

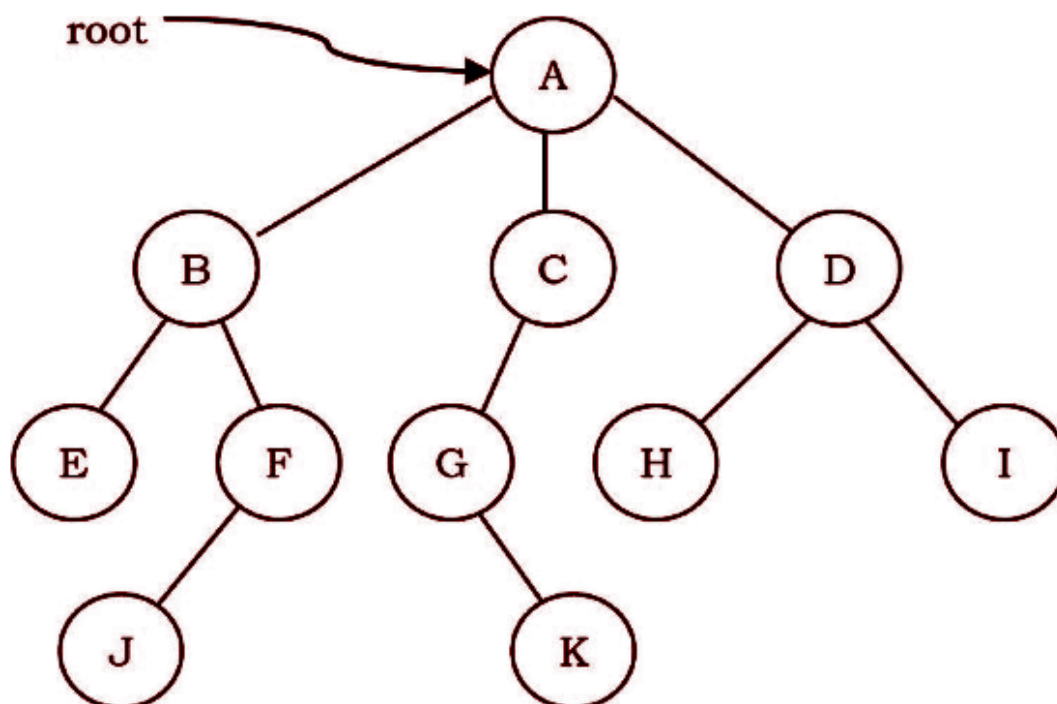
# Binary Trees- 1

---

## What is A Tree?

- 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 several nodes.
- A tree is an example of a non- linear data structure.
- A tree structure is a way of representing the hierarchical nature of a structure in a graphical form.

## Terminology Of Trees

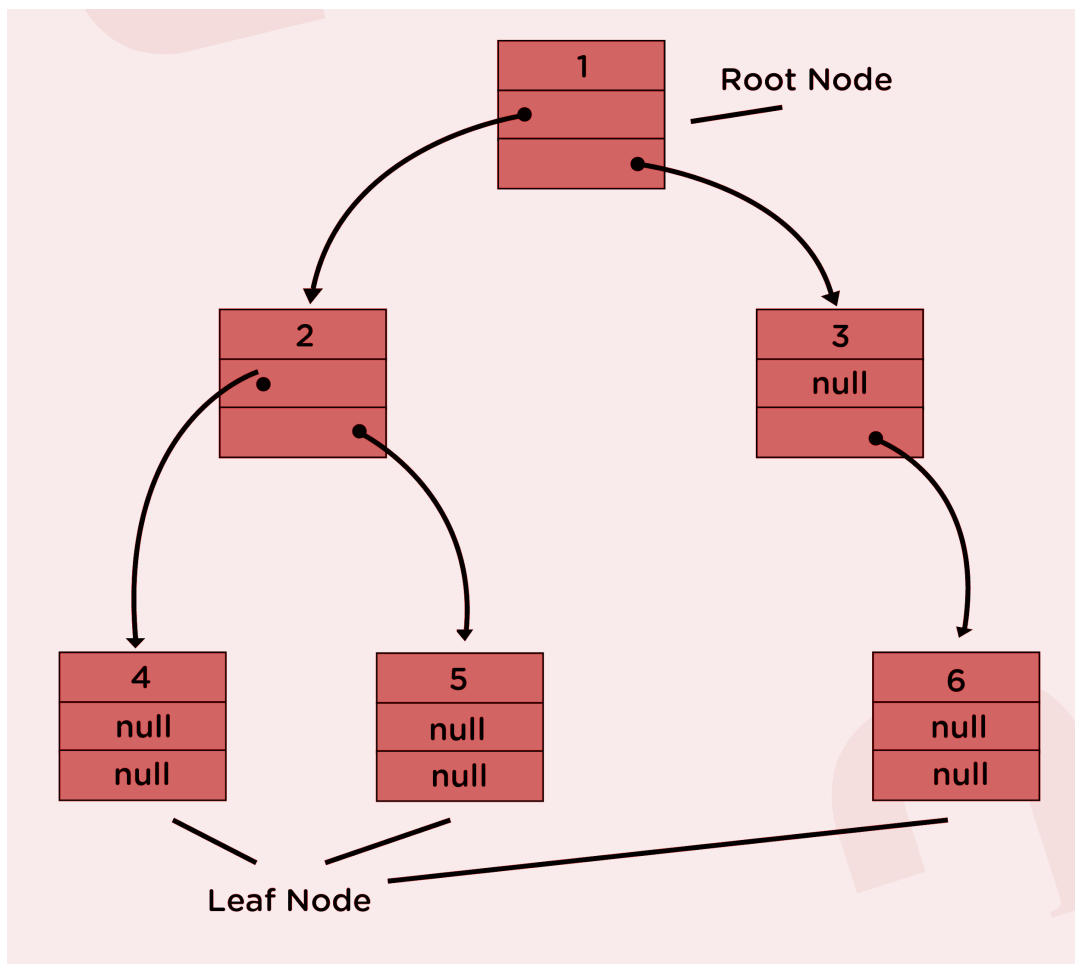


- The root of a tree is the node with no parents. There can be at most one root node in a tree (**node A in the above example**).

- An **edge** refers to the link from a parent to a child (**all links in the figure**).
- A node with no children is called a **leaf node** (E, J, K, H, and I).
- The children nodes of the same parent are called **siblings** (B, C, D are **siblings of parent A** and E, F are **siblings of parent B**).
- 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.
- The **depth** of a node is the length of the path from the root to the node (depth of G is 2, A → C → G).
- The **height** of a node is the length of the path from that node to the deepest node.
- 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.

## Binary Trees

- A generic tree with at most two child nodes for each parent node is known as a binary tree.
- A binary tree is made of nodes that constitute a **left** pointer, a **right** pointer, and a data element. The **root** pointer is the topmost node in the tree.
- The left and right pointers recursively point to smaller **subtrees** on either side.
- An empty tree is also a valid binary tree.
- *A formal definition is:* A **binary tree** is either empty (represented by a None pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.

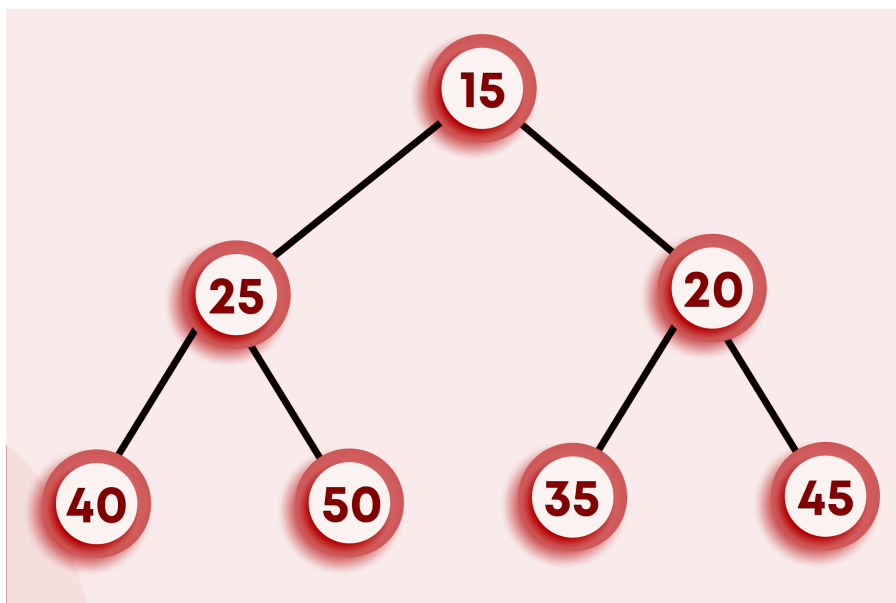
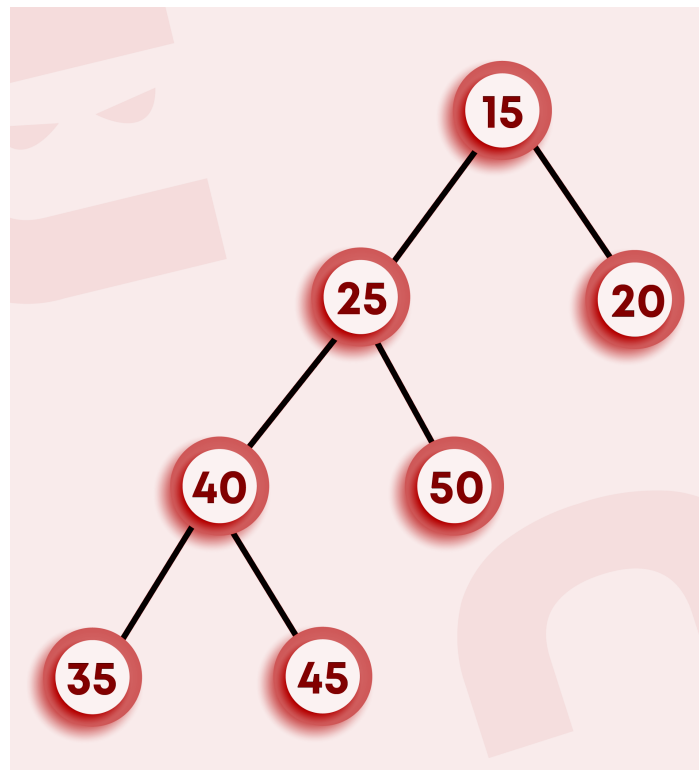


## Types of binary trees:

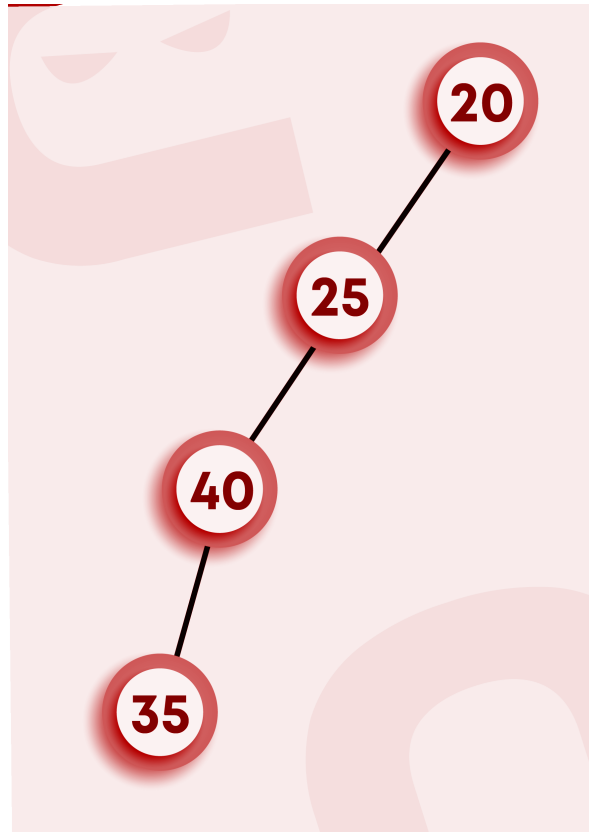
**Full binary trees:** A binary tree in which every node has 0 or 2 children is termed as a full binary tree.

**Complete binary tree:** A complete binary tree has all the levels filled except for the last level, which has all its nodes as much as to the left.

**Perfect binary tree:** A binary tree is termed perfect when all its internal nodes have two children along with the leaf nodes that are at the same level.

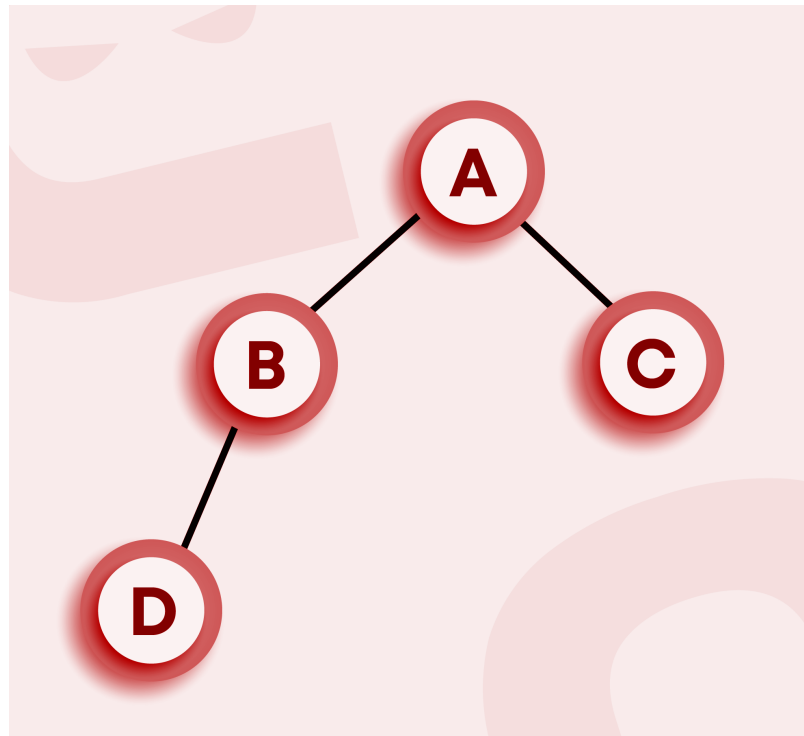


**A degenerate tree:** In a degenerate tree, each internal node has only one child.



The tree shown above is degenerate. These trees are very similar to linked-lists.

**Balanced binary tree:** A binary tree in which the difference between the depth of the two subtrees of every node is at most one is called a balanced binary tree.



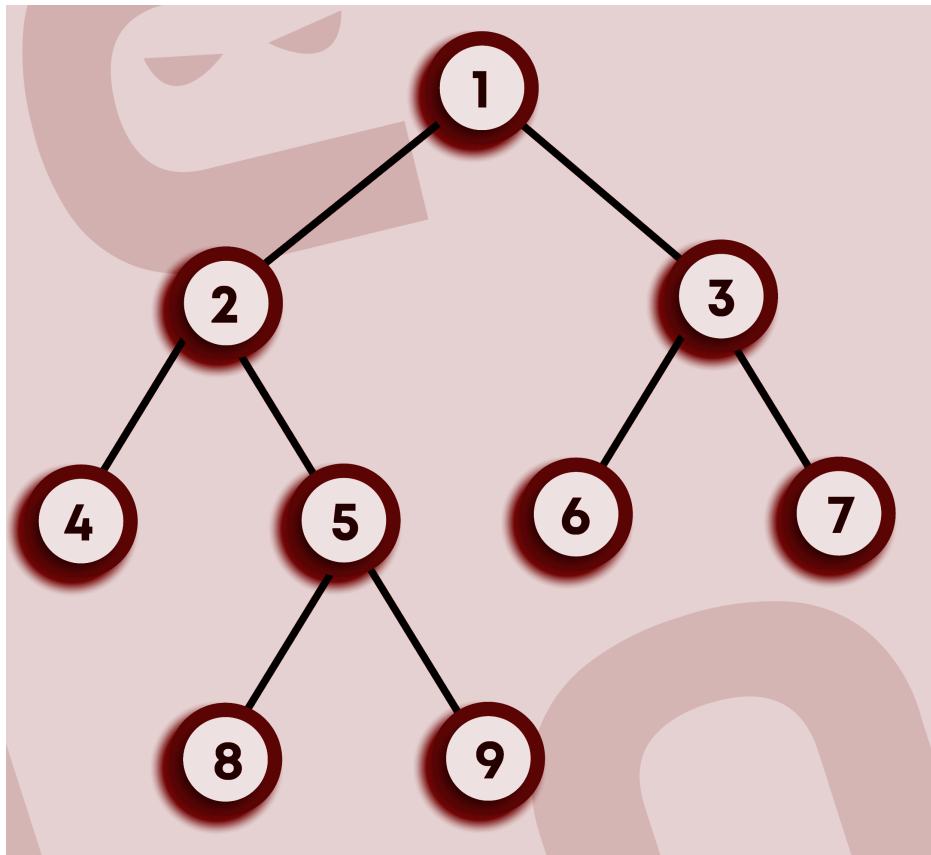
## Binary tree representation:

Binary trees can be represented in two ways:

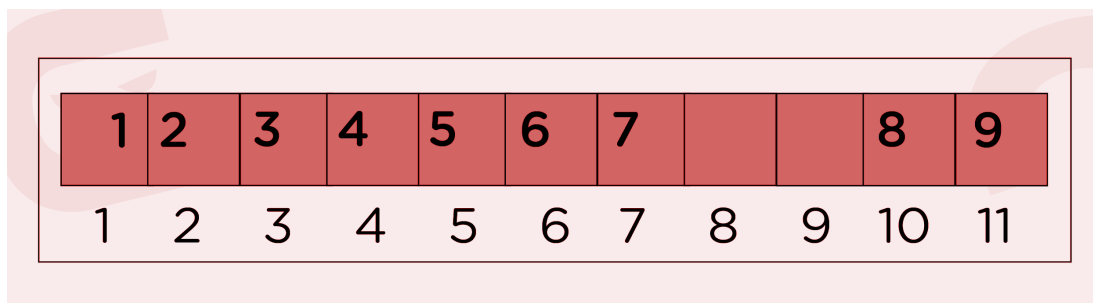
### Sequential representation

- This is the most straightforward technique to store a tree data structure. An array is used to store the tree nodes.
- The number of nodes in a tree defines the size of the array.
- The root node of the tree is held at the first index in the array.
- In general, if a node is stored at the  $i^{\text{th}}$  location, then its **left** and **right** child are kept at  $(2i)^{\text{th}}$  and  $(2i+1)^{\text{th}}$  locations in the array, respectively.

Consider the following binary tree:



The array representation of the above binary tree is as follows:



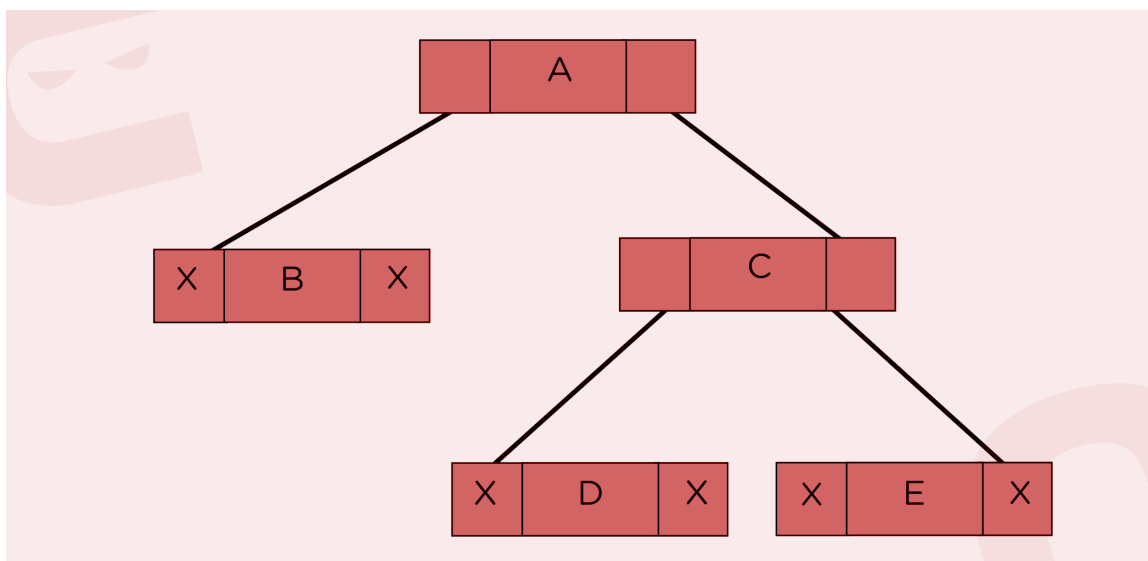
As discussed above, we see that the left and right child of each node is stored at locations  **$2 * (\text{nodePosition})$**  and  **$2 * (\text{nodePosition}) + 1$** , respectively.

**For Example,** The location of node 3 in the array is 3. So its left child will be placed at  $2*3 = 6$ . Its right child will be at the location  $2*3 + 1 = 7$ . As we can see in the array, children of 3, which are 6 and 7, are placed at locations 6 and 7 in the array.

**Note:** The sequential representation of the tree is not preferred due to the massive amount of memory consumption by the array.

### Linked list representation:

In this type of model, a linked list is used to store the tree nodes. The nodes are connected using the parent-child relationship like a tree. The following diagram shows a linked list representation for a tree.



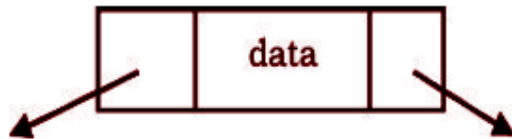
As shown in the above representation, each linked list node has three components:

- Pointer to the left child
- Data
- Pointer to the right child

**Note:** If there are no children for a given node (leaf node), then the left and right pointers for that node are set to **None**.

Let's now check the implementation of the **Binary tree class**.





Or



```
class BinaryTreeNode:
    def __init__(self, data):
        self.left = None #To store data
        self.right = None #For storing the reference to left pointer
        self.data = data #For storing the reference to right pointer
```

## Operations on Binary Trees

### Basic Operations

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

### Auxiliary Operations

- Finding the size of the tree
- Finding the height of the tree
- Finding the level which has the maximum sum and many more...

## Print Tree Recursively

Let's first write a program to print a binary tree recursively. Follow the comments in the code below:

```
def printTree(root):
    root == None: #Empty tree
        return
    print(root.data, end ":") #Print root data
    if root.left != None:
        print("L", root.left.data, end=",") #Print left child
    if root.right != None:
        print("R", root.right.data, end="") #Print right child
    Print #New Line
```

```
printTree(root.left) #Recursive call to print left subtree
printTree(root.right) #Recursive call to print right subtree
```

## Input Binary Tree

We will be following the level-wise order for taking input and -1 denotes the **None** pointer.

```
def treeInput():
    rootData = int(input())
    if rootData == -1: #Leaf Node is denoted by -1
        return None

    root = BinaryTreeNode(rootData) #Create a tree node
    leftTree = treeInput() #Take input for left subtree
    rightTree = treeInput() #Take input for right subtree
    root.left = leftTree #Assign the left subtree to the left child
    root.right = rightTree #Right subtree to the right child
    return root
```

## Count nodes

- Unlike the Generic trees, where we need to traverse the children vector of each node, in binary trees, we just have at most left and right children for each node.
- Here, we just need to recursively call on the right and left subtrees independently with the condition that the node pointer is not None.
- Follow the comments in the upcoming code for better understanding:

```
def count_nodes(node):
    if node is None: #Check if root node is None
        return 0
```

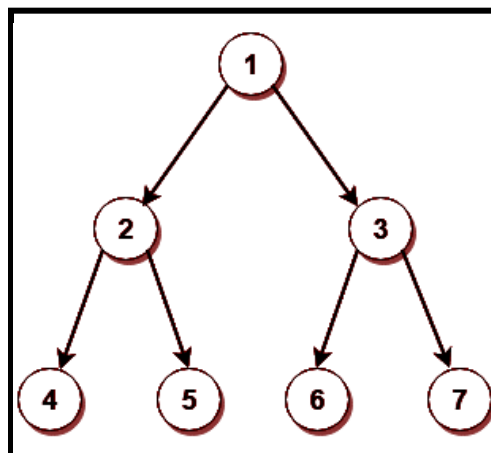
```
return 1 + count_nodes(node.left) + count_nodes(node.right)
#Recursively count number of nodes in left and right subtree and add
```

## Binary tree traversal

Following are the ways to traverse a binary tree and their orders of traversal:

- **Preorder traversal** : ROOT -> LEFT -> RIGHT
- **Postorder traversal** : LEFT -> RIGHT-> ROOT
- **Inorder traversal** : LEFT -> ROOT -> RIGHT

Some examples of the above-stated traversal methods:



- ❖ **Preorder traversal**: 1, 2, 4, 5, 3, 6, 7
- ❖ **Postorder traversal**: 4, 5, 2, 6, 7, 3, 1
- ❖ **Inorder traversal**: 4, 2, 5, 1, 6, 3, 7

Let's look at the code for inorder traversal, below:

```
# A function to do inorder tree traversal
def printInorder(root):
    if root: #If tree is not empty
        printInorder(root.left) # First recur on left child
```

```
print(root.val) # Then print the data of the node
printInorder(root.right) # Now recur on right child
```

Now, from this inorder traversal code, try to code preorder and postorder traversal yourselves. If you get stuck, refer to the solution tab for the same.

## Node with the Largest Data

In a Binary Tree, we must visit every node to figure out the maximum. So the idea is to traverse the given tree and for every node return the maximum of 3 values:

- Node's data.
- Maximum in node's left subtree.
- Maximum in node's right subtree.

Below is the implementation of the above approach.

```
def findMaximum(root):
    # Base case
    if (root == None):
        return float('-inf') ***

    # Return maximum of 3 values:
    # 1) Root's data 2) Max in Left Subtree
    # 3) Max in right subtree
    max = root.data
    lmax = findMaximum(root.left) #Maximum of Left subtree
    rmax = findMaximum(root.right) #Maximum of right subtree
    if (lmax > max):
        max = lmax
    if (rmax > max):
        max = rmax
    return max
```

**Note\*\*:** In python, float values can be used to represent an infinite integer. One can use **float('-inf')** as an integer to represent it as "Negative" infinity or the smallest possible integer.