**Student Name:** AMAN

**Branch:** BE-CSE **Semester:** 6th

**Subject Name:** AP LAB-II

# Aim:

**Experiment-6**

**UID:** 22BCS15783

**Section/Group:** 640/B

**Date of Performance:** 21/02/25

**Subject Code:** 22CSP-351

A tree is a hierarchical data structure that consists of nodes connected by edges. Unlike linear data structures like arrays and linked lists, trees are non-linear, making them ideal for representing hierarchical relationships such as file systems, database indexes, and decision-making processes.

# Introduction to the Searching and Sorting:

## Tree Traversal Methods:

Traversal is the process of visiting nodes in a tree.

## Depth-First Search (DFS)

* + - Explores as deep as possible before backtracking.
    - Preorder (Root → Left → Right): Used for copying a tree.
    - Inorder (Left → Root → Right): Used in Binary Search Trees (BSTs) to retrieve sorted values.
    - Postorder (Left → Right → Root): Used for deleting trees (deletes child nodes before the parent).

## Breadth-First Search (BFS)

* + - Also called Level Order Traversal.
    - Explores all nodes at one level before moving to the next.

# Implementation/Code:

## 104 Maximum Depth of Binary Tree:

import java.util.\*; class Solution {

public int maxDepth(TreeNode root) { if (root == null) return 0;

Queue<TreeNode> queue = new LinkedList<>(); queue.add(root);

int depth = 0;

while (!queue.isEmpty()) { int size = queue.size(); depth++;

for (int i = 0; i < size; i++) { TreeNode node = queue.poll();

if (node.left != null) queue.add(node.left);

if (node.right != null) queue.add(node.right);

}

}

return depth;

}

}

## 104Output:

* **98 Validate Binary Search Tree:**

import java.util.Stack; class Solution {

public boolean isValidBST(TreeNode root) { Stack<TreeNode> stack = new Stack<>(); TreeNode current = root;

TreeNode prev = null;

while (!stack.isEmpty() || current != null) { while (current != null) {

stack.push(current); current = current.left;

}

current = stack.pop();

if (prev != null && current.val <= prev.val) { return false;

}

prev = current;

current = current.right;

}

return true;

}

}

## 98 Output:

* **108 Convert Sorted Array to Binary Search Tree:**

class Solution {

public TreeNode sortedArrayToBST(int[] nums) { return buildBST(nums, 0, nums.length - 1);

}

private TreeNode buildBST(int[] nums, int left, int right) { if (left > right) return null; // Base case

int mid = (left + right) / 2; // Find middle index

TreeNode root = new TreeNode(nums[mid]); // Create root node root.left = buildBST(nums, left, mid - 1); // Left Subtree root.right = buildBST(nums, mid + 1, right); // Right Subtree return root;

}

}

## 108 Output:

* **94 Binary Search Tree Inorder Traversal:**

import java.util.\*;

class Solution {

public List<Integer> inorderTraversal(TreeNode root) { List<Integer> result = new ArrayList<>(); inorder(root, result);

return result;

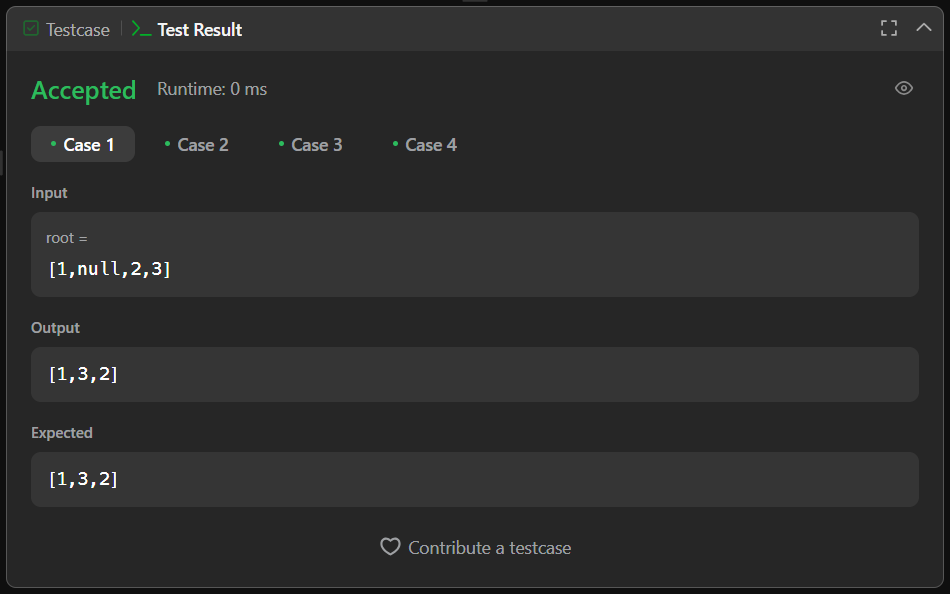
}

private void inorder(TreeNode node, List<Integer> result) { if (node == null) return;

inorder(node.left, result); // Left result.add(node.val); // Root inorder(node.right, result); // Right

}

}

**94 Output:**

# Learning Outcome

* Understand the breaking down the sorted array into smaller parts to build a height- balanced BST.
* Understanding binary search tree beyond simple sorted arrays.
* How to validate if a binary tree is a BST by checking left and right subtrees recursively.