**LAB 12**

**Name: Omerullah Ansari**

**ID: 65584**

Q1

from collections import deque

class TreeNode:

def \_\_init\_\_(self, value=0, left=None, right=None):

self.value = value

self.left = left

self.right = right

def level\_order\_traversal(root):

if not root:

return []

result = []

queue = deque([root])

while queue:

level\_size = len(queue)

current\_level = []

for \_ in range(level\_size):

node = queue.popleft()

current\_level.append(node.value)

if node.left:

queue.append(node.left)

if node.right:

queue.append(node.right)

result.append(current\_level)

return result

root = TreeNode(1)

root.left = TreeNode(2)

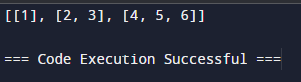
root.right = TreeNode(3)

root.left.left = TreeNode(4)

root.left.right = TreeNode(5)

root.right.right = TreeNode(6)

print(level\_order\_traversal(root))



Q2

class Node:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.value = key

def height(node):

if node is None:

return -1

left\_height = height(node.left)

right\_height = height(node.right)

return max(left\_height, right\_height) + 1

root = Node(1)

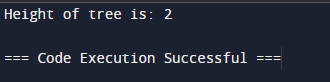
root.left = Node(2)

root.right = Node(3)

root.left.left = Node(4)

root.left.right = Node(5)

print("Height of tree is:", height(root))



Q3

class Node:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.value = key

def update\_keys\_with\_sum\_of\_greater\_keys(node):

def reverse\_in\_order\_traversal(node, acc\_sum):

if node is None:

return acc\_sum

acc\_sum = reverse\_in\_order\_traversal(node.right, acc\_sum)

acc\_sum += node.value

node.value = acc\_sum

acc\_sum = reverse\_in\_order\_traversal(node.left, acc\_sum)

return acc\_sum

reverse\_in\_order\_traversal(node, 0)

root = Node(5)

root.left = Node(3)

root.right = Node(8)

root.left.left = Node(2)

root.left.right = Node(4)

root.right.left = Node(7)

root.right.right = Node(9)

update\_keys\_with\_sum\_of\_greater\_keys(root)

def in\_order\_traversal(node):

if node is None:

return

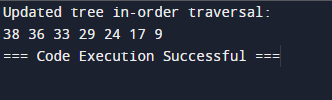
in\_order\_traversal(node.left)

print(node.value, end=' ')

in\_order\_traversal(node.right)

print("Updated tree in-order traversal:")

in\_order\_traversal(root)



Q4

class Node:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.value = key

def remove\_out\_of\_range\_nodes(node, min\_val, max\_val):

if node is None:

return None

node.left = remove\_out\_of\_range\_nodes(node.left, min\_val, max\_val)

node.right = remove\_out\_of\_range\_nodes(node.right, min\_val, max\_val)

if node.value < min\_val:

return node.right

elif node.value > max\_val:

return node.left

return node

def insert(node, key):

if node is None:

return Node(key)

if key < node.value:

node.left = insert(node.left, key)

else:

node.right = insert(node.right, key)

return node

def in\_order\_traversal(node):

if node is None:

return

in\_order\_traversal(node.left)

print(node.value, end=' ')

in\_order\_traversal(node.right)

root = None

keys = [5, 3, 8, 2, 4, 7, 9]

for key in keys:

root = insert(root, key)

print("Original tree in-order traversal:")

in\_order\_traversal(root)

print()

min\_val = 4

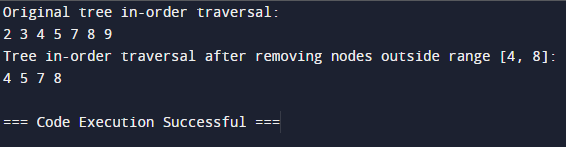
max\_val = 8

root = remove\_out\_of\_range\_nodes(root, min\_val, max\_val)

print(f"Tree in-order traversal after removing nodes outside range [{min\_val}, {max\_val}]:")

in\_order\_traversal(root)

print()



Q5

class Node:

def \_\_init\_\_(self, key):

self.left = None

self.right = None

self.value = key

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, key):

if self.root is None:

self.root = Node(key)

else:

self.\_insert(self.root, key)

def \_insert(self, node, key):

if key < node.value:

if node.left is None:

node.left = Node(key)

else:

self.\_insert(node.left, key)

else:

if node.right is None:

node.right = Node(key)

else:

self.\_insert(node.right, key)

def desc\_traversal(self):

result = []

self.\_desc\_traversal(self.root, result)

return result

def \_desc\_traversal(self, node, result):

if node is not None:

self.\_desc\_traversal(node.right, result)

result.append(node.value)

self.\_desc\_traversal(node.left, result)

def find\_min(self):

if self.root is None:

return None

current = self.root

while current.left is not None:

current = current.left

return current.value

def find\_max(self):

if self.root is None:

return None

current = self.root

while current.right is not None:

current = current.right

return current.value

def delete\_min(self):

if self.root is not None:

self.root = self.\_delete\_min(self.root)

def \_delete\_min(self, node):

if node.left is None:

return node.right

node.left = self.\_delete\_min(node.left)

return node

bst = BinarySearchTree()

keys = [5, 3, 8, 2, 4, 7, 9]

for key in keys:

bst.insert(key)

print("Descending order traversal:")

print(bst.desc\_traversal())

print("Minimum value in the BST:", bst.find\_min())

print("Maximum value in the BST:", bst.find\_max())

bst.delete\_min()

print("Descending order traversal after deleting the minimum value:")

print(bst.desc\_traversal())

