Data Structures Lab 8

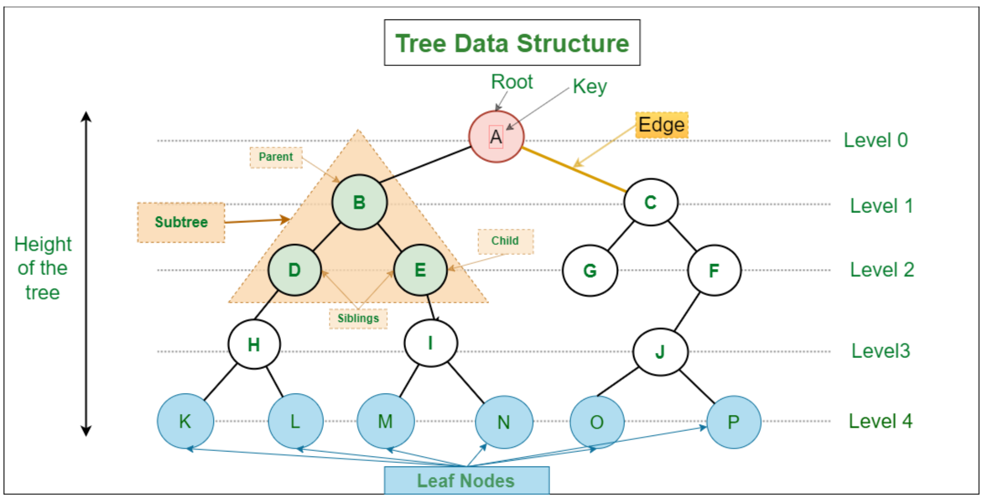
Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif

Note:

- Lab manual covers following below topics:
  {Tree, Binary Tree, Binary Search Tree, Traverse the tree with the three common orders, Operation such as searches, insertions, and removals on a binary search tree and its applications}
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

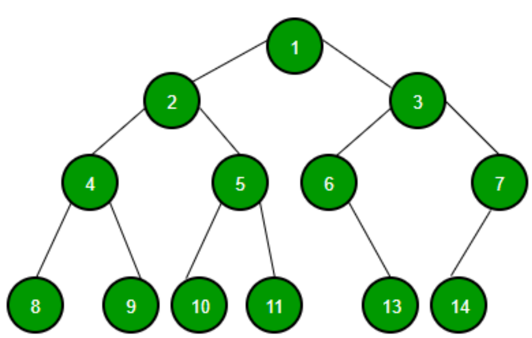
Tree

A tree data structure (**non-linear**) is a **hierarchical structure** that is used to **represent and organize data** in a way that is **easy to navigate and search**. It is a collection of nodes that are connected by **edges** and has a hierarchical relationship between the **nodes**.



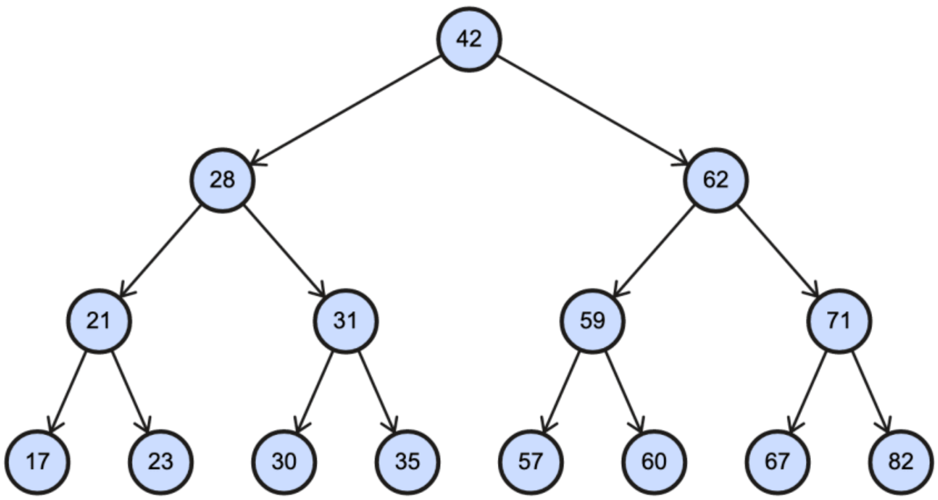
Binary Tree

Binary Tree is defined as a tree data structure where each node has **at most 2 children**. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



BINARY SEARCH TREE

Binary Search Tree is a node-based binary tree data structure which has the **left subtree of a node** that contains only nodes with **keys less than the node's key**. Similarly, the **right subtree of a node** contains only nodes with **keys greater than the node's key**. The left and right **subtree** each must also be a **binary search tree**.



BST Insertion

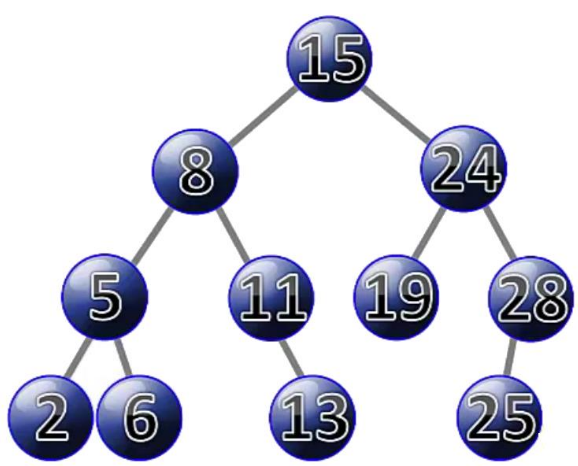
Sample Code of class Nodes:

```
Class Node:
    Attributes:
        - data: integer
        - leftChild: Node (pointer to left child)
        - rightChild: Node (pointer to right child)

    Constructor():
        data <- 0
        leftChild <- null
        rightChild <- null

    Constructor(d: integer):
        data <- d
        leftChild <- null
        rightChild <- null

------------------------------------------------------------

Class BinaryTree:
    Attribute:
        - root: Node (pointer to root)

    Constructor():
        root <- null

    Constructor(d: integer):
        root <- new Node(d)
```

```
Method insertNode(data: integer):
    // Create a new Node
    Node newNode <- new Node(data)

    // If there is no root, this becomes the root
    If root == null:
        root <- newNode
    Else:
        // Start with the root node
        Node temp <- root
        Node parent <- null

        While temp != null:
            parent <- temp  // Keep track of the parent node

            // Check if the new node should go on the left side
            If data < temp.data:
                temp <- temp.leftChild  // Move to the left child
            Else:
                temp <- temp.rightChild  // Move to the right child

        // If there's no left child, place the new node here
        If data < parent.data:
            parent.leftChild <- newnode
        // If there's no right child, place the new node here
        Else:
            parent.rightChild <- newnode

    Method inOrderTraversal(temp: Node):
        If temp != null:
            inOrderTraversal(temp.leftChild)
            print("Data:", temp.data)
            inOrderTraversal(temp.rightChild)

    Method display():
        inOrderTraversal(root)
```

Example:

Add New Node with Key’s value is 12.



Tree Traversals: Inorder, PreOrder, PostOrder

Tree Traversals:

```
preorder(node):
  if node == null: return
  print(node.value)
  preorder(node.left)
  preorder(node.right)
```

preorder prints before the recursive calls

```
inorder(node):
  if node == null: return
  inorder(node.left)
  print(node.value)
  inorder(node.right)
```

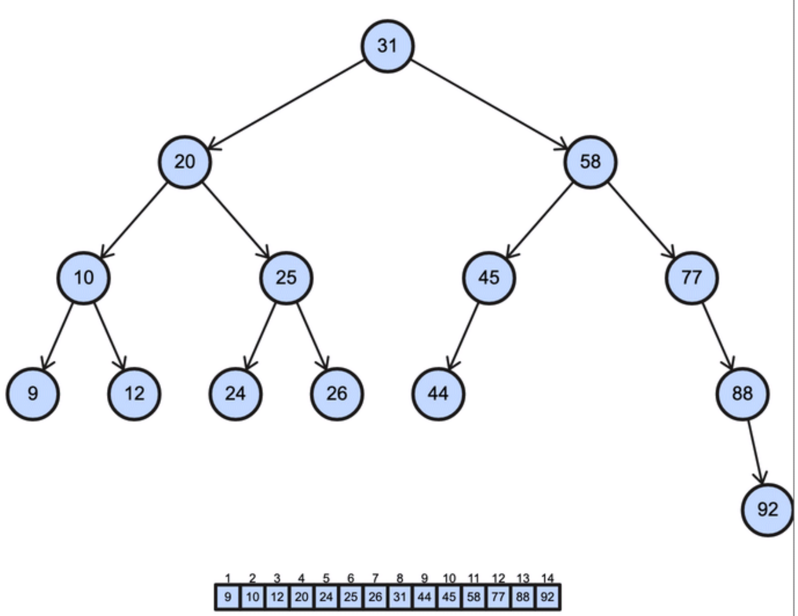
inorder prints between the recursive calls

```
postorder(node):
  if node == null: return
  postorder(node.left)
  postorder(node.right)
  print(node.value)
```

postorder prints after the recursive calls

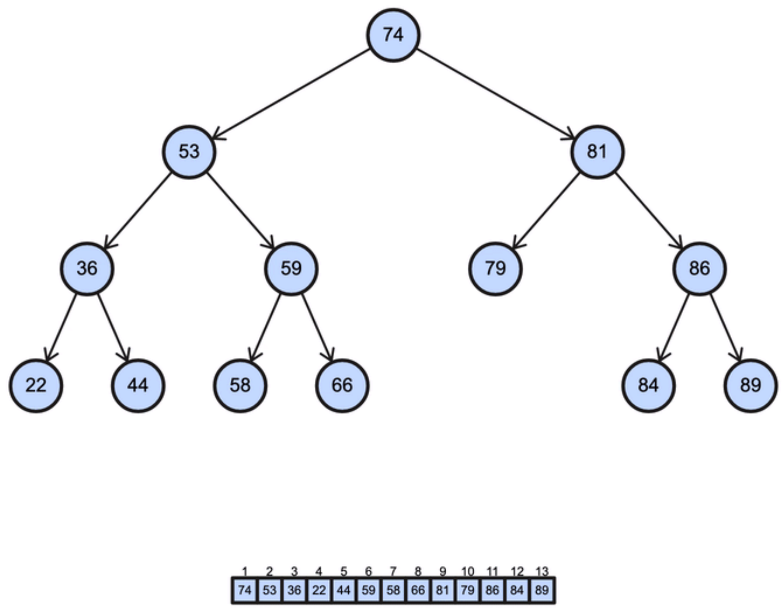
Pseudo code for InOrder Traversal (Iterative)

```
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current
is NULL
4) If current is NULL and stack is not empty then
     a) Pop the top item from the stack.
     b) Print the popped item, set current = popped_item->right
     c) Go to step 3.
5) If the current is NULL and stack is empty then we are done.
```



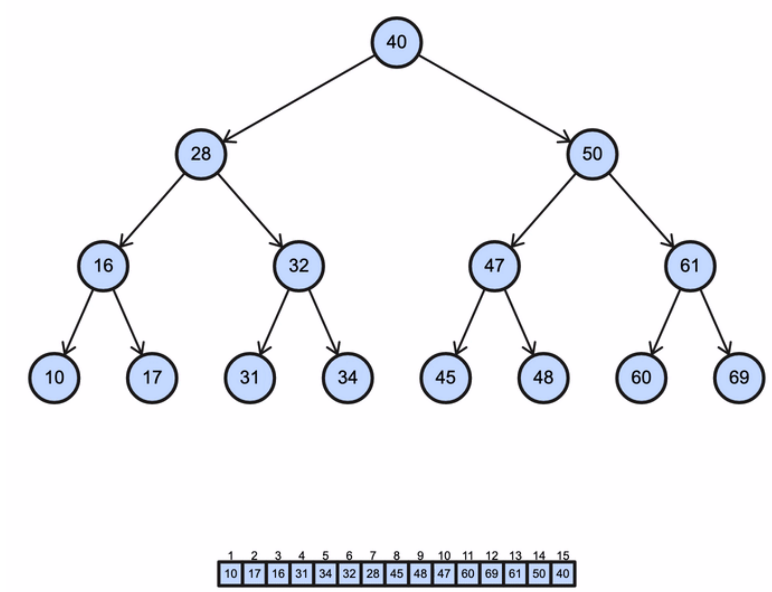
Pre-Order Traversal (Recursive):

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: Write TREE -> DATA
Step 3: PREORDER(TREE -> LEFT)
Step 4: PREORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END
```



Post-Order Traversal (Recursive):

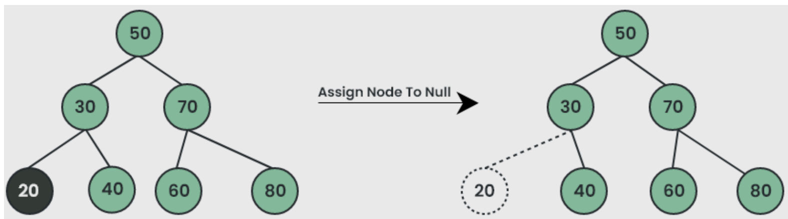
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: POSTORDER(TREE -> LEFT)
Step 3: POSTORDER(TREE -> RIGHT)
Step 4: Write TREE -> DATA
[END OF LOOP]
Step 5: END
```



BST Deletion

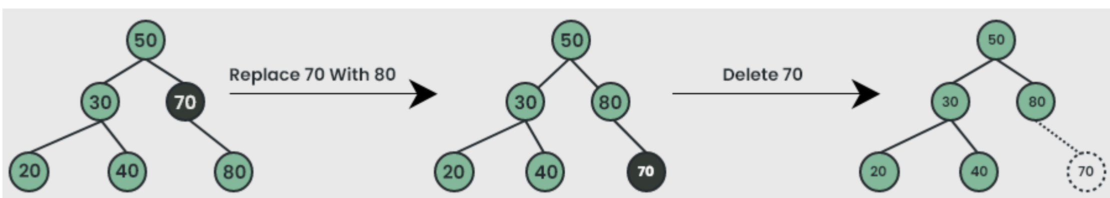
1) ***Node to be deleted is the leaf:*** Simply remove from the tree.

```
          50                         50
       /     \        delete(20)    /   \
      30      70      --------->  30     70
     /  \    /  \                   \   /  \
   20   40  60   80                 40 60   80
```



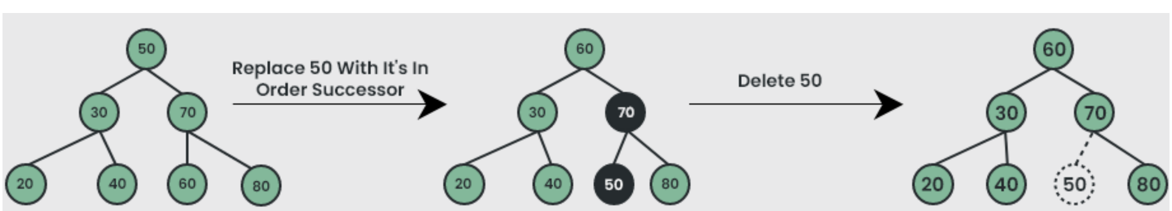
2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
          50                         50
       /     \        delete(30)    /   \
      30      70      --------->  40     70
        \    /  \                       /  \
        40  60   80                    60   80
```



3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
          50                         60
       /     \        delete(50)    /   \
      40      70      --------->  40     70
             /  \                          \
            60   80                         80
```



The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the **minimum value in the right child of the node**.

```
    Method deleteNode(key: integer):
        root <- deleteNodeHelper(root, key)

    Method deleteNodeHelper(temp: Node, key: integer):
        // Base case: if the tree is empty
        If temp == null:
            Return temp

        // Traverse the tree
        If key < temp.data:
            temp.leftChild <- deleteNodeHelper(temp.leftChild, key)//Go left
        Else If key > temp.data:
            temp.rightChild<-deleteNodeHelper(temp.rightChild, key)//Go right
        Else:
            // Node to be deleted found
            // Case 1: Node has no child (leaf node)
            If temp.leftChild == null and temp.rightChild == null:
                delete temp
                Return null  // Return null to the parent

            // Case 2: Node has one child (left or right)
            Else If temp.leftChild == null:
                Node* right <- temp.rightChild
                delete temp
                Return right  // Return the right child to the parent
            Else If temp.rightChild == null:
                Node* left <- temp.leftChild
                delete temp
                Return left  // Return the left child to the parent

            // Case 3: Node has two children
            Node* minRight<-findMin(temp.rightChild)//Find the in-order
successor
            temp.data<-minRight.data //Replace the data with the successor's
data
            temp.rightChild <- deleteNodeHelper(temp.rightChild,
minRight.data)  // Delete the successor node

        Return temp  // Return the current node (which may be the same or
modified)

      // Find the node with the minimum value in a subtree
    Method findMin(temp: Node):
        If temp == null:
            Return null
        While temp.leftChild != null:
            temp <- temp.leftChild
        Return temp
```

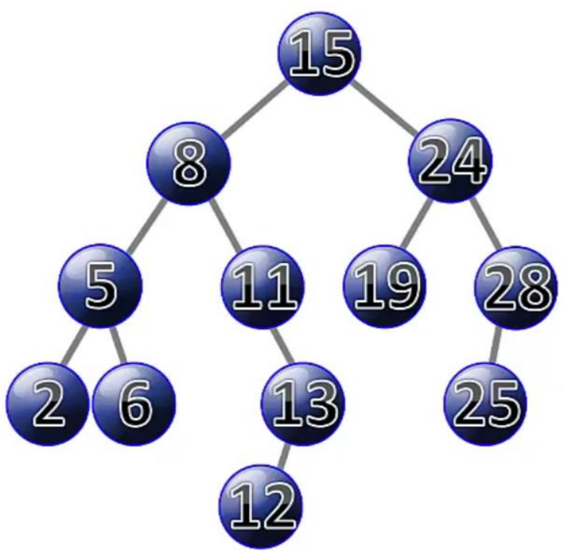
Searching:

Given a pointer to the root of the tree and a key *k*, TREE-SEARCH returns a pointer to a node with key *k* if one exists; otherwise, it returns NIL. The procedure begins its search at the root and traces a path downward in the tree. For each node *node->data== k* it encounters, it compares the key *k* with *node->data*. If the two keys are equal, the search terminates. If *k* is smaller than *node.data*, the search continues in the left subtree of *node->leftChild*, since the binary-search-tree property implies that *k* could not be stored in the right subtree. Symmetrically, if *k* is larger than *node->data*, the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

```
Method search(key: integer):
    Return searchHelper(root, key)

Method searchHelper(node: Node, key: integer):
    // Base case: if node is null or key is found
    If node == null:
        Return node  // Key not found

    If node.data == key:
        Return node  // Key found

    // If key is less than the current node's data, search in the left
subtree
    If key < node.data:
        Return searchHelper(node.leftChild, key)  // Go left
    Else:
        // If key is greater than the current node's data, search in the
right subtree
        Return searchHelper(node.rightChild, key)  // Go right
```

Example:

Search the node having key value is 19.

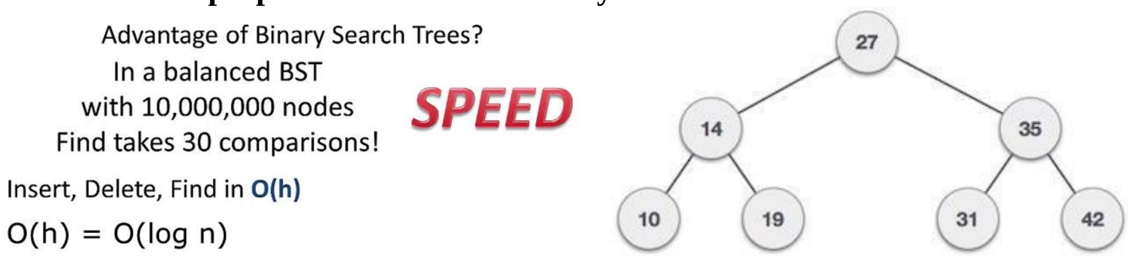


Some points to Note:

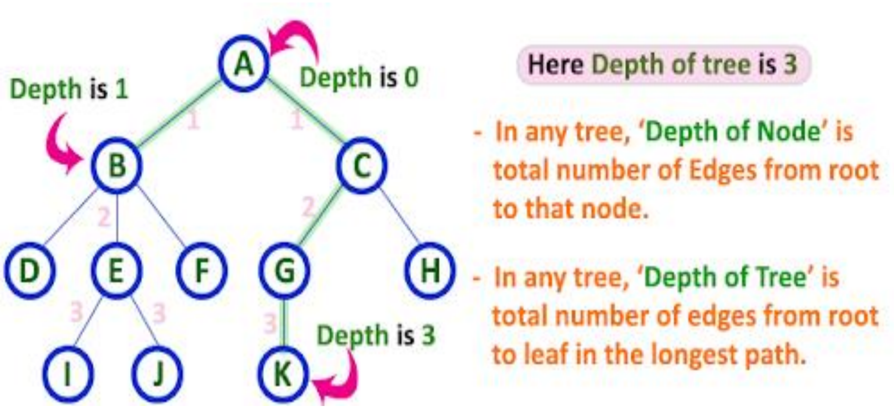
Binary search tree (BST)

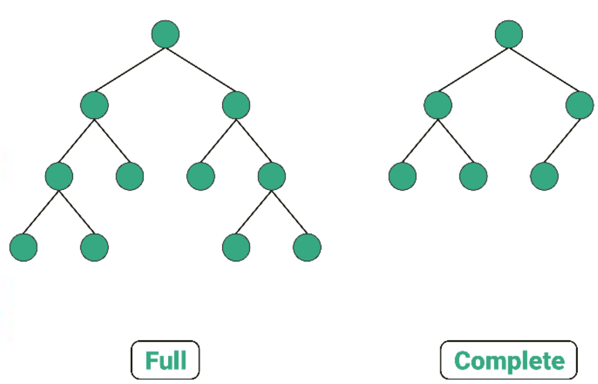
Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

- Each **node has a value**.
- The key value of the **left child of a node is less than to the parent's key value**.
- The key value of the **right child of a node is greater than (or equal) to the parent's key** value.
- And these **properties hold true for every node** in the tree.



- **Subtree**: any node in a tree and its descendants.
- **Depth of a node**: the number of steps to hop from the current node to the root node of the tree.
- **Depth of a tree**: the maximum depth of any of its leaves.
- **Height of a node**: the length of the longest downward path to a leaf from that node.
- **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**.
- **Complete binary tree:** A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is **filled from the left**. A complete binary tree is just like a full binary tree, but with two major differences:
  1. All the leaf elements must lean towards the left.
  2. 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.
- **Traversal**: an organized way to visit every member in the structure.





Traversals:

The binary search tree property allows us to obtain all the keys in a binary search tree in **a sorted order by a simple traversing algorithm, called an inorder** tree walk, that traverses the left subtree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.
The algorithms are described below, with Node initialized to the tree's root.

• **Preorder Traversal**

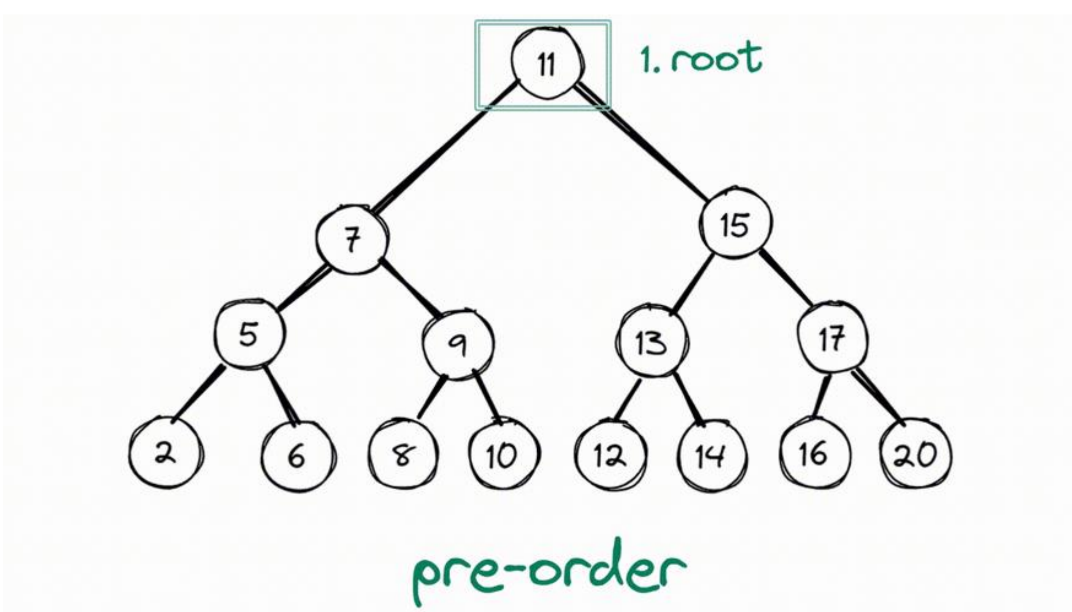
1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

• **In-order Traversal**

1. Traverse Node's left sub-tree.
2. Visit Node.
3. Traverse Node's right sub-tree

• **Post-order Traversal**

1. Traverse Node's left sub-tree.
2. Traverse Node's right sub-tree.
3. Visit Node



Tasks

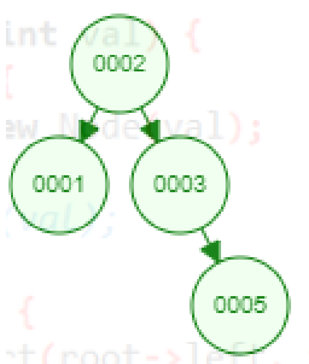
Q1: Implement a binary search tree and perform all operations you learned above like: Inserting, Deleting, Searching, and Traversing

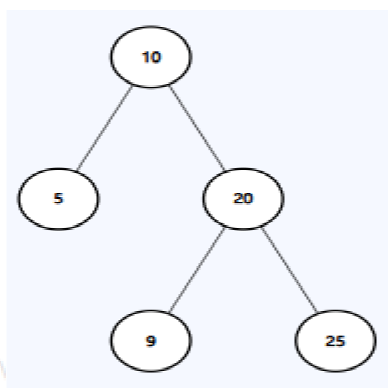
Q2: Search for the value defined by the user in the tree. If the value does not exist insert it and print the new tree.

Q3: Given two BST's (Down Below) You are tasked to merge them together to form a new BST the output of this will be 1 2 2 3 3 4 5 6 6 7.

```
BST1:
       5
     /   \
    3     6
   / \
  2   4
BST2:
        2
      /   \
     1     3
            \
             7
            /
           6
```

Q4: Given the root of a binary tree. Check whether these are Binary Search Tree or not.





Q5: Suppose you are working on a project for a small online retailer that needs to keep track of their inventory using a binary search tree. The retailer's inventory consists of a unique product ID and its corresponding quantity in stock. Write a C++ class for the binary search tree and add the required functionalities to insert new products into the tree, update the quantity of existing products, and search for products by their ID.

Additionally, the retailer would like to keep track of the product with the highest quantity in stock. Implement a function that returns the ID of this product, along with its quantity.

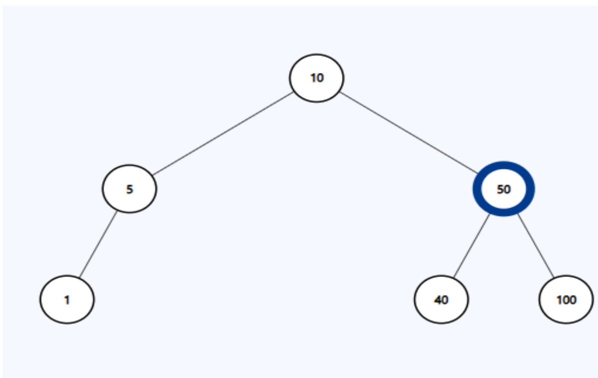
Q6: Given a Binary Search Tree, find the median of it.

- If the number of nodes are even: then median = ((n/2 + ((n)/2+1)) /2
- If the number of nodes are odd: then median = (n+1)/2

Q7: Given a Binary Search Tree (BST) and a range [a, b], the task is to count the number of nodes in the BST that lie in the given range.

Examples:

Input: a= 5, b = 45



Output: 3

Explanation: There are three nodes in range [5, 45] = 5, 10 and 40.