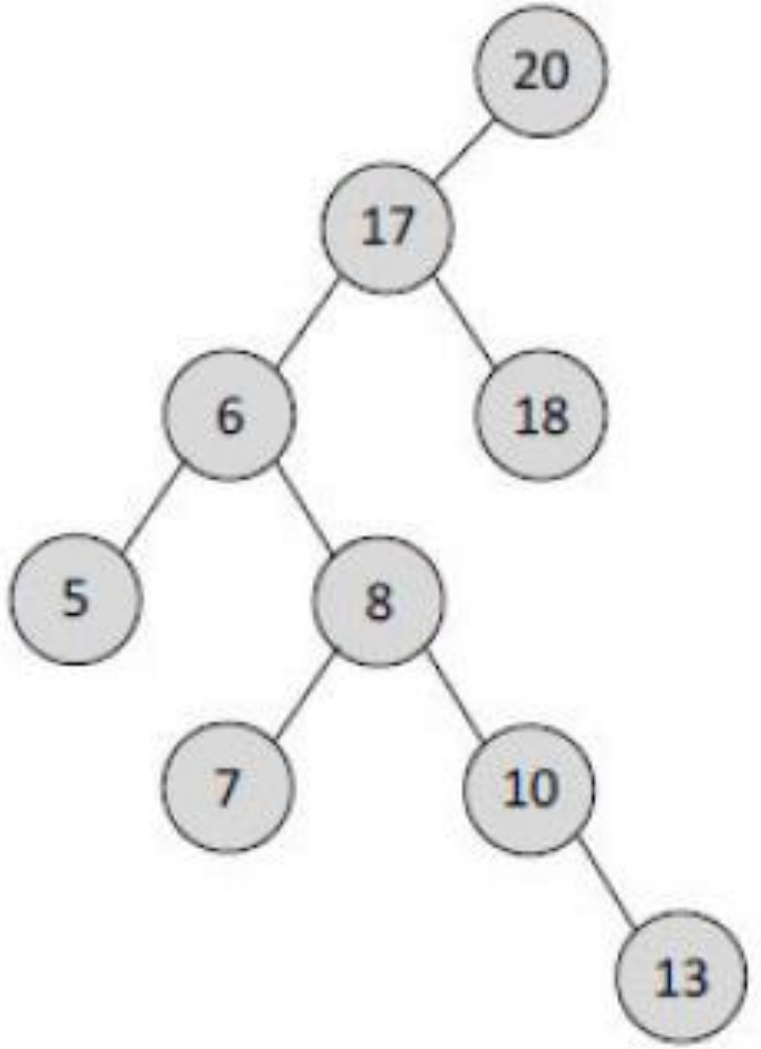
Searching of a Node in BST

1. Example: Search for 7 in the given BST
2. **Efficient because at each step, we eliminate half of the tree to be searched**.
3. We know that we have to scan only the right sub-tree of node 3, since 7 is greater than 3.
4. Likewise, when we descend down the tree and reach 9 we have to search only its left sub-tree as 7 is less than 9.
5. And so on.



Traversal of a BST

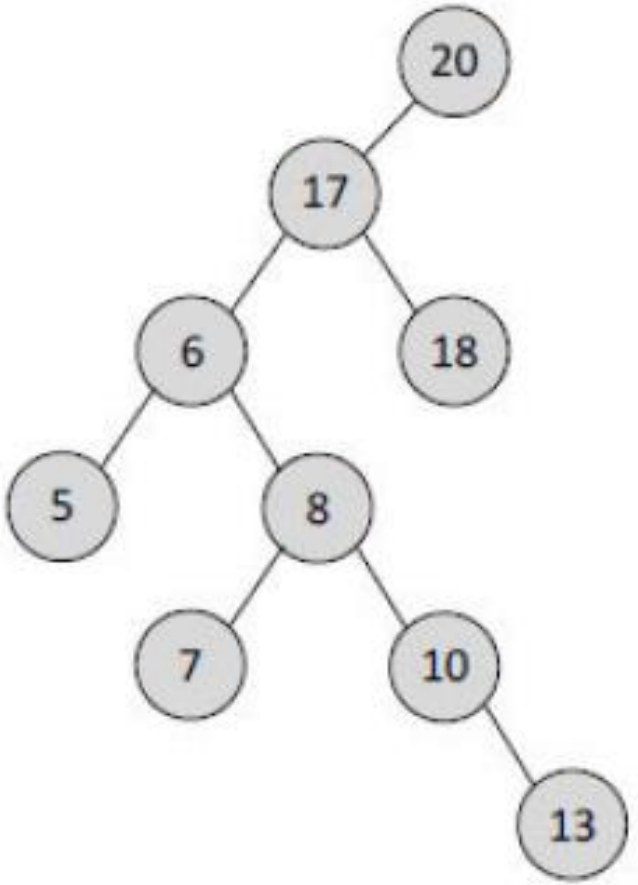
1. The traversal of a BST is to visit each node in the tree exactly once.
2. There are three popular methods of BST traversal—
3. Inorder traversal:
   - (1) Traverse the **left sub-tree** in in-order
   - (2) Visit the **root**
   - (3) Traverse the **right** sub-tree in in-order



Inorder : 5, 6, 7, 8, 10, 13, 17, 18, 20

Traversal of a BST

1. Pre-order traversal:
   - (1) Visit the root
   - (2) Traverse the left sub-tree in pre-order
   - (3) Traverse the right sub-tree in pre-order
2. Post-order traversal:
   - (1) Traverse the left sub-tree in post-order
   - (2) Traverse the right sub-tree in post-order
   - (3) Visit the root



Inorder : 5, 6, 7, 8, 10, 13, 17, 18, 20

Prorder : 20, 17, 6, 5, 8, 7, 10, 13, 18

Postorder : 5, 7, 13, 10, 8, 6, 18, 17, 20

Construct BST from given preorder traversal

1. Input: {10, 5, 1, 7, 40, 50}
2. The first element of preorder traversal is always the root. We first construct the root.
3. Then we find the index of the first element which is greater than the root. Let the index be ‘i’.
4. The values between root and ‘i’ will be part of the left subtree, and the values between ‘i'(inclusive) and ‘n-1’ will be part of the right subtree.
5. Divide the given pre[] at index “i” and recur for left and right sub-trees.

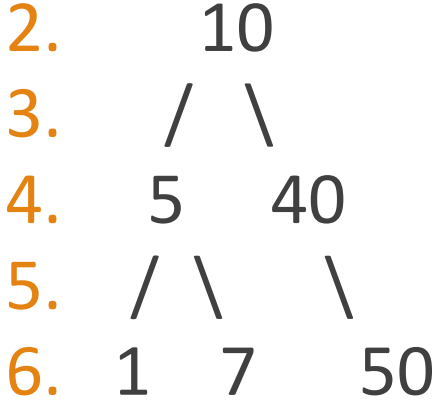
Construct BST from given preorder traversal

1. For example in {10, 5, 1, 7, 40, 50}, 10 is the first element, so we make it root. Now we look for the first element greater than 10, we find 40. So we know the structure of BST is as follows.:
2. 10
3. / \
4. {5, 1, 7} {40, 50}

5. We recursively follow the above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree.

Construct BST from given postorder traversal

1. given traversal is {1, 7, 5, 50, 40, 10}, then following tree should be constructed and root of the tree should be returned.



Construct BST from given postorder traversal

1. The last element of postorder traversal is always root. We first construct the root.
2. Then we find the index of last element which is smaller than root.
3. Let the index be ‘i’. The values between 0 and ‘i’ are part of left subtree, and the values between ‘i+1’ and ‘n-2’ are part of right subtree.
4. Divide given post[] at index “i” and recur for left and right sub-trees.
5. 10
6. / \
7. / \
8. {1, 7, 5} {50, 40}

Construct BST from given postorder traversal

1. We recursively follow above steps for subarrays {1, 7, 5} and {40, 50}, and get the complete tree.

Thank You