# Experiment 6

**Student Name:** Tarun Baliyan **UID:** 22BET10184

**Branch:** IT **Section/Group:** 22BET\_IOT-702/A

**Semester:** 6th **Date of Performance:** 28/02/25

**Subject Name:** Advance Programming-II **Subject Code:** 22ITP-367

**Problem: 1.5.1:** [**Maximum Depth of Binary Tree**](https://leetcode.com/problems/maximum-depth-of-binary-tree/)

**Problem Statement:** Given the root of a binary tree, determine its maximum depth. The maximum depth is defined as the number of nodes along the longest path from the root down to the farthest leaf node.

1. **Objective:** Traverse the tree to evaluate and compare the depth of its left and right subtrees.
2. **Code:**

## class Solution:

## def longestNiceSubstring(self, s: str) -> str:

## # Base case: if the string is empty or has only one character, return ""

## if len(s) < 2:

## return ""

## 

## # Check for invalid characters

## for i, ch in enumerate(s):

## if ch.swapcase() not in s:

## # Split around the invalid character and check both parts

## left = self.longestNiceSubstring(s[:i])

## right = self.longestNiceSubstring(s[i+1:])

## # Return the longer substring

## return left if len(left) >= len(right) else right

## 

## # If all characters are valid, return the entire string

## return s

## 3. Result:

## 

**Problem 1.5.2: Validate Binary Search Tree**

**Problem Statement:** Given the root of a binary tree, determine if it is a valid Binary Search Tree (BST). A valid BST satisfies the following conditions for every node.

1. **Objective:** Design and implement an algorithm that efficiently verifies whether a binary tree adheres to BST properties by ensuring every node's value falls within valid lower and upper bounds.
2. **Code:**

class Solution {

public:

bool isValidBST(TreeNode\* root) {

return validate(root, nullptr, nullptr);

}

private:

bool validate(TreeNode\* node, TreeNode\* lower, TreeNode\* upper) {

if (!node) return true;

// Ensure the current node's value is greater than the lower bound (if it exists)

if (lower && node->val <= lower->val) return false;

// Ensure the current node's value is less than the upper bound (if it exists)

if (upper && node->val >= upper->val) return false;

// Recursively validate the left and right subtrees with updated bounds

return validate(node->left, lower, node) && validate(node->right, node, upper);

}

};

1. **Result:**

## 

**Problem 1.4.3: Kth Largest Element in an Array**

**Problem 1.5.3: Symmetric Tree**

**Problem Statement:** Given the root of a binary tree, determine if the tree is symmetric around its center. In other words, check if the tree is a mirror of itself.

1. **Objective:** Recursively or iteratively compare the left and right subtrees to verify mirror symmetry.
2. **Code:**

class Solution {

public:

bool isSymmetric(TreeNode\* root) {

if (!root) return true;

return isMirror(root->left, root->right);

}

private:

bool isMirror(TreeNode\* left, TreeNode\* right) {

// Both subtrees are empty: symmetric

if (!left && !right) return true;

// Only one subtree is empty: not symmetric

if (!left || !right) return false;

// The current nodes must be equal, and the left subtree of left must mirror the right subtree of right, and vice versa

return (left->val == right->val) &&

isMirror(left->left, right->right) &&

isMirror(left->right, right->left);

}

};

1. **Result:**

## 

**Problem 1.3.4:** [**Binary Tree Inorder Traversal**](https://leetcode.com/problems/binary-tree-inorder-traversal/)

**Problem Statement:** Given the root of a binary tree, return its inorder traversal. In an inorder traversal, the nodes are recursively visited in the following order.

1. **Objective:** Develop a solution that efficiently performs an inorder traversal of a binary tree, returning the values of the nodes in the correct order.
2. **Code:**

class Solution

{

public:

vector<int> inorder;

vector<int> inorderTraversal(TreeNode\* root)

{

if (root!=nullptr)

{

inorderTraversal(root->left);

inorder.push\_back(root->val);

inorderTraversal(root->right);

}

return inorder;

}

};

1. **Result:**

## 

**Problem 1.4.5: Kth Smallest element in a BST**

**Problem Statement:** Given the root of a Binary Search Tree (BST) and an integer k, return the kth smallest element in the BST. In a BST, an inorder traversal produces elements in sorted order.

1. **Objective:** Develop an efficient solution that leverages the properties of a BST to find the kth smallest element by performing an inorder traversal.
2. **Code:**

#include <stack>

using namespace std;

class Solution {

public:

int kthSmallest(TreeNode\* root, int k) {

stack<TreeNode\*> st;

while (true) {

// Traverse to the leftmost node

while (root != nullptr) {

st.push(root);

root = root->left;

}

// Process the node on top of the stack

root = st.top();

st.pop();

k--;

// If k reaches 0, we've found the kth smallest element

if (k == 0)

return root->val;

// Move to the right subtree

root = root->right;

}

}

};

1. **Result:**

## 