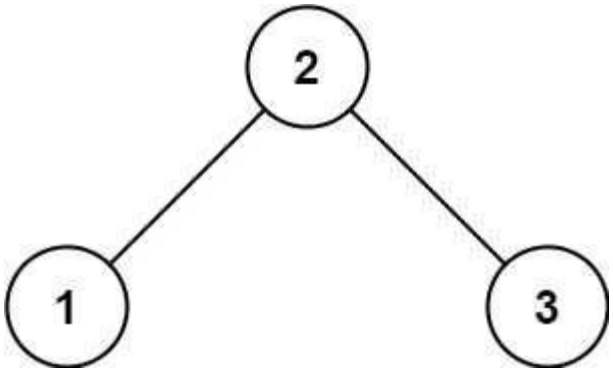
Yakshita Rakholiya
Dhruv Ranpariya
Suraj Salunkhe
Course: CS-608-21141
Assignment-3
Team-2

Q-1

Determine if a given root of a tree is a valid binary search tree (BST) A valid BST is defined as follows:

- Given root, the **left subtree** of a node contains only nodes with keys **less than the node's key**.
- Given root, the **right subtree** of a node contains only nodes with keys **greater than the node's key**.
- Ensure that both the left and right subtrees are also binary search trees.

Example:



Input: root = [2,1,3]
Output: true

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
# Function to validate if a given tree is a valid BST
def isValidBST(root):
    def helper(node, lower=float('-inf'), upper=float('inf')):
        # Base case: if node is None, it's valid
        if node is None:
            return True
        # Check if the node violates the BST property
        if node.val <= lower or node.val >= upper:
            return False
        # Recursively check left and right subtrees
        return (helper(node.left, lower, node.val) and
                helper(node.right, node.val, upper))
    return helper(root)
# Function to accept input values for the tree nodes
def buildTree():
    nodes = input("Enter values for tree nodes (separated by commas): ").split(',')
    nodes = [int(val) if val != 'None' else None for val in nodes]
    if not nodes:
        return None
    root = TreeNode(nodes[0])
    queue = [root]
    i = 1
    # Build the tree level-wise using a queue
    while i < len(nodes):
        node = queue.pop(0)
        # Create left child if not None
        if nodes[i] is not None:
            node.left = TreeNode(nodes[i])
            queue.append(node.left)
        i += 1
        # Create right child if not None
        if i < len(nodes) and nodes[i] is not None:
            node.right = TreeNode(nodes[i])
            queue.append(node.right)
        i += 1
    return root
# Example usage
# Input values for tree nodes
root = buildTree()
# Validate if the given tree is a valid BST
is_valid_bst = isValidBST(root)
# Print the result
print(is_valid_bst)
```

```
Enter values for tree nodes (separated by commas): 2,1,3
True
```

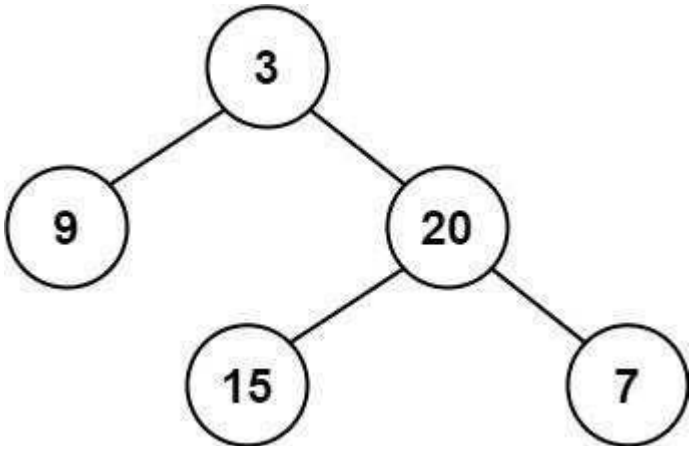
Q-2

Balanced Binary Tree

Determine if a binary tree is height-balanced.

A height-balanced binary tree is defined as a binary tree in which the left and right subtrees of every node differ in height by no more than 1.

Example:



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

Output: true

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
# Function to determine if a binary tree is height-balanced
def is_balanced(root):
    def height(node):
        # Base case: height of None is -1
        if node is None:
            return -1
        # Recursively calculate height of left and right subtrees
        return 1 + max(height(node.left), height(node.right))
    # Check if the height difference between left and right subtrees is no more than 1
    def check_balance(node):
        if node is None:
            return True
        left_height = height(node.left)
        right_height = height(node.right)
        if abs(left_height - right_height) > 1:
            return False
        return check_balance(node.left) and check_balance(node.right)
    return check_balance(root)
# Function to accept input values for the tree nodes
def build_tree():
    nodes = input("Enter values for tree nodes (separated by commas): ").split(',')
    nodes = [int(val) if val != 'None' else None for val in nodes]
    if not nodes:
        return None
    root = TreeNode(nodes[0])
    queue = [root]
    i = 1
    # Build the tree level-wise using a queue
    while i < len(nodes):
        node = queue.pop(0)
        # Create left child if not None
        if nodes[i] is not None:
            node.left = TreeNode(nodes[i])
            queue.append(node.left)
        i += 1
        # Create right child if not None
        if i < len(nodes) and nodes[i] is not None:
            node.right = TreeNode(nodes[i])
            queue.append(node.right)
        i += 1
    return root
# Example usage
# Input values for tree nodes
root = build_tree()
# Check if the given tree is height-balanced
is_balanced_tree = is_balanced(root)
# Print the result
print(is_balanced_tree)
```

```
Enter values for tree nodes (separated by commas): 3,9,20,None,None,15,7
True
```

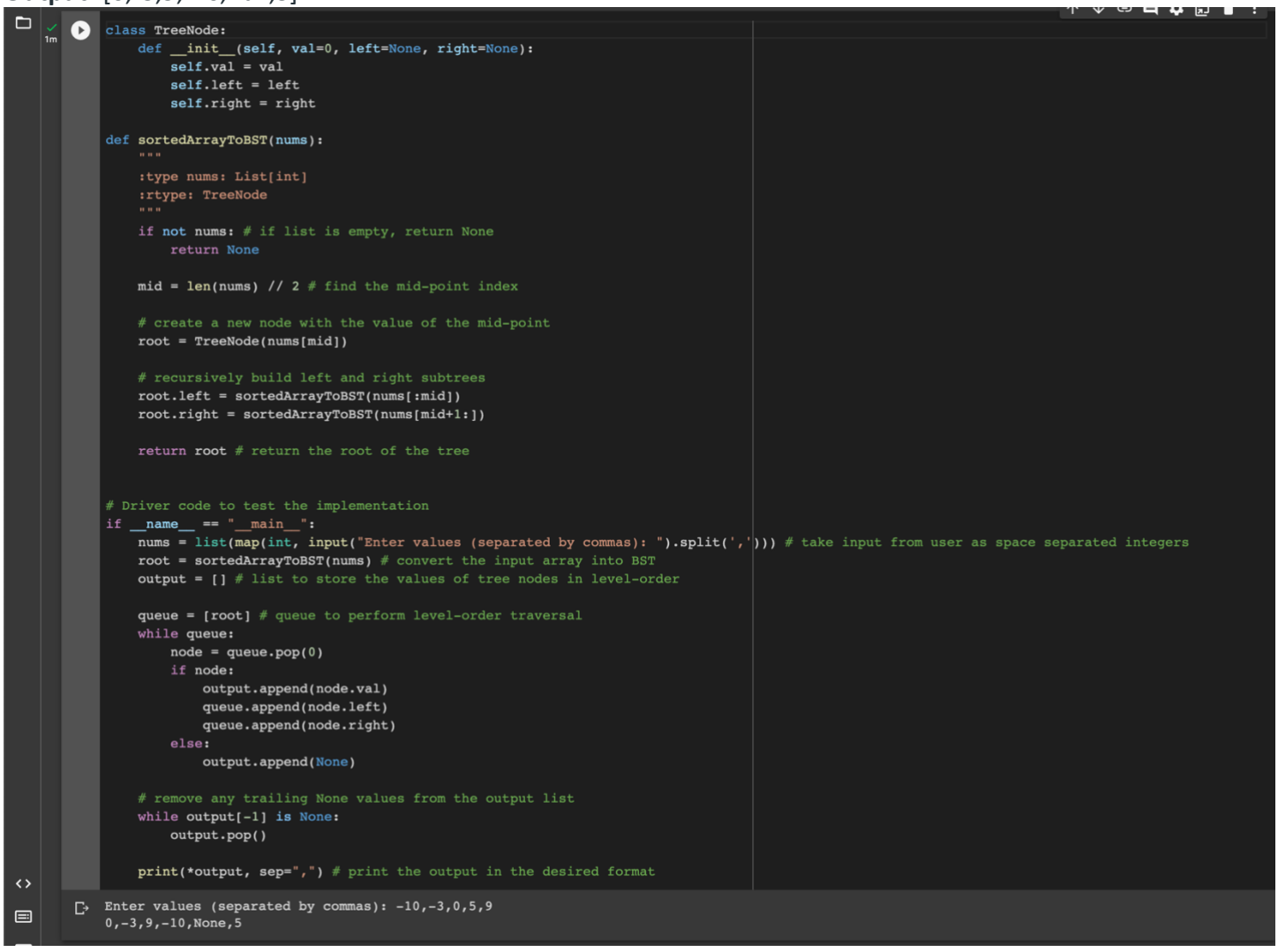
3. Convert Sorted Array to Binary Search Tree

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

A **height-balanced** binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

Input: nums = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]



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

def sortedArrayToBST(nums):
    """
    :type nums: List[int]
    :rtype: TreeNode
    """
    if not nums: # if list is empty, return None
        return None

    mid = len(nums) // 2 # find the mid-point index

    # create a new node with the value of the mid-point
    root = TreeNode(nums[mid])

    # recursively build left and right subtrees
    root.left = sortedArrayToBST(nums[:mid])
    root.right = sortedArrayToBST(nums[mid+1:])

    return root # return the root of the tree


# Driver code to test the implementation
if __name__ == "__main__":
    nums = list(map(int, input("Enter values (separated by commas): ").split(','))) # take input from user as space separated integers
    root = sortedArrayToBST(nums) # convert the input array into BST
    output = [] # list to store the values of tree nodes in level-order

    queue = [root] # queue to perform level-order traversal
    while queue:
        node = queue.pop(0)
        if node:
            output.append(node.val)
            queue.append(node.left)
            queue.append(node.right)
        else:
            output.append(None)

    # remove any trailing None values from the output list
    while output[-1] is None:
        output.pop()

    print(*output, sep=",") # print the output in the desired format
```

```
Enter values (separated by commas): -10,-3,0,5,9
0,-3,9,-10,None,5
```