# Assignment 3

## Student Name: Anuj Kumar UID:22BCS11581

## Branch: BE-CSE (General) Section/Group: FL\_IOT-602 A

## Semester:6th Date of Performance: 14-02-25

## Subject Name: Advanced Programming Lab-2 Subject Code: 22CSP-351

# Aim: 94.Binary Tree Inorder Traversal

# Implementation/ Code:

# class Solution {

# public List<Integer> inorderTraversal(TreeNode root) {

# List<Integer> result = new ArrayList<>();

# inorderHelper(root, result);

# return result;

# }

# 

# private void inorderHelper(TreeNode node, List<Integer> result) {

# if (node == null) return;

# inorderHelper(node.left, result);

# result.add(node.val);

# inorderHelper(node.right, result);

# }

# }

# Output:

# 

# Aim: 101.Symmetric Tree

# Implementation/ Code:

# class Solution {

# public boolean isSymmetric(TreeNode root) {

# return isMirror(root, root);

# }

# 

# private boolean isMirror(TreeNode t1, TreeNode t2) {

# if (t1 == null && t2 == null) return true;

# if (t1 == null || t2 == null) return false;

# return t1.val == t2.val && isMirror(t1.left, t2.right) && isMirror(t1.right, t2.left);

# }

# }

# Output:

# 

# Aim: 104.Maximum Depth of Binary Tree

# Implementation/ Code:

# class Solution {

# public int maxDepth(TreeNode root) {

# if(root==null) return 0 ;

# int left = maxDepth(root.left);

# int right = maxDepth(root.right);

# return Math.max(left, right) + 1;

# }

# }

# Output:

# 

# Aim: 98.Validate Binary Search Tree

# Implementation/ Code:

# class Solution {

# public boolean isValidBST(TreeNode root) {

# return isValid(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

# }

# 

# private boolean isValid(TreeNode node, long min, long max) {

# if (node == null) return true;

# if (node.val <= min || node.val >= max) return false;

# return isValid(node.left, min, node.val) && isValid(node.right, node.val, max);

# }

# }

# Output:

# 

# Aim: 230.Kth Smallest Element in a BST

# Implementation/ Code:

# class Solution {

# public int kthSmallest(TreeNode root, int k) {

# List<Integer> result = new ArrayList<>();

# inorder(root, result);

# return result.get(k - 1);

# }

# 

# private void inorder(TreeNode node, List<Integer> result) {

# if (node == null) return;

# inorder(node.left, result);

# result.add(node.val);

# inorder(node.right, result);

# }

# }

# Output:

# 

# Aim: 102. Binary Tree Level Order Traversal

# Implementation/ Code:

# class Solution {

# public List<List<Integer>> levelOrder(TreeNode root) {

# List<List<Integer>> result = new ArrayList<>();

# if (root == null) return result;

# Queue<TreeNode> queue = new LinkedList<>();

# queue.offer(root);

# 

# while (!queue.isEmpty()) {

# int levelSize = queue.size();

# List<Integer> currentLevel = new ArrayList<>();

# for (int i = 0; i < levelSize; i++) {

# TreeNode node = queue.poll();

# currentLevel.add(node.val);

# if (node.left != null) queue.offer(node.left);

# if (node.right != null) queue.offer(node.right);

# }

# result.add(currentLevel);

# }

# return result;

# }

# }

# Output:

# 

# Aim: 107.Binary Tree Level Order Traversal II

# Implementation/ Code:

# class Solution {

# public List<List<Integer>> levelOrderBottom(TreeNode root) {

# List<List<Integer>> result = new ArrayList<>();

# if (root == null) return result;

# Queue<TreeNode> queue = new LinkedList<>();

# queue.offer(root);

# 

# while (!queue.isEmpty()) {

# int levelSize = queue.size();

# List<Integer> currentLevel = new ArrayList<>();

# for (int i = 0; i < levelSize; i++) {

# TreeNode node = queue.poll();

# currentLevel.add(node.val);

# if (node.left != null) queue.offer(node.left);

# if (node.right != null) queue.offer(node.right);

# }

# result.add(currentLevel);

# }

# Collections.reverse(result);

# return result;

# }

# }

# Output:

# 

# Aim: 103.Binary Tree Zigzag Level Order Traversal

# Implementation/ Code:

# class Solution {

# public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

# List<List<Integer>> result = new ArrayList<>();

# if (root == null) return result;

# 

# Queue<TreeNode> queue = new LinkedList<>();

# queue.offer(root);

# boolean leftToRight = true;

# while (!queue.isEmpty()) {

# int levelSize = queue.size();

# List<Integer> currentLevel = new ArrayList<>();

# for (int i = 0; i < levelSize; i++) {

# TreeNode node = queue.poll();

# if (leftToRight) currentLevel.add(node.val);

# else currentLevel.add(0, node.val);

# if (node.left != null) queue.offer(node.left);

# if (node.right != null) queue.offer(node.right);

# }

# result.add(currentLevel);

# leftToRight = !leftToRight;

# }

# return result;

# }

# }

# Output:

# 

# Aim: 199.Binary Tree Right Side View

# Implementation/ Code:

# class Solution {

# public List<Integer> rightSideView(TreeNode root) {

# List<Integer> result = new ArrayList<>();

# if (root == null) return result;

# 

# Queue<TreeNode> queue = new LinkedList<>();

# queue.offer(root);

# while (!queue.isEmpty()) {

# int levelSize = queue.size();

# for (int i = 0; i < levelSize; i++) {

# TreeNode node = queue.poll();

# if (i == levelSize - 1) result.add(node.val);

# if (node.left != null) queue.offer(node.left);

# if (node.right != null) queue.offer(node.right);

# }

# }

# return result;

# }

# }

# Output:

# 

# Aim: 106.Construct Binary Tree from Inorder and Postorder Traversal

# Implementation/ Code:

# class Solution {

# public TreeNode buildTree(int[] inorder, int[] postorder) {

# HashMap<Integer, Integer> rec = new HashMap<>();

# for (int i = 0; i < inorder.length; i++) {

# rec.put(inorder[i], i);

# }

# return helper(inorder, postorder, 0, inorder.length - 1, 0, postorder.length - 1, rec);

# }

# private TreeNode helper(int[] inorder, int[] postorder,

# int inStart, int inEnd,

# int postStart, int postEnd,

# HashMap<Integer, Integer> rec) {

# if (inStart > inEnd || postStart > postEnd) return null;

# int val = postorder[postEnd];

# TreeNode root = new TreeNode(val);

# int idx = rec.get(val);

# int leftSubtreeSize = idx - inStart;

# root.left = helper(inorder, postorder,

# inStart, idx - 1,

# postStart, postStart + leftSubtreeSize - 1,

# rec);

# root.right = helper(inorder, postorder,

# idx + 1, inEnd,

# postStart + leftSubtreeSize, postEnd - 1,

# rec);

# return root;

# }

# }

# Output:

# 

# Aim: 513.Find Bottom Left Tree Value

# Implementation/ Code:

# public class Solution {

# public int findBottomLeftValue(TreeNode root) {

# int last = 0;

# Queue<TreeNode> q = new LinkedList<>();

# q.add(root);

# while (!q.isEmpty()) {

# int count = q.size();

# for (int i = 0; i < count; i++) {

# TreeNode curr = q.poll();

# if (i == 0)

# last = curr.val; // last leftMost val

# if (curr.left != null)

# q.add(curr.left);

# if (curr.right != null)

# q.add(curr.right);

# }

# }

# return last;

# }

# }

# Output:

# 

# Aim: 124. Binary Tree Maximum Path Sum

# Implementation/ Code:

# class TreeNode {

# int val;

# TreeNode left, right;

# TreeNode(int val) { this.val = val; }

# }

# class Solution {

# private int ans = Integer.MIN\_VALUE;

# public int maxPathSum(TreeNode root) {

# helper(root);

# return ans;

# }

# private int helper(TreeNode root) {

# if (root == null) return 0;

# int left = Math.max(0, helper(root.left));

# int right = Math.max(0, helper(root.right));

# ans = Math.max(ans, root.val + left + right);

# return root.val + Math.max(left, right);

# }

# }

# Output:

# 

# Aim: 987.Vertical Order Traversal of a Binary Tree

# Implementation/ Code:

# class Solution {

# Map<Integer, TreeMap<Integer, PriorityQueue<Integer>>> map;

# public List<List<Integer>> verticalTraversal(TreeNode root) {

# if (root == null)

# return null;

# map = new TreeMap<>();

# dfs(root, 0, 0);

# List<List<Integer>> res = new LinkedList<>();

# for (int key : map.keySet()){

# List<Integer> list = new LinkedList<>();

# TreeMap<Integer, PriorityQueue<Integer>> tm = map.get(key);

# for (int k : tm.keySet()){

# PriorityQueue<Integer> pq = tm.get(k);

# while (!pq.isEmpty()){

# list.add(pq.poll());

# }

# }

# res.add(list);

# }

# return res;

# }

# private void dfs(TreeNode root, int index, int level){

# if (root == null)

# return;

# map.putIfAbsent(index, new TreeMap<>());

# map.get(index).putIfAbsent(level, new PriorityQueue<>());

# map.get(index).get(level).add(root.val);

# dfs(root.left, index - 1, level + 1);

# dfs(root.right, index + 1, level + 1);

# }

# }

# Output:

# 