Experiment - 3(A)

Student Name: Paras Bhatt **UID:** 22BCS15683
Branch: CSE **Section/Group:** NTPP_602-A
Semester: 6 **Date of Performance:** 17-02-25
Subject Name: Advanced Programming Lab-2 **Subject Code:** 22CSH-359

1. **Title:** Divide and Conquer (Longest Nice Substring)
https://leetcode.com/problems/maximum-depth-of-binary-tree/

2. **Aim:** Maximum Depth of Binary Tree

3. **Objective:** The objective is to find the maximum depth of a binary tree. The depth is defined as the number of nodes along the longest path from the root node down to the farthest leaf node.

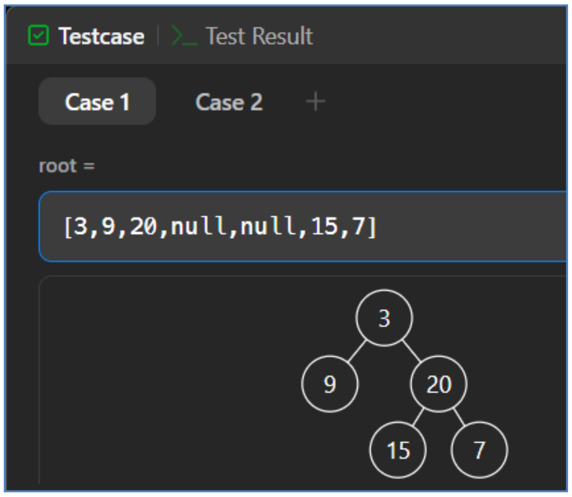
4. **Algorithm:**

   1. If the root is `None`, return `0`.
   2. Recursively calculate the maximum depth of the left and right subtrees.
   3. Return `1 + max(left_depth, right_depth)`.

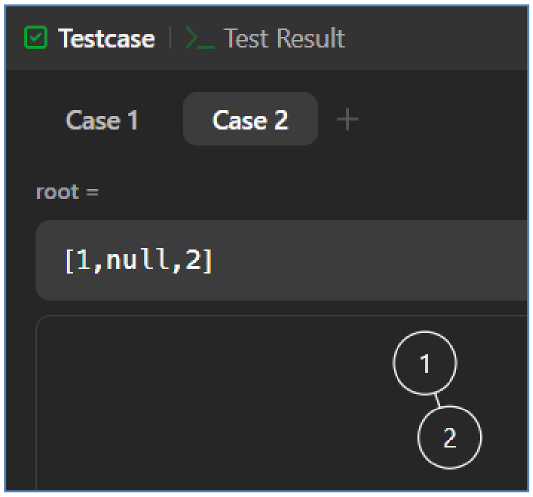
5. **Implementation/Code:**

```
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 1 + max(left_depth, right_depth)
```

6. **Output:**



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



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

7. **Time Complexity:** O(N)

8. **Space Complexity:** O(N)

9. **Learning Outcomes:**
   - Understand the concept of tree depth and its calculation using recursion.
   - Apply depth-first search (DFS) to solve binary tree problems.
   - Analyze time and space complexity of recursive algorithms.
   - Communicate algorithmic steps and complexities clearly.

Experiment 3(B)

1. **Title:** Reverse Bits (https://leetcode.com/problems/validate-binary-search-tree/)

2. **Objective:** The objective is to validate if a given binary tree is a Binary Search Tree (BST). A BST must follow the rule that for each node, all nodes in its left subtree must contain values smaller than the node's value, and all nodes in its right subtree must contain values greater than the node's value. Additionally, the left and right subtrees must also be valid BSTs.

3. Algorithm:

a. Start by checking if the root of the tree is None. If so, return True as an empty tree is a valid BST.

b. Define a helper function validate(node, low, high) that will check if the value of the current node is within the range defined by low and high.

c. Recursively check if the left child of the node is within the valid range [low, node.val) and if the right child is within the range (node.val, high].

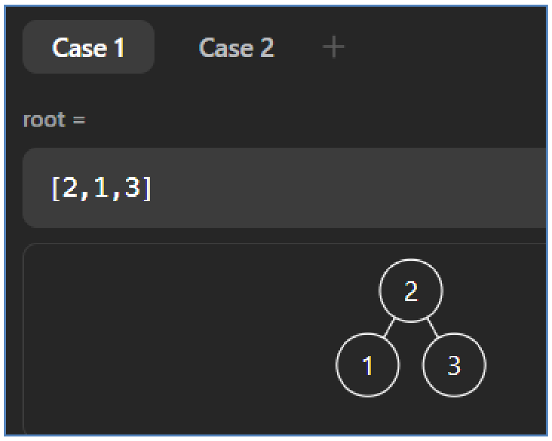
d. If any node violates the BST property, return False. If the entire tree satisfies the conditions, return True.

e. Start the recursive check from the root node with the range (-∞, ∞).

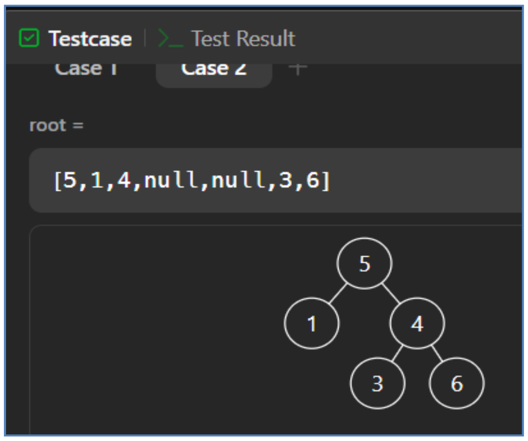
4. Implementation/Code:

```
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):

        def validate(node, low, high):
            if not node:
                return True
            if not (low < node.val < high):
                return False
            return (validate(node.left, low, node.val) and
                    validate(node.right, node.val, high))

        return validate(root, float('-inf'), float('inf'))
```

6. Output:



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



```
[5,1,4,null,null,3,6]
```

8. Time Complexity: O(N)

9. Space Complexity: O(H)

9. Learning Outcomes:

1. Understand the key properties of a Binary Search Tree (BST).
2. Learn how to recursively validate the structure of a tree.
3. Apply depth-first search (DFS) to solve tree-related problems.

Experiment 6(C)

1. **Title:** Number of 1 Bits (https://leetcode.com/problems/binary-tree-level-order-traversal/)

2. **Objective:** The objective is to return the level order traversal of the nodes in a binary tree. The traversal should be performed level by level, starting from the root node, moving from left to right at each level.

4. Algorithm:

a. Start by checking if the root of the tree is `None`. If so, return an empty list `[]`, as the tree is empty.
b. Initialize an empty list `result` to store the nodes' values level by level.
c. Use a queue (FIFO structure) to perform a breadth-first search (BFS).
d. Begin by enqueuing the root node into the queue.
e. While the queue is not empty:

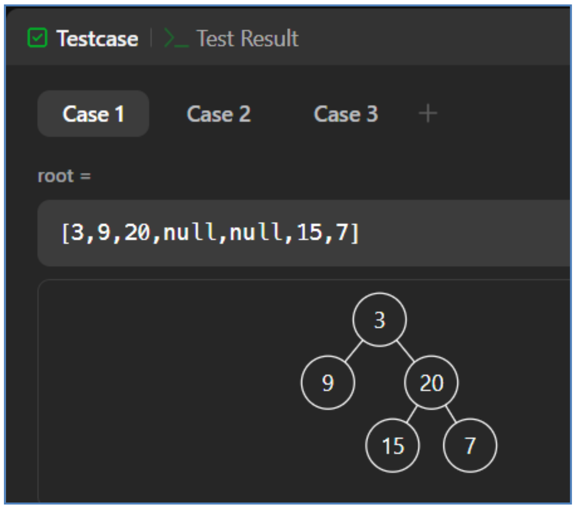
- Determine the number of nodes at the current level (this is equal to the size of the queue).
- Initialize a temporary list `level_values` to store values of the nodes at this level.
- For each node at the current level, dequeue it, append its value to `level_values`, and enqueue its children (if any).
- After processing all nodes at the current level, append `level_values` to `result`. f. After the loop ends, return `result`, which contains the level order traversal.

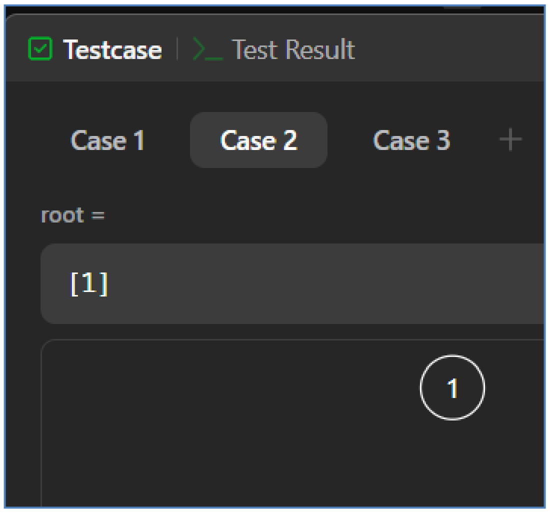
5. Implementation/Code:

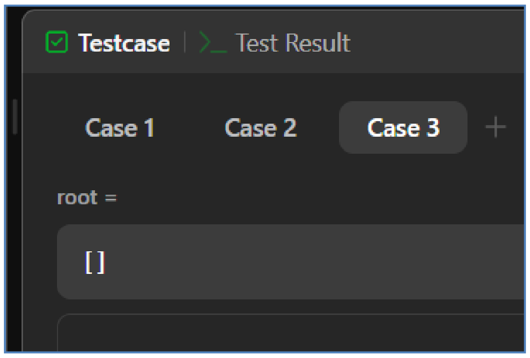
```
from collections import deque

class Solution(object):
    def levelOrder(self, root):
        """
        :type root: Optional[TreeNode]
        :rtype: List[List[int]]
        """
        if not root:
            return []

        result = []
        queue = deque([root])   # Initialize queue with the root node

        while queue:
            level_size = len(queue)   # Get the number of nodes at the
current level
            level_values = []
```

```
              for _ in range(level_size):
                  node = queue.popleft()  # Pop the leftmost node
                  level_values.append(node.val)   # Add its value to the
current level list

                  # Enqueue the left and right children of the current node
(if they exist)
                  if node.left:
                      queue.append(node.left)
                  if node.right:
                      queue.append(node.right)

              result.append(level_values)   # Add the current level values to
the result

          return result
```

6. Output:


Testcase | Test Result
Case 1 Case 2 Case 3
root =
[3,9,20,null,null,15,7]


Testcase | Test Result
Case 1 Case 2 Case 3
root =
[1]


Testcase | Test Result
Case 1 Case 2 Case 3
root =
[]

8. Time Complexity: O(N)

9. Space Complexity: O(N)