CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

# Experiment 3

**Student Name: Karan Goyal** **UID:-22BCS15864**

**Branch: BE-CSE** **Group-Ntpp_IOT_602-A**

**Semester: 5TH** **Dateof Performance:23-1-25**

**Subject Name:-Advance Programming lab-2** **Subject Code:-22CSP-351**

**Aim**:- Given the root of a binary tree, return *its maximum depth*.A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Objective**:- The objective is to determine the maximum depth of a binary tree, which represents the longest path from the root to the deepest leaf node. This helps in understanding the tree's structure, height, and balance, which are essential in various applications like searching, sorting, and optimizing tree-based algorithms.

## Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

**Algorithm for the Two Sum Problem**:

1. **Check Base Condition** – If the root is nullptr, return 0 (empty tree has depth 0).
2. **Recursively Compute Left Depth** – Call maxDepth(root->left) to compute the depth of the left subtree.
3. **Recursively Compute Right Depth** – Call maxDepth(root->right) to compute the depth of the right subtree.
4. **Compare Depths** – Take the maximum of the left and right subtree depths.
5. **Increment Depth** – Add 1 to include the current root node in the depth count.
6. **Return Result** – Return the computed depth value.

**Code:**

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

**Time Complexity:** O(N) – Each node is visited once.

**Space Complexity:** O(N) – Worst case (skewed tree), O(log N) – Best case (balanced tree).

**Output-** All the test cases passed

Testcase | Test Result

• Case 1 • Case 2

Input

root =
[3,9,20,null,null,15,7]

Output

3

Expected

3

Testcase | Test Result

Input

root =
[1,null,2]

Output

2

Expected

2

## Problem-2

**Aim:-** Given the root of a binary tree, return *its maximum depth*. A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Objective-**The objective is to determine whether a given binary tree is a valid binary search tree (BST). A valid BST ensures that for each node, its left subtree contains only smaller values, and its right subtree contains only larger values. This validation is crucial for efficient searching, sorting, and maintaining ordered data structures.

### Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

**Algorithm to Check if a Binary Tree is a Valid BST**

1. **Initialize Function** – Call valid(root, LONG_MIN, LONG_MAX) to start validation.
2. **Check Base Condition** – If the current node is nullptr, return true (empty subtree is valid).
3. **Validate Node Value** – Ensure the node's value is within the given range (minimum < node->val < maximum).
4. **Return False if Invalid** – If the node violates the BST property, return false.
5. **Recursively Check Left Subtree** – Call valid(node->left, minimum, node->val).
6. **Recursively Check Right Subtree** – Call valid(node->right, node->val, maximum) and return the combined result.

**Code-**

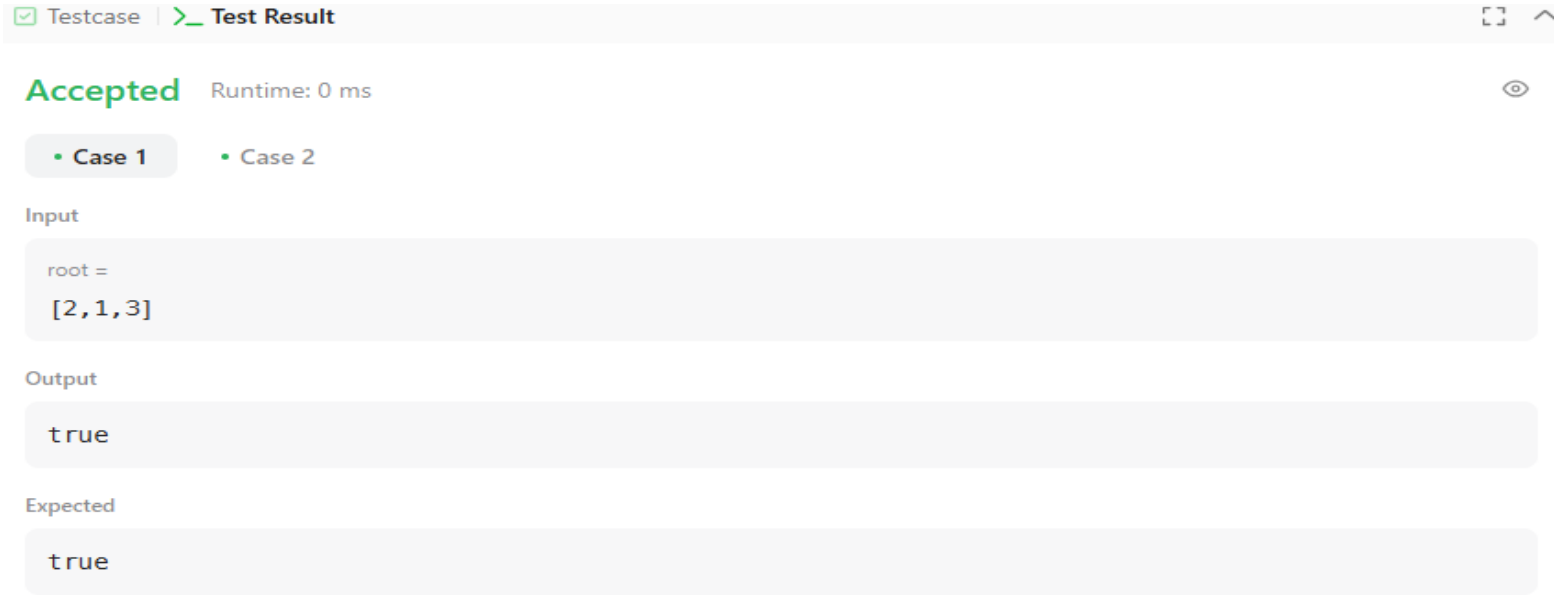
```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return valid(root, LONG_MIN, LONG_MAX);
    }

private:
    bool valid(TreeNode* node, long minimum, long maximum) {
        if (!node) return true;
        if (!(node->val > minimum && node->val < maximum)) return false;
        return valid(node->left, minimum, node->val) && valid(node->right, node->val, maximum);
    }
};
```

**Time Complexity: O(N) – Each node is visited once.**

**Space Complexity: O(N) – Worst case (skewed tree), O(log N) – Best case (balanced tree).**

**Result**-All test cases passes




## Problem-3

**Aim-**Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

**Objective-** The objective is to determine whether a given binary tree is symmetric around its center. This involves checking if the left and right subtrees are mirror images of each other. The solution should efficiently compare corresponding nodes using recursion or iteration to verify structural and value-based symmetry.

### Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

**Algorithm to Check if a Binary Tree is Symmetric**

1. Base Case Check: If the root is nullptr, return true (an empty tree is symmetric).
2. Call Helper Function: Use a helper function isMirror() to check if the left and right subtrees are mirror images.
3. Check Null Nodes: If both nodes are nullptr, return true. If only one is nullptr, return false.
4. Compare Values: If the values of the two nodes do not match, return false.
5. Recursive Check: Recursively check if the left subtree of one tree matches the right subtree of the other and vice versa.
6. Return Result: Return the final result after all recursive comparisons.

**Code-**

```
class Solution {

public:

   bool isSymmetric(TreeNode* root) {

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

private:    bool isMirror(TreeNode* n1, TreeNode* n2) {

      if (n1 == nullptr && n2 == nullptr) {

         return true; }

            if (n1 == nullptr || n2 == nullptr) {

         return false; }

            return n1->val == n2->val && isMirror(n1->left, n2->right) && isMirror(n1->right, n2->left);

   }};
```

**Result-** All test cases passes



**Learning Outcomes**:

1. **Understanding Tree Symmetry:** Learned how to check if a binary tree is symmetric by comparing left and right subtrees recursively.
2. **Validating Binary Search Trees:** Gained knowledge of verifying whether a binary tree follows BST properties using recursion and value constraints.
3. **Computing Tree Depth:** Explored recursive depth calculation to determine the height of a binary tree efficiently.
4. **Recursive Problem-Solving:** Developed skills in solving tree-related problems using recursion and understanding base cases.
5. **Handling Edge Cases:** Improved understanding of handling nullptr scenarios to ensure robustness in tree traversal algorithms.