**Department of Information Technology**

**UIT2201— Programming and Data Structures 2022 – 2023**

**Exercise — 12**

**Part – A**

**1)Provide an implementation of Binary Search Trees with various operations of Insert, Delete, Find, Findmin and Findmax. Use Linked Binary Tree for the implementation.**

**Code:**

from LinkedBinarytree import LinkedBinaryTree

class Binarysearchtree(LinkedBinaryTree):

    def \_\_init\_\_(self,item=None,t\_left=None,t\_right=None):

        super().\_\_init\_\_(item,t\_left,t\_right)

    def Construct(self,item,pos):

        if pos is None:

            self.addRoot(item)

        elif item < pos.item :

            if pos.left is  None :

                pos.left = self.addLeft(item,pos)

            else:

                self.Construct(item,pos.left)

        elif item > pos.item:

            if pos.right is None:

                pos.right = self.addRight(item,pos)

            else:

                self.Construct(item,pos.right)

    def search(self,item,pos):

        if pos is None:

            return None

        if item < pos.item :

            return self.search(item,pos.left)

        if item > pos.item :

            return self.search(item,pos.right)

        if item ==pos.item :

            return pos

    def findmax(self,pos):

        if pos is None:

            return None

        if pos.right is not None:

            return self.findmax(pos.right)

        else:

            return pos

    def findmin(self,pos):

        if pos is None:

            return None

        if pos.left is not None:

            return self.findmin(pos.left)

        else:

            return pos

    def delete(self,item,pos):

        if pos is None:

            return None

        if item < pos.item:

            pos.left =self.delete(item,pos.left)

        elif item > pos.item :

            pos.right =self.delete(item,pos.right)

        else:

            if pos.left is None:

                temp = pos.right

                pos = None

                return temp

            elif pos.right is None:

                temp = pos.left

                pos = None

                return temp

            else:

                temp = self.findmin(pos.right)

                pos.item = temp.item

                pos.right = self.delete(temp.item,pos.right )

        return pos

b=Binarysearchtree(40)

b.Construct(50,b.root)

b.Construct(60,b.root)

b.Construct(70,b.root)

b.Construct(80,b.root)

b.Construct(90,b.root)

b.Construct(30,b.root)

b.Construct(20,b.root)

b.Construct(10,b.root)

print("Binary Search Tree")

print(b)

print("Search")

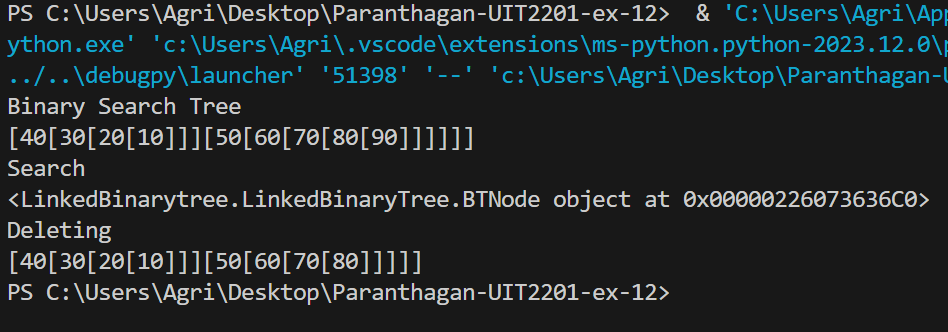
print(b.search(70,b.root))

print("Deleting")

b.delete(90,b.root)

print(b)

**Output:**

****

**2. Write a Python code to implement AVL tree, a self-balancing binary search tree. Ensure that the balance factor of the tree nodes is maintained during insertion, if not perform appropriate single rotation or double rotation to maintain the same.**

**Code:**

from LinkedBinarytree import LinkedBinaryTree

class AVLnode:

    def \_\_init\_\_(self, item):

        self.item = item

        self.left = None

        self.right = None

        self.height = 1

class AVLTree:

    def \_\_init\_\_(self):

        self.pos = None

    def insert(self, item):

        self.pos = self.\_insert(self.pos, item)

    def \_insert(self, pos, item):

        if pos is None:

            return AVLnode(item)

        elif item < pos.item:

            pos.left = self.\_insert(pos.left, item)

        else:

            pos.right = self.\_insert(pos.right, item)

        pos.height = 1 + max(self.\_get\_height(pos.left), self.\_get\_height(pos.right))

        balance\_factor = self.\_get\_balance\_factor(pos)

        if balance\_factor > 1:  # Left subtree is heavier

            if item < pos.left.item:

                return self.\_rotate\_right(pos)

            else:

                pos.left = self.\_rotate\_left(pos.left)

                return self.\_rotate\_right(pos)

        if balance\_factor < -1:  # Right subtree is heavier

            if item > pos.right.item:

                return self.\_rotate\_left(pos)

            else:

                pos.right = self.\_rotate\_right(pos.right)

                return self.\_rotate\_left(pos)

        return pos

    def \_get\_height(self, node):

        if node is None:

            return 0

        return node.height

    def \_get\_balance\_factor(self, node):

        if node is None:

            return 0

        return self.\_get\_height(node.left) - self.\_get\_height(node.right)

    def \_rotate\_right(self, z):

        y = z.left

        T3 = y.right

        y.right = z

        z.left = T3

        z.height = 1 + max(self.\_get\_height(z.left), self.\_get\_height(z.right))

        y.height = 1 + max(self.\_get\_height(y.left), self.\_get\_height(y.right))

        return y

    def \_rotate\_left(self, z):

        y = z.right

        T2 = y.left

        y.left = z

        z.right = T2

        z.height = 1 + max(self.\_get\_height(z.left), self.\_get\_height(z.right))

        y.height = 1 + max(self.\_get\_height(y.left), self.\_get\_height(y.right))

        return y

    def display(self):

        self.\_display(self.pos)

    def \_display(self, pos):

        if pos is not None:

            self.\_display(pos.left)

            print(pos.item, end=' ')

            self.\_display(pos.right)

#Driver Code

avl\_tree = AVLTree()

print("Inserting the Elements")

avl\_tree.insert(90)

avl\_tree.insert(80)

avl\_tree.insert(70)

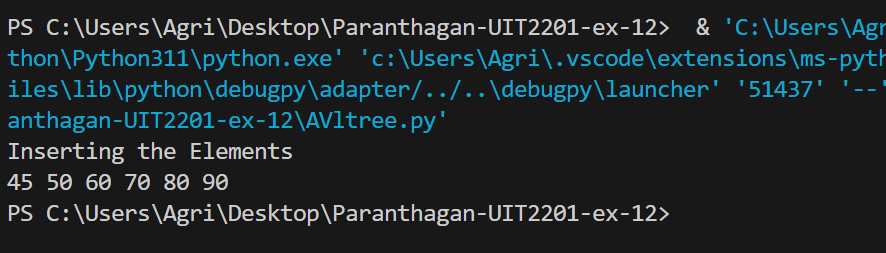
avl\_tree.insert(60)

avl\_tree.insert(50)

avl\_tree.insert(45)

avl\_tree.display()

**Output:**

****