```
import random
import string


class AVLTree:
    root = None


class AVLNode:
    parent = None
    leftnode = None
    rightnode = None
    key = None
    value = None
    bf = 0

    def display(self):
        print("\n\nAVL TREE (key, bf):\n")
        lines, *_ = self._display_aux()
        for line in lines:
            print(line)
        print('')

    def _display_aux(self):
        """Returns list of strings, width, height, and horizontal coordinate of the
root."""
        # No child.
        if self.rightnode is None and self.leftnode is None:
            line = f"({self.key}, {self.bf})"
            width = len(line)
            height = 1
            middle = width // 2
            return [line], width, height, middle

        # Only leftnode child.
        if self.rightnode is None:
            lines, n, p, x = self.leftnode._display_aux()
            s = f"({self.key}, {self.bf})"
            u = len(s)
            first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
            second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
            shifted_lines = [line + u * ' ' for line in lines]
            return [first_line, second_line
                    ] + shifted_lines, n + u, p + 2, n + u // 2

        # Only rightnode child.
        if self.leftnode is None:
            lines, n, p, x = self.rightnode._display_aux()
            s = f"({self.key}, {self.bf})"
            u = len(s)
            first_line = s + x * '_' + (n - x) * ' '
            second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
            shifted_lines = [u * ' ' + line for line in lines]
            return [first_line, second_line
                    ] + shifted_lines, n + u, p + 2, u // 2

        # Two children.
        left, n, p, x = self.leftnode._display_aux()
        right, m, q, y = self.rightnode._display_aux()
        s = f"({self.key}, {self.bf})"
        u = len(s)
        first_line = (x + 1) * ' ' + (n - x -
                                      1) * '_' + s + y * '_' + (m - y) * ' '
        second_line = x * ' ' + '/' + (n - x - 1 + u +
                                       y) * ' ' + '\\' + (m - y - 1) * ' '
        if p < q:
            left += [n * ' '] * (q - p)
        elif q < p:
            right += [m * ' '] * (p - q)
```

```
        zipped_lines = zip(left, right)
        lines = [first_line, second_line
                 ] + [a + u * ' ' + b for a, b in zipped_lines]
        return lines, n + m + u, max(p, q) + 2, n + u // 2


def newNode(key, val=None):
    if val is None:
        val = key
    new = AVLNode()
    new.key = key
    new.value = val
    return new


def access(AVL, key):
    def accessR(node, key):
        if node is None:
            return None

        if key < node.key:
            return accessR(node.leftnode, key)
        elif key > node.key:
            return accessR(node.rightnode, key)
        else:
            return node

    return accessR(AVL.root, key)


def insert(AVL, key, val=None):
    def insertR(node, new):
        if node is None:
            return new
        new.parent = node
        if new.key < node.key:
            node.leftnode = insertR(node.leftnode, new)
        else:
            node.rightnode = insertR(node.rightnode, new)
        return node

    new = newNode(key, val)
    AVL.root = insertR(AVL.root, new)
    return key


def delete(AVL, key):
    def deleteR(node, key):
        if node is None:
            return None
        if node.key == key:
            if node.leftnode is None and node.rightnode is None:
                return None
            elif node.leftnode is not None and node.rightnode is None:
                return node.leftnode
            elif node.leftnode is None and node.rightnode is not None:
                return node.rightnode
            else:
                temp = maximum(node.leftnode)  # or minimum(node.rightnode)
                node.key = temp.key
                node.value = temp.value
                node.leftnode = deleteR(node.leftnode, node.value)
        node.leftnode = deleteR(node.leftnode, key)
        node.rightnode = deleteR(node.rightnode, key)
        return node

    AVL.root = deleteR(AVL.root, key)
    return key
```

```
def minimum(node):
    if node.leftnode is None and node.rightnode is None:
        return node
    return minimum(node.leftnode)


def maximum(node):
    if node.leftnode is None and node.rightnode is None:
        return node
    return maximum(node.rightnode)


def traverseIn(AVL):
    def traverseInR(node, lst):
        if node is None:
            return lst
        traverseInR(node.leftnode, lst)
        lst.append(node)
        traverseInR(node.rightnode, lst)
        return lst

    return traverseInR(AVL.root, [AVL.root])


def height(node):
    if node is None:
        return 0
    return 1 + max(height(node.leftnode), height(node.rightnode))


def balanceFactor(node):
    if node is None:
        return 0
    leftH = height(node.leftnode)
    rightH = height(node.rightnode)
    return leftH - rightH


def calculateBalance(AVL):
    lst = traverseIn(AVL)
    for node in lst:
        node.bf = balanceFactor(node)
    return AVL


def traverseBreadth(root):
    def traverseBreadthR(Q, current):
        L = []
        Q.append(current)
        while len(Q) > 0:
            current = Q.pop(0)
            L.append(current)
            if current.leftnode:
                Q.append(current.leftnode)
            if current.rightnode:
                Q.append(current.rightnode)
        return L

    if root is not None:
        return traverseBreadthR([], root)


def rotateLeft(AVL, p):
    q = p.rightnode
    if q.bf == -1 or q.bf == 0:
        p.rightnode = q.leftnode
        if q.leftnode:
            q.leftnode.parent = p
```

```
        q.leftnode = p
        p.parent = q
        q.parent = None
        calculateBalance(AVL)
        return q

    elif q.bf == 1:
        p.rightnode = rotateRight(AVL, q)
        calculateBalance(AVL)
        return rotateLeft(AVL, p)


def rotateRight(AVL, p):
    q = p.leftnode

    if q.bf == 0 or q.bf == 1:
        p.leftnode = q.rightnode
        if q.rightnode:
            q.rightnode.parent = p
        q.rightnode = p
        p.parent = q
        q.parent = None
        calculateBalance(AVL)
        return q

    elif q.bf == -1:
        p.leftnode = rotateLeft(AVL, q)
        calculateBalance(AVL)
        return rotateRight(AVL, p)


def find_lowest_unbalanced(AVL):
    nodes = traverseBreadth(AVL.root)
    nodes.reverse()
    # [print(node.key, end=" ") for node in ll]
    for node in nodes:
        if abs(node.bf) == 2:
            return node
    return None


def rebalance(AVL):
    def rebalanceR(node, root):
        if node is None:
            return root

        left = None
        nodeP = node.parent
        if nodeP:
            if node.parent.leftnode is node:
                left = True
            elif node.parent.rightnode is node:
                left = False

        r = None
        if node.bf == 2:
            r = rotateRight(AVL, node)
        elif node.bf == -2:
            r = rotateLeft(AVL, node)

        if left is True:
            nodeP.leftnode = r
        elif left is False:
            nodeP.rightnode = r
        calculateBalance(AVL)
        return rebalanceR(node.parent, root)

    start = find_lowest_unbalanced(AVL)
    if start:
```

```
        AVL.root = rebalanceR(start, AVL.root)
    return AVL


def create_tree(size):
    random.seed(10)
    A = AVLTree()
    for i in range(size):
        val = random.choice(string.ascii_letters.upper())
        key = random.randint(-20, 20)
        insert(A, key, val)
    return A


AVL = create_tree(10)
AVL.root.display()


def tests():
    print("================================================\nSIMPLE ROTATION CASE")
    AVL = AVLTree
    AVL.root = newNode(10)
    insert(AVL, 50)
    insert(AVL, 30)
    AVL = calculateBalance(AVL)
    AVL.root.display()
    AVL.root = rotateLeft(AVL, AVL.root)
    # print(AVL.root.key)
    AVL = calculateBalance(AVL)
    AVL.root.display()

    print("================================================\nDOUBLE ROTATION CASE")
    AVL = AVLTree
    AVL.root = newNode(10)
    insert(AVL, 20)
    insert(AVL, 30)
    AVL = calculateBalance(AVL)
    AVL.root.display()
    AVL.root = rotateLeft(AVL, AVL.root)
    #     print(AVL.root.key)
    AVL = calculateBalance(AVL)
    AVL.root.display()

    print("================================================\nDOUBLE ROTATION CASE v2")
    AVL = AVLTree
    AVL.root = newNode(30)
    insert(AVL, 20)
    insert(AVL, 60)
    insert(AVL, 50)
    insert(AVL, 70)
    insert(AVL, 40)
    AVL = calculateBalance(AVL)
    AVL.root.display()
    AVL.root = rotateLeft(AVL, AVL.root)
    #     print(AVL.root.key)
    AVL = calculateBalance(AVL)
    AVL.root.display()
```