**Data Structures Lab 08**

**Course:** Data Structures (CL2001) **Semester:** Fall 2024
**Instructor:** Sameer Faisal

---

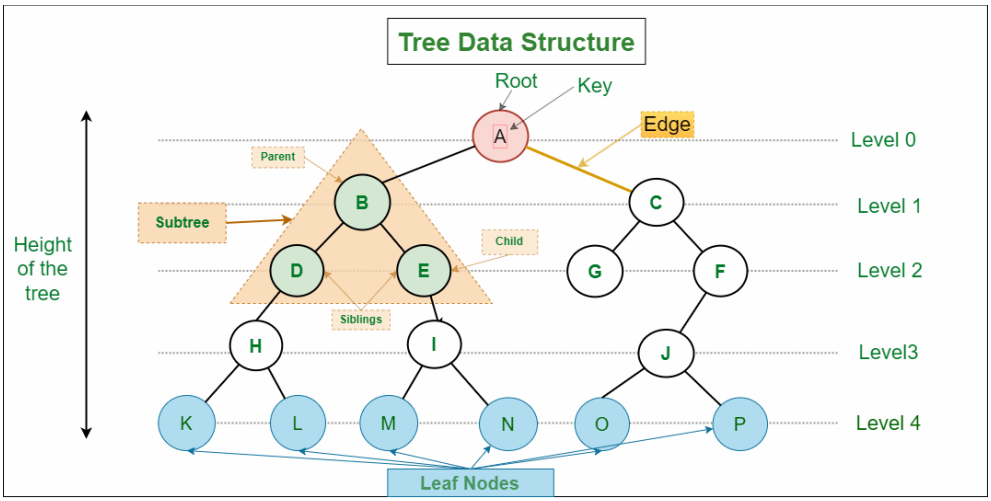
**Note:**

- Lab manual covers following below topics:
  **{Tree, BST, Design and implement classes for binary tree nodes and nodes for general 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.

---

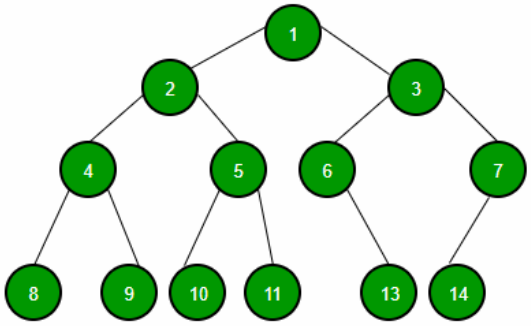
## Tree

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**.



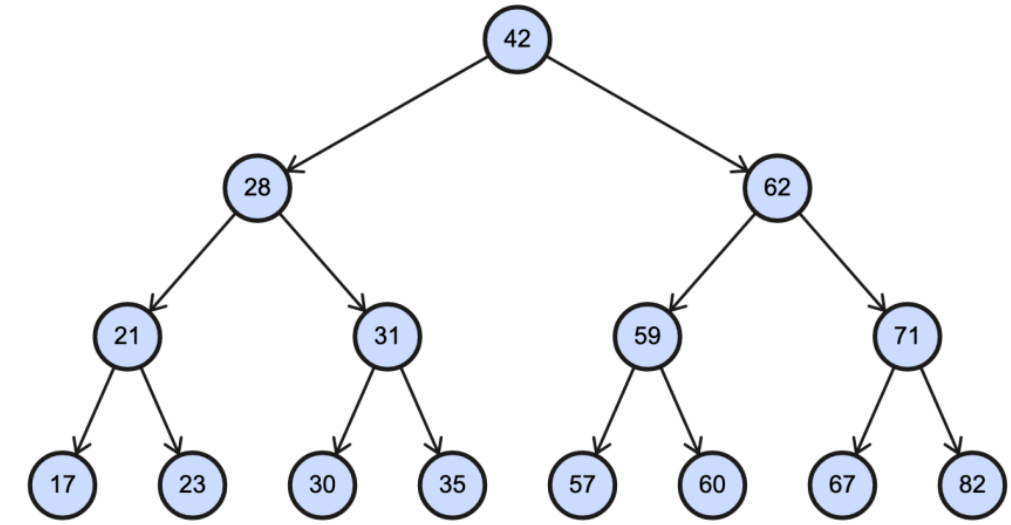

## Binary Tree

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

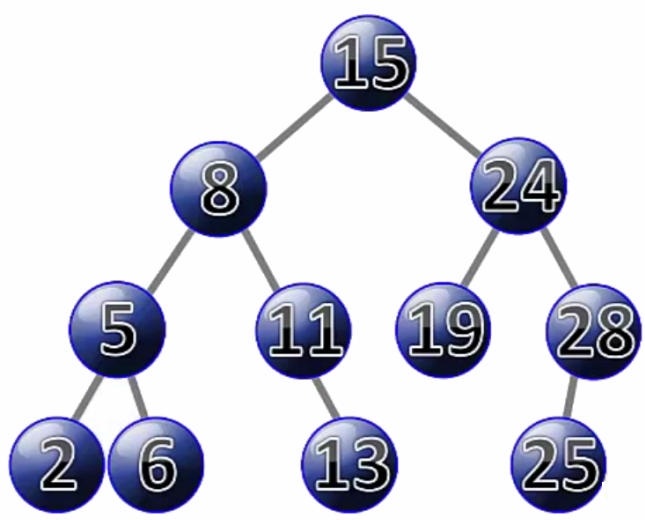
**Sample Code of class Nodes**

```
Create class Nodes
class Node {    private:
 int key;
 string name;

 Node leftChild;
  Node rightChild; public:
 Node(int key, string name) {

       this.key = key;
       this.name = name;

 }
string toString() {

       return cout<<name<< " has the key " <<key<<endl;
        } };

class BinaryTree {
  private:   Node root;
 public:
 void addNode(int key, string name) {

       ----------------------// Create a new Node and initialize it
```

```
// If there is no root this becomes root

if (root == NULL) {

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

} else {

	// Set root as the Node we will start with as we traverse the tree

	-----------------------------
	// Future parent for new Node

	Node parent;

	while (true) {

		// root is the top set the parent node to the root node

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

		// Check if the new node should go on
		// the left side of the parent node
		Key is compared with that of root. If the key is less than root,
		it is compared with root's left child key. If greater, it is
		compared with the root's right child. Continue this process until
		the new node is compared with a leaf node and added either on the
		right or left child depending on its key.

		}
```

**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)

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

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

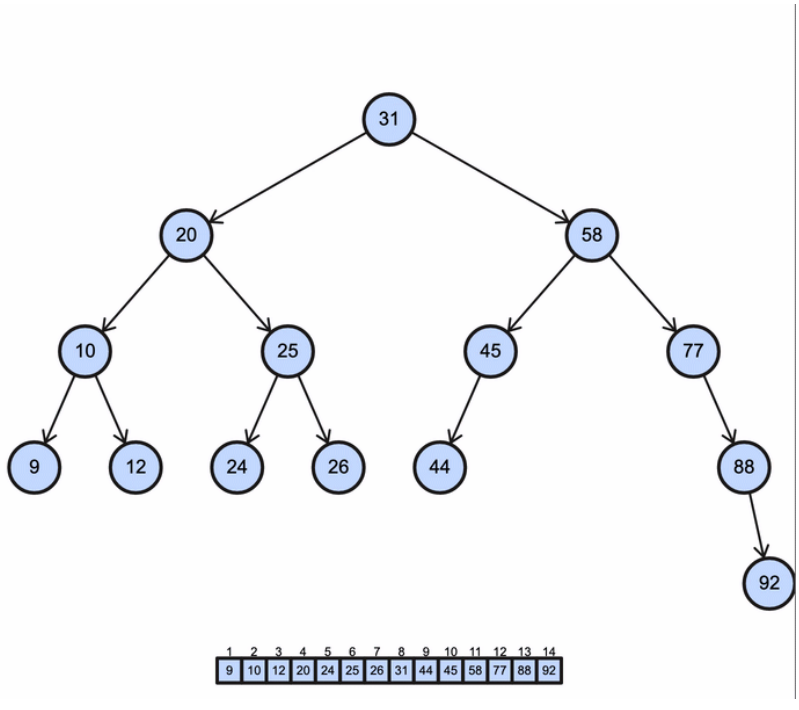
preorder prints before the recursive calls

inorder prints between the recursive calls

postorder prints after the recursive calls

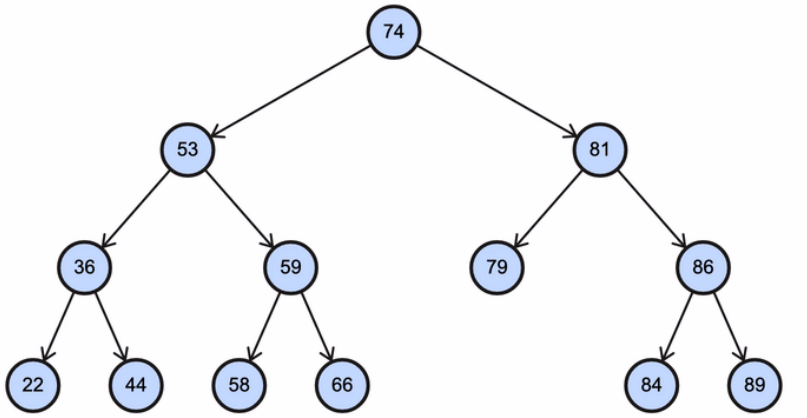
### Pseudo code for Inorder Traversal (iteration)

```
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 stack.
     b) Print the popped item, set current = popped_item->right
     c) Go to step 3.
5) If current is NULL and stack is empty then we are done.
```



**Pre-Order Traversal**

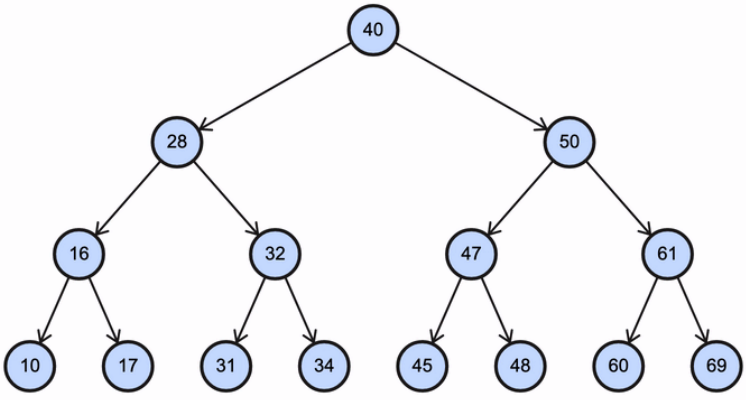
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: Write TREE -> DATA
Step 3: PREORDER(TREE -> LEFT)
Step 4: PREORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END
```




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 74 | 53 | 36 | 22 | 44 | 59 | 58 | 66 | 81 | 79 | 86 | 84 | 89 |

**Post-Order Traversal**

```
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
```




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 17 | 16 | 31 | 34 | 32 | 28 | 45 | 48 | 47 | 60 | 69 | 61 | 50 | 40 |

## BST Deletion

**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
```

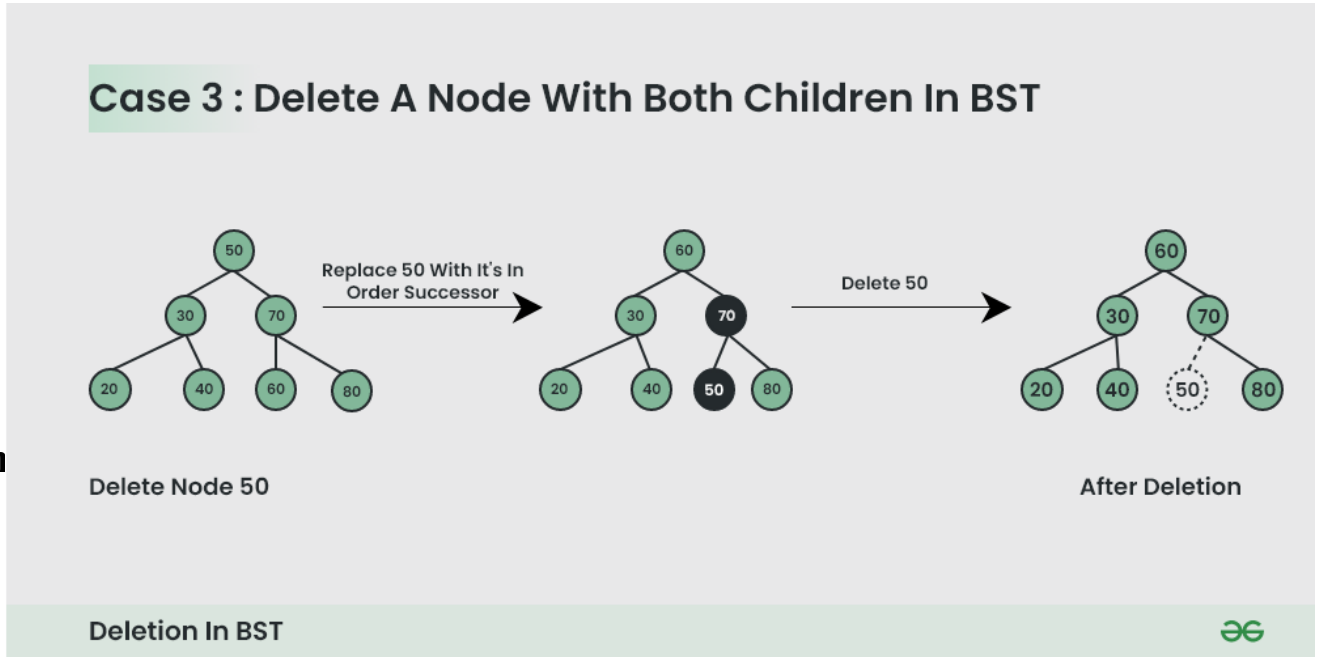
**2)** ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
          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
```

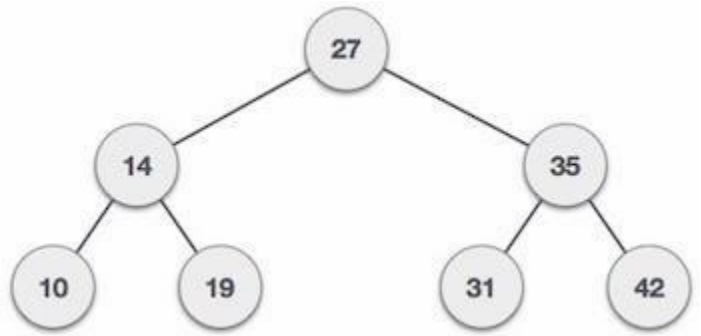
The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the **minimum value in the right child of the node**.




**Son**

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**

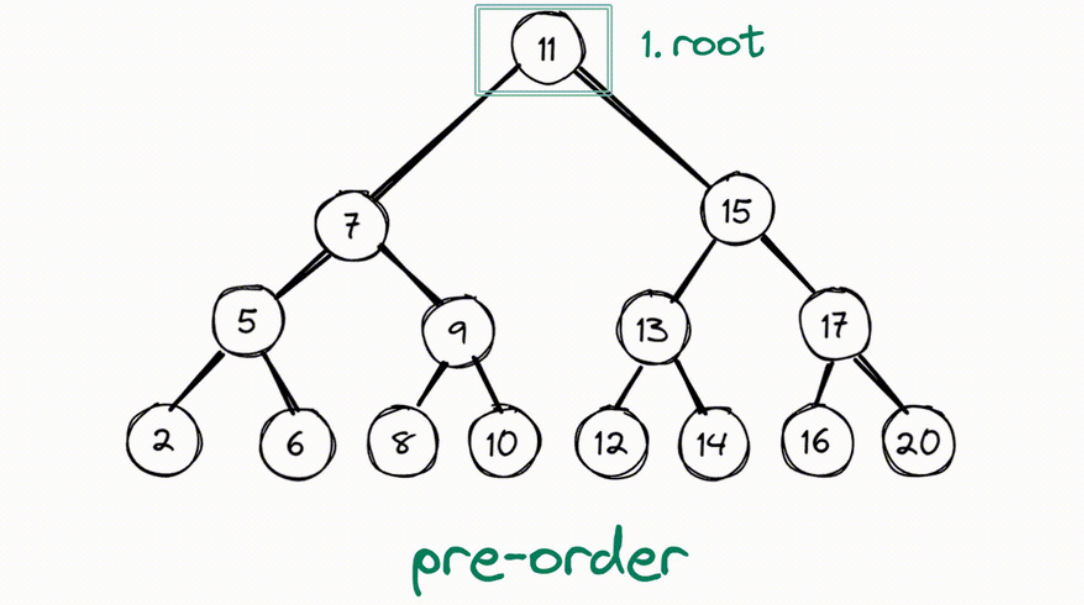
1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

**• 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




## Searching:

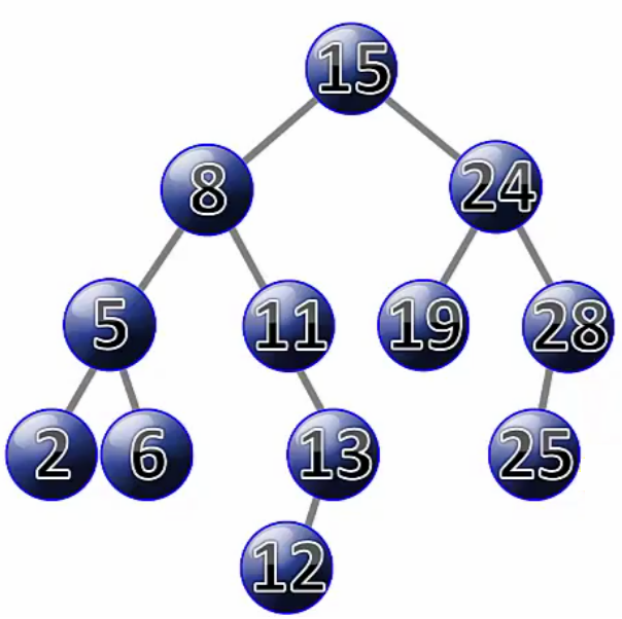
We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key *k,* TREE-SEARCH returns a pointer to a node with key *k* if one exists; otherwise, it returns NIL.

```
TREE-SEARCH (x, k)
1 if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH (left[x], k)
5     else return TREE-SEARCH (right[x], k)
```

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 13.2. For each node *x* it encounters, it compares the key *k* with *key*[*x*]. If the two keys are equal, the search terminates. If *k* is smaller than *key*[*x*], the search continues in the left subtree of *x,* since the binary-search-tree property implies that *k* could not be stored in the right subtree. Symmetrically, if *k* is larger than *key*[*k*], 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.

## Example:

Search the node having key value is 19.



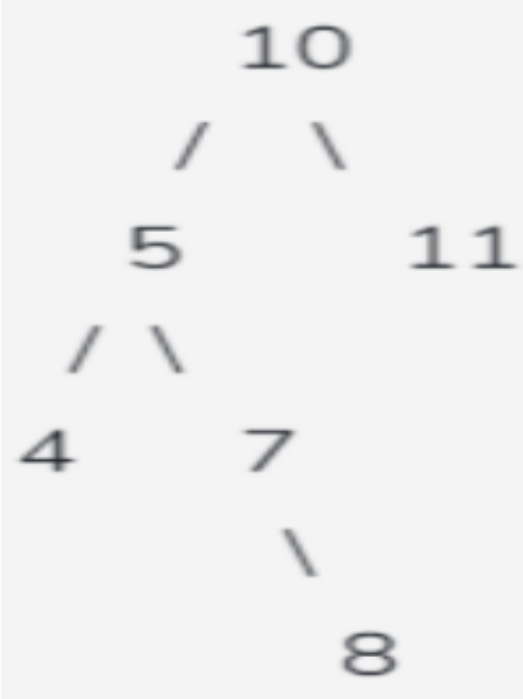

# Lab Exercises

Q1: Given an array of {1,2,3,4,5} implement a binary tree according.

Q2: Using the same tree that you made insert the values 6,7,8 accordingly. Now check if the tree made is a complete binary tree or if it's a full binary tree if not delete the nodes accordingly, to make it a complete and full binary tree. (I know the tree will be complete but implement a logic to check if it is complete).

Q3: Search for the value defined by the user in the tree. If the value does not exist insert it then identify its location based on the level of the tree and if it's the root at level, left child or right child.

Q4: Given a Tree of nodes (Down below):




You are tasked to take a value X from the user and ask whether to ceil or floor the value. If ceil is selected find the closest value to X in our case 7 at level three right child because if we ceil 6 we get 7 (Yes, I am familiar with the logic of how ceil and floor works with floating points but this scenario is different so in the case of ceil add 1 to X and in the case of floor subtract 1 from x then find the value). Also find multiple occurrences if it as well.

Q5: 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 (You can visualize this by drawing the array to a tree in your notebook).

**BST1:**

**BST2:**