Data Structure Algorithms & Applications (CT-159)
Lab 08

Tree and Binary Tree

Objectives

The objective of the lab is to get students familiar with Binary Tree data structure, depth-first-traversal, breadth-first-traversal, insertion and deletion operations.

Tools Required

Dev C++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Introduction

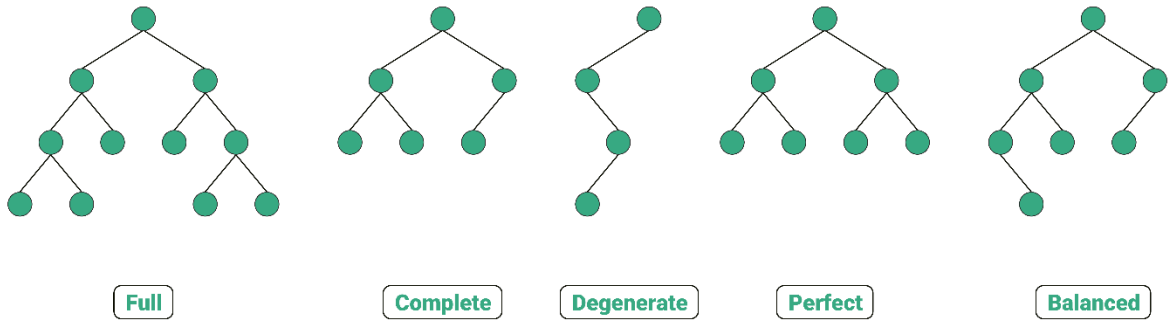
A **tree** is a hierarchical data structure consisting of nodes connected by edges. It is defined recursively as a collection of nodes, where the topmost node is called the **root**. Each node may have zero or more **child nodes**. Nodes with no children are called **leaf nodes**. Each child node has exactly one **parent** node, and there are no cycles, meaning a node cannot be its own ancestor.

Types of Trees: There are various types of trees based on different properties and applications. Below are the key types:

1. **General Tree**: No restrictions on the number of children per node. Used for representing hierarchical data structures like organizational charts, file systems, etc.
2. **Binary Tree**: Each node has at most two children, referred to as the **left** and **right** children. Binary trees are useful in representing expression trees, decision trees, etc.
3. **Complete Binary Tree**: All levels are completely filled except possibly the last level, which is filled from left to right. Used in heaps.
4. **Perfect Binary Tree**: A binary tree in which all interior nodes have exactly two children, and all leaves are at the same level.
5. **Balanced Tree**: A binary tree where the height is kept minimal, ensuring efficient operations (usually O(logn)).
6. **Full Binary Tree**: Every node other than the leaves has exactly two children.
7. **Binary Search Tree (BST)**: A binary tree where for every node, all nodes in the left subtree have smaller values, and all nodes in the right subtree have larger values. Allows for efficient searching, insertion, and deletion.
8. **Heap**: A complete binary tree used primarily in priority queues. **Max-Heap**: Parent nodes are always greater than or equal to their children. **Min-Heap**: Parent nodes are always less than or equal to their children.
9. **B-tree**: A balanced tree used in databases and file systems for efficient disk access. Nodes can have multiple children, and it maintains sorted data with efficient insertion, deletion, and search.
10. **Trie (Prefix Tree)**: A specialized tree used to store strings where each edge represents a character. Commonly used in autocomplete and spell-checking.

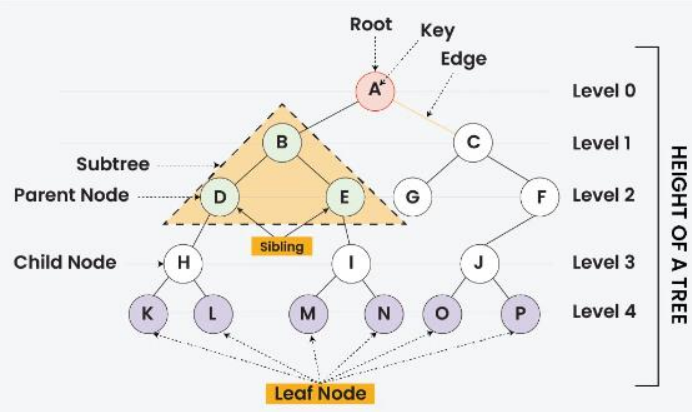
Binary Tree

A **binary tree** is a hierarchical data structure in which each node has at most two children. These children are referred to as the **left child** and the **right child**. The binary tree starts with a single node called the **root**.



Binary Tree Properties:

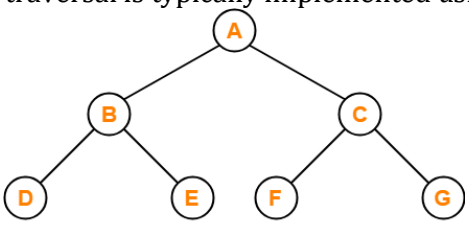
- **Depth of a Node**: The depth of a node is the number of edges from the root to the node. The root node has depth 0.
- **Degree of a Node**: The degree is the number of children of a node. The degree of a tree is the degree of the root node. In a binary tree, the degree of a node can be 0, 1 or 2.
- **Number of nodes at level l**: The level of a node is its depth. At level l, a binary tree can have at most 2^l nodes.
- **Maximum number of nodes in a binary tree of height h**: The maximum number of nodes is $2^h - 1$.
- **Minimum height of a binary tree with n nodes**: The minimum height is $\lceil log_2(n+1) \rceil$.



Binary Tree Traversals:

Binary tree traversal refers to the process of visiting each node in a binary tree in a specific order. Traversals are essential for many operations, such as searching, printing, or modifying tree nodes. There are two primary categories of binary tree traversals: **depth-first traversal** and **breadth-first traversal**.

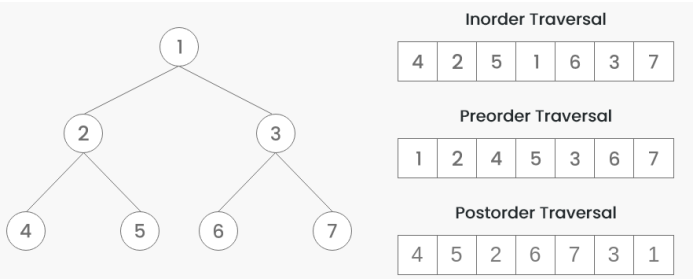
In **breadth-first traversal,** the tree is explored level by level, starting from the root. Nodes at each level are visited from left to right. This traversal is useful for searching the shortest path in an unweighted tree. Breadth-first traversal is typically implemented using a **queue** data structure.



Level Order Traversal : A , B , C , D , E , F , G

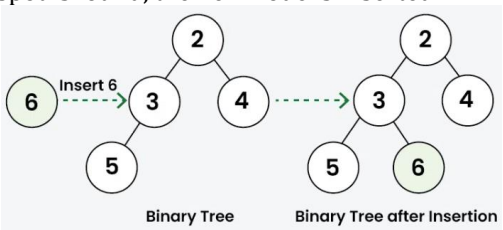
In **depth-first traversal**, the algorithm explores as far down one branch as possible before backtracking. There are three types of depth-first traversals:

1. Inorder Traversal (Left, Root, Right): Traverses the left subtree, visits the root, and then traverses the right subtree.
2. Preorder Traversal (Root, Left, Right): Visits the root first, then traverses the left subtree, and finally the right subtree.
3. Postorder Traversal (Left, Right, Root): Traverses the left subtree, then the right subtree, and finally visits the root.



Insert a Node in Binary Tree

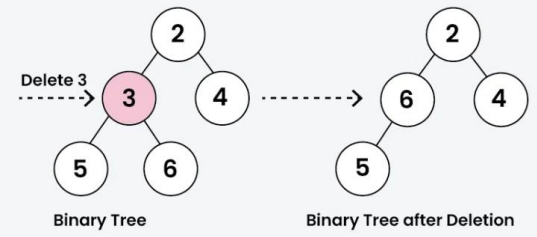
1. If the binary tree is empty, the new node becomes the root.
2. Otherwise, a level-order traversal (using a queue) is performed to find the first empty spot (either in the left or right child).
3. Once the first available spot is found, the new node is inserted.



This approach ensures that the new node is added in a complete binary tree fashion, filling levels from left to right.

Delete a Node from a Binary Tree

1. **Finding the node to delete**: We first perform a level-order traversal to find the node that needs to be deleted (nodeToDelete) and keep track of the deepest node in the tree (deepestNode).
2. **Replacement**: Once the nodeToDelete is found, we replace its value with the value of the deepestNode.
3. **Deleting the deepest node**: We then delete the deepest node to ensure that the binary tree structure remains valid.
4. **Edge case**: If the tree is empty or only has one node, handle that accordingly.



This approach keeps the binary tree complete while ensuring that the node is properly deleted.

Example 01: Implementation of a binary tree with traversal, search, insert and delete operations.

```
#include <iostream>
#include <queue>
using namespace std;

class Node {   // Node structure for the binary tree
   public:
   int data;
   Node* left;
   Node* right;

   Node(int value) {    // Constructor to initialize node
      data = value;
      left = nullptr;
      right = nullptr;
   }
};
class BinaryTree {    // Binary Tree Class
private:
   Node* root;

public:
   BinaryTree() {
      root = nullptr;
   }

   void insert(int data) {     // Function to insert data in level-order (simple binary tree, not based on value comparison)
      Node* newNode = new Node(data);
      if (root == nullptr) {       // If the tree is empty, set the new node as the root
         root = newNode;
         return;
      }
      queue<Node*> q;             // Use a queue for level-order traversal to find the first available position
      q.push(root);
      while (!q.empty()) {
         Node* current = q.front();
         q.pop();

         // Check the left child
         if (current->left == nullptr) {
            current->left = newNode;
            return;
         } else {
            q.push(current->left);
         }

         // Check the right child
         if (current->right == nullptr) {
            current->right = newNode;
            return;
         } else {
```

```
            q.push(current->right);
        }
      }
    }

    void inorder() {   // Function to perform inorder traversal (left -> root -> right)
      inorderTraversal(root);
      cout << endl;
    }
private:
    void inorderTraversal(Node* node) {          // Recursive inorder traversal function
      if (node == nullptr) return;
      inorderTraversal(node->left);
      cout << node->data << " ";
      inorderTraversal(node->right);
    }
};

int main() {
    BinaryTree tree;
    // Insert values into the binary tree in level-order
    tree.insert(1);
    tree.insert(2);s
    tree.insert(3);
    tree.insert(4);
    tree.insert(5);
    tree.insert(6);
    tree.insert(7);

    cout << "Inorder Traversal: ";
    tree.inorder();
}
```

Key Use Cases for Different Traversals:

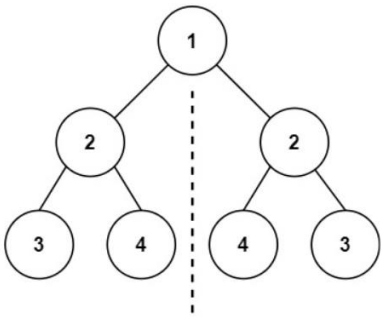
- Inorder Traversal: Used in binary search trees to retrieve nodes in sorted order.
- Preorder Traversal: Used to copy the tree or to prefix-express an arithmetic expression stored in a tree.
- Postorder Traversal: Useful for deleting or freeing nodes in a tree or evaluating postfix arithmetic expressions.
- Level-Order Traversal: Used in breadth-first search and is ideal for finding the shortest path in unweighted trees or graphs.

Applications of Binary Trees:

- Expression Trees: Used to represent arithmetic expressions.
- Binary Search Trees: Used for searching, inserting, and deleting data efficiently.
- Heap Data Structure: A special binary tree used for implementing priority queues.
- Huffman Coding Tree: Used in data compression algorithms.

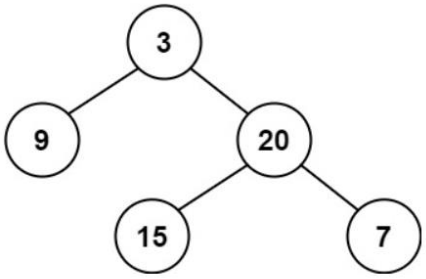
Exercise

1. Implement methods for the following operations in Binary tree class given in example 01.
    - Searching a node based on a given value.
    - Pre-order traversal
    - Post-order traversal
2. Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

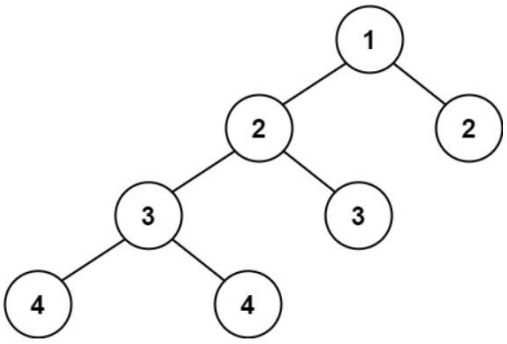



```
Input: root = [1,2,2,3,4,4,3]
Output: true
```

3. Given a binary tree, determine if it is height-balanced i.e., the absolute difference between the left and right subtree of each node is not greater than 1.

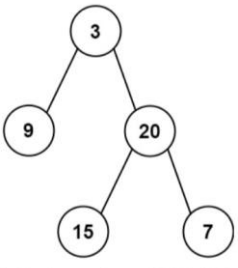



```
Input: root = [3,9,20,null,null,15,7]
Output: true
```



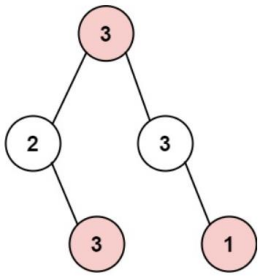

```
Input: root = [1,2,2,3,3,null,null,4,4]
Output: false
```

4. Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

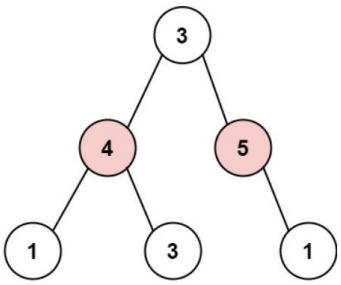



```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

5. The thief has found himself a new place for his thievery again. There is only one entrance to this area, called root. Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that all houses in this place form a binary tree. It will automatically contact the police if two directly-linked houses were broken into on the same night. Given the root of the binary tree, return the maximum amount of money the thief can rob without alerting the police.



Input: root = [3,2,3,null,3,null,1]
Output: 7
Explanation: Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.



Input: root = [3,4,5,1,3,null,1]
Output: 9
Explanation: Maximum amount of money the thief can rob = 4 + 5 = 9.

Lab 08 Evaluation		
Student Name:	**Student ID:**	**Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		