CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

EXPERIMENT 6

Name: Aaditya maheshwari **UID:** 22BET10094

Section: 22BET_IOT-702 **Group:** B

Subject: Advance Programming **Subject Code:** 22ITP-351

Date of Performance: 28/03/2025

Aim: Maximum Depth of Binary Tree

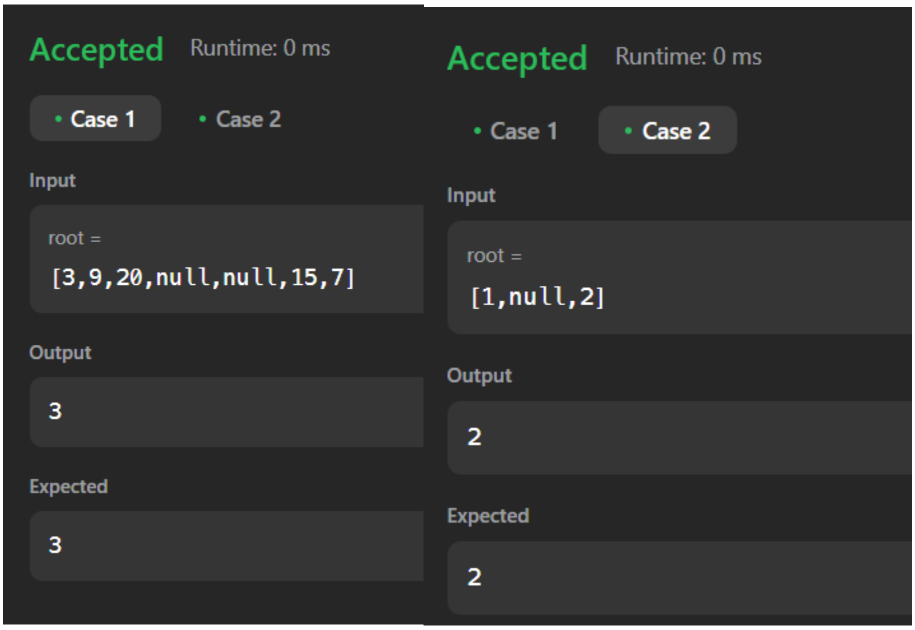
Objective: Given the root of a binary tree, return *its maximum depth*.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Code:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def maxDepth(self, root):
        if not root:
            return 0
        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)
        return max(left_depth, right_depth) + 1
```

Output:



Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```
root =
[3,9,20,null,null,15,7]
```

Output

```
3
```

Expected

```
3
```

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```
root =
[1,null,2]
```

Output

```
2
```

Expected

```
2
```

Aim: Validate Binary Search Tree

Objective: Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

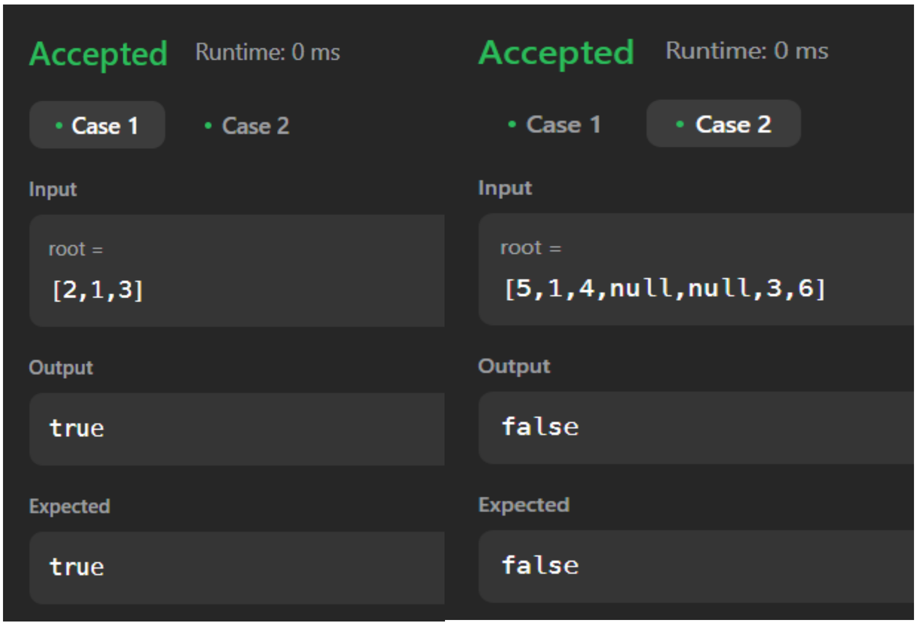
A valid BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Code:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def isValidBST(self, root, low=float('-inf'), high=float('inf')):
        if not root:
            return True

        if not (low < root.val < high):
            return False

        return (self.isValidBST(root.left, low, root.val) and
                self.isValidBST(root.right, root.val, high))
```

Output:



Accepted Runtime: 0 ms

Input

root =
[2,1,3]

Output

true

Expected

true

Accepted Runtime: 0 ms

Input

root =
[5,1,4,null,null,3,6]

Output

false

Expected

false

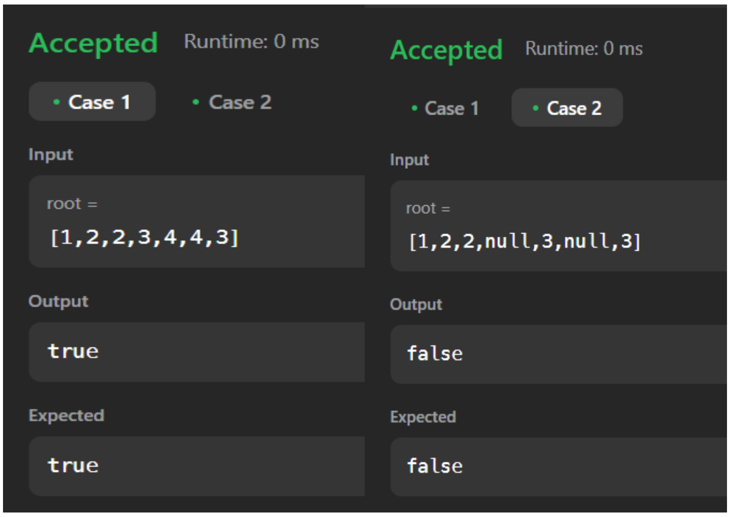
Aim: Symmetric Tree

Objective: Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

Code:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def isSymmetric(self, root):
        if not root:
            return True

        def isMirror(t1, t2):
            if not t1 and not t2:
                return True
            if not t1 or not t2:
                return False

            return (t1.val == t2.val and
                    isMirror(t1.left, t2.right) and
                    isMirror(t1.right, t2.left))

        return isMirror(root.left, root.right)
```

Output



Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

root =

```
[1,2,2,3,4,4,3]
```

Output

```
true
```

Expected

```
true
```

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

root =

```
[1,2,2,null,3,null,3]
```

Output

```
false
```

Expected

```
false
```

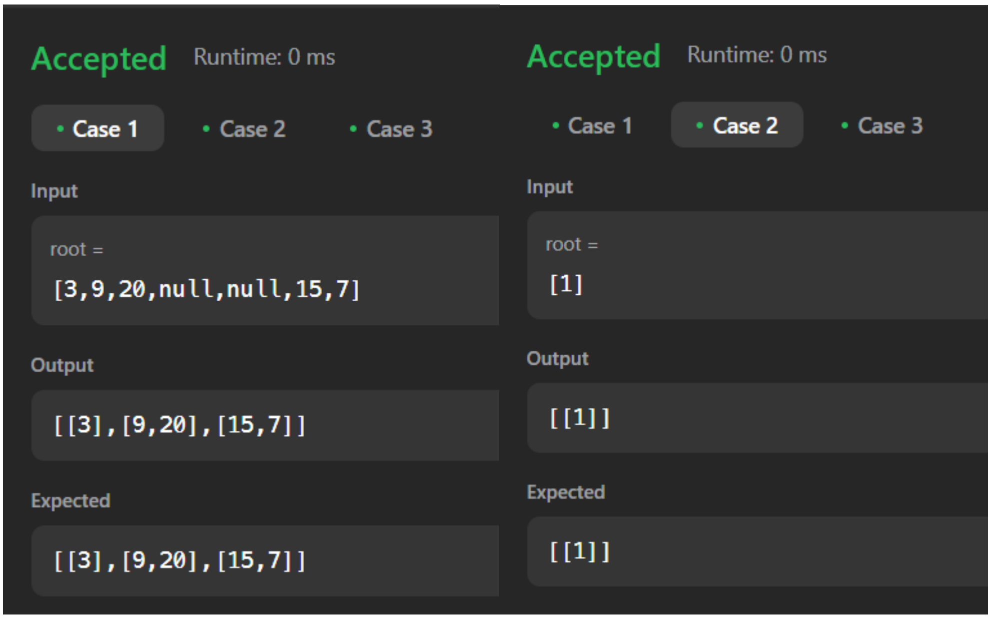
Aim: Binary Tree Level Order Traversal

Objective: Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Code:

```
from collections import deque


class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def levelOrder(self, root):
        if not root:
            return []

        result = []
        queue = deque([root])

        while queue:
            level = []
            for _ in range(len(queue)):
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(level)

        return result
```

Output:



Accepted Runtime: 0 ms

Input

root =
[3,9,20,null,null,15,7]

Output

[[3],[9,20],[15,7]]

Expected

[[3],[9,20],[15,7]]

Accepted Runtime: 0 ms

Input

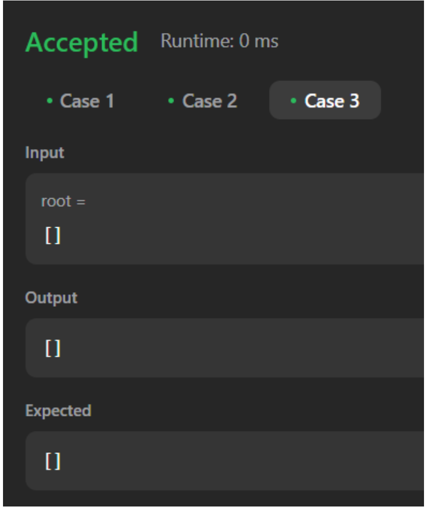
root =
[1]

Output

[[1]]

Expected

[[1]]



Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
root =
[]
```

Output

```
[]
```

Expected

```
[]
```

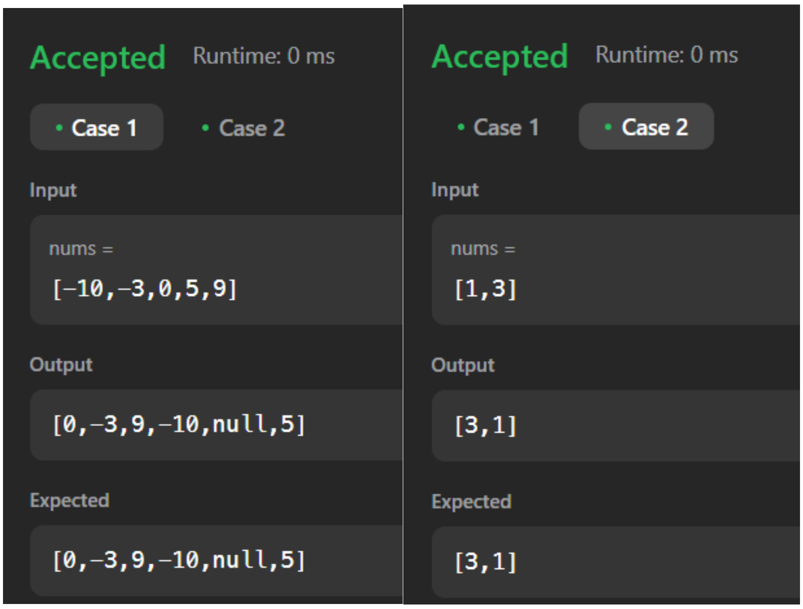
Aim: Convert Sorted Array to Binary Search Tree

Objective: Given an integer array nums where the elements are sorted in ascending order, convert *it to a height-balanced binary search tree.*

Code:

```
class Solution(object):
    def sortedArrayToBST(self, nums):
        if not nums:
            return None

        mid = len(nums) // 2
        root = TreeNode(nums[mid])
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid+1:])
        return root
```

Output:



Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

nums =

```
[-10,-3,0,5,9]
```

Output

```
[0,-3,9,-10,null,5]
```

Expected

```
[0,-3,9,-10,null,5]
```

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

nums =

```
[1,3]
```

Output

```
[3,1]
```

Expected

```
[3,1]
```

Aim: Binary Tree Inorder Traversal

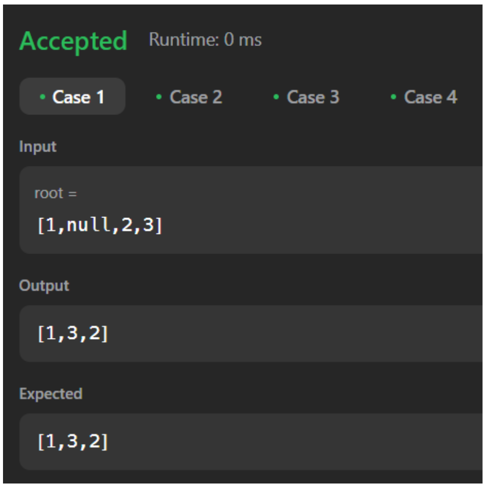
Objective: Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

Code:

```
from collections import deque


class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class Solution(object):
    def inorderTraversal(self, root):
        result = []
        stack = []
        current = root

        while current or stack:
            while current:
                stack.append(current)
                current = current.left
            current = stack.pop()
            result.append(current.val)
            current = current.right

        return result
```

Output:



Accepted Runtime: 0 ms

Input

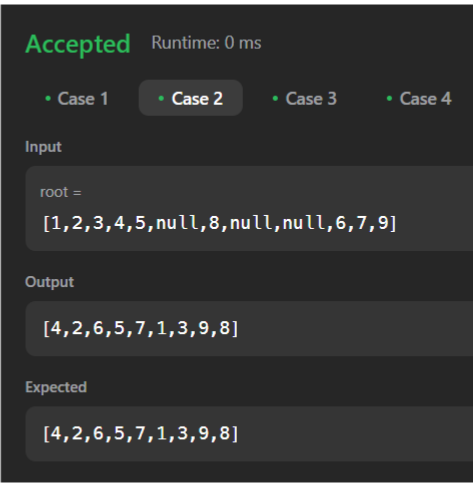
root =
[1,null,2,3]

Output

[1,3,2]

Expected

[1,3,2]



Accepted Runtime: 0 ms

Input

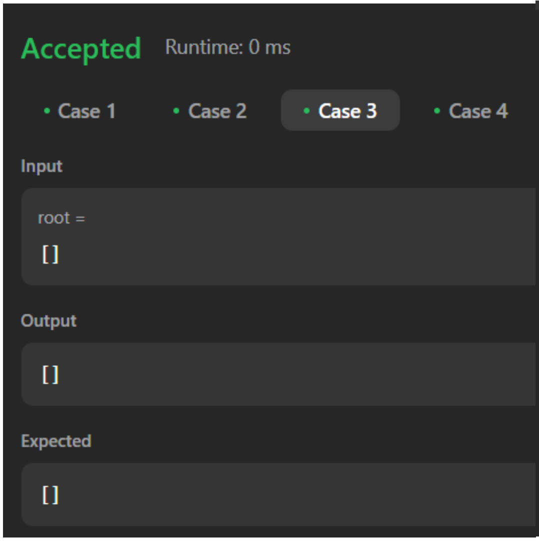
root =
[1,2,3,4,5,null,8,null,null,6,7,9]

Output

[4,2,6,5,7,1,3,9,8]

Expected

[4,2,6,5,7,1,3,9,8]



Accepted Runtime: 0 ms

Input

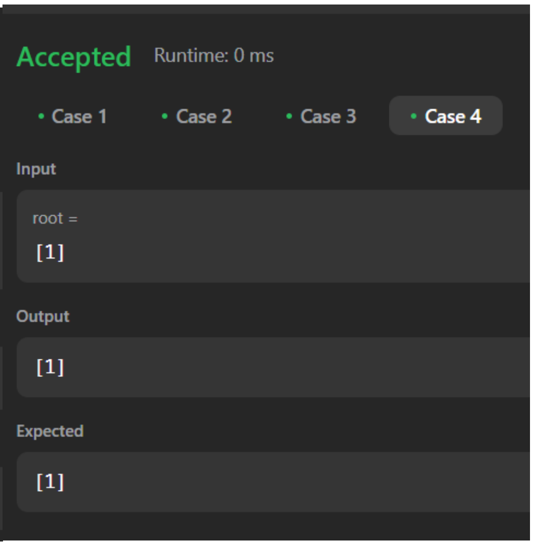
```
root =
[]
```

Output

```
[]
```

Expected

```
[]
```



Accepted Runtime: 0 ms

Input

```
root =
[1]
```

Output

```
[1]
```

Expected

```
[1]
```

Aim: Construct Binary Tree from Inorder and Postorder Traversal

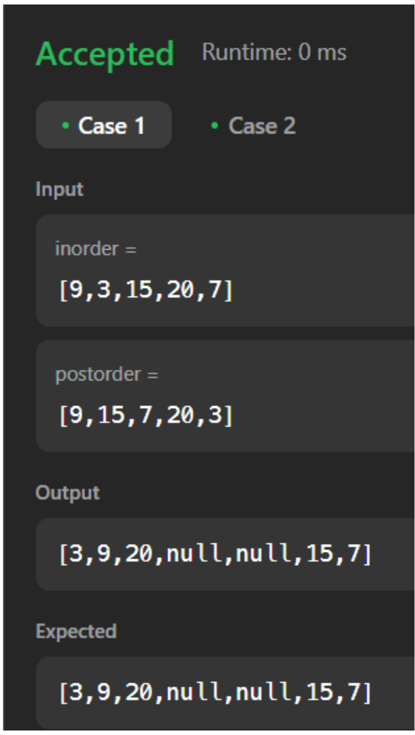
Objective: Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.

Code:

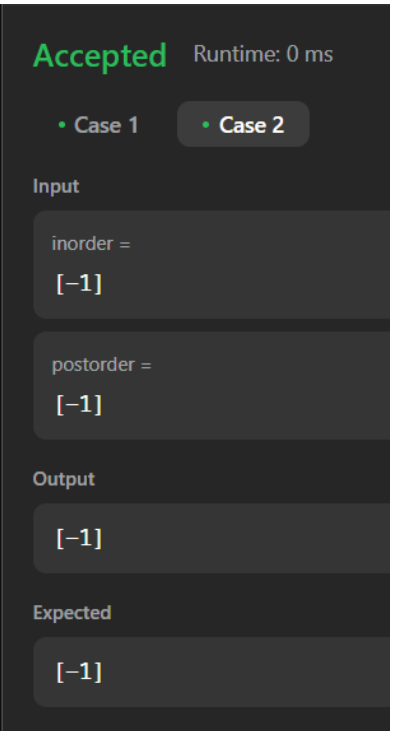
```
class Solution(object):
    def buildTree(self, inorder, postorder):
        if not inorder or not postorder:
            return None
        root_val = postorder.pop()
        root = TreeNode(root_val)
        root_index = inorder.index(root_val)
        root.right = self.buildTree(inorder[root_index + 1:], 
postorder)
```

```
        root.left = self.buildTree(inorder[:root_index], postorder)

        return root
```

Output:



```
Accepted   Runtime: 0 ms

• Case 1   • Case 2

Input

inorder =
[9,3,15,20,7]

postorder =
[9,15,7,20,3]

Output

[3,9,20,null,null,15,7]

Expected

[3,9,20,null,null,15,7]
```



```
Accepted   Runtime: 0 ms

• Case 1   • Case 2

Input

inorder =
[-1]

postorder =
[-1]

Output

[-1]

Expected

[-1]
```

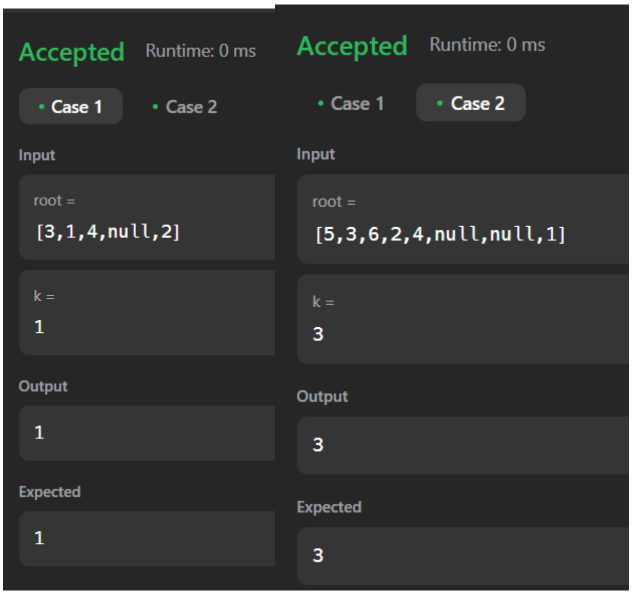
Aim: Kth Smallest element in a BST

Objective: Given the root of a binary search tree, and an integer k, return *the* kth *smallest value (1-indexed) of all the values of the nodes in the tree*.

Code:

```
class TreeNode(object):

    def __init__(self, val=0, left=None, right=None):

        self.val = val

        self.left = left
        self.right = right


class Solution(object):
    def kthSmallest(self, root, k):
        stack = []
        while True:
            while root:
                stack.append(root)
                root = root.left
            root = stack.pop()
            k -= 1
            if k == 0:
                return root.val
            root = root.right
```

Output:



```
Accepted   Runtime: 0 ms
Input
root =
[3,1,4,null,2]
k =
1
Output
1
Expected
1
```

```
Accepted   Runtime: 0 ms
Input
root =
[5,3,6,2,4,null,null,1]
k =
3
Output
3
Expected
3
```

Aim: Populating Next Right Pointers in Each Node

Objective: You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

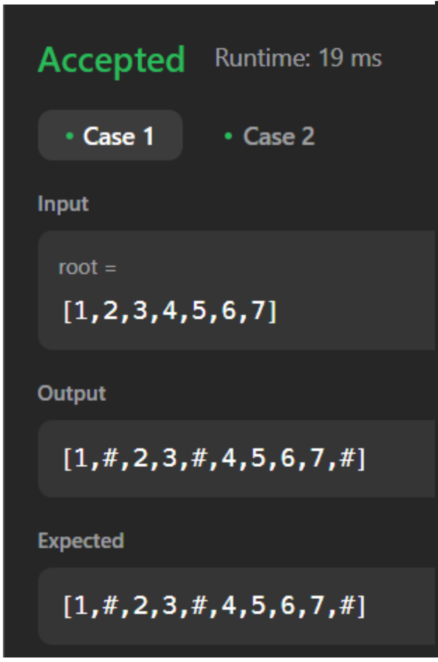
Initially, all next pointers are set to NULL.

Code:

```
class Node(object):
    def __init__(self, val=0, left=None, right=None,
next=None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next
```

```
class Solution(object):
    def connect(self, root):
        if not root:
            return None

        leftmost = root

        while leftmost.left:
            head = leftmost
            while head:
                head.left.next = head.right

                if head.next:
                    head.right.next = head.next.left

                head = head.next

            leftmost = leftmost.left

        return root
```

Output:



Accepted Runtime: 19 ms

Input

root =

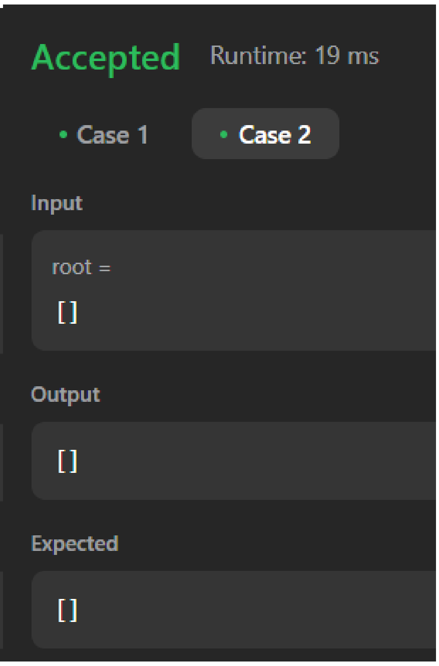
```
[1,2,3,4,5,6,7]
```

Output

```
[1,#,2,3,#,4,5,6,7,#]
```

Expected

```
[1,#,2,3,#,4,5,6,7,#]
```



Accepted Runtime: 19 ms

Input

root =

```
[]
```

Output

```
[]
```

Expected

```
[]
```