# Ejercicio 1

```
def rotateLeft(t, node):

    if node.rightnode.bf > 0:
        rotateRight(t, node.rightnode)

    a = node
    b = node.rightnode


    a.rightnode = None
    b.parent = a.parent
    if b.parent:
        if b.key > b.parent.key:
            b.parent.rightnode = b
        else:
            b.parent.leftnode = b
    else:
        t.root = b

    if b.leftnode:
        a.rightnode = b.leftnode
        b.leftnode.parent = a
        b.leftnode = None


    b.leftnode = a
    a.parent = b

    return b

def rotateRight(t, node):

    if node.leftnode.bf < 0:
        rotateLeft(t, node.leftnode)

    a = node
    b = node.leftnode


    a.leftnode = None
    b.parent = a.parent
    if b.parent:
        if b.key > b.parent.key:
            b.parent.rightnode = b
        else:
            b.parent.leftnode = b
    else:
        t.root = b


    if b.rightnode:
        a.leftnode = b.rightnode
        b.rightnode.parent = a
        b.rightnode = None


    b.rightnode = a
    a.parent = b

    return b
```

# Ejercicio 2

```
def calculateBalance(AVLTree):
    if not AVLTree:
        return None


    def calculateNodeBalance(node):
        if not node:
            return 0

        leftHeight = calculateNodeBalance(node.leftnode)
        rightHeight = calculateNodeBalance(node.rightnode)
        node.bf = leftHeight - rightHeight
        return max(leftHeight, rightHeight) + 1

    calculateNodeBalance(AVLTree.root)
    return AVLTree
```

# Ejercicio 3

```
def reBalanceNode(t, node):
    if not node:
        return False

    wasReBalanced = reBalanceNode(t, node.leftnode) or reBalanceNode(t,
node.rightnode)

    if wasReBalanced:
        return True

    if abs(node.bf) < 2:
        return False

    if node.bf == 2:
        return rotateRight(t, node)
    elif node.bf == -2:
        return rotateLeft(t, node)
    elif abs(node.bf) > 2:
        print(f'Node con key = {node.key} tiene un balance factor = {node.bg}
incorregible!')
        return False


def reBalance(t):
    if not t:
        return None

    calculateBalance(t)
    reBalanceNode(t, t.root)
    calculateBalance(t)
    return t
```

# Ejercicio 4

```
def _insertNode(currentNode, newNode):
    if newNode.key < currentNode.key:
        if not currentNode.leftnode:
            currentNode.leftnode = newNode
            newNode.parent = currentNode
            return newNode.key
        else: # Si lo hay, llamamos a la recursión
            return _insertNode(currentNode.leftnode, newNode)
    elif newNode.key > currentNode.key:
        if not currentNode.rightnode:
            currentNode.rightnode = newNode
            newNode.parent = currentNode
            return newNode.key
        else: # Si lo hay, llamamos a la recursión
            return _insertNode(currentNode.rightnode, newNode)
    else:
        print("Error! Ya existe un elemento para la key indicada!")
        return None


def insert(t, element, key):
    if not t:
        return None


    newNode = AVLNode()
    newNode.key = key
    newNode.value = element
    newNode.bf = 0


    if not t.root:
        t.root = newNode
        return key

    key = _insertNode(t.root, newNode)
    reBalance(t)
    return key
```

# Ejercicio 5

```
def _deleteNode(B, node):
    if not node:
        return None

        newNode = _deleteNode(B, _findSmallest(node.rightnode) or
_findLargest(node.leftnode))

    if newNode:
        newNode.leftnode = node.leftnode
        newNode.rightnode = node.rightnode
        if newNode.leftnode:
            newNode.leftnode.parent = newNode
        if newNode.rightnode:
            newNode.rightnode.parent = newNode
        newNode.parent = node.parent


    if not node.parent:
        B.root = newNode
    elif node.parent.leftnode == node:
        node.parent.leftnode = newNode
    else:
        node.parent.rightnode = newNode

    return node


def delete(B, element):
    node = _deleteNode(B, _findNodeByValue(B.root, element))
    reBalance(B)
    return node.key if node else None
```