Experiment :- 6

Student Name: Shreyash Moghe
UID: 22BET10041
Branch: BE-IT
Section/Group: 22BET_IOT_703/A
Semester: 6th
Date of Performance: -01-25
Subject Name: Advanced Programming Lab-2
Subject Code: 22ITP-351

- **Aim: Maximum Depth of Binary Tree**

- **Objective:**

- . Find the maximum depth (height) of a binary tree.
- Use DFS (recursion) or BFS (iteration) to traverse the tree.
- Handle empty tree (depth = 0) and single-node tree (depth = 1).
- Return the longest path from root to the deepest leaf node.

- **Implementation/Code:**

```
class Solution {
public:
int maxDepth(TreeNode* root) {
if (!root) return 0;
int l = maxDepth(root->left), r = maxDepth(root->right);
return 1 + max(l, r);
  }
};
```

- **Aim: Validate Binary Search Tree**

- **Objective:**

- Check if a given binary tree is a valid BST.
- Every node must satisfy left < root < right for all subtrees.
- Use in-order traversal (should be strictly increasing) or DFS with min-max range.
- Handle empty tree, single node, and trees with duplicate values.

- **Code:**

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        TreeNode* prev = nullptr;
        function<bool(TreeNode*)> dfs = [&](TreeNode*
root) {
            if (!root) {
                return true;
            }
            if (!dfs(root->left)) {
                return false;
            }
            if (prev && prev->val >= root->val) {
                return false;
            }
            prev = root;
            return dfs(root->right);
        };
        return dfs(root);
    }
};
```

- **Aim: Symmetric Tree**

- **Objective:**

- Check if a given binary tree is mirror-symmetric around its center.
- Left subtree should be a mirror of the right subtree.
- Use recursive DFS (compare left-right pairs) or iterative BFS (queue-based level order check).
- Handle empty tree (symmetric), single node (symmetric), and asymmetric structures.

- **Code:**

```
class Solution {
public:
   bool isSymmetric(TreeNode* root) {
      auto dfs = [&](this auto&& dfs, TreeNode* root1, TreeNode* root2) -> bool {
         if (root1 == root2) {
            return true;
         }
         if (!root1 || !root2 || root1->val != root2->val) {
            return false;
         }
         return dfs(root1->left, root2->right) && dfs(root1->right, root2->left);
      };
      return dfs(root->left, root->right);
   }
};
```

- **Aim:** **<u>Binary Tree Level Order Traversal</u>**

- **Objective:**
  - Traverse a binary tree level by level (left to right).
  - Use BFS (Queue-based traversal) to visit nodes level-wise.
  - Return a list of lists, where each list contains nodes at that level.
  - Handle empty tree (return []) and single-node tree ([[root]]).

- **Code:**

```
class Solution {
public:
   vector<vector<int>> levelOrder(TreeNode* root) {
      vector<vector<int>> ans;
      if (!root) return ans;
      queue<TreeNode*> q{{root}};
      while (!q.empty()) {
         vector<int> t;
         for (int n = q.size(); n; --n) {
            auto node = q.front();
            q.pop();
            t.push_back(node->val);
            if (node->left) {
               q.push(node->left);
```

```
}
            if (node->right) {
                q.push(node->right);
            }
        }
        ans.push_back(t);
    }
    return ans;
  }
};
```

- **Aim:** **<u>Convert Sorted Array to Binary Search Tree</u>**

- **Objective:**

  - Convert a sorted array into a height-balanced BST.
  - Use recursion with divide and conquer, selecting the middle element as the root.
  - A balanced BST where left and right subtrees have nearly equal nodes.
  - Handle empty array (return null) and single-element array (root with no children).

- **Code:**

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        auto dfs = [&](this auto&& dfs, int l, int r) -> TreeNode* {
            if (l > r) {
                return nullptr;
            }
            int mid = (l + r) >> 1;
            return new TreeNode(nums[mid], dfs(l, mid - 1), dfs(mid + 1, r));
        };
        return dfs(0, nums.size() - 1);
    }
};
```

- **Aim:** **Binary Tree Inorder Traversal**

- **Code:**

```
class Solution {
public:
   vector<int> inorderTraversal(TreeNode* root) {
      vector<int> ans;
      function<void(TreeNode*)> dfs = [&](TreeNode* root) {
         if (!root) {
            return;
         }
         dfs(root->left);
         ans.push_back(root->val);
         dfs(root->right);
      };
      dfs(root);
      return ans;
   }
};
```

- **Aim:** **Binary Tree Zigzag Level Order Traversal**

- **Code:**

```
class Solution {
public:
   vector<vector<int>> zigzagLevelOrder(TreeNode* root)
{
      vector<vector<int>> ans;
      if (!root) {
         return ans;
      }
      queue<TreeNode*> q{{root}};
      int left = 1;
      while (!q.empty()) {
         vector<int> t;
         for (int n = q.size(); n; --n) {
            auto node = q.front();
            q.pop();
            t.emplace_back(node->val);
            if (node->left) {
               q.push(node->left);
            }
```

```
            if (node->right) {
                q.push(node->right);
            }
        }
        if (!left) {
            reverse(t.begin(), t.end());
        }
        ans.emplace_back(t);
        left ^= 1;
    }
    return ans;
  }
};
```

- **Aim:** **Construct Binary Tree from Inorder and Postorder Traversal**

- **Code:**

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int, int> d;
        int n = inorder.size();
        for (int i = 0; i < n; ++i) {
            d[inorder[i]] = i;
        }
        function<TreeNode*(int, int, int)> dfs = [&](int i, int j, int n) -> TreeNode* {
            if (n <= 0) {
                return nullptr;
            }
            int v = postorder[j + n - 1];
            int k = d[v];
            auto l = dfs(i, j, k - i);
            auto r = dfs(k + 1, j + k - i, n - k + i - 1);
            return new TreeNode(v, l, r);
        };
        return dfs(0, 0, n);
    }
};
```

- **Aim:** **<u>Kth Smallest Element in a BST</u>**

- **Code:**

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        stack<TreeNode*> stk;
        while (root || !stk.empty()) {
            if (root) {
                stk.push(root);
                root = root->left;
            } else {
                root = stk.top();
                stk.pop();
                if (--k == 0) return root->val;
                root = root->right;
            }
        }
        return 0;
    }
};
```