1 > Top View of tree

Code :

import java.util.\*;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

public List<Integer> topView(TreeNode root) {

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

if (root == null)

return result;

Queue<Pair<TreeNode, Integer>> queue = new LinkedList<>();

Map<Integer, Integer> verticalLevels = new TreeMap<>();

queue.offer(new Pair<>(root, 0));

while (!queue.isEmpty()) {

Pair<TreeNode, Integer> current = queue.poll();

TreeNode node = current.getKey();

int level = current.getValue();

if (!verticalLevels.containsKey(level))

verticalLevels.put(level, node.val);

if (node.left != null)

queue.offer(new Pair<>(node.left, level - 1));

if (node.right != null)

queue.offer(new Pair<>(node.right, level + 1));

}

for (int val : verticalLevels.values())

result.add(val);

return result;

}

}

2 > Bottom View of tree

Code :

import java.util.\*;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

public List<Integer> bottomView(TreeNode root) {

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

if (root == null)

return result;

Queue<Pair<TreeNode, Integer>> queue = new LinkedList<>();

Map<Integer, Integer> verticalLevels = new TreeMap<>();

queue.offer(new Pair<>(root, 0));

while (!queue.isEmpty()) {

Pair<TreeNode, Integer> current = queue.poll();

TreeNode node = current.getKey();

int level = current.getValue();

verticalLevels.put(level, node.val);

if (node.left != null)

queue.offer(new Pair<>(node.left, level - 1));

if (node.right != null)

queue.offer(new Pair<>(node.right, level + 1));

}

for (int val : verticalLevels.values())

result.add(val);

return result;

}

}

3 > Diagonal View

Code :

import java.util.\*;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

public List<Integer> diagonalView(TreeNode root) {

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

if (root == null)

return result;

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

Map<TreeNode, Integer> diagonalLevels = new HashMap<>();

queue.offer(root);

diagonalLevels.put(root, 0);

while (!queue.isEmpty()) {

TreeNode node = queue.poll();

int diagonalLevel = diagonalLevels.get(node);

while (node != null) {

if (!diagonalLevels.containsKey(node))

diagonalLevels.put(node, diagonalLevel);

if (node.left != null) {

queue.offer(node.left);

diagonalLevels.put(node.left, diagonalLevel + 1);

}

node = node.right;

}

}

for (Map.Entry<TreeNode, Integer> entry : diagonalLevels.entrySet()) {

result.add(entry.getKey().val);

}

return result;

}

}

4 > Boundary Traversal

Code :

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

public List<Integer> boundaryTraversal(TreeNode root) {

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

if (root == null)

return result;

result.add(root.val);

addLeftBoundary(root.left, result);

addLeaves(root, result);

addRightBoundary(root.right, result);

return result;

}

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

if (node == null || (node.left == null && node.right == null))

return;

result.add(node.val);

if (node.left != null)

addLeftBoundary(node.left, result);

else

addLeftBoundary(node.right, result);

}

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

if (node == null)

return;

if (node.left == null && node.right == null)

result.add(node.val);

addLeaves(node.left, result);

addLeaves(node.right, result);

}

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

if (node == null || (node.left == null && node.right == null))

return;

if (node.right != null)

addRightBoundary(node.right, result);

else

addRightBoundary(node.left, result);

result.add(node.val); }}

5 > Width of a tree

Code :

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

public int widthOfTree(TreeNode root) {

if (root == null)

return 0;

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

Queue<Integer> levelQueue = new LinkedList<>();

queue.offer(root);

levelQueue.offer(0);

int maxWidth = 1;

while (!queue.isEmpty()) {

int levelSize = queue.size();

int leftmostIndex = 0;

int rightmostIndex = 0;

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

TreeNode node = queue.poll();

int currentIndex = levelQueue.poll();

if (i == 0)

leftmostIndex = currentIndex;

if (i == levelSize - 1)

rightmostIndex = currentIndex;

if (node.left != null) {

queue.offer(node.left);

levelQueue.offer(2 \* currentIndex);

}

if (node.right != null) {

queue.offer(node.right);

levelQueue.offer(2 \* currentIndex + 1);

}

}

int currentWidth = rightmostIndex - leftmostIndex + 1;

maxWidth = Math.max(maxWidth, currentWidth);

}

return maxWidth;

}

}

6 > Diameter of a tree

Code :

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

public int diameterOfTree(TreeNode root) {

if (root == null)

return 0;

int leftHeight = height(root.left);

int rightHeight = height(root.right);

int leftDiameter = diameterOfTree(root.left);

int rightDiameter = diameterOfTree(root.right);

return Math.max(leftHeight + rightHeight + 1, Math.max(leftDiameter, rightDiameter));

}

private int height(TreeNode root) {

if (root == null)

return 0;

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

}

}

7 > longest path b/w 2 leaf nodes

Code :

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

private int maxPath;

public int longestPathBetweenLeaves(TreeNode root) {

maxPath = 0;

findLongestPath(root);

return maxPath;

}

private int findLongestPath(TreeNode node) {

if (node == null)

return 0;

int leftHeight = findLongestPath(node.left);

int rightHeight = findLongestPath(node.right);

int currentPath = 1 + leftHeight + rightHeight;

maxPath = Math.max(maxPath, currentPath);

return 1 + Math.max(leftHeight, rightHeight);

}

}

8 > Check for Balance Tree

Code :

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

public boolean isBalanced(TreeNode root) {

return checkBalance(root) != -1;

}

private int checkBalance(TreeNode node) {

if (node == null)

return 0;

int leftHeight = checkBalance(node.left);

if (leftHeight == -1)

return -1;

int rightHeight = checkBalance(node.right);

if (rightHeight == -1)

return -1;

if (Math.abs(leftHeight - rightHeight) > 1)

return -1;

return 1 + Math.max(leftHeight, rightHeight);

}

}

9 >Construct Binary Tree from given inorder & postorder

Code :

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

left = null;

right = null;

}

}

public class BinaryTree {

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

if (inorder == null || postorder == null || inorder.length == 0 || postorder.length == 0)

return null;

Map<Integer, Integer> inorderMap = new HashMap<>();

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

inorderMap.put(inorder[i], i);

}

int postIndex = postorder.length - 1;

return buildTreeHelper(inorder, postorder, 0, inorder.length - 1, postIndex, inorderMap);

}

private TreeNode buildTreeHelper(int[] inorder, int[] postorder, int inStart, int inEnd, int postIndex,

Map<Integer, Integer> inorderMap) {

if (inStart > inEnd || postIndex < 0)

return null;

int rootVal = postorder[postIndex];

TreeNode root = new TreeNode(rootVal);

int rootIndexInorder = inorderMap.get(rootVal);

root.right = buildTreeHelper(inorder, postorder, rootIndexInorder + 1, inEnd, postIndex - 1, inorderMap);

root.left = buildTreeHelper(inorder, postorder, inStart, rootIndexInorder - 1, postIndex - (inEnd - rootIndexInorder) - 1, inorderMap);

return root;

}}