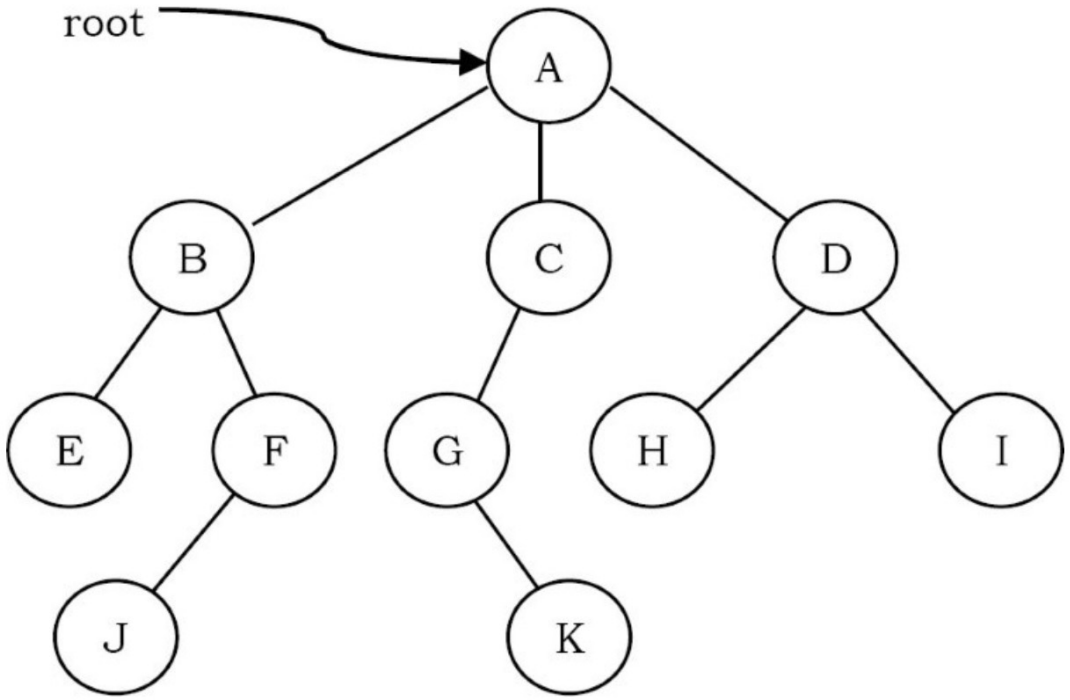
Trees

Trees

- A tree is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to many nodes.
- A tree is an example of **nonlinear data structures**.
- A tree structure is a way of representing the hierarchical nature of a structure in a graphical form.

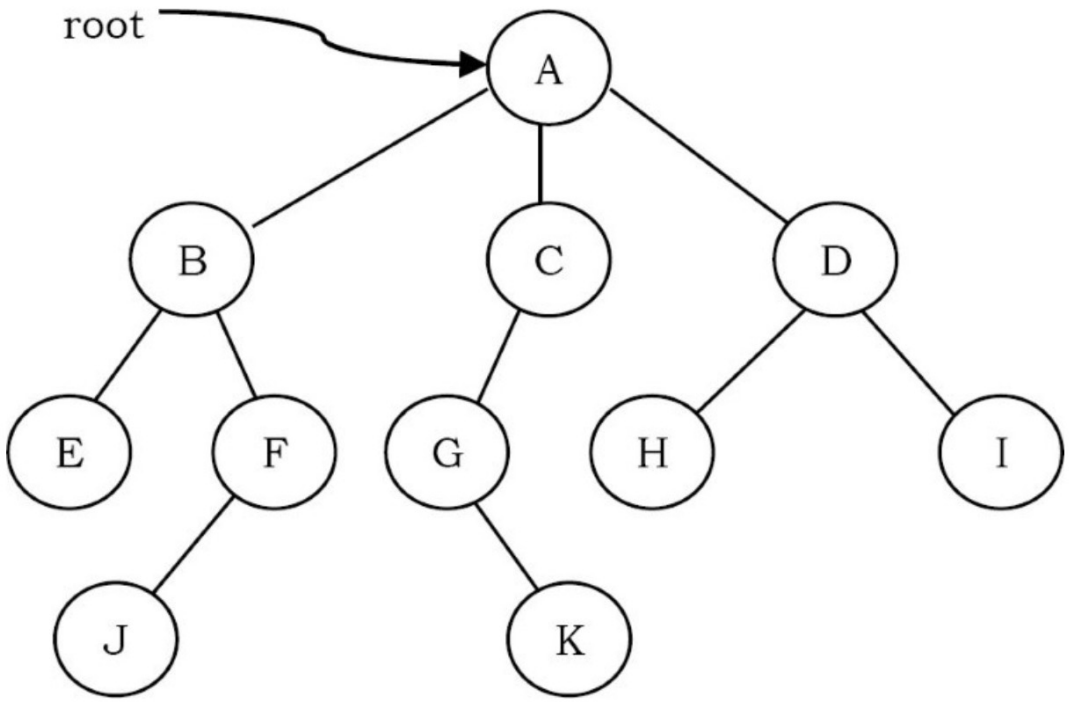
Trees-related Terms

- The **root** of a tree is the node with no parents. There can be at most one root node in a tree (node A in the example).
- An **edge** refers to the link from parent to child (all links in the figure).
- A node with no children is called a **leaf** node (E, J, K, H and I).



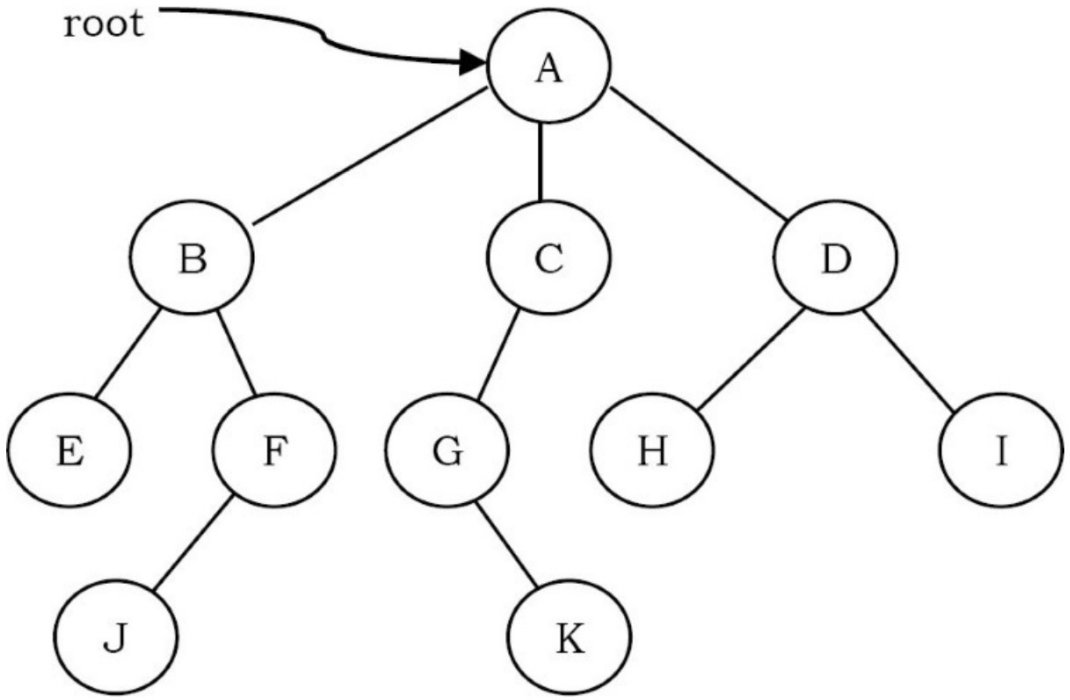
Trees-related Terms

- Children of the same parent are called siblings (B, C, D are siblings as children of A, and E, F are siblings as children of B).
- A node **p** is an **ancestor** of a node **q** if there exists a path from root to **q** and **p** appears on the path. The node **q** is called a **descendant** of **p**. For example, A, C and G are the ancestors of K.



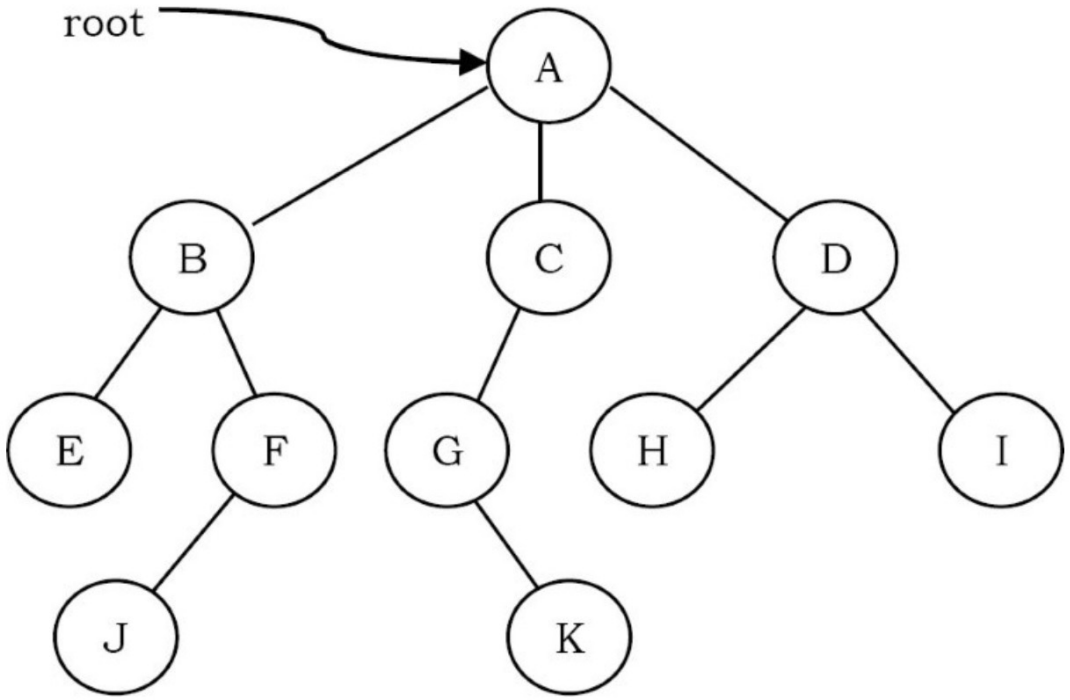
Trees-related Terms

- The **depth** of a node is the length of the path (or no. of edges) from the root to the node itself (depth of G is 2, A – C – G).
- The set of all nodes at a given depth is called the **level** of the tree (B, C and D are the same level). The root node is at level zero.



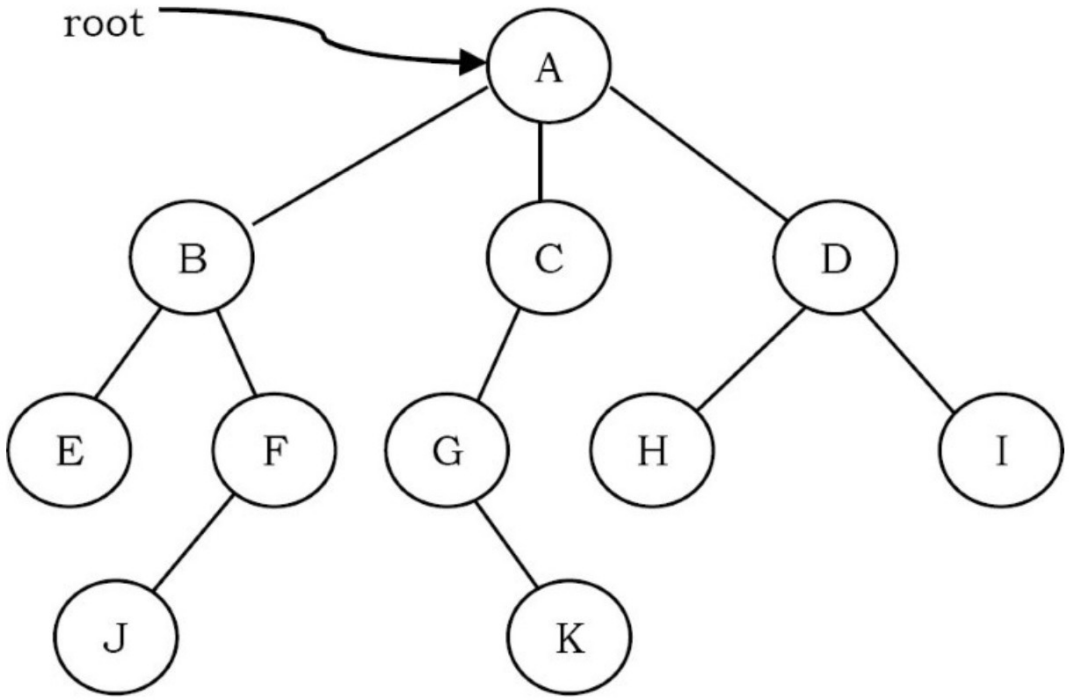
Trees-related Terms

- The **height** of a node is the length of the path from that node to the deepest node (in its descendants). The height of a tree is the length of the path from the root to the deepest node in the tree.
- A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the height of B is 2 (B – F – J).



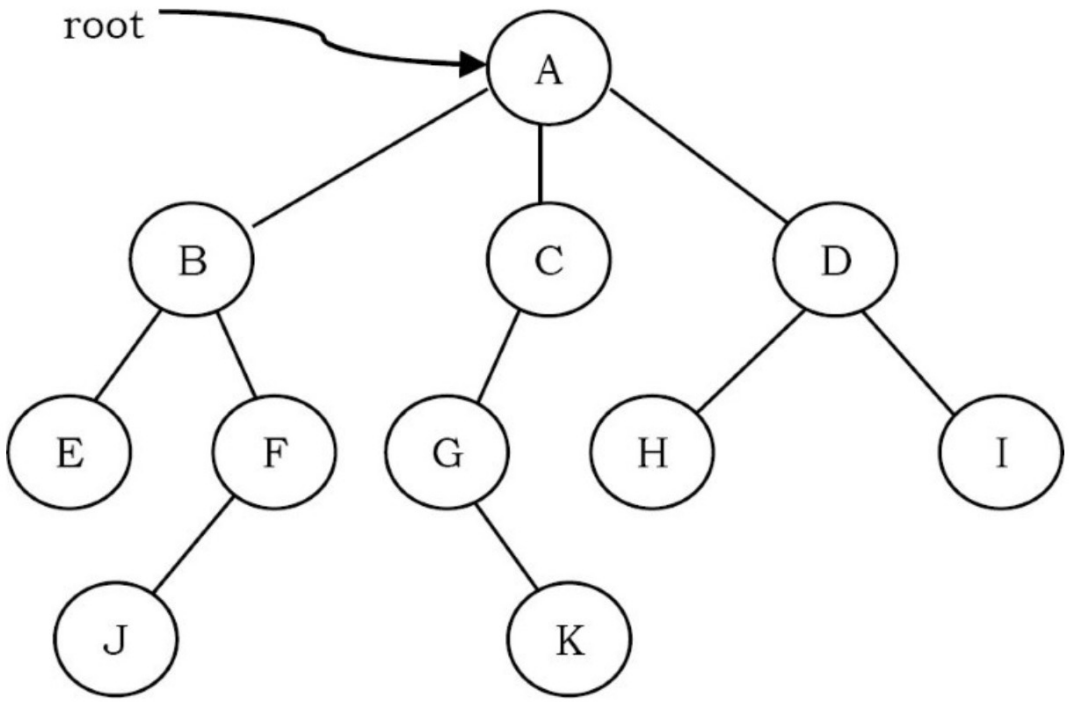
Trees-related Terms

- The **height** of the tree (itself) is the maximum height among all the nodes in the tree (i.e., the height of the root) and the **depth** of the tree (itself) is the maximum depth among all the nodes in the tree (i.e., path from root to the deepest leaf node).
- For a given tree, depth and height return the same value. But for individual nodes, we may get different results.



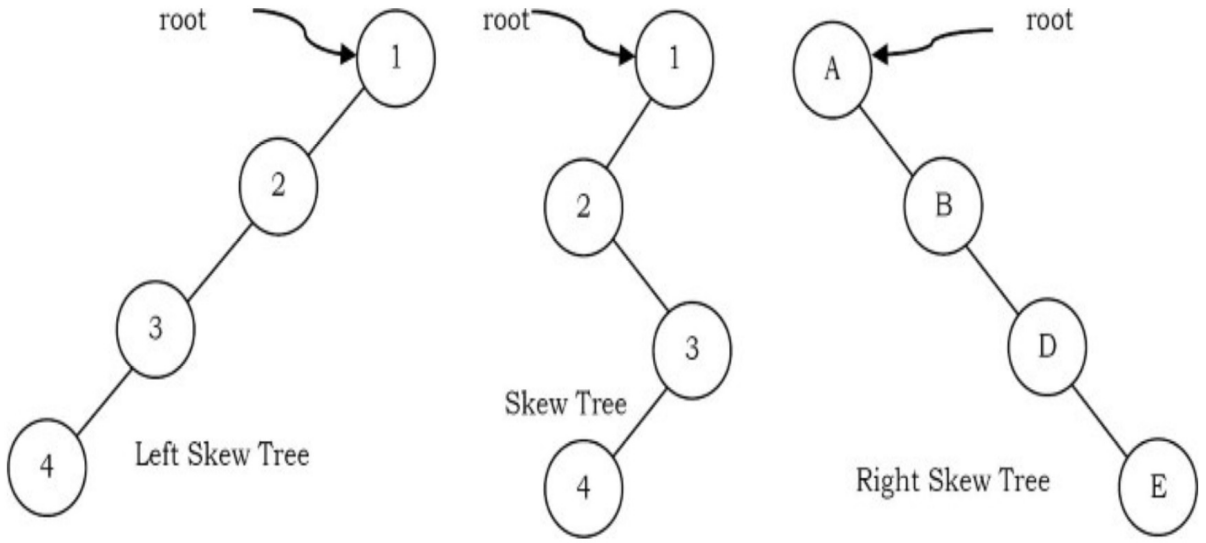
Trees-related Terms

- The **size** of a node is the number of descendants it has including itself (the size of the subtree C is 3).
- Essentially, it represents the total count of nodes in the subtree rooted at that node.



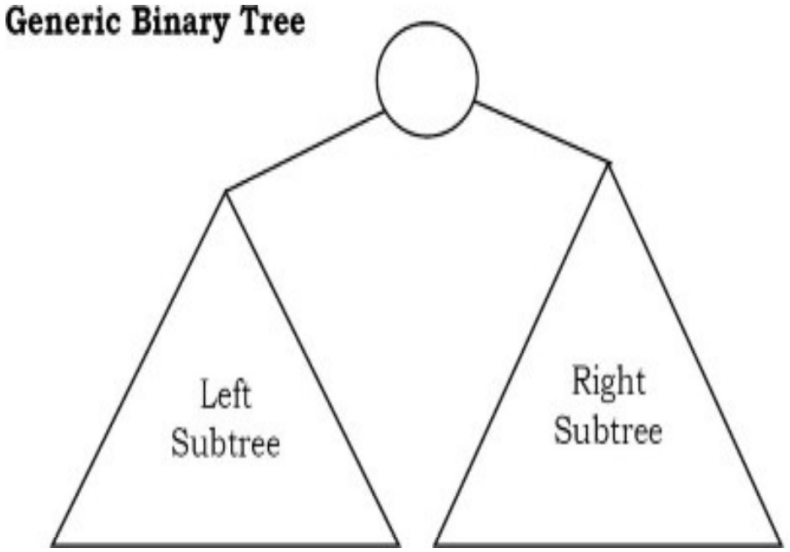
Trees-related Terms

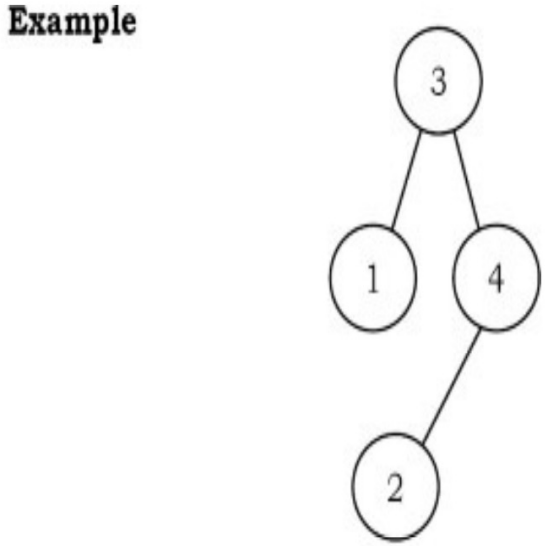
- If every node in a tree has only one child (except leaf nodes) then we call such trees **skew trees**.
- If every node has only (one) left child then we call them **left skew trees**.
- Similarly, if every node has only (one) right child then we call them **right skew trees**.



Binary Tree

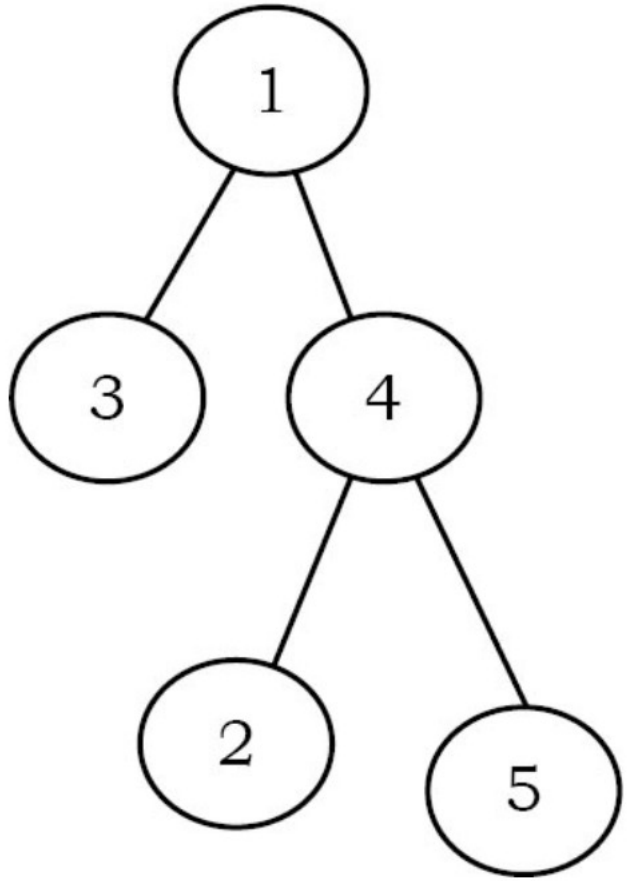
- A tree is called a **binary tree** if each node has zero child, one child, or two children (not more than two). An empty tree is also a valid binary tree.
- We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.





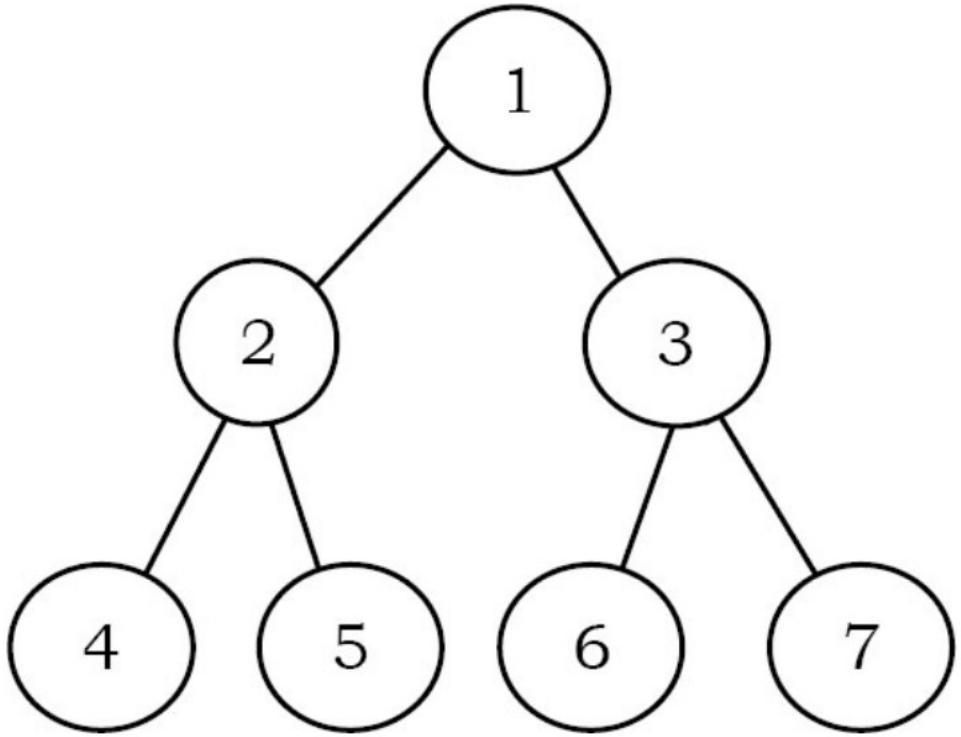
Binary Tree Types

- **Strict Binary Tree**: A binary tree is called a strict binary tree if each node has exactly two children or no children.
- It is also known as a **Full Binary Tree** or a **Proper Binary Tree** in some books.



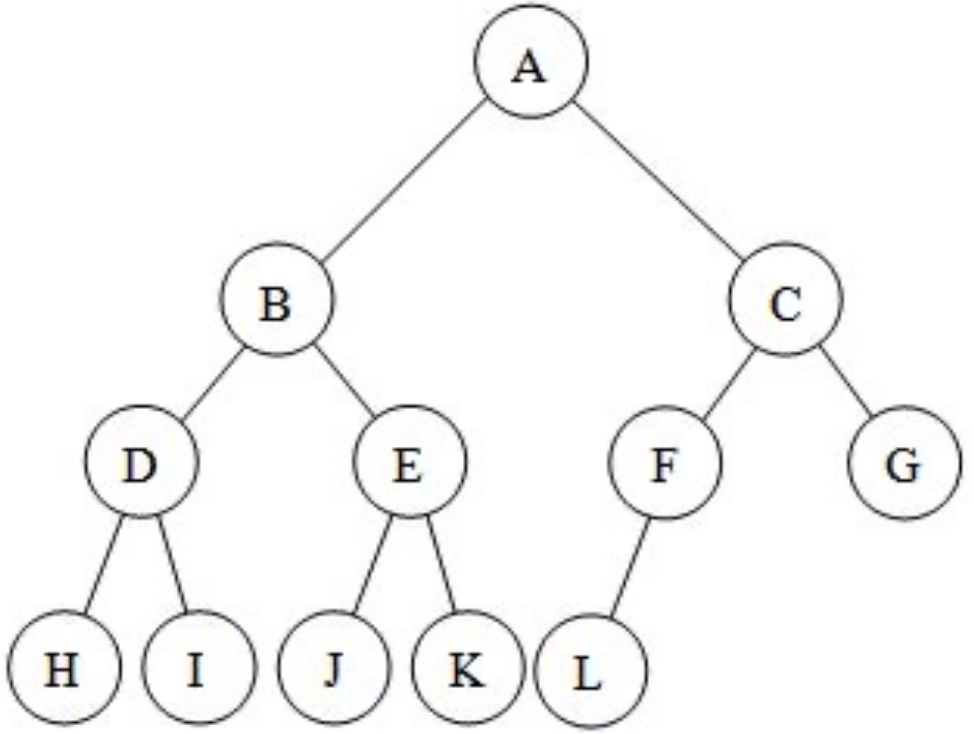
Binary Tree Types

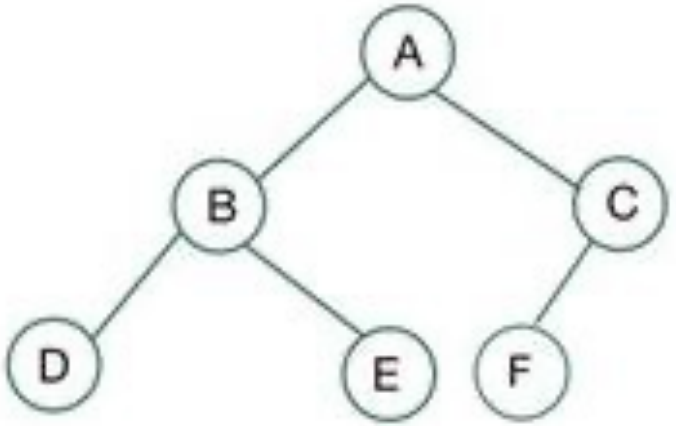
- **Perfect Binary Tree**: A binary tree is called a perfect binary tree if each node has exactly two children and all leaf nodes are at the same level (i.e., same depth from the root).

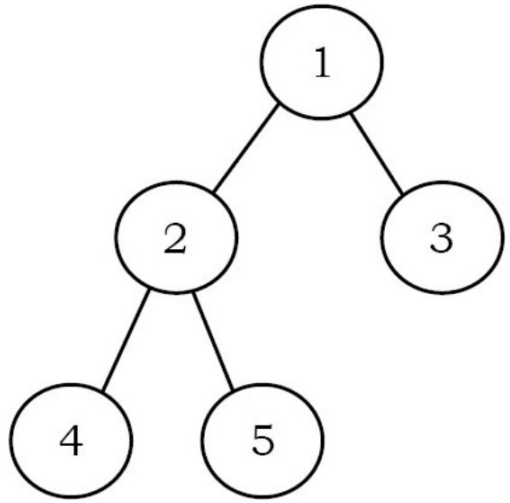


Binary Tree Types

- **Complete Binary Tree**: A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled in a left-to-right way.

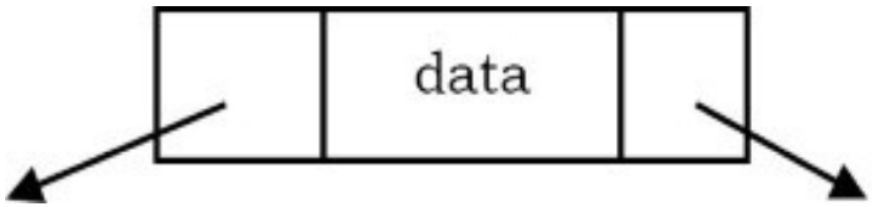






Binary Tree Structure

- Now let us define the structure of the binary tree. For simplicity, assume that the data of the nodes are integers.
- One way to represent a node (which contains data) is to have two links that point to left and right children along with data fields as shown below:



Or

Binary Tree Operations

- Basic Operations
  - Inserting an element into a tree
  - Deleting an element from a tree
  - Searching for an element
  - Traversing the tree

Binary Tree Applications

- Following are some of the applications where binary trees play an important role:
  - Expression trees are used in compilers.
  - Huffman coding trees that are used in data compression algorithms.
  - Binary Search Tree (BST), which supports search, insertion, and deletion on a collection of items in $O(\log n)$ (average)
  - Priority Queue (PQ), which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (in worst case).

Binary Tree Traversal

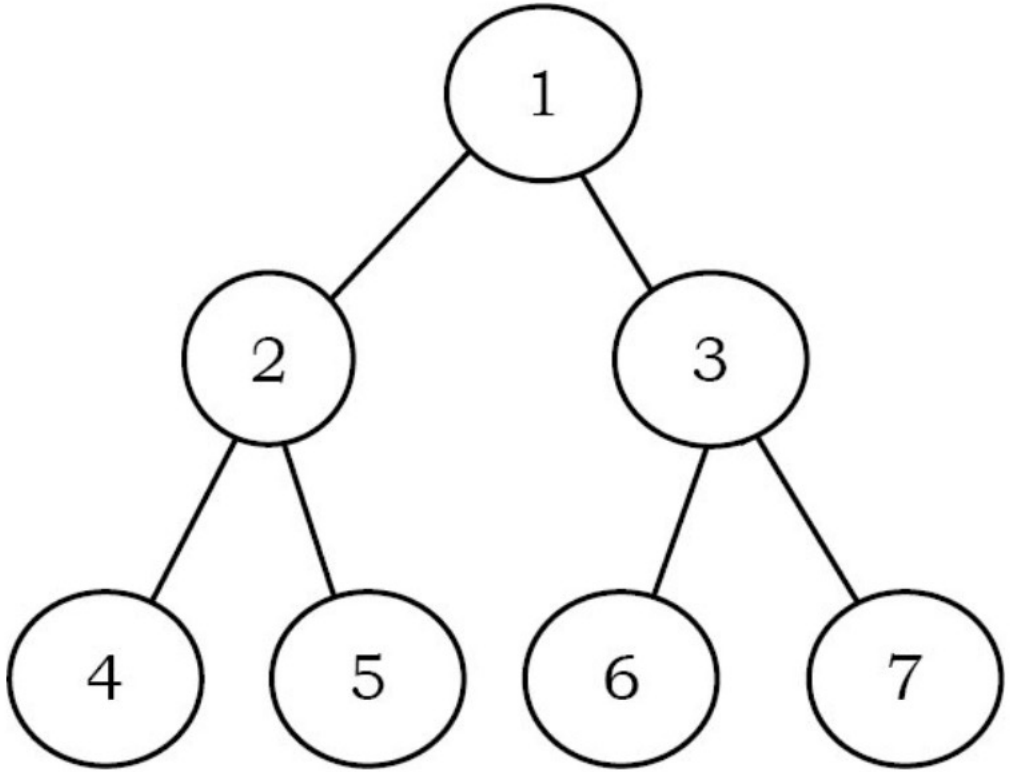
- The process of visiting all nodes of a tree is called **tree traversal**.
- Each node is processed only once but it may be visited more than once.
- As we have already seen in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order.
- But, in tree structures, there are many different ways.
- Tree traversal is like searching the tree, except that in traversal the goal is to move through the tree in a particular order.
- In addition, all nodes are processed in the traversal but searching stops when the required node is found.

Binary Tree Traversal

- There are three common ways to traverse a tree
  - Preorder Traversal
  - Inorder Traversal
  - Postorder Traversal
- There is another traversal method that does not depend on the above orders and it is:
  - Level Order Traversal: This method is inspired by Breadth First Traversal (BFS of Graph algorithms).

PreOrder Traversal

- In preorder traversal, each node is processed before (pre) either of its subtrees.
- It is defined as:
  - Visit the root (of the current tree/subtree).
  - Traverse the left subtree in Preorder.
  - Traverse the right subtree in Preorder.
- For the given tree: 1 2 4 5 3 6 7

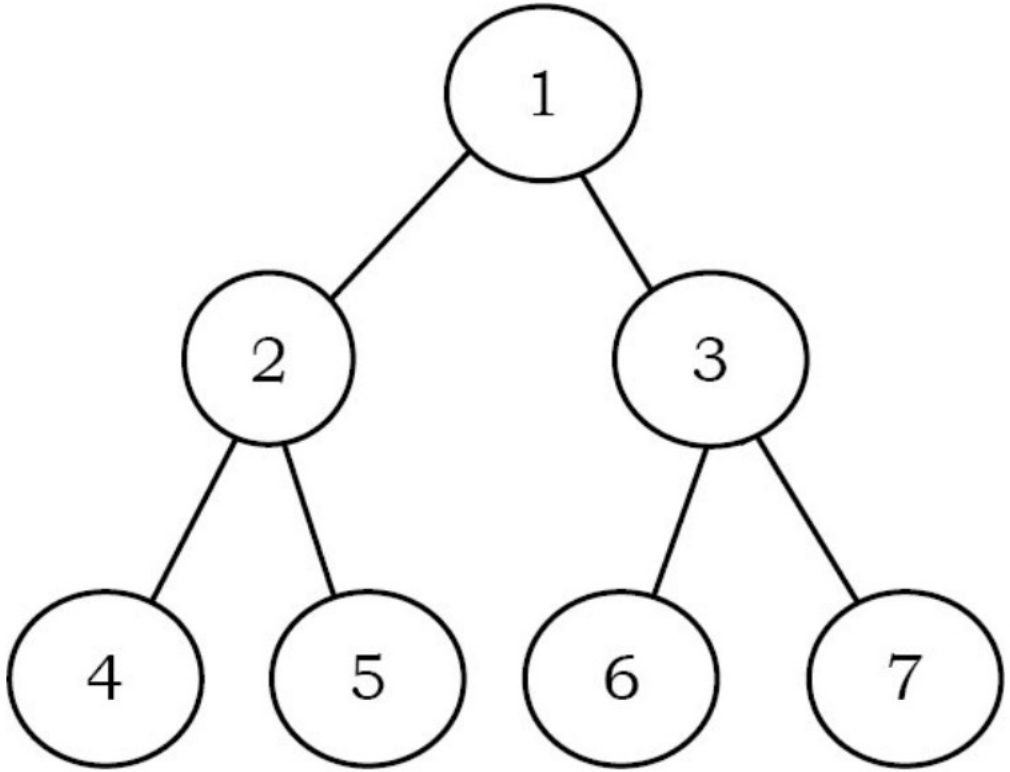


PreOrder Traversal

```
PreOrderTraversal(tree)
if tree = nil:
    return
Print(tree.key)
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

InOrder Traversal

- In Inorder Traversal the root is visited between the subtrees. Inorder traversal is defined as:
  - Traverse the left subtree in Inorder.
  - Visit the root (of the current tree/subtree).
  - Traverse the right subtree in Inorder.
- For the given tree: 4 2 5 16 3 7

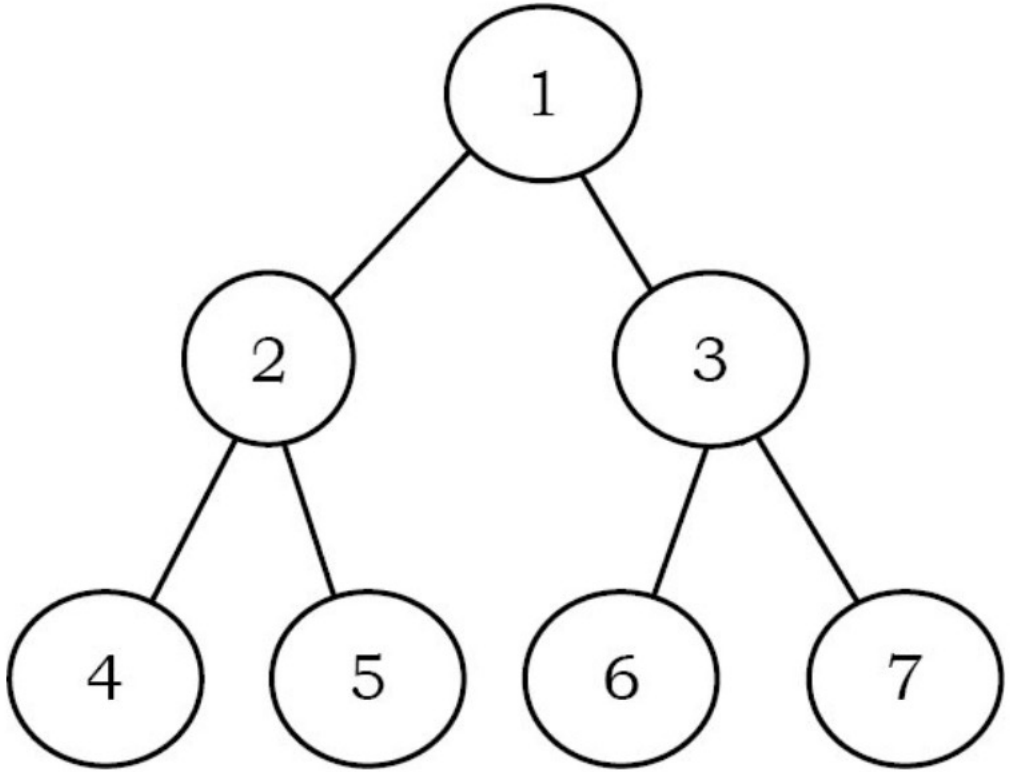


InOrder Traversal

```
InOrderTraversal(tree)
if tree = nil:
    return
InOrderTraversal(tree.left)
Print(tree.key)
InOrderTraversal(tree.right)
```

PostOrder Traversal

- In postorder traversal, the root is visited after both subtrees. Postorder traversal is defined as:
  - Traverse the left subtree in Postorder.
  - Traverse the right subtree in Postorder.
  - Visit the root (of the current tree/subtree).
- For the given tree: 4 5 2 6 7 3 1



PostOrder Traversal

```
PostOrderTraversal(tree)

if tree = nil:
    return
PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
Print(tree.key)
```