DATA STRUCTURES (ITPC-203)

Trees – Part I

Dr. Sanga Chaki

Department of Information Technology

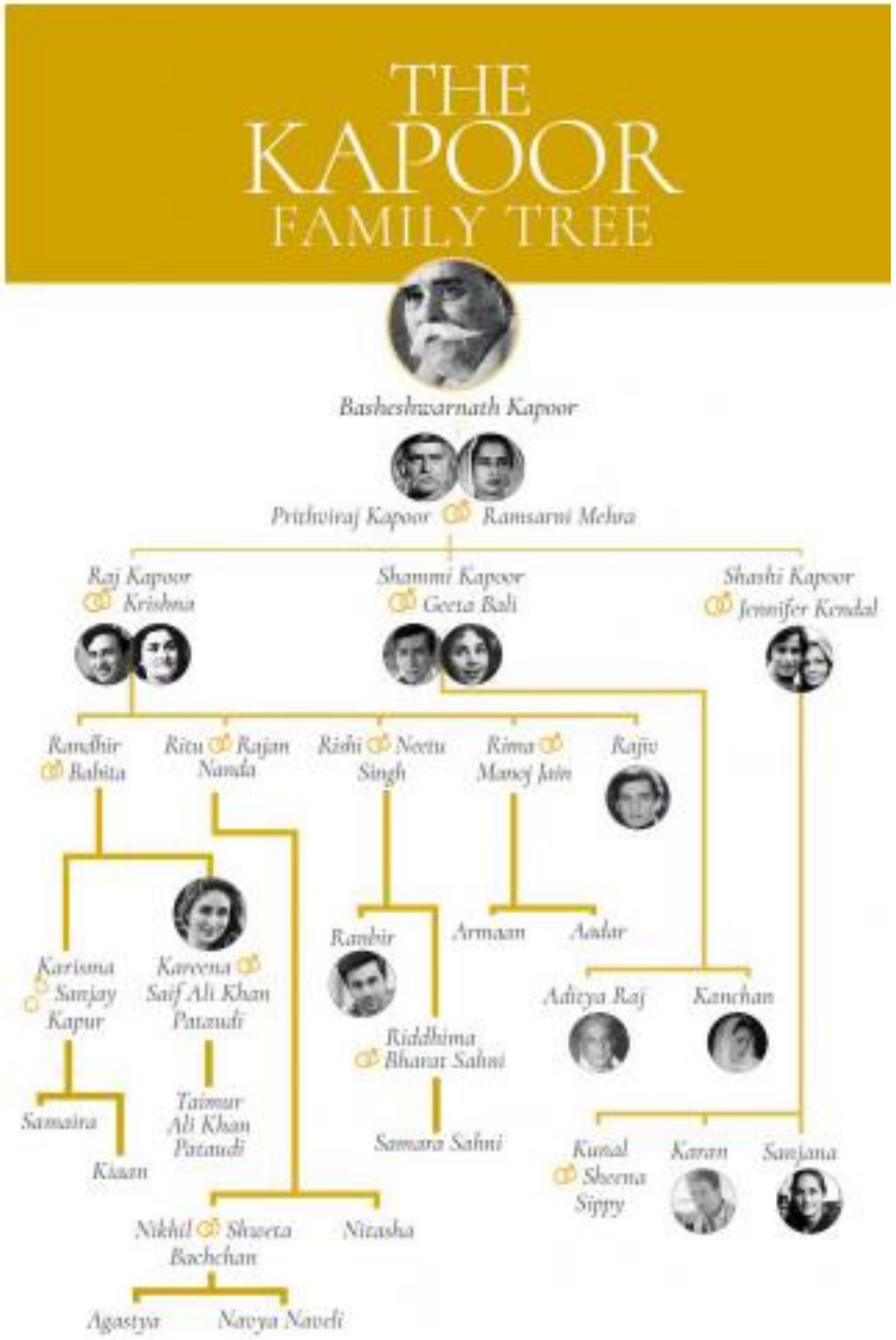
Dr. B. R. Ambedkar National Institute of Technology, Jalandhar

Contents

Tree Terminology

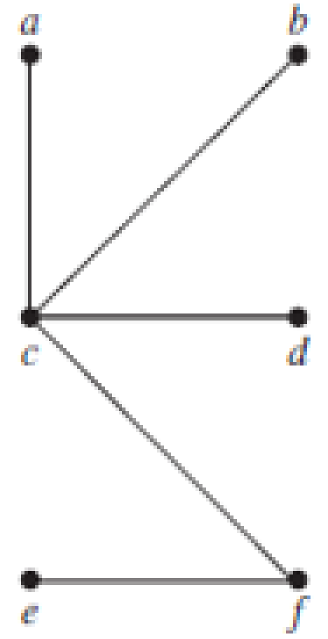
Trees

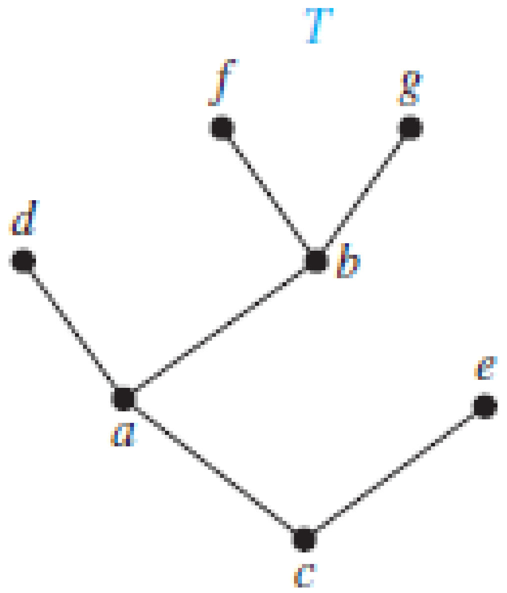
1. A particular type of graph
2. So named because such graphs resemble inverted trees.
3. Example: family trees are graphs that represent genealogical charts.
4. Family trees use vertices to represent the members of a family and edges to represent parent–child relationships.



Trees

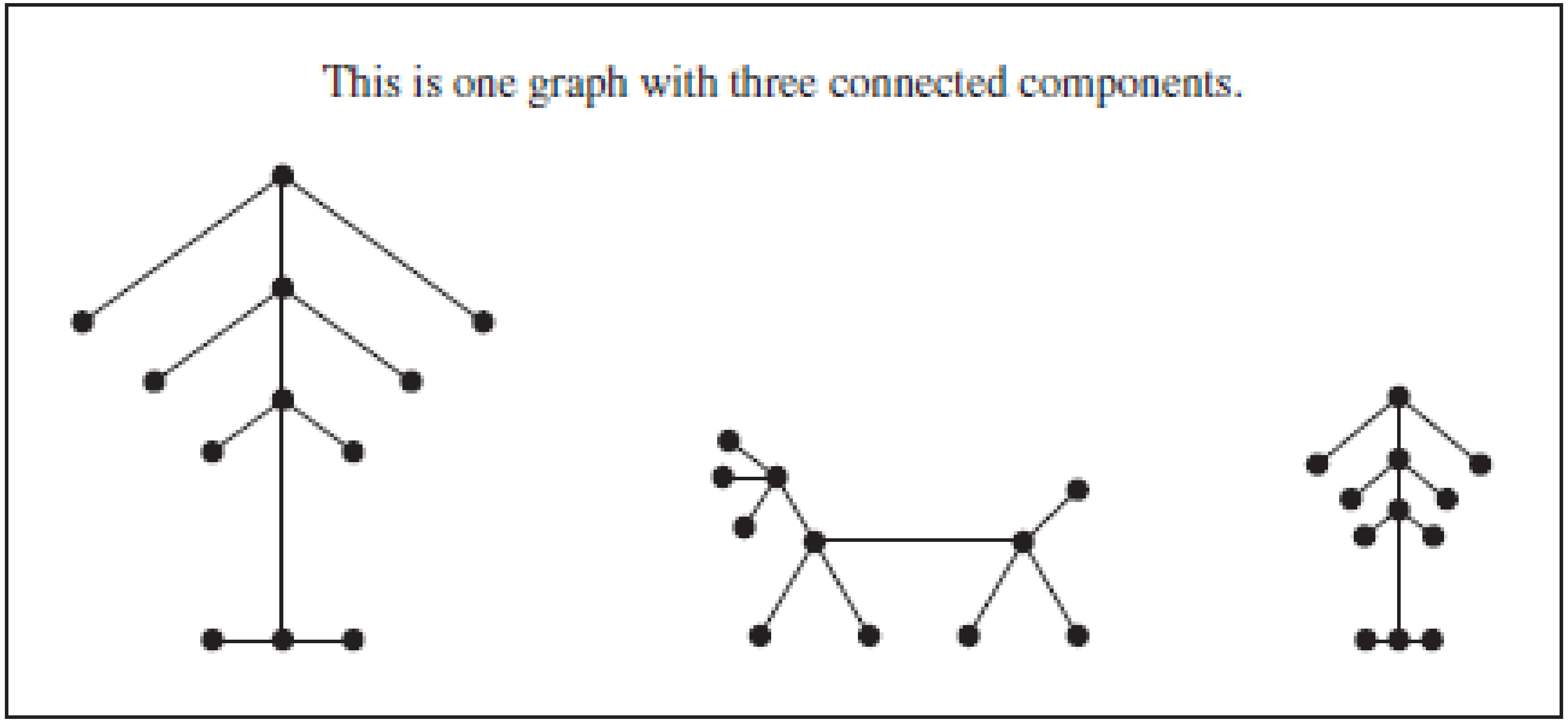
1. **Definition: A tree is a <u>connected</u> undirected graph with no <u>simple circuits</u>.**
2. Connected graph: A graph is a connected graph if, for each pair of vertices, there exists at least one single path which joins them
3. Circuits: A circuit is path that begins and ends at the same vertex.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





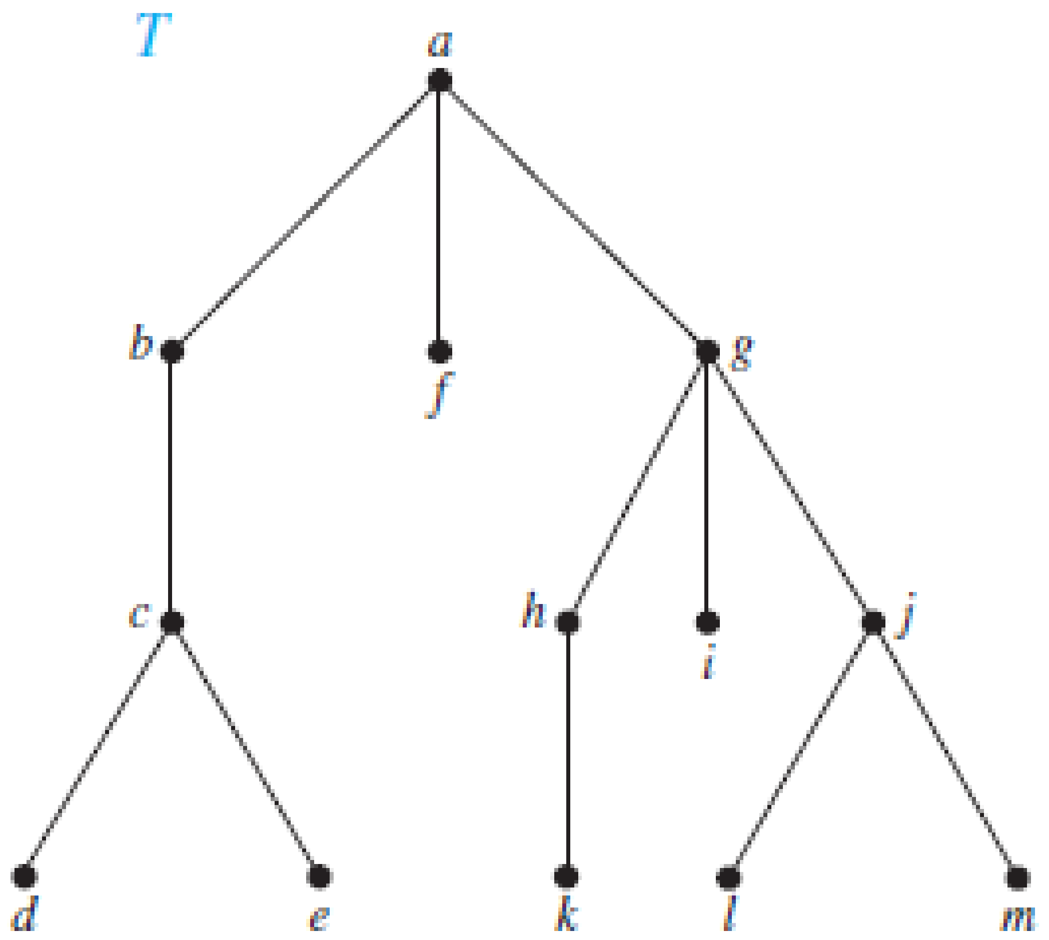
Forests

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



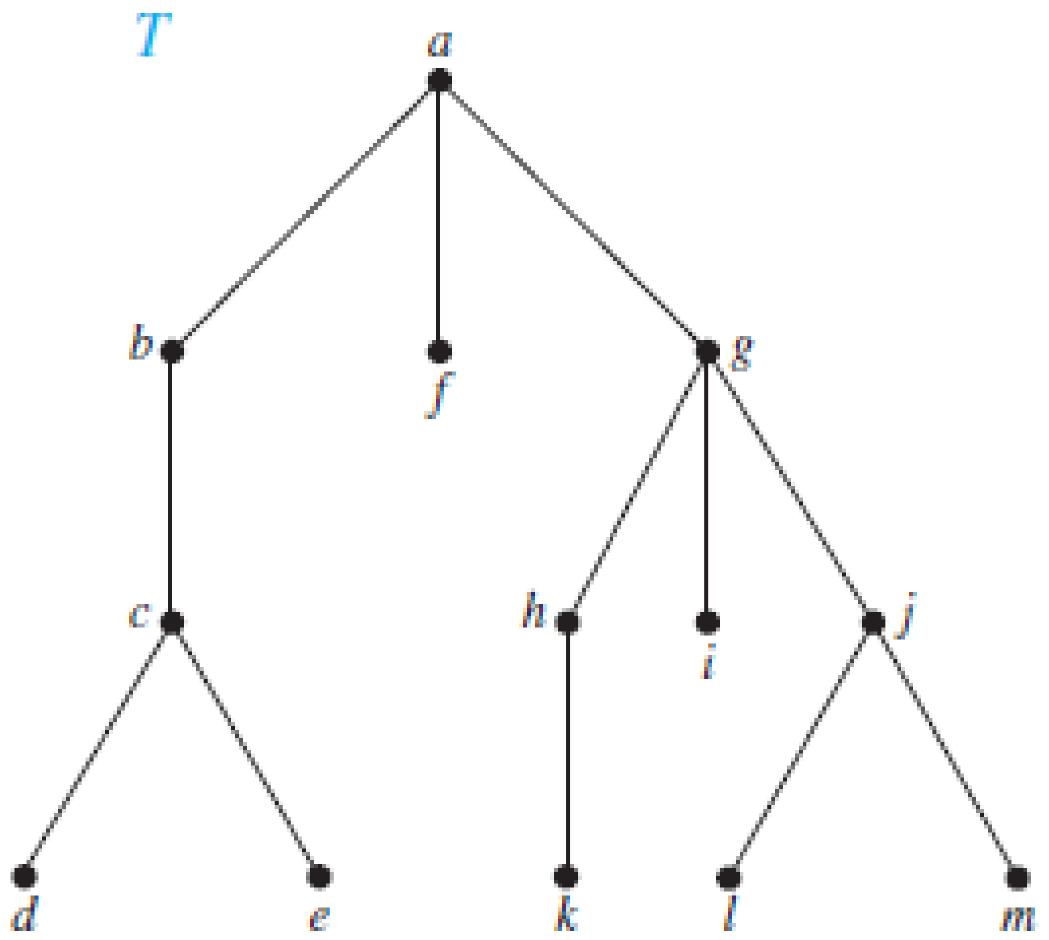
Rooted Trees

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



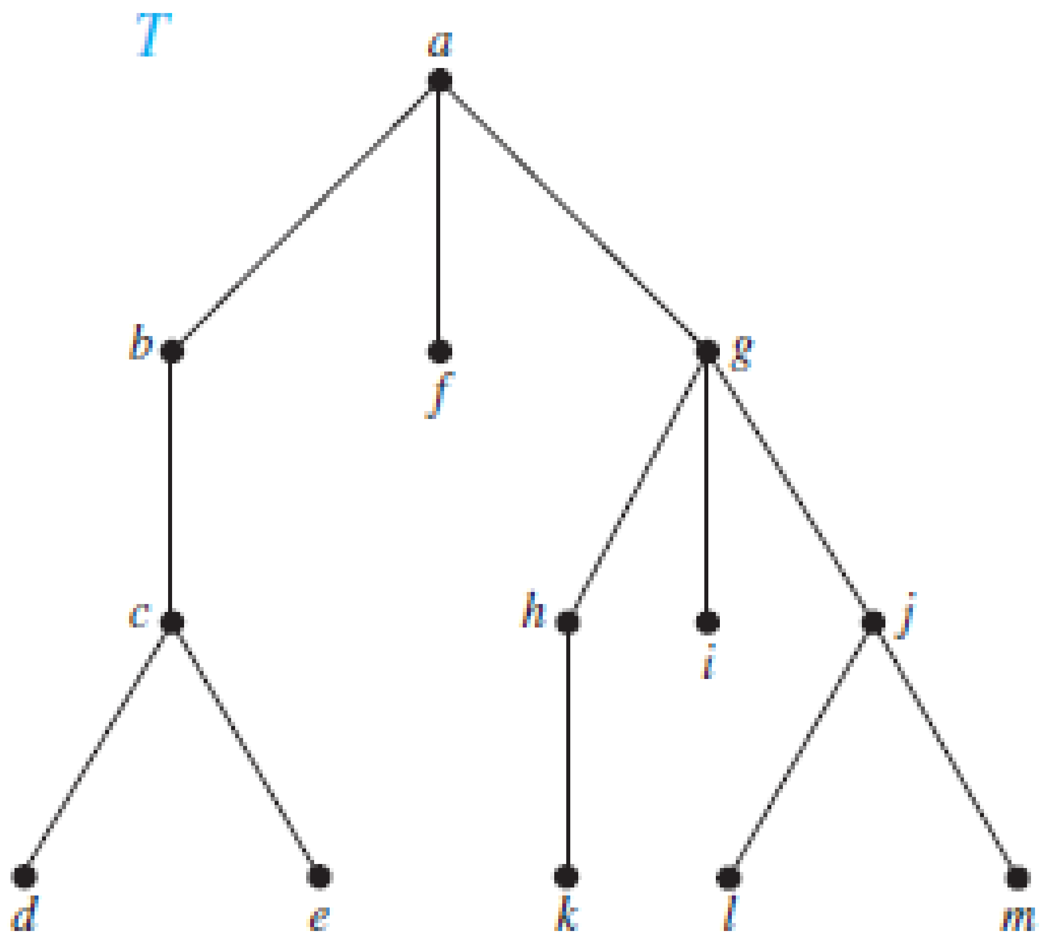
Rooted Tree Terminology

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



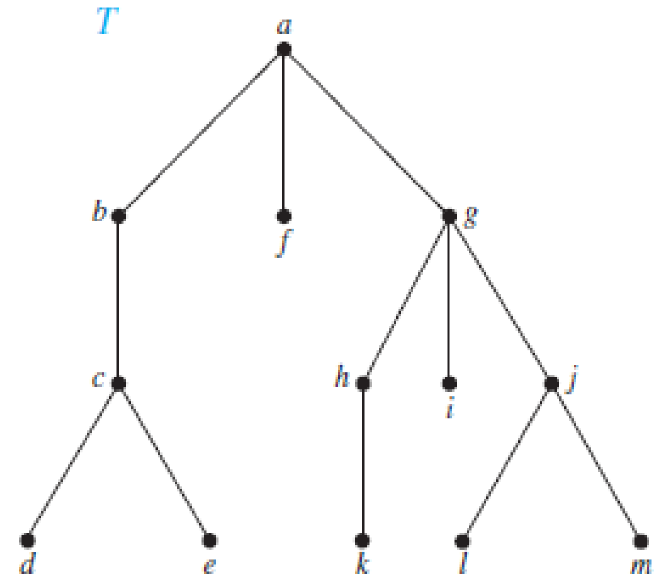
Rooted Tree Terminology

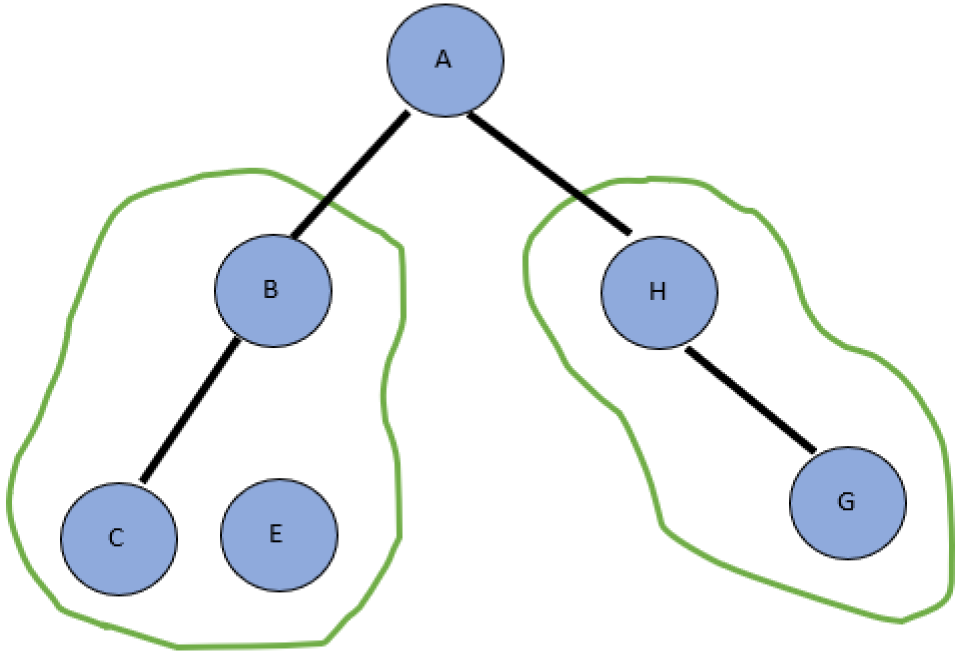
1. The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root
   - its parent, its parent's parent, and so on, until the root is reached
   - Ancestors of m are g and j
2. The **descendants** of a vertex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



Rooted Tree Terminology

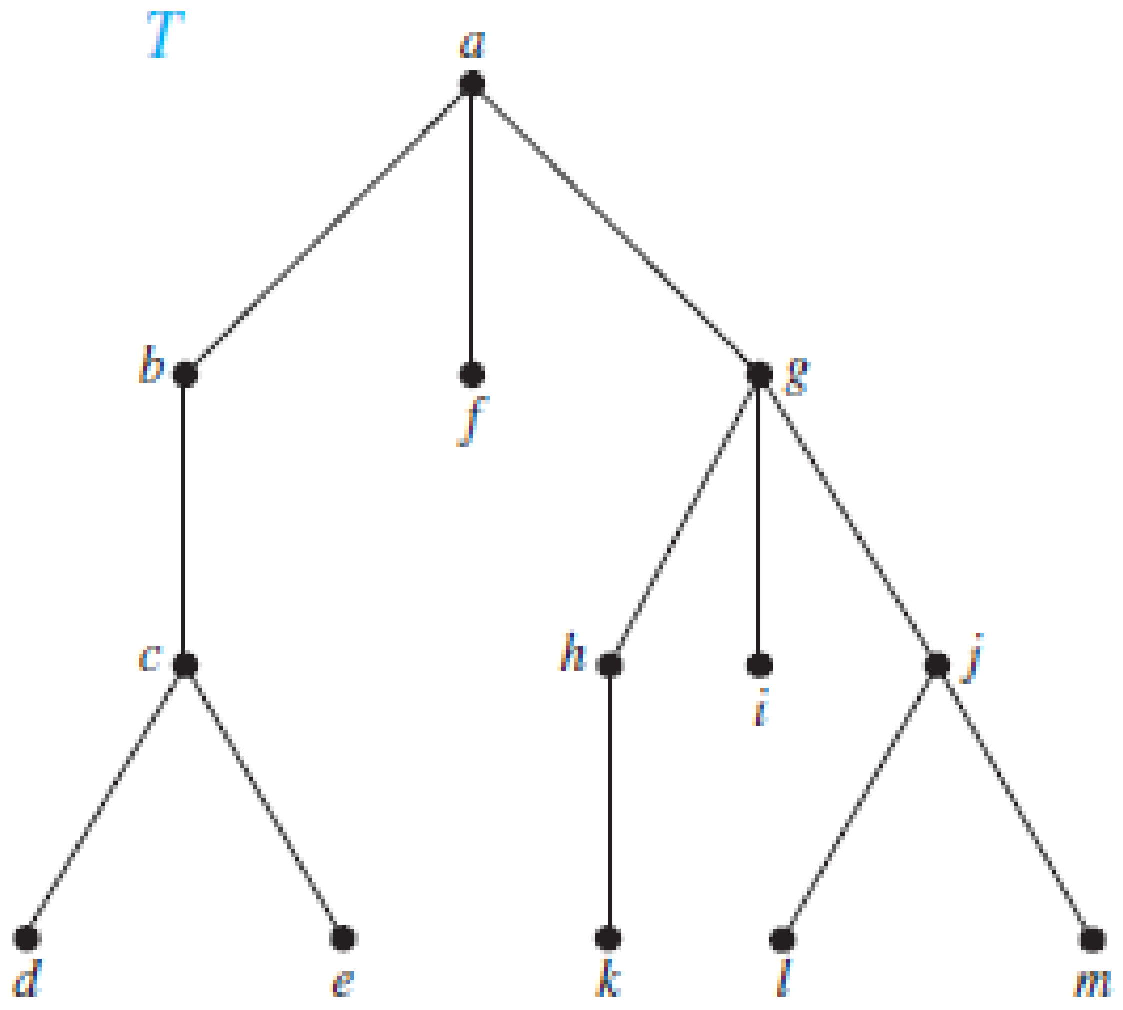
1. A vertex of a rooted tree is called a **leaf** if it has no children.
   - Eg. d, e etc
2. Vertices that have children are called **internal vertices**.
   - Eg. b, c etc
3. If ***a*** is a vertex in a tree, the **subtree** with ***a*** as its root is the subgraph of the tree consisting of ***a***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





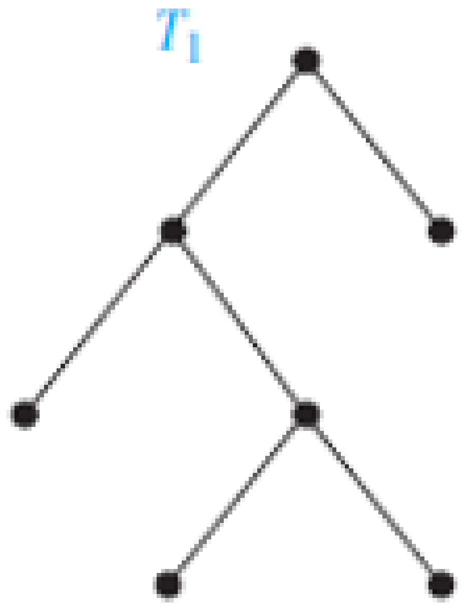
Rooted Tree

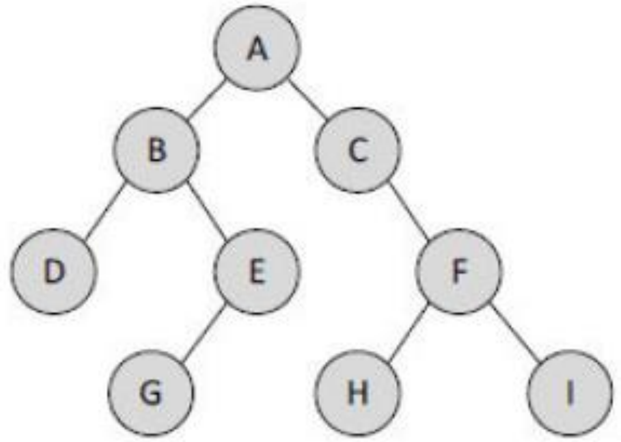
1. **Example**: In the below rooted tree T (with root a), find
   - the parent of c,
   - the children of g,
   - The siblings of h,
   - all ancestors of e,
   - all descendants of b,
   - all internal vertices, and
   - all leaves.
   - What is the subtree rooted at g?



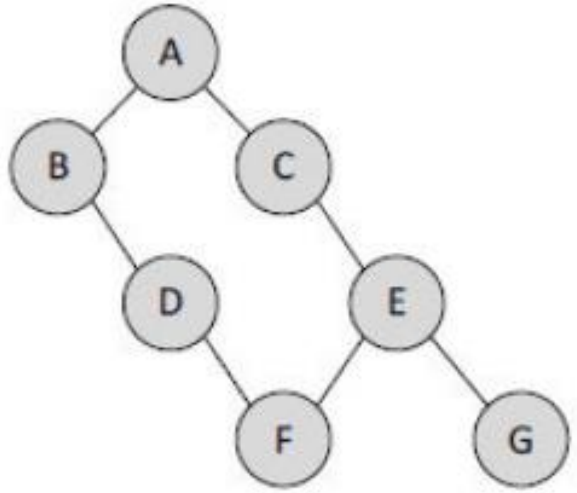
M-ary Tree – Binary Tree

1. A rooted 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.
3. An m-ary tree with m = 2 is called a **binary tree**.
4. So in a binary tree, each internal vertex can have a maximum of 2 children.





(a) Binary Tree



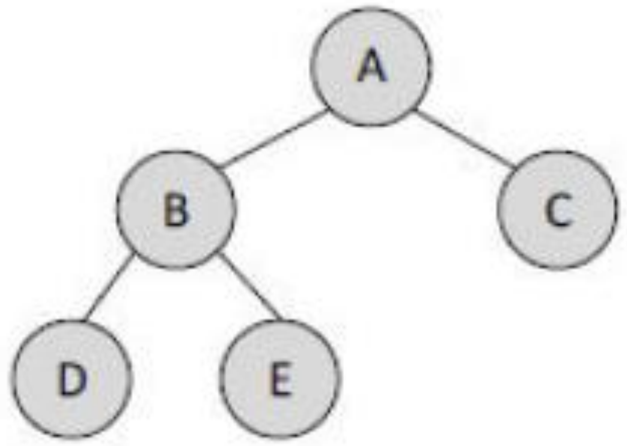
(b) Invalid Binary Tree

Binary Trees - Terminology

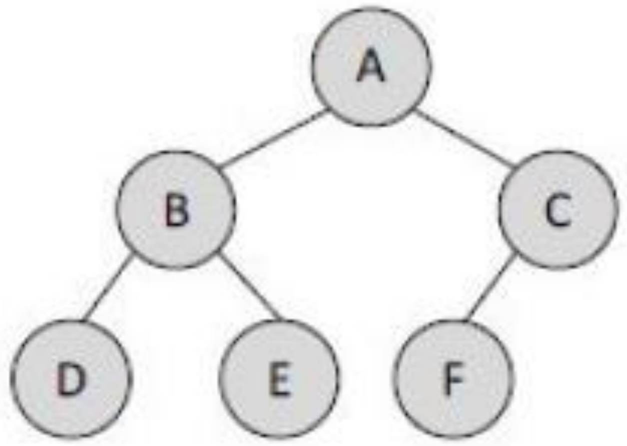
1. **Climbing**: The root of the tree is at the top and the leaves at the bottom. Going from the leaves to the root is called climbing the tree.
2. **Descending**: going from the root to the leaves is called descending the tree.
3. **Degree of a node** : The number of nodes connected to a particular node is called the degree of a particular node.
4. **Level** : The root of the tree has level 0. Level of any other node in the tree is one more than the level of its parent
5. **Depth** : Depth of a node is the maximum number of links from root to that node. The depth of a binary tree is the maximum level of any leaf in the tree. This equals the length of the longest path from the root to any leaf.
6. **Height** : Height of a node is the maximum number of links from that node to leaf node. Height of a binary tree is height of its root node.

Binary Trees - Terminology

1. **Strictly binary tree** : If every non-leaf node in a binary tree has non- empty left and right sub-trees, the tree is termed a strictly binary. So each node has either 2 or 0 children
2. **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.



(a) Strictly Binary Tree



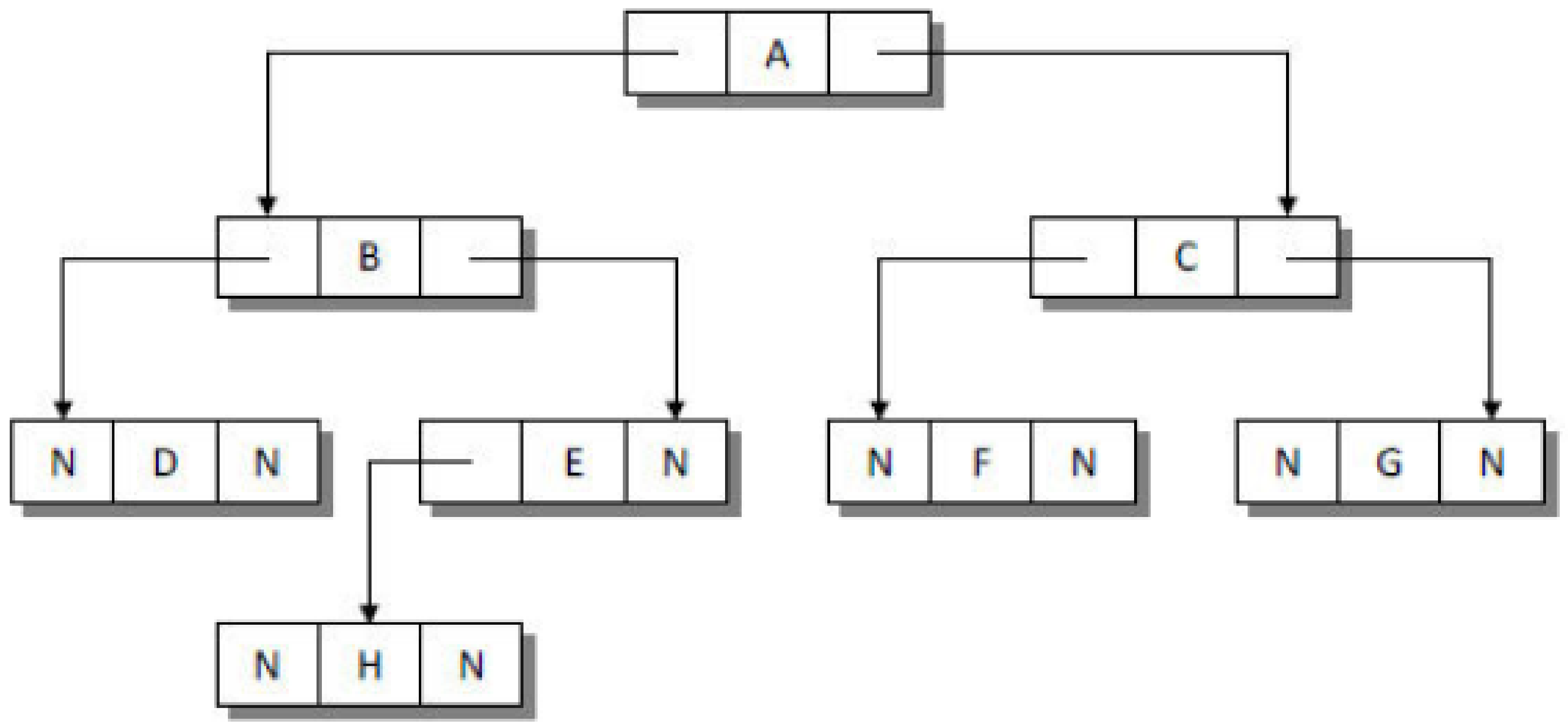
(b) Complete Binary Tree

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

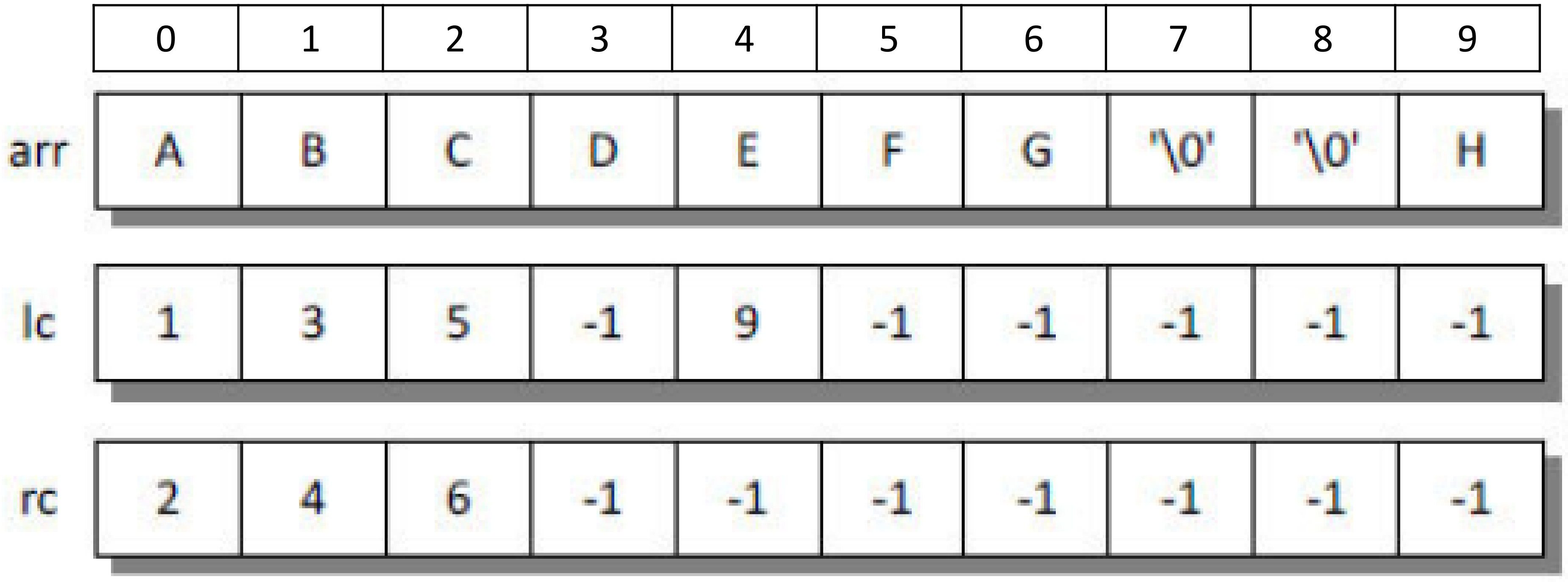
1. In linked representation each node contains addresses of its left child and right child, along with data value.
2. If a child is absent, the link contains a NULL value.



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

Array Representation of Binary Trees

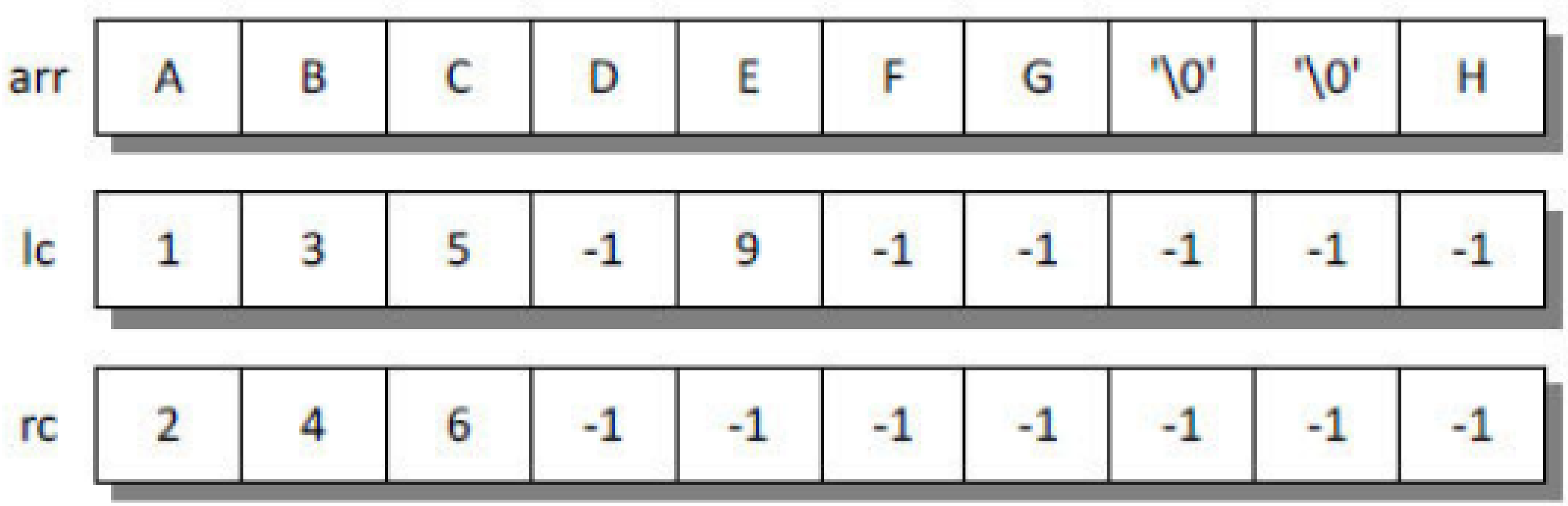
1. When a binary tree is represented by arrays three separate arrays are required.
2. One array arr stores the data fields of the nodes.
3. The other two arrays lc and rc represents the left child and right child of the nodes
4. Example array representation of the previous tree.



	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

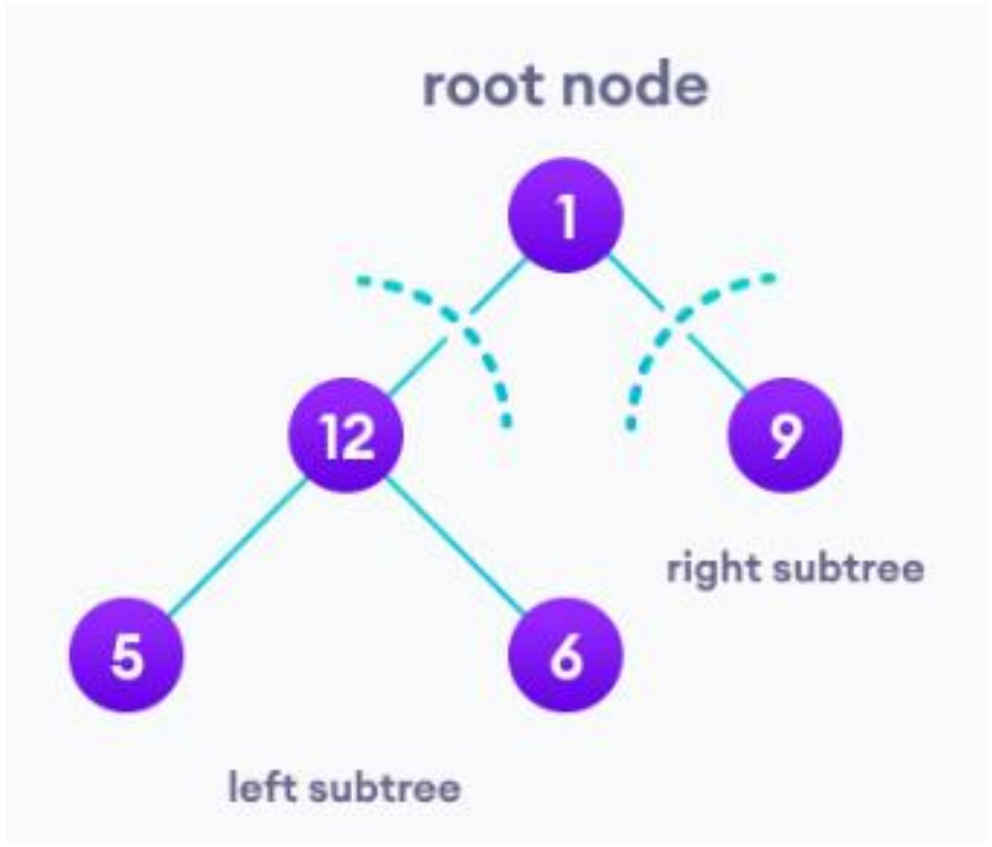
Tree Traversal algorithms

Tree Traversals

1. The goal: to visit each node
2. So we need to visit
   - all the nodes in the left subtree,
   - visit the root node and
   - visit all the nodes in the right subtree as well.
3. Depending on the order in which we do this, there can be three types of traversals.
   - ➢Inorder traversal
   - ➢Preorder traversal and
   - ➢Postorder traversal

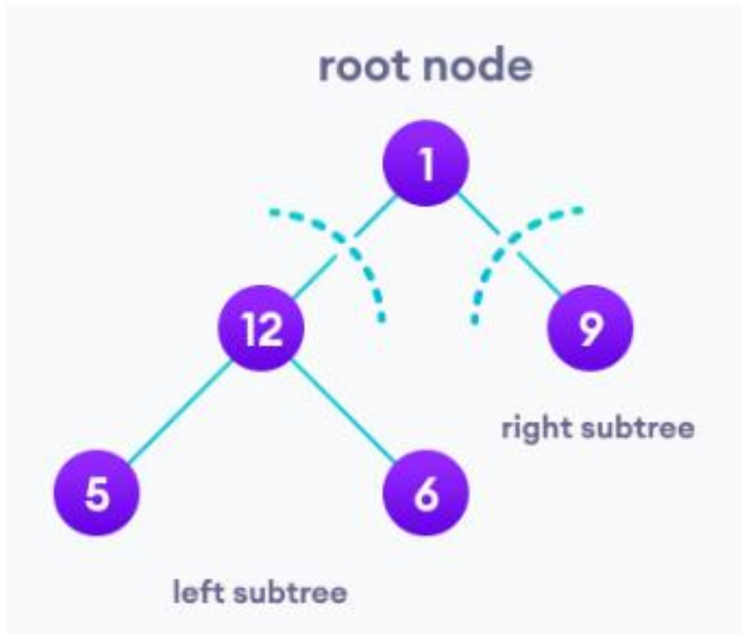
Inorder Traversal

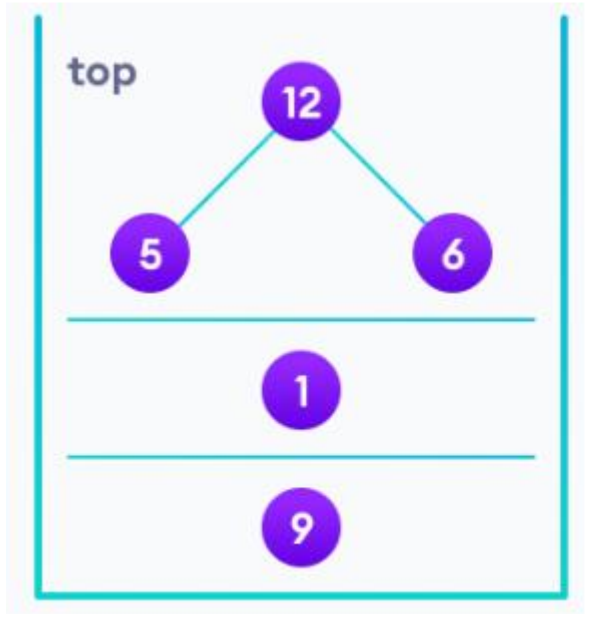
1. Follow the below steps to implement the idea (**left-root-right**):
   - ➢Traverse left subtree
   - ➢Visit the root and print the data.
   - ➢Traverse the right subtree
2. Performed recursively



Inorder Traversal

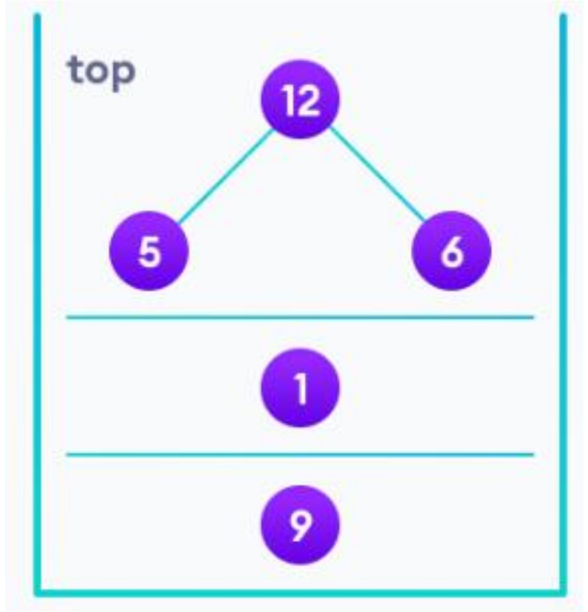
1. We traverse the left subtree first.
2. We also need to remember to visit the root node and the right subtree when this tree is done.
3. Use a stack so that we remember what needs to be done.

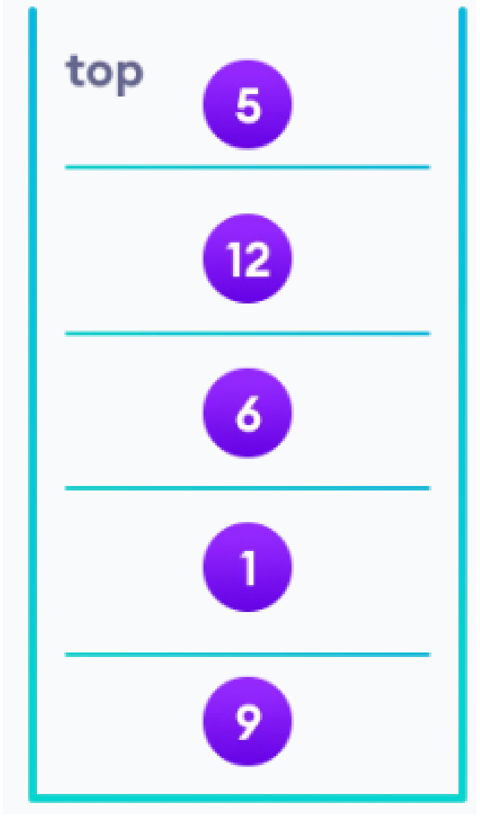




Inorder Traversal

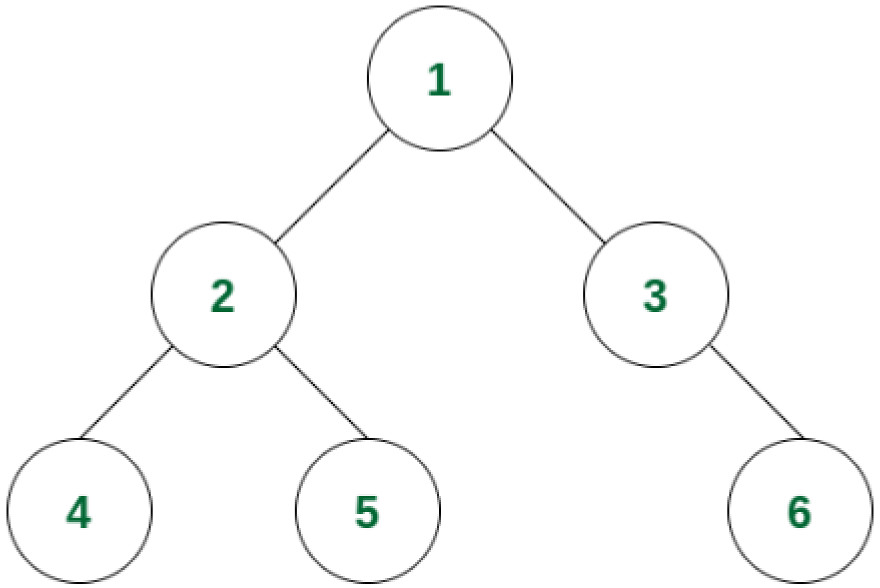
1. Now we traverse to the subtree pointed on the TOP of the stack.
2. Again, we follow the same rule of inorder:
   - Left subtree → root → right subtree
3. After traversing the left subtree, we are left with this stack.
4. Since there are no subtrees left to traverse, we can pop the elements from the stack to get the inorder traversal of the tree.
5. 5 – 12 – 6 – 1 – 9





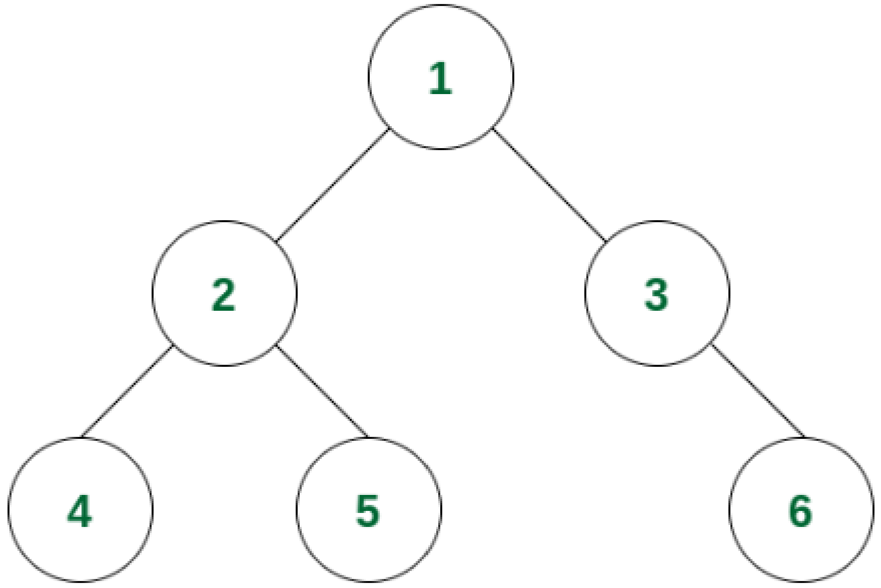
Preorder Traversal

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.



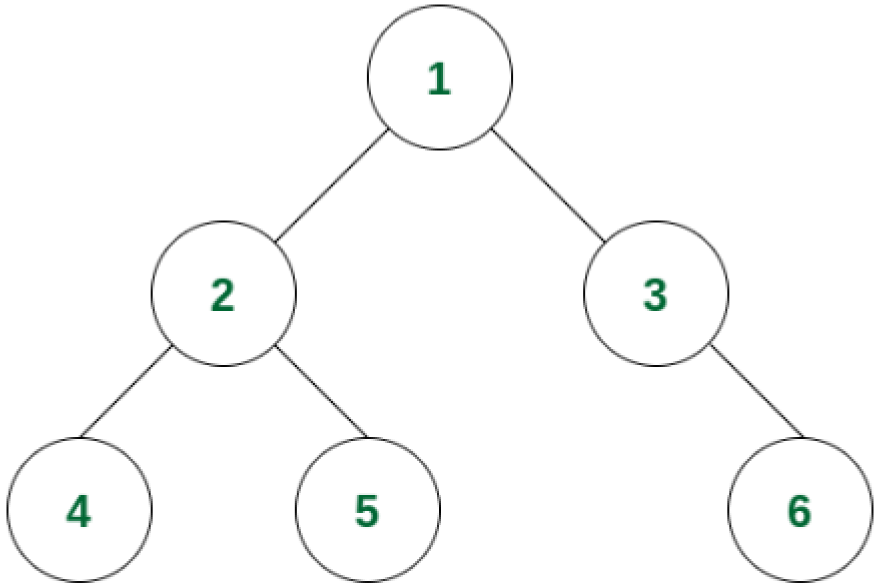
Preorder Traversal

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.
4. 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 traversal ends.
5. 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
2. 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.
3. 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.
3. Step 5: All the subtrees of node 3 are traversed. So now node 3 is visited.
4. 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.
5. So the order of traversal of nodes is
   4 -> 5 -> 2 -> 6 -> 3 -> 1.