class TreeNode(object):

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

self.data = data

self.left = left

self.right = right

self.height = 0

class AVLTree(object):

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

self.root = None

def Order(self):

if self.root:

self.order(self.root)

print()

def order(self,TreeNode):

if TreeNode:

print(TreeNode.data,end=" ")

self.order(TreeNode.left)

self.order(TreeNode.right)

def find(self, key):

if not self.root:

return None

else:

return self.\_find(key, self.root)

def \_find(self, key, node):

if not node:

return None

elif key < node.data:

return self.\_find(key, node.left)

elif key > node.data:

return self.\_find(key, node.right)

else:

return node

def findMin(self):

if self.root is None:

return None

else:

return self.\_findMin(self.root)

def \_findMin(self, node):

if node.left:

return self.\_findMin(node.left)

else:

return node

def findMax(self):

if self.root is None:

return None

else:

return self.\_findMax(self.root)

def \_findMax(self, node):

if node.right:

return self.\_findMax(node.right)

else:

return node

def height(self, node):

if node is None:

return -1

else:

return node.height

def singleLeftRotate(self, node):

k1 = node.left

node.left = k1.right

k1.right = node

node.height = max(self.height(node.right), self.height(node.left)) + 1

k1.height = max(self.height(k1.left), node.height) + 1

return k1

def singleRightRotate(self, node):

k1 = node.right

node.right = k1.left

k1.left = node

node.height = max(self.height(node.right), self.height(node.left)) + 1

k1.height = max(self.height(k1.right), node.height) + 1

return k1

def doubleRightRotate(self, node):

node.right = self.singleLeftRotate(node.right)

return self.singleRightRotate(node)

def doubleLeftRotate(self, node):

node.left = self.singleRightRotate(node.left)

return self.singleLeftRotate(node)

def insert(self, key):

if not self.root:

self.root = TreeNode(key)

else:

self.root = self.\_insert(key, self.root)

def \_insert(self, key, node):

if node is None:

node = TreeNode(key)

elif key < node.data:

node.left = self.\_insert(key, node.left)

if (self.height(node.left) - self.height(node.right)) == 2:

if key < node.left.data:

node = self.singleLeftRotate(node)

else:

node = self.doubleLeftRotate(node)

elif key > node.data:

node.right = self.\_insert(key, node.right)

if (self.height(node.right) - self.height(node.left)) == 2:

if key > node.right.data:

node = self.singleRightRotate(node)

else:

node = self.doubleRightRotate(node)

node.height = max(self.height(node.right), self.height(node.left)) + 1

return node

def delete(self, key):

if self.root is None:

raise KeyError('Error,empty tree')

else:

self.root = self.\_delete(key, self.root)

def \_delete(self, key, node):

if node is None:

raise KeyError('Error,key not in tree')

elif key < node.data:

node.left = self.\_delete(key, node.left)

if (self.height(node.right) - self.height(node.left)) == 2:

if self.height(node.right.right) >= self.height(node.right.left):

node = self.singleRightRotate(node)

else:

node = self.doubleRightRotate(node)

node.height = max(self.height(node.left), self.height(node.right)) + 1

elif key > node.data:

node.right = self.\_delete(key, node.right)

if (self.height(node.left) - self.height(node.right)) == 2:

if self.height(node.left.left) >= self.height(node.left.right):

node = self.singleLeftRotate(node)

else:

node = self.doubleLeftRotate(node)

node.height = max(self.height(node.left), self.height(node.right)) + 1

elif node.left and node.right:

if node.left.height <= node.right.height:

minNode = self.\_findMin(node.right)

node.key = minNode.key

node.right = self.\_delete(node.key, node.right)

else:

maxNode = self.\_findMax(node.left)

node.key = maxNode.key

node.left = self.\_delete(node.key, node.left)

node.height = max(self.height(node.left), self.height(node.right)) + 1

else:

if node.right:

node = node.right

else:

node = node.left

return node

T=AVLTree()

print("中序遍历:")

T.insert(5)

T.Order()

T.insert(4)

T.Order()

T.insert(6)

T.Order()

T.insert(1)

T.Order()

T.insert(3)

T.Order()

T.delete(6)

T.Order()

