# Problem 1

## Aim:

Maximum Depth of Binary Tree

## Code:

```
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }
}
```

## Output:


Case 1


Case 2

# Problem 2

## Aim:

Validate Binary Search Tree

## Code:

```
class Solution {
    private long minVal = Long.MIN_VALUE;
    public boolean isValidBST(TreeNode root) {
        if (root == null) return true;
        if (!isValidBST(root.left)) return false;

        if (minVal >= root.val) return false;

        minVal = root.val;

        if (!isValidBST(root.right)) return false;

        return true;
    }
}
```

## Output:


Test Case 1


Test Case 2

# Problem 3

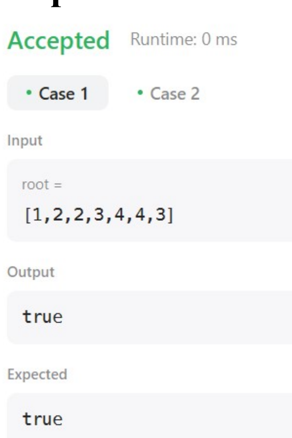
## Aim:

Symmetric Tree

## Code:

```java
class Solution {
    public boolean isSymmetric(TreeNode root) {
        return isMirror(root.left, root.right);
    }

    private boolean isMirror(TreeNode n1, TreeNode n2) {
        if (n1 == null && n2 == null) {
            return true;
        }

        if (n1 == null || n2 == null) {
            return false;
        }

        return n1.val == n2.val && isMirror(n1.left, n2.right) && isMirror(n1.right, n2.left);
    }
}
```

## Output:




Case 1


Case 2

# Problem 4

## Aim:
Binary Tree Level Order Traversal

## Code:
```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root)
    {
        List<List<Integer>>al=new ArrayList<>();
        pre(root,0,al);
        return al;
    }
    public static void pre(TreeNode root,int l,List<List<Integer>>al)
    {
        if(root==null)
            return;
        if(al.size()==l)
        {
            List<Integer>li=new ArrayList<>();
            li.add(root.val);
            al.add(li);
        }
        else
            al.get(l).add(root.val);
        pre(root.left,l+1,al);
        pre(root.right,l+1,al);
    }
}
```

## Output:




Case 1 Case 2 Case 3

# Problem 5

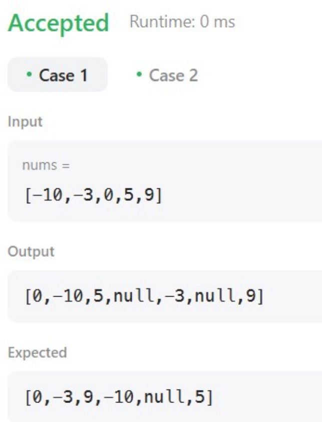
## Aim:

Convert Sorted Array to Binary Search Tree

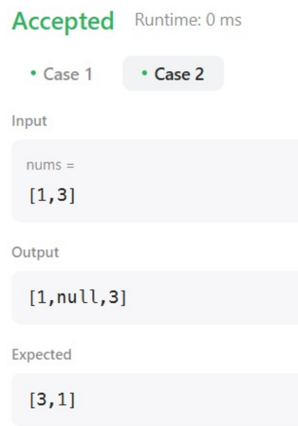
## Code:

```
// Definition for a binary tree node.
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return helper(nums, 0, nums.length - 1);
    }

    private TreeNode helper(int[] nums, int left, int right) {
        if (left > right) return null;
        int mid = (left + right) / 2;
        TreeNode root = new TreeNode(nums[mid]);
        root.left = helper(nums, left, mid - 1);
        root.right = helper(nums, mid + 1, right);
        return root;
    }
}
```

## Output:




Case 1




Case 2

# Problem 6

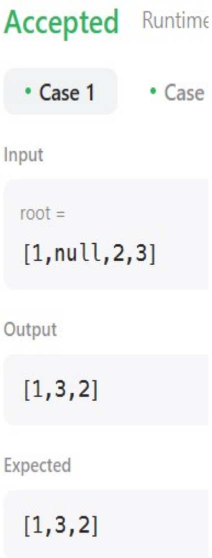
## Aim:

Binary Tree Inorder Traversal

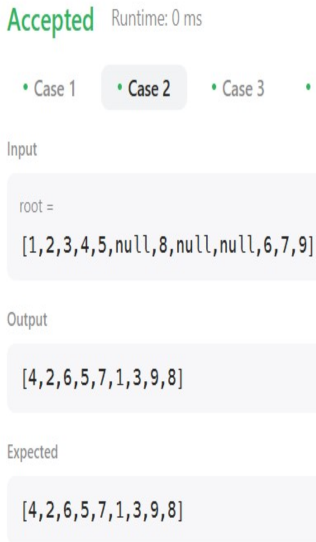
## Code:

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();

        inorder(root, res);
        return res;
    }

    private void inorder(TreeNode node, List<Integer> res) {
        if (node == null) {
            return;
        }
        inorder(node.left, res);
        res.add(node.val);
        inorder(node.right, res);
    }
}
```

## Output:



Case 1



Case 2

Case 3

Case 4

# Problem 7

## Aim:

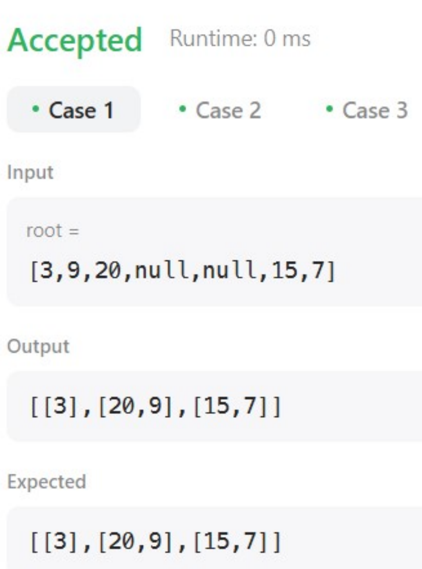
Binary Zigzag Level Order Traversal

## Code:

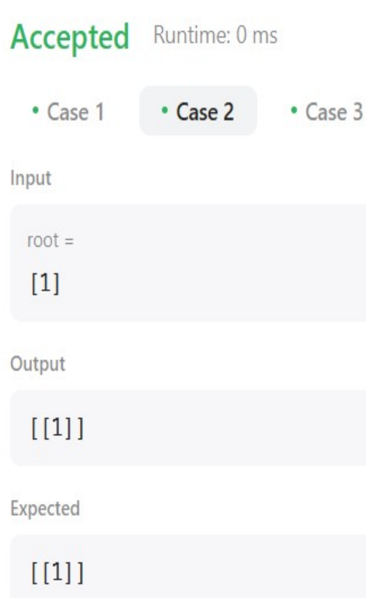
```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        if(root == null)return new ArrayList<>();
```

```
        ArrayDeque<TreeNode> dq = new ArrayDeque<>();
        dq.offer(root);
        List<List<Integer>> result = new ArrayList<>();
        boolean leftToRight = true;

        while(!dq.isEmpty()){
            List<Integer> currLevel = new ArrayList<>();
            for(int i = dq.size(); i > 0; i--){
                TreeNode curr = (leftToRight)?dq.pollFirst():dq.pollLast();
                currLevel.add(curr.val);
                if(leftToRight){
                    if(curr.left != null)
                        dq.offerLast(curr.left);
                    if(curr.right != null)
                        dq.offerLast(curr.right);
                }
                else{
                    if(curr.right != null)
                        dq.offerFirst(curr.right);
                    if(curr.left != null)
                        dq.offerFirst(curr.left);
                }
            }

            leftToRight = !leftToRight;
            result.add(currLevel);
        }
        return result;
    }
}
```

# Output:




Case 1




Case 2


Case 3

# Problem 8

## Aim:

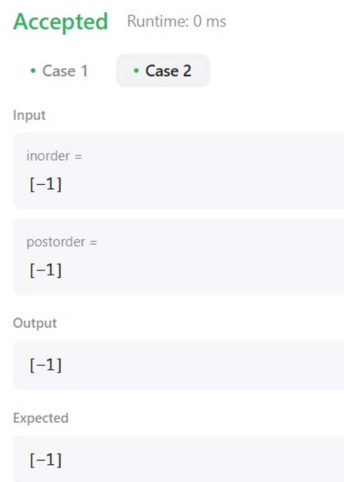
Construct Binary Tree from Inorder and Postorder Traversal

## Code:

```
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int, int> index;
        for (int i = 0; i < inorder.size(); i++) {
            index[inorder[i]] = i;
        }
        return buildTreeHelper(inorder, postorder, 0, inorder.size() - 1, 0, postorder.size() - 1, index);
    }

    TreeNode* buildTreeHelper(vector<int>& inorder, vector<int>& postorder, int inorderStart, int inorderEnd, int
postorderStart, int postorderEnd, unordered_map<int, int>& index) {
        if (inorderStart > inorderEnd || postorderStart > postorderEnd) {
            return nullptr;
        }
        int rootVal = postorder[postorderEnd];
        TreeNode* root = new TreeNode(rootVal);
        int inorderRootIndex = index[rootVal];
        int leftSubtreeSize = inorderRootIndex - inorderStart;
```

```
        root->left = buildTreeHelper(inorder, postorder, inorderStart, inorderRootIndex - 1, postorderStart, postorderStart + leftSubtreeSize - 1, index);
        root->right = buildTreeHelper(inorder, postorder, inorderRootIndex + 1, inorderEnd, postorderStart + leftSubtreeSize, postorderEnd - 1, index);
        return root;
    }
};
```

## Output:


Case 1 Case 2

# Problem 9

## Aim:

Kth Smallest element in a BST

## Code:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private int count = 0; // Counter for visited nodes

    public int kthSmallest(TreeNode root, int k) {
        TreeNode result = helper(root, k);
        return result != null ? result.val : 0; // Return value or 0 if not found
    }
```

```
    private TreeNode helper(TreeNode root, int k) {
        if (root == null) return null;

        // Traverse left subtree
        TreeNode left = helper(root.left, k);
        if (left != null) return left; // If found in left subtree

        count++; // Increment count for current node
        if (count == k) return root; // Found k-th smallest

        // Traverse right subtree
        return helper(root.right, k);
    }
}
```

## Output:


Case 1


Case 2

# Problem 10

## Aim:

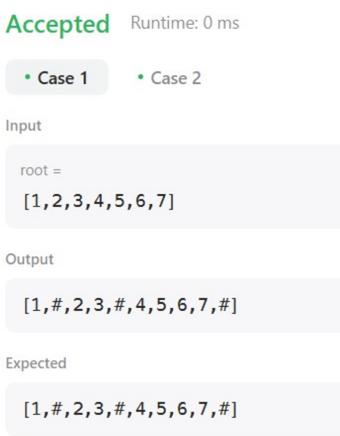
Populating Next Right Pointers in Each Node

## Code:

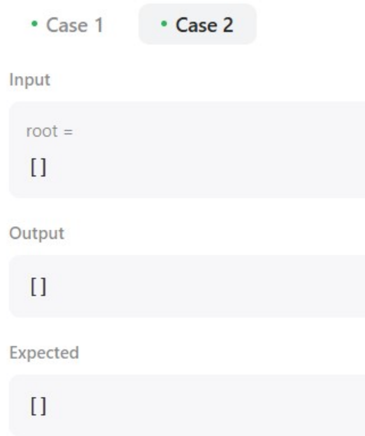
```
/*
// Definition for a Node.
class Node {
    public int val;
    public Node left;
    public Node right;
    public Node next;

    public Node() {}

    public Node(int _val) {
```

```
        val = _val;
    }

    public Node(int _val, Node _left, Node _right, Node _next) {
        val = _val;
        left = _left;
        right = _right;
        next = _next;
    }
};
*/

class Solution {
    public Node connect(Node root) {
        Queue<Node> q = new LinkedList<>();
        if (root == null ) return root;
        q.offer(root);
        while(!q.isEmpty()){
            int level = q.size();
            for(int i =0; i< level; i++){
                Node cur = q.poll();
                if (cur.left != null && cur.right !=null) {
                    q.offer(cur.left);
                    q.offer(cur.right);
                }

                if (q.isEmpty() || i == level -1)
                    cur.next = null;
                else
                    cur.next = q.peek();
            }

        }
        return root;
}
}
```

**Output:**


Accepted
Runtime: 0 ms
Case 1
Case 2
Input
root =
[1,2,3,4,5,6,7]
Output
[1,#,2,3,#,4,5,6,7,#]
Expected
[1,#,2,3,#,4,5,6,7,#]

Case 1


Case 1
Case 2
Input
root =
[]
Output
[]
Expected
[]

Case 2