Trees

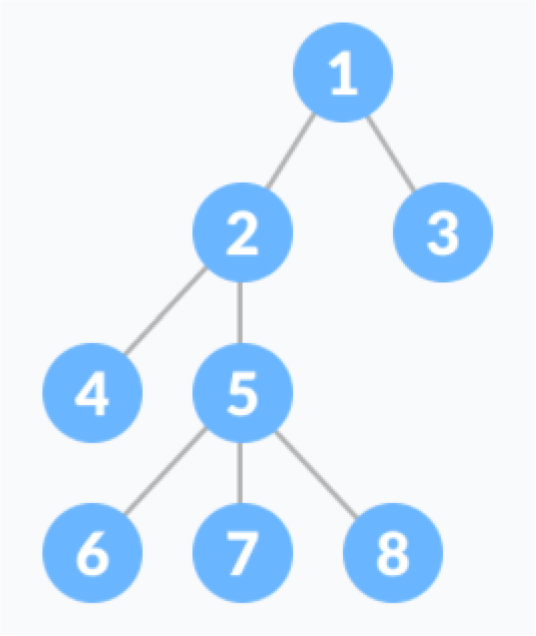
Anoop S Babu
Faculty Associate
Dept. of Computer Science & Engineering
bsanoop@am.amrita.edu

Tree Data Structure

- A tree is a **nonlinear hierarchical** data structure that consists of **nodes connected by edges**.

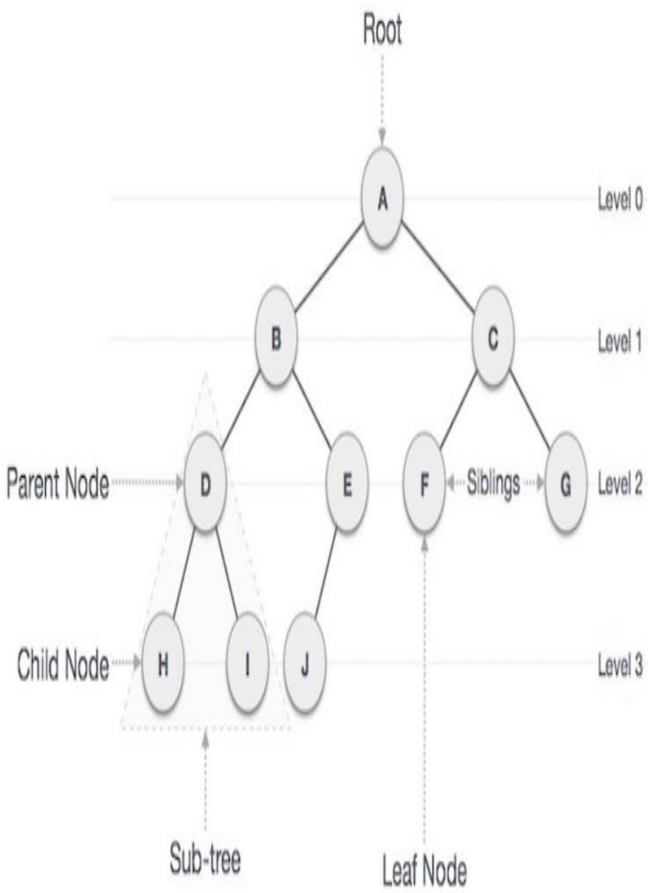
Formal Definition

- A tree T as a **set of nodes** storing elements such that the **nodes have a parent-child relationship** that satisfies the following properties:
  - If T is nonempty, it has a **special node**, called the **root** of T, that **has no parent.**
  - Each node v of T different from the root has a **unique parent node w**; every node with parent w is a child of w.



Tree Terminologies

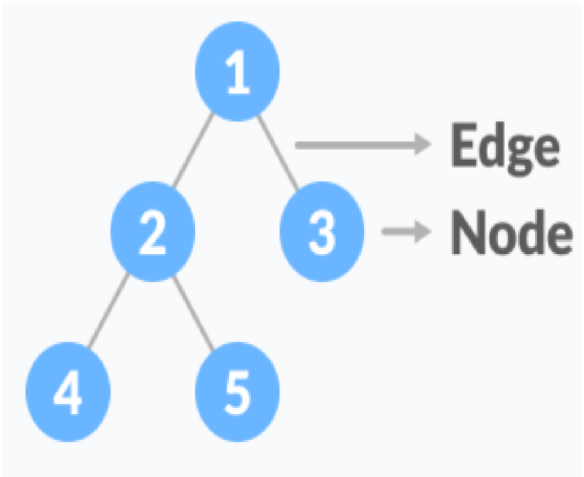
- **Root**: The node at the top of the tree is called root.
- **Parent Node**: If the **node contains any sub-node**, then that node is said to be the parent of that sub-node. The sub-node is called the **child node**.
- **Sibling**: The nodes that **have the same parent** are known as siblings.
- **Leaf Node**: The node of the tree, which **doesn't have any child node**, is called a leaf node (**external node**).
- **Internal node**: A node has **atleast one child node** known as an internal node
- **Ancestor node**: An ancestor of a node is any predecessor node on a path from the root to that node.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node.
- **Subtree:** Subtree represents the node and its descendants.

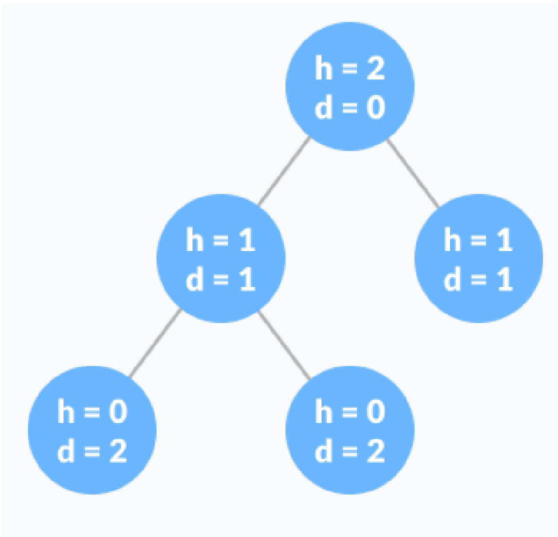


Tree Terminologies

- **Node:** A node is an entity that contains a key or value and pointers to its child nodes.
- **Edge** It is the link between any two nodes.

- **Height of a Node:** It is the number of edges from the node to the deepest leaf.
- **Depth of a Node:** It is the number of edges from the root to the node.
- **Height of a Tree**: It is the height of the root node or the depth of the deepest node.
- **Degree of a Node:** It is the total number of branches of that node.





Height and depth of each node in a tree

Tree Traversal

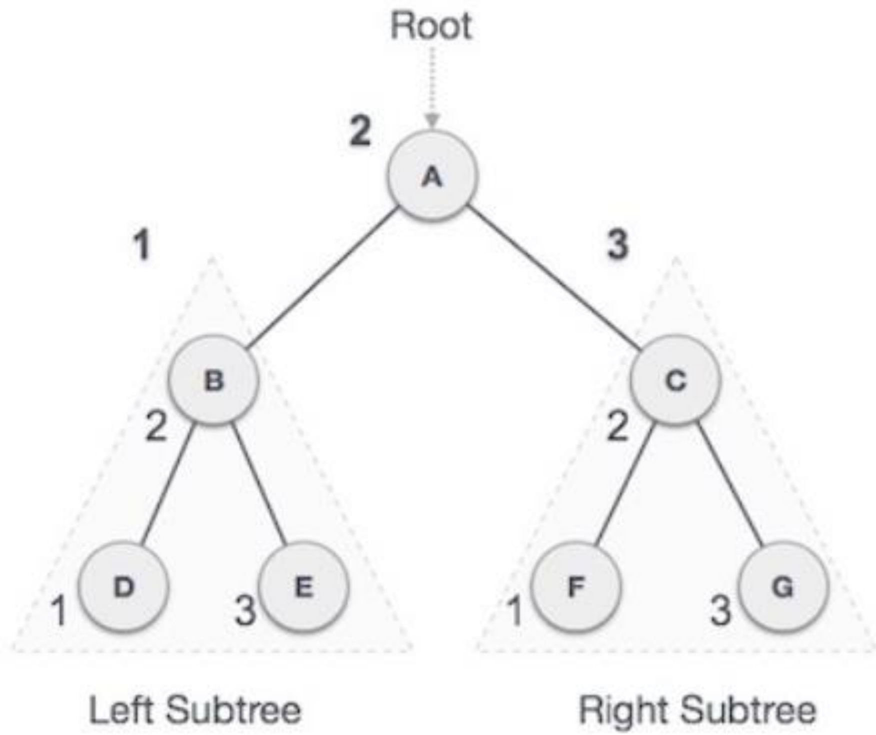
- Traversing a tree means **visiting every node** in the tree.
- There are **three ways** which we use to traverse a tree.
  - In-order Traversal (LNR)
  - Pre-order Traversal (NLR)
  - Post-order Traversal (LRN)

In-order Traversal

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.

Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```



The output of inorder traversal

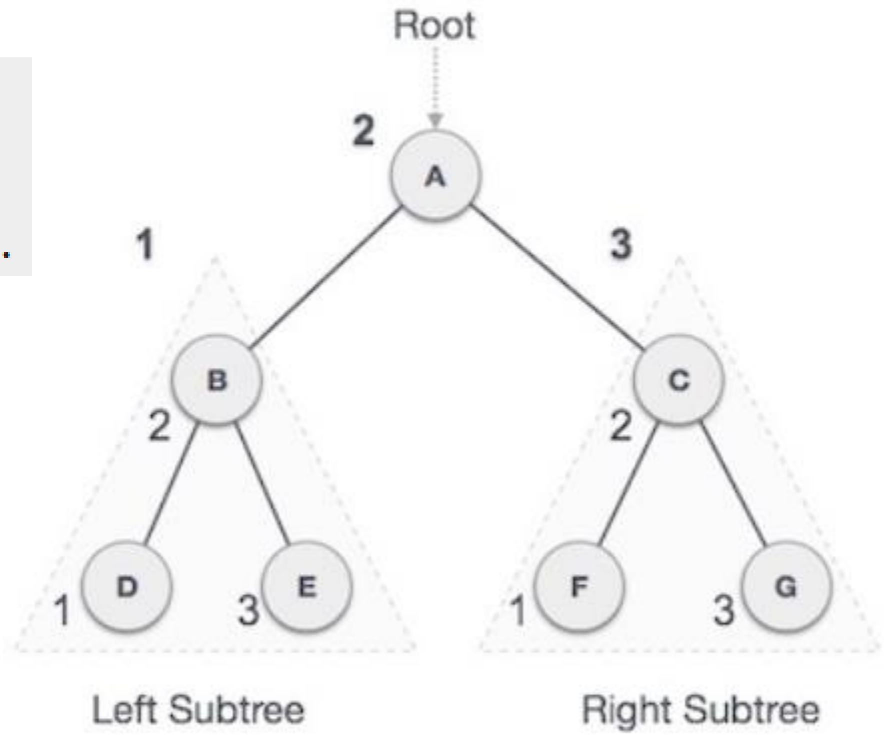
$$D \to B \to E \to A \to F \to C \to G$$

Pre-order Traversal

- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Algorithm

```
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```



The output of preorder traversal

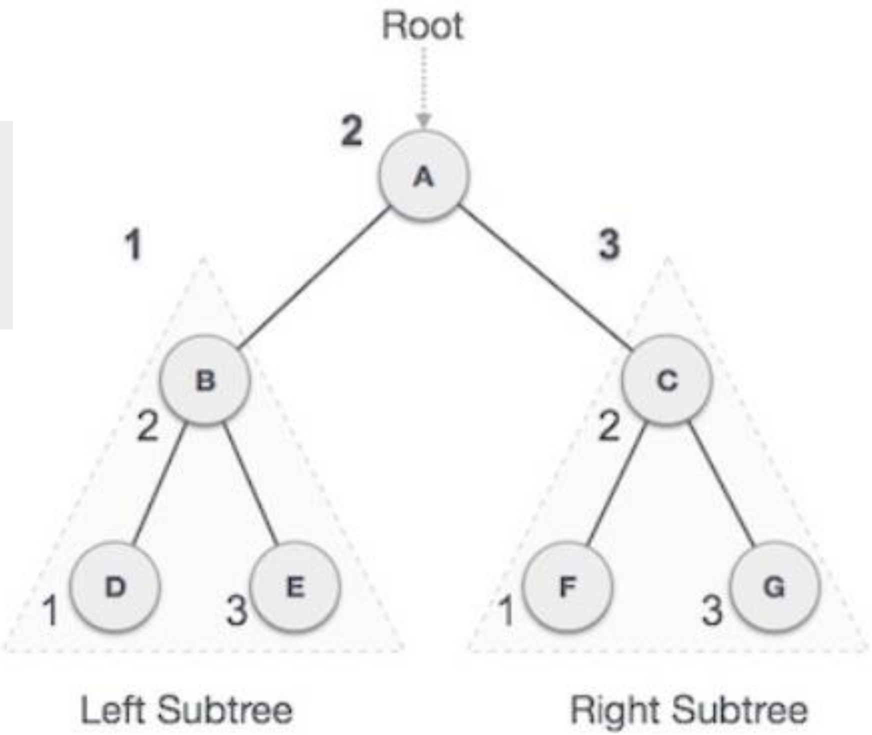
$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Post-order Traversal

- In this traversal method, the root node is visited last. First we traverse the left subtree, then the right subtree and finally the root node.

Algorithm

```
Until all nodes are traversed –
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```



The output of postorder traversal

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Tree Traversal: Python Implementation

```
# Tree Traversal
class Node:
    def __init__(self, item):
        self.left = None
        self.right = None
        self.val = item
def inorder(root):
    if root:
        # Traverse left
        inorder(root.left)
        # Traverse root
        print(str(root.val) + "->", end='')
        # Traverse right
        inorder(root.right)
def postorder(root):
    if root:
        # Traverse left
        postorder(root.left)
        # Traverse right
        postorder(root.right)
        # Traverse root
        print(str(root.val) + "->", end='')
```

Tree Traversal: Python Implementation

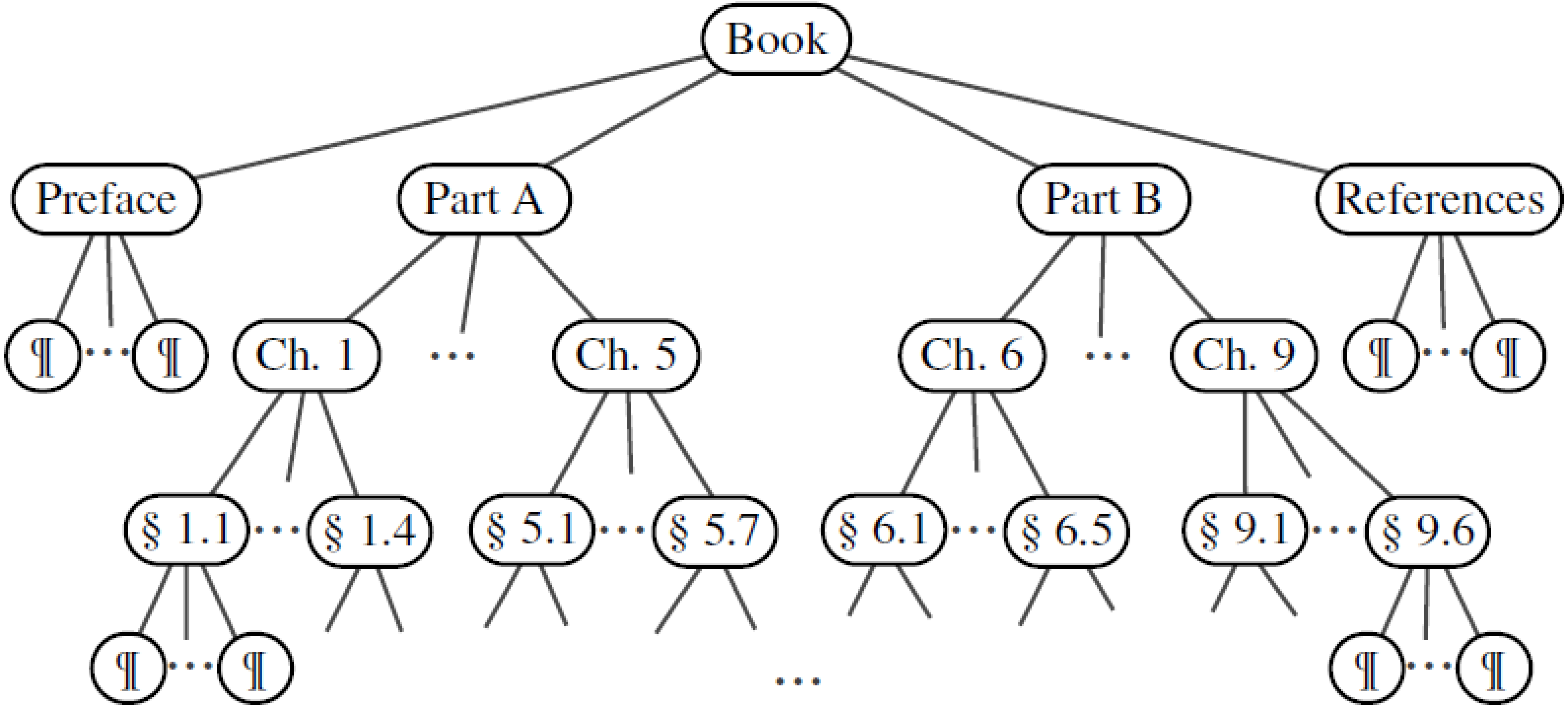
```python
def preorder(root):
    if root:
        # Traverse root
        print(str(root.val) + "->", end='')
        # Traverse left
        preorder(root.left)
        # Traverse right
        preorder(root.right)

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print("Inorder traversal ")
inorder(root)
print("\nPreorder traversal ")
preorder(root)
print("\nPostorder traversal ")
postorder(root)
```

Output

```
Inorder traversal
4->2->5->1->3->
Preorder traversal
1->2->4->5->3->
Postorder traversal
4->5->2->3->1->
```

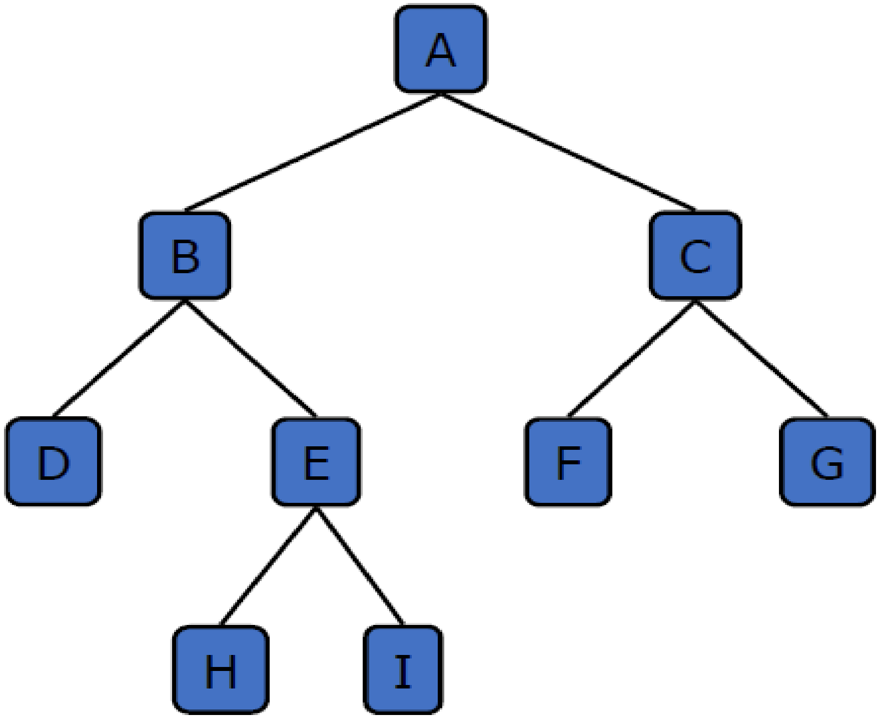
Ordered Tree

- A tree is ***ordered*** if there is a **meaningful linear order** among the children of each node.



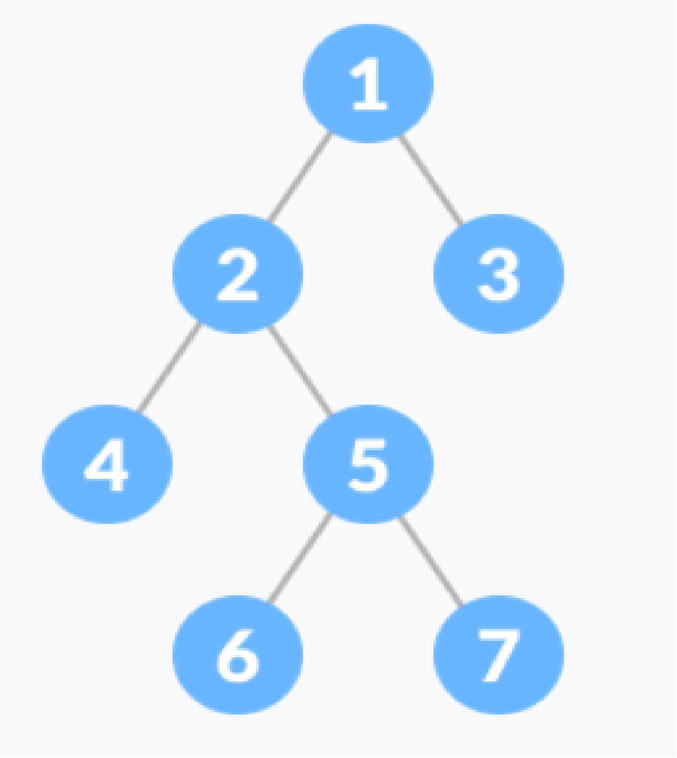
Binary Tree

- A ***binary tree*** is an ordered tree with the following properties:
  - Every node has at **most two children**.
  - Each child node is labeled as being either a ***left child*** or a ***right child***.
  - A left child precedes a right child in the order of children of a node.



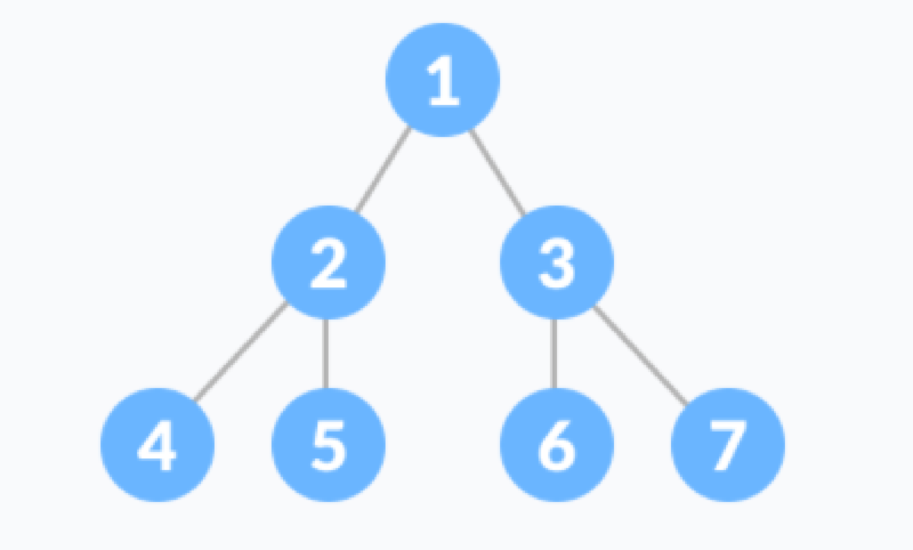
Types of Binary Tree

- **Full Binary Tree**: A full Binary tree is a special type of binary tree in which **every parent node/internal node has either two or no children.**
- It is also known as a **proper binary tree.**



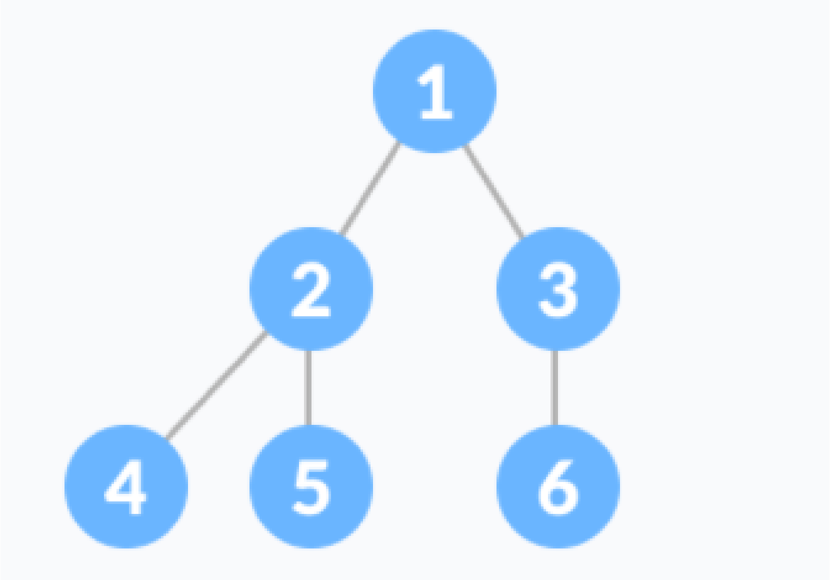
Types of Binary Tree

- **Perfect Binary Tree:** A perfect binary tree is a type of binary tree in which **every internal node has exactly two child nodes** and **all the leaf nodes are at the same level.**

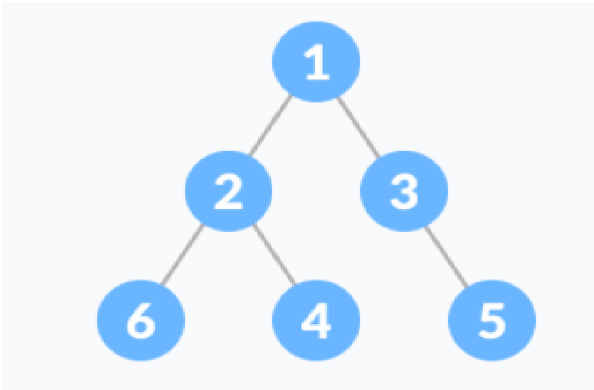


Types of Binary Tree

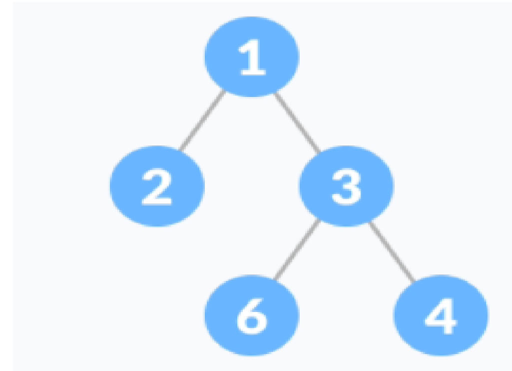
- **Complete Binary Tree:** It is just like a full binary tree, but with two major differences
  - All the leaf elements must lean towards the left.
  - The last leaf element might not have a right sibling. (i.e. a complete binary tree doesn't have to be a full binary tree.)



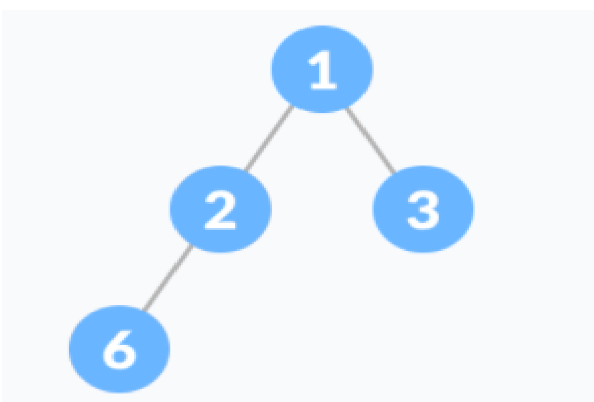
Full Binary Tree vs Complete Binary Tree



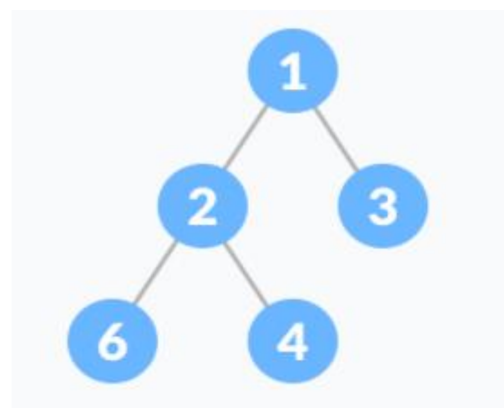
✗ Full Binary Tree
✗ Complete Binary Tree



✓ Full Binary Tree
✗ Complete Binary Tree



✗ Full Binary Tree
✓ Complete Binary Tree

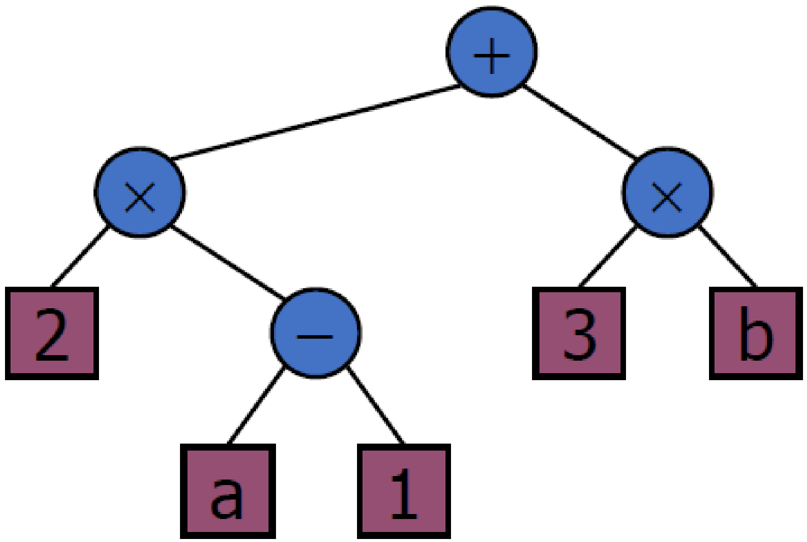


✓ Full Binary Tree
✓ Complete Binary Tree

Applications of Binary Tree

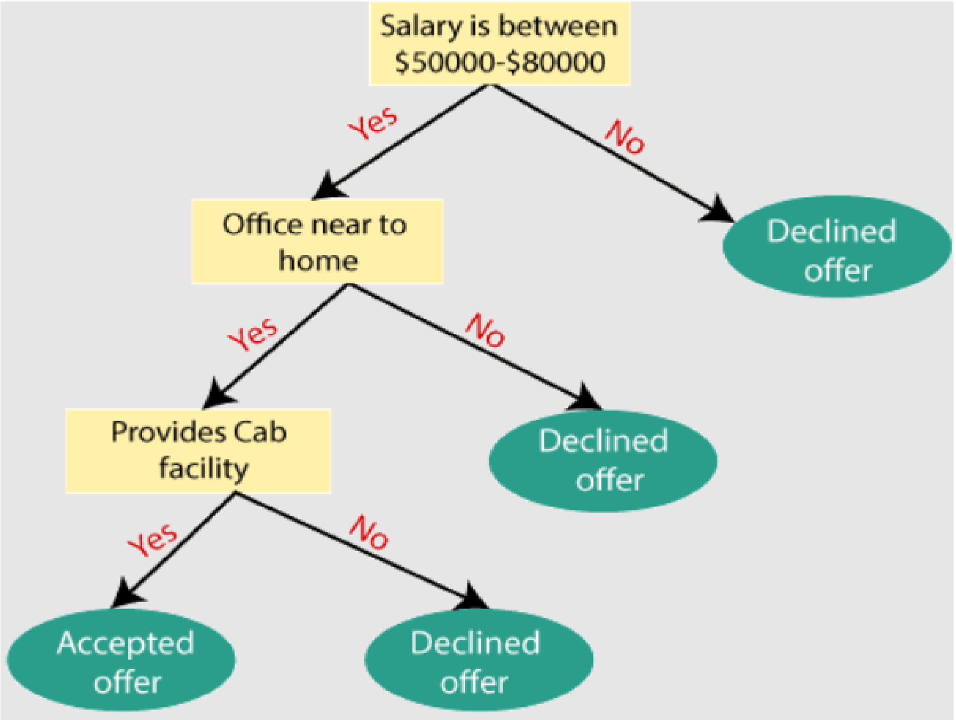
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: Arithmetic expression tree for the expression (2 x (a - 1) + (3 x b))



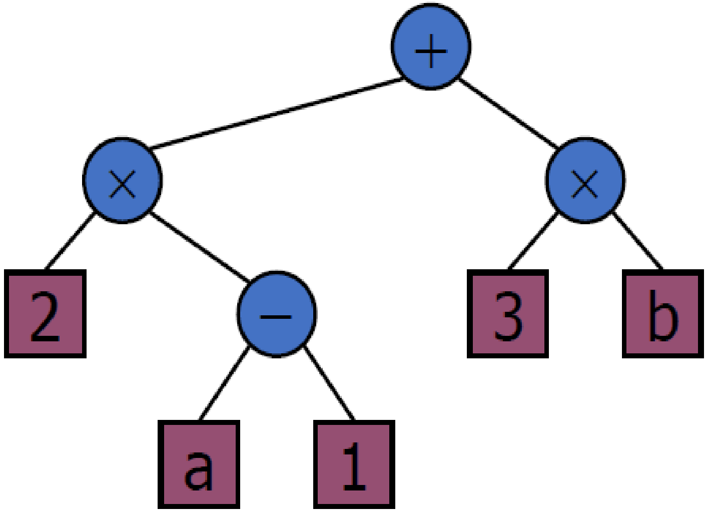
Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print “(“ before traversing left subtree
  - print “)“ after traversing right subtree



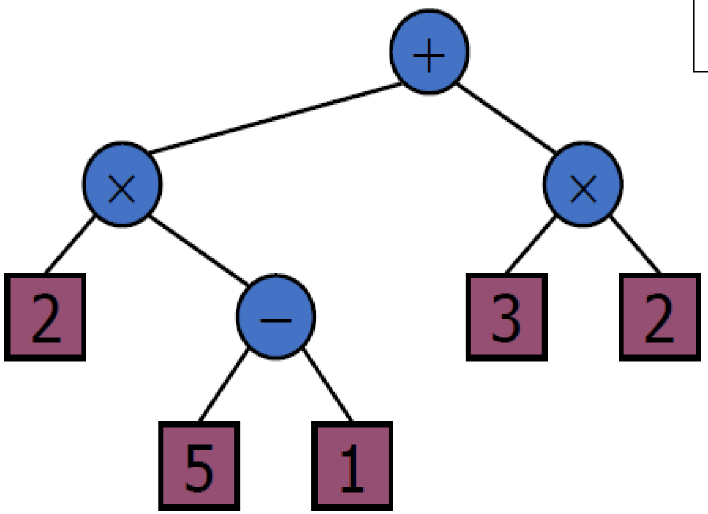
Algorithm *printExpression*(*v*)
    if *hasLeft* (*v*)
        print(“(”)
        inOrder (*left*(*v*))
    print(*v.element* ())
    if *hasRight* (*v*)
        inOrder (*right*(*v*))
        print (“)”)

$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions

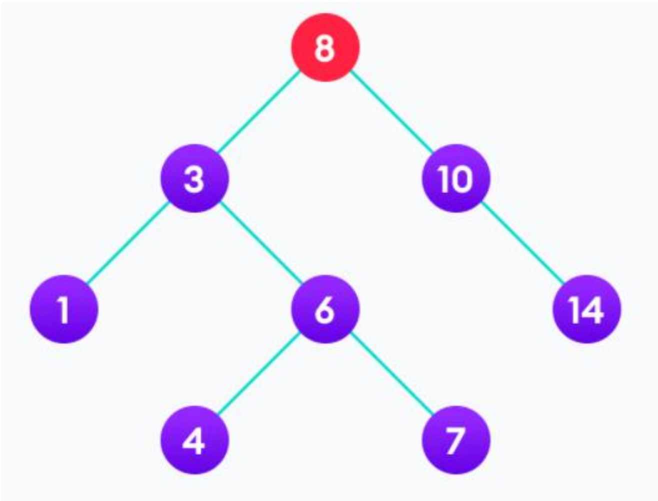
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(v)
  if *isExternal* (v)
    return *v.element* ()
  else
    $x \leftarrow$ *evalExpr*(*leftChild* (v))
    $y \leftarrow$ *evalExpr*(*rightChild* (v))
    $\Diamond \leftarrow$ operator stored at v
    return $x \Diamond y$

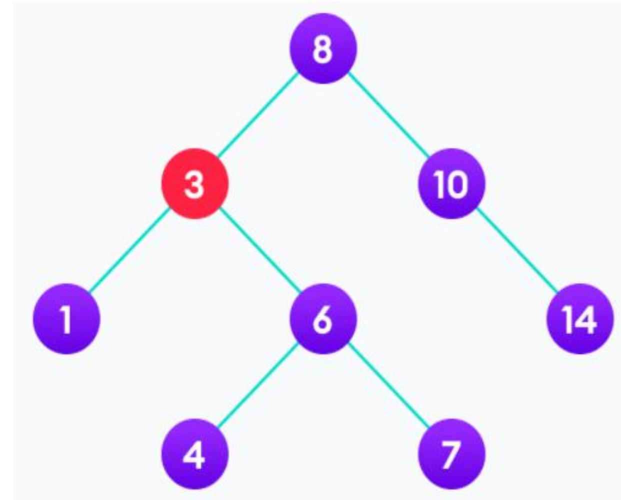


Answer = 14

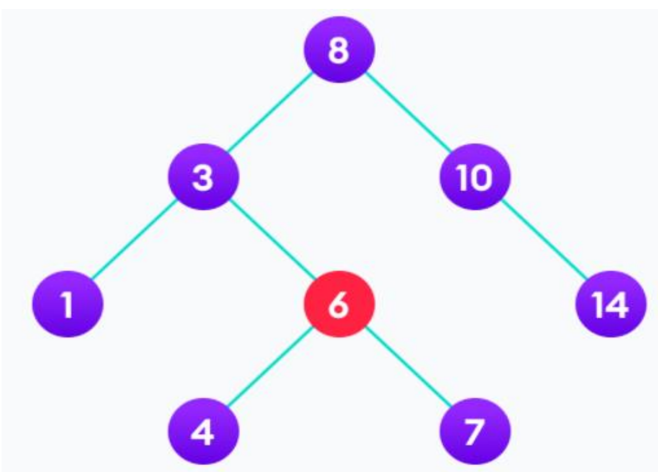
Searching: BST



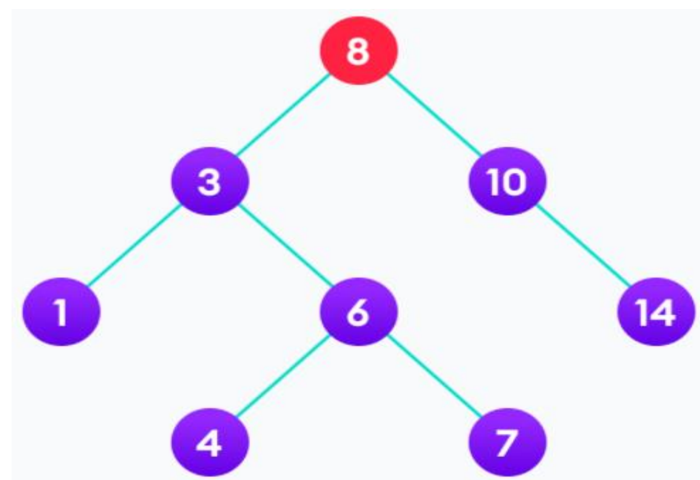
4 is not found so, traverse through the left subtree of 8



4 is not found so, traverse through the right subtree of 3



4 is not found so, traverse through the left subtree of 6



4 is found