Q1:

class TreeNode {

public:

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

class Solution {

public:

int minDepth(TreeNode\* root) {

if (root == nullptr) {

return 0;

}

std::queue<std::pair<TreeNode\*, int>> q;

q.push({root, 1});

while (!q.empty()) {

int size = q.size();

while (size--) {

auto current = q.front();

q.pop();

TreeNode\* node = current.first;

int level = current.second;

if (node->left == nullptr && node->right == nullptr) {

return level;

}

level++;

if (node->left) {

q.push({node->left, level});

}

if (node->right) {

q.push({node->right, level});

}

}

}

return -1; // This line is reached if the tree is somehow empty

}

};

Q2:

class Solution {

private List<Integer> values = new ArrayList<>();

public int countNodes(TreeNode root) {

traverse(root);

return values.size();

}

private void traverse(TreeNode node) {

if (node == null) {

return;

}

values.add(node.val);

traverse(node.left);

traverse(node.right);

}

}

Q3:

public class Solution {

public List<Integer> largestValues(TreeNode root) {

if (root == null) {

return new ArrayList<>();

}

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

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

queue.offer(root);

while (!queue.isEmpty()) {

int currentLevelSize = queue.size();

int maxVal = Integer.MIN\_VALUE;

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

TreeNode node = queue.poll();

maxVal = Math.max(maxVal, node.val);

if (node.left != null) {

queue.offer(node.left);

}

if (node.right != null) {

queue.offer(node.right);

}

}

result.add(maxVal);

}

return result;

}

}

Q4:

public class Solution {

class LeafIterator implements Iterator<Integer> {

private ArrayDeque<TreeNode> stack = new ArrayDeque<>();

LeafIterator(TreeNode head) {

stack.addFirst(head);

}

public boolean hasNext() {

while (!stack.isEmpty()) {

TreeNode current = stack.peekFirst();

if (current.left == null && current.right == null) {

return true;

} else if (current.left != null) {

TreeNode next = current.left;

current.left = null;

if (current.right == null) {

stack.removeFirst();

}

stack.addFirst(next);

} else {

current = current.right;

stack.removeFirst();

stack.addFirst(current);

}

}

return false;

}

public Integer next() {

if (!hasNext()) {

throw new NoSuchElementException();

}

return stack.removeFirst().val;

}

}

public boolean leafSimilar(TreeNode root1, TreeNode root2) {

LeafIterator iterator1 = new LeafIterator(root1);

LeafIterator iterator2 = new LeafIterator(root2);

while (iterator1.hasNext() && iterator2.hasNext()) {

if (!iterator1.next().equals(iterator2.next())) {

return false;

}

}

return !iterator1.hasNext() && !iterator2.hasNext();

}

}

Q5:

public class Solution {

public int deepestLeavesSum(TreeNode root) {

int sum = 0;

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

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

sum = 0;

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

TreeNode node = queue.poll();

sum += node.val;

if (node.right != null) {

queue.add(node.right);

}

if (node.left != null) {

queue.add(node.left);

}

}

}

return sum;

}

}