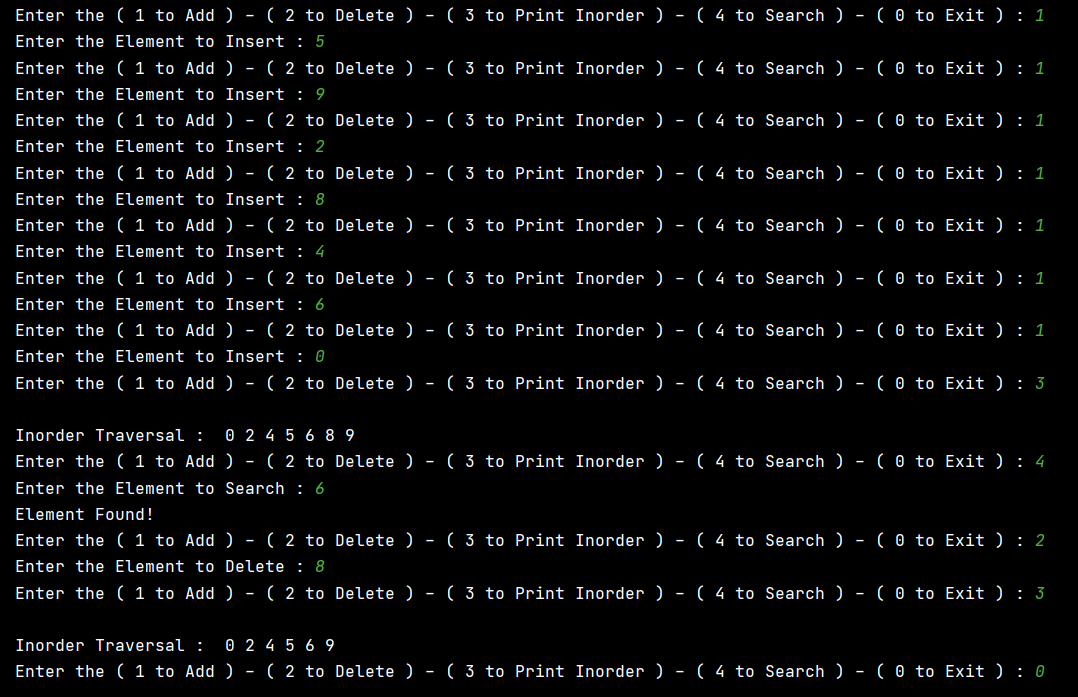
**S Abhishek AM.EN.U4CSE19147**

**Data Structures**

**1. Implement the binary search tree and its operations: insertion, deletion and searching.**

class Node:  
 def \_\_init\_\_(self**,** key):  
 self.key = key  
 self.left = None  
 self.right = None  
 self.parent = None  
  
 def insert(self**,** node):  
 if self.key > node.key:  
 if self.left is None:  
 self.left = node  
 node.parent = self  
 else:  
 self.left.insert(node)  
 elif self.key < node.key:  
 if self.right is None:  
 self.right = node  
 node.parent = self  
 else:  
 self.right.insert(node)  
  
 def inorder(self):  
 if self.left is not None:  
 self.left.inorder()  
 print(self.key**,** end=' ')  
 if self.right is not None:  
 self.right.inorder()  
  
 def rep\_parent(self**,** new\_node):  
 if self.parent is not None:  
 if new\_node is not None:  
 new\_node.parent = self.parent  
 if self.parent.left == self:  
 self.parent.left = new\_node  
 elif self.parent.right == self:  
 self.parent.right = new\_node  
 else:  
 self.key = new\_node.key  
 self.left = new\_node.left  
 self.right = new\_node.right  
 if new\_node.left is not None:  
 new\_node.left.parent = self  
 if new\_node.right is not None:  
 new\_node.right.parent = self  
  
 def find\_min(self):  
 current = self  
 while current.left is not None:  
 current = current.left  
 return current  
  
 def pop(self):  
 if self.left is not None and self.right is not None:  
 successor = self.right.find\_min()  
 self.key = successor.key  
 successor.pop()  
 elif self.left is not None:  
 self.rep\_parent(self.left)  
 elif self.right is not None:  
 self.rep\_parent(self.right)  
 else:  
 self.rep\_parent(None)  
  
 def search(self**,** key):  
 if self.key > key:  
 if self.left is not None:  
 return self.left.search(key)  
 else:  
 return None  
 elif self.key < key:  
 if self.right is not None:  
 return self.right.search(key)  
 else:  
 return None  
 return self  
  
 def find(self**,** key):  
 if self.key > key:  
 if self.left is not None:  
 return self.left.find(key)  
 else:  
 print("Element not Found!")  
 return  
 elif self.key < key:  
 if self.right is not None:  
 return self.right.find(key)  
 else:  
 print("Element not Found!")  
 return  
 print("Element Found!")  
  
  
class Tree:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def inorder(self):  
 if self.root is not None:  
 self.root.inorder()  
  
 def push(self**,** key):  
 new\_node = Node(key)  
 if self.root is None:  
 self.root = new\_node  
 else:  
 self.root.insert(new\_node)  
  
 def pop(self**,** key):  
 to\_pop = self.search(key)  
 if self.root == to\_pop and self.root.left is None and self.root.right is None:  
 self.root = None  
 else:  
 if to\_pop is None:  
 print("No Node!")  
 else:  
 to\_pop.pop()  
  
 def search(self**,** key):  
 if self.root is not None:  
 return self.root.search(key)  
  
 def find(self**,** key):  
 if self.root is not None:  
 return self.root.find(key)  
  
  
Tree = Tree()  
  
while (**1**):  
 op = int(input("Enter the ( 1 to Add ) - ( 2 to Delete ) - ( 3 to Print Inorder ) - ( 4 to Search ) - ( 0 to Exit "  
 ") : "))  
  
 if op == **1**:  
 ele = int(input("Enter the Element to Insert : "))  
 Tree.push(ele)  
  
 elif op == **2**:  
 ele = int(input("Enter the Element to Delete : "))  
 Tree.pop(ele)  
  
 elif op == **3**:  
 print("\nInorder Traversal : "**,** end=" ")  
 Tree.inorder()  
 print()  
  
 elif op == **4**:  
 ele = int(input("Enter the Element to Search : "))  
 Tree.find(ele)  
  
 elif op == **0**:  
 break  
  
 else:  
 print("Wrong option!")

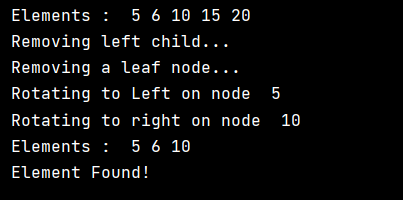


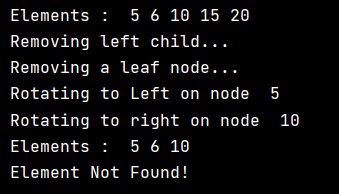
**2. Implement the AVL tree and its operations: insertion, deletion and searching.**

class Node(object):  
  
 def \_\_init\_\_(self**,**data):  
 self.data = data  
 self.leftChild = None  
 self.rightChild = None  
 self.height = **0**class AVL(object):  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def calcHeight(self**,**node):  
 if not node:  
 return -**1** #print('\nHeight: ', node.height)  
 return node.height  
  
  
 def insert(self**,** data):  
 self.root = self.insertNode(data**,** self.root)  
  
 def insertNode(self**,** data**,** node):  
 if not node:  
 return Node(data)  
 if data < node.data:  
 node.leftChild = self.insertNode(data**,** node.leftChild)  
 else:  
 node.rightChild = self.insertNode(data**,** node.rightChild)  
  
 node.height = max(self.calcHeight(node.leftChild)**,** self.calcHeight(node.rightChild)) + **1** #print('Node {} Inserted'.format(data))  
 return self.settleViolation(data**,** node)  
  
  
  
 def settleViolation(self**,** data**,** node):  
 balance = self.calcBalance(node)  
  
 if balance > **1** and data < node.leftChild.data:  
 return self.rotateRight(node)  
  
 if balance < -**1** and data > node.rightChild.data:  
 return self.rotateLeft(node)  
  
 if balance > **1** and data > node.leftChild.data:  
 node.leftChild = self.rotateLeft(node.leftChild)  
 return self.rotateRight(node)  
  
 if balance < -**1** and data < node.rightChild.data:  
 node.rightChild = self.rotateRight(node.rightChild)  
 return self.rotateLeft(node)  
 return node  
  
  
 def calcBalance(self**,**node):  
 if not node:  
 return **0** return self.calcHeight(node.leftChild) - self.calcHeight(node.rightChild)  
  
  
 def rotateRight(self**,**node):  
 print('Rotating to right on node '**,** node.data)  
 tempLeftChild = node.leftChild  
 t = tempLeftChild.rightChild  
  
 tempLeftChild.rightChild = node  
 node.leftChild = t  
  
 node.height = max(self.calcHeight(node.leftChild)**,** self.calcHeight(node.rightChild)) + **1** tempLeftChild.height = max(self.calcHeight(tempLeftChild.leftChild)**,** self.calcHeight(tempLeftChild.rightChild)) + **1** return tempLeftChild  
  
 def rotateLeft(self**,**node):  
 print('Rotating to Left on node '**,** node.data)  
 tempRightChild = node.rightChild  
 t = tempRightChild.leftChild  
  
 tempRightChild.leftChild = node  
 node.rightChild = t  
  
 node.height = max(self.calcHeight(node.leftChild)**,** self.calcHeight(node.rightChild)) + **1** tempRightChild.height = max(self.calcHeight(tempRightChild.leftChild)**,** self.calcHeight(tempRightChild.rightChild)) + **1** return tempRightChild  
  
  
 def remove(self**,** data):  
 if self.root:  
 self.root = self.removeNode(data**,** self.root)  
  
 def removeNode(self**,** data**,** node):  
 if not node:  
 return node  
 if data < node.data:  
 node.leftChild = self.removeNode(data**,**node.leftChild)  
 if data > node.data:  
 node.rightChild = self.removeNode(data**,** node.rightChild)  
 else:  
 if not node.leftChild and not node.rightChild:  
 print('Removing a leaf node...')  
 del node  
 return None  
 if not node.leftChild:  
 print('\nRemoving right child...')  
 tempNode = node.rightChild  
 del node  
 return tempNode  
 if not node.rightChild:  
 print('\nRemoving left child...')  
 tempNode = node.leftChild  
 return tempNode  
 print('\nRemoving Node with two children...')  
 tempNode = self.getPredecessor(node.leftChild)  
 node.data = tempNode.data  
 node.leftChild = self.removeNode(tempNode.data**,** node.leftChild)  
 if not node:  
 return node  
  
 node.height = max(self.calcHeight(node.leftChild)**,** self.calcHeight(node.rightChild)) + **1** balance = self.calcBalance(node)  
  
  
 if balance > **1** and self.calcBalance(node.leftChild) >= **0**:  
 return self.rotateRight(node)  
  
 if balance < -**1** and self.calcBalance(node.rightChild) <= **0**:  
 return self.rotateLeft(node)  
  
 if balance > **1** and self.calcBalance(node.leftChild) < **0**:  
 node.leftChild = self.rotateLeft(node.leftChild)  
 return self.rotateRight(node)  
  
 if balance < -**1** and self.calcBalance(node.rightChild) > **0**:  
 node.rightChild = self.rotateRight(node.rightChild)  
 return self.rotateLeft(node)  
 return node  
  
  
 def getPredecessor(self**,** node):  
 if node.rightChild:  
 return self.getPredecessor(node.rightChild)  
 return node  
  
  
 def traverse(self):  
 if self.root:  
 print("Elements : "**,**end= " ")  
 self.traverseInOrder(self.root)  
  
  
 def traverseInOrder(self**,** node):  
 if node.leftChild:  
 self.traverseInOrder(node.leftChild)  
 print(node.data**,**end = " ")  
 if node.rightChild:  
 self.traverseInOrder(node.rightChild)  
  
 def search(self**,**data):  
 if self.root:  
 self.find(self.root**,**data)  
  
  
 def find(self**,** node**,**data):  
 if node.leftChild:  
 if node.data == data:  
 print("\nElement Found!")  
 f=**1** return  
 self.find(node.leftChild**,**data)  
  
 if node.rightChild:  
 if node.data == data:  
 print("\nElement Found!")  
 return  
 self.find(node.rightChild**,**data)  
  
 else:  
 print("\nElement Not Found!")  
 return

if \_\_name\_\_ == '\_\_main\_\_':  
  
 avl = AVL()  
 avl.insert(**10**)  
 avl.insert(**20**)  
 avl.insert(**5**)  
 avl.insert(**6**)  
 avl.insert(**15**)  
 avl.traverse()  
  
 avl.remove(**20**)  
 avl.remove(**15**)  
 avl.traverse()  
 avl.search(**6**)

avl.search(**7**)

****

****

**One Drive :** [**Click Me!!**](https://amritauniv-my.sharepoint.com/:f:/g/personal/sabhishek_am_students_amrita_edu/EpbzHuqZ-uxIoa0RBVcJ5boBCoZytRoFUnghOpwQECTWPQ?e=LFrEPM)

**Thankyou!!**