CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

Experiment 5

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

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

Semester: 5TH **Dateof Performance:-12-2-25**

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

Problem 1

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 given binary tree by finding the longest path from the root to the farthest leaf node. This helps in understanding the tree's structure, balancing, and traversal depth, which are essential for various applications in computer science and algorithm design.

Apparatus Used:

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

Algorithm for the Two Sum Problem:

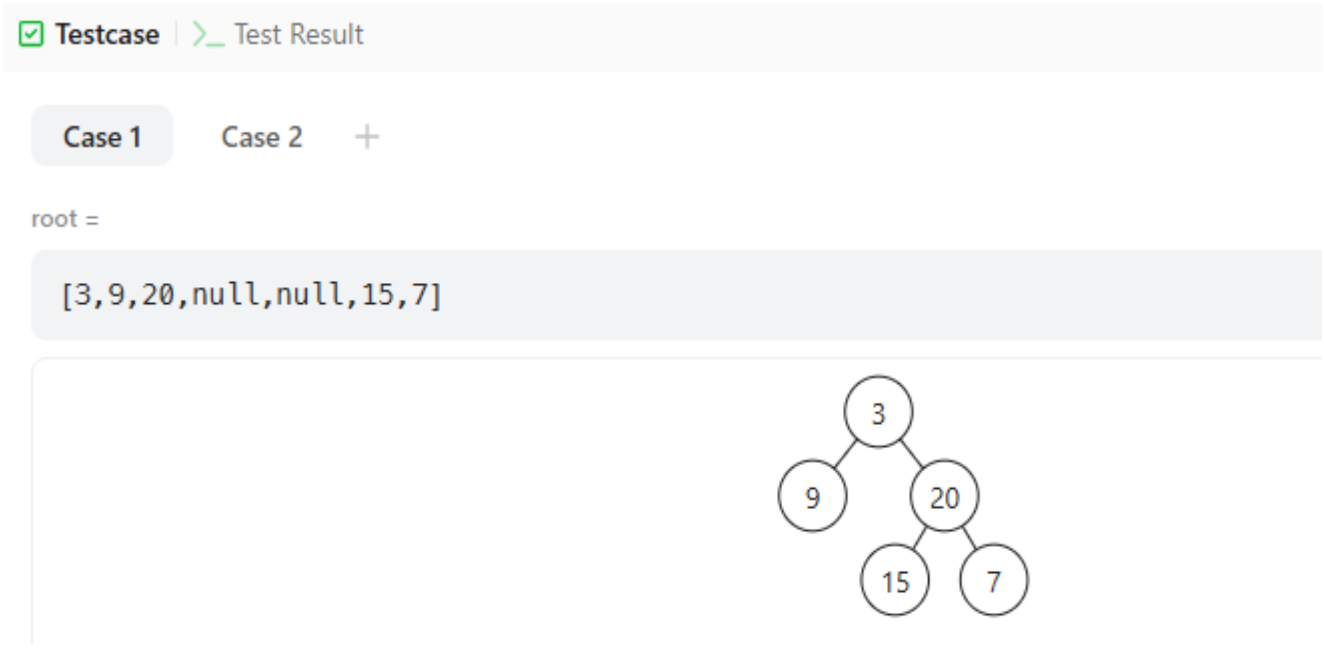
1. **Check Base Condition** – If the root is nullptr, return 0 (empty tree has depth 0).
2. **Recursively Compute Left Depth** – Call maxDepth(root->left) to compute the depth of the left subtree.
3. **Recursively Compute Right Depth** – Call maxDepth(root->right) to compute the depth of the right subtree.
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.

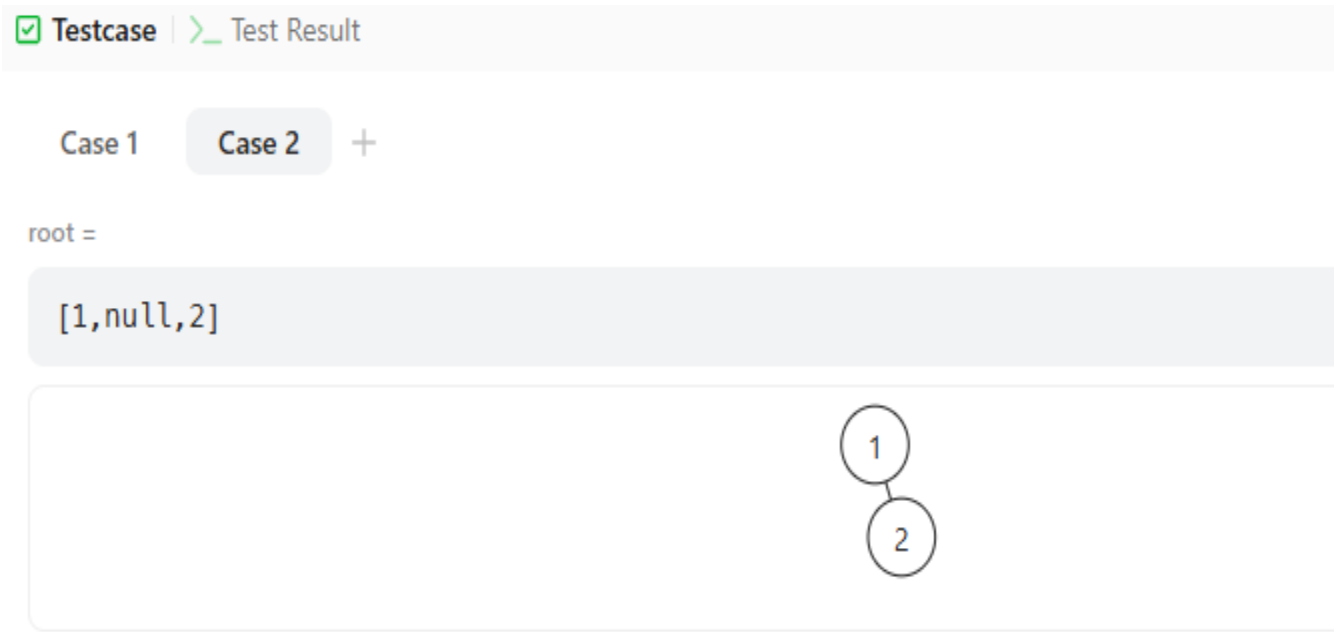
Code:

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

- **Time complexity**: $O(n)$
- **Space complexity:** $O(H)$

Output- All the test cases passed





Learning Outcomes:-

1. Understand recursion in computing the depth of a binary tree.
2. Learn how to traverse a tree using depth-first search (DFS).
3. Analyze time and space complexity of recursive tree algorithms.
4. Gain insight into balanced vs. skewed tree depth behavior.
5. Apply the concept to real-world hierarchical data structures.

Problem-2

Aim:- Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

The left subtree of a node contains only nodes with keys **less than** the node's key.

- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Objective- The objective is to determine whether a given binary tree satisfies the properties of a Binary Search Tree (BST). This involves verifying that all left subtree nodes are smaller, all right subtree nodes are greater, and both subtrees are valid BSTs. Ensuring BST validity is crucial for efficient searching and sorting operations.

Apparatus Used:

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

Algorithm

1. **Define Helper Function** – Use a recursive function with range limits (min, max).
2. **Check Base Case** – If the node is nullptr, return true (empty tree is valid).
3. **Validate Node** – Ensure the node's value is within the (min, max) range.
4. **Recursive Check (Left)** – Recursively validate the left subtree with an updated max limit.
5. **Recursive Check (Right)** – Recursively validate the right subtree with an updated min limit.
6. **Return Result** – Return true if both left and right subtrees are valid BSTs.

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).
- **Space Complexity:** O(H)

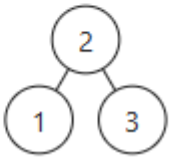
Result-All test cases passes

Testcase | Test Result

Case 1 Case 2 +

root =

```
[2,1,3]
```

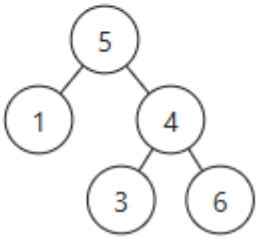


Testcase | Test Result

Case 1 Case 2 +

root =

```
[5,1,4,null,null,3,6]
```



Learing Outcomes:-

1. Understand the properties and structure of a Binary Search Tree (BST).
2. Learn how to validate a BST using recursive depth-first traversal with range constraints.
3. Analyze the time and space complexity of recursive BST validation algorithms.
4. Explore edge cases such as duplicate values, skewed trees, and empty trees.
5. Apply BST validation concepts in real-world applications like database indexing and searching.

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- Given the root of a binary tree, determine if it is symmetric around its center. The tree is symmetric if the left and right subtrees are mirror images. This involves recursively comparing corresponding nodes and their subtrees.

Apparatus Used:

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

Algorithm

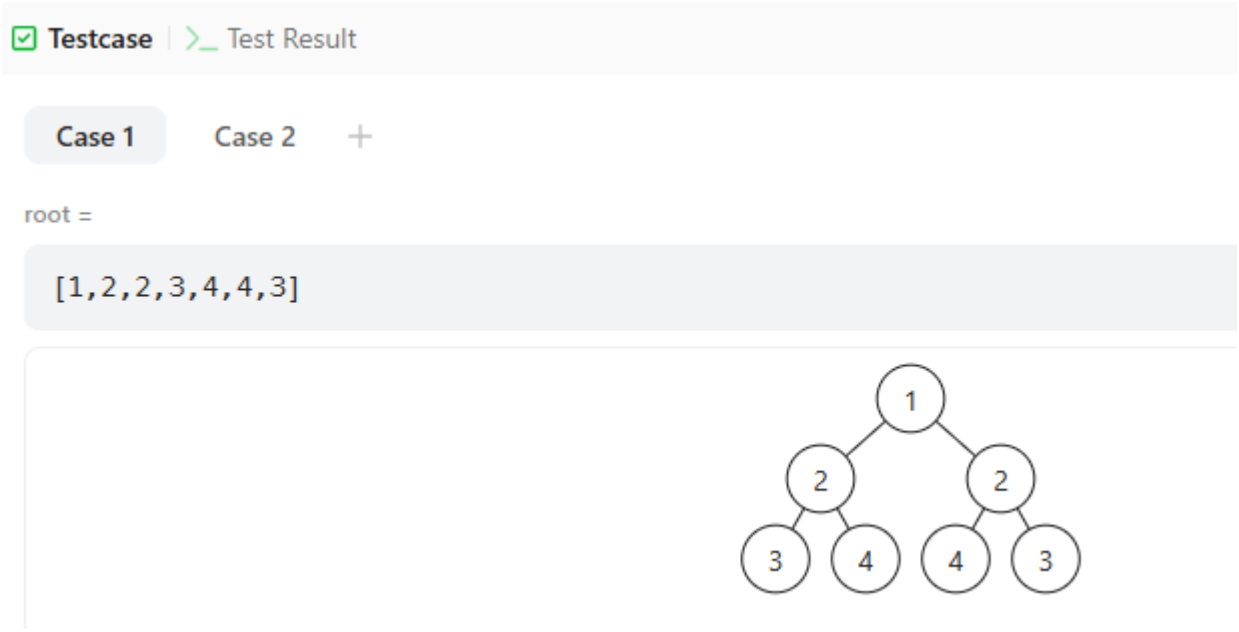
1. Base Check: If the tree is empty, return true.
2. Call Helper Function: Compare the left and right subtrees using isMirror().
3. Base Case in isMirror(): If both nodes are nullptr, return true.
4. Null Check: If only one node is nullptr, return false.
5. Compare Values: Check if the values of both nodes are equal.
6. Recursive Check: Recursively compare n1->left with n2->right and n1->right with n2->left.

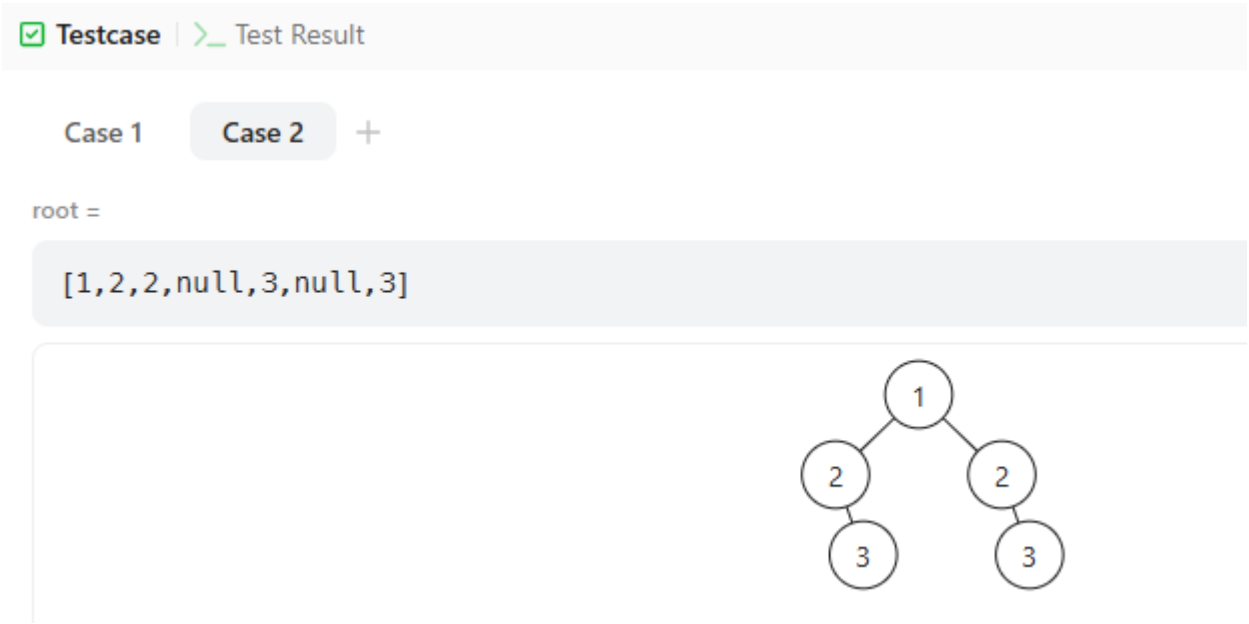
Code-

```
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);
    }};
```

Time Complexity: O(n)
Space Complexity: O(H)

Result- All test cases passes





Learning Outcomes:

1. **Understanding Tree Symmetry:** Learn how to check if a binary tree is symmetric around its center.
2. **Recursive Approach:** Apply recursion to compare corresponding nodes in left and right subtrees.
3. **Base Case Handling:** Understand the importance of handling nullptr cases to avoid errors.
4. **Time and Space Complexity Analysis:** Analyze why the algorithm runs in **O(N)** time and **O(H)** space.
5. **Practical Problem-Solving:** Gain experience in solving tree-based problems using recursion efficiently.