```
================================================================================================================
UNIVERSIDAD NACIONAL DE CUYO - LCC - TP ÁRBOLES AVL

Adriano Fabris
================================================================================================================
```

Ejercicio 1:

```python
def rotateLeft(AVL, node) -> AVLNode:
    """Returns the new root"""

    nodeR = node.rightnode
    node.rightnode = nodeR.leftnode
    nodeR.leftnode = node
    node.parent = node.rightnode
    nodeR.parent = None
    refresh_parents(AVL)
    calculateBalance(AVL)
    return nodeR


def rotateRight(AVL, node) -> AVLNode:
    """Returns the new root"""

    nodeL = node.leftnode
    node.leftnode = node.leftnode.rightnode
    nodeL.rightnode = node
    node.parent = node.leftnode
    nodeL.parent = None
    refresh_parents(AVL)
    calculateBalance(AVL)
    return nodeL
```

Ejercicio 2:

```python
def height(node) -> int:
    if node is None:
        return 0
    return 1 + max(height(node.leftnode), height(node.rightnode))


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


def calculateBalance(AVL):
    """Calculates balance factors by reference, no need to reassign. O(n^2)"""

    trv_in = traverseIn(AVL)
    for node in trv_in:
        node.bf = balanceFactor(node)
    return AVL
```

Ejercicio 3:

```python
def refresh_parents(AVL) -> None:
    """Assign each node its current parent"""

    def refresh_parentsR(node):
        if node is None:
            return
```

```
        if node.leftnode is not None:
            node.leftnode.parent = node
        if node.rightnode is not None:
            node.rightnode.parent = node
        refresh_parentsR(node.leftnode)
        refresh_parentsR(node.rightnode)
        return

    return refresh_parentsR(AVL.root)


def rebalance(AVL):
    """Given a nearby-unbalanced AVL tree construction, rebalances and returns the AVL"""

    refresh_parents(AVL)
    calculateBalance(AVL)

    while True:
        low = find_low_unb(AVL)
        if low is None:
            break
        if low is AVL.root:
            AVL.root = rotateCases(AVL, low)
            refresh_parents(AVL)
            calculateBalance(AVL)
            break

        lowP = low.parent
        if lowP.leftnode is low:
            lowP.leftnode = rotateCases(AVL, low)
        else:  # lowP.rightnode is low
            lowP.rightnode = rotateCases(AVL, low)

        refresh_parents(AVL)
        calculateBalance(AVL)

    return AVL


Ejercicio 4:


def insert(AVL, key, val=None):
    """Inserts an AVLNode using the divide and conquer method, returns the current root"""

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

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


Ejercicio 5:

def maximum(node) -> AVLNode:
    """Given a Tree node, returns its maximum successor"""
    if node.leftnode is None and node.rightnode is None:
        return node
    return maximum(node.rightnode)
```

```
def delete(AVL, key):
    """Deletes an AVLNode by key, returns the current root"""

    def deleteR(node, key):
        if node is None:
            return None
        if node.key == key:
            if node.leftnode is None and node.rightnode is None:  # the node is a leaf
                return None
            elif node.leftnode is not None and node.rightnode is None:  # only has left
child
                return node.leftnode
            elif node.leftnode is None and node.rightnode is not None:  # only has right
child
                return node.rightnode
            else:
                #  has both left and right children
                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)
    rebalance(AVL)
    return AVL.root



"""
Ejercicio 6:

a) Falso: contraejemplo: insertar 5, 3, 2, 1, 4, 7, 6
b) Verdadero: ya que el bf aplica también para el penúltimo nivel, asegurando así que
todos los nodos de ese nivel tengan 2 hijos, lo propio ocurre con todos los subárboles de
menor nivel
c) Falso: el nodo desbalanceado no necesariamente deber estar justo encima, puede estar en
cualquier parte del recorrido que está por encima del nodo insertado; contraejemplo:
insertar 2, 1, 3, 4, 5
d) Verdadero: lo demostramos planteando: "todo árbol que contenga una hoja, tendrá un nodo
con bf = 0 (en particular, es el de la hoja)". Como todo árbol tiene al menos una hoja (el
caso extremo es el árbol de un solo nodo), entonces siempre existirá un nodo con bf = 0,
en particular, se cumple para los árboles AVL


Ejercicio 7:

Se calculan ambas alturas hA y hB, operaciones O(log n) por ser AVLs.
Supongamos que hA > hB, caso contrario, intercambiar A con B y B con A para el análisis.

Si abs(hA - hB) < 2:
  Se conecta A a la izquierda de X y B a su derecha, el bf de la raíz, X, es a lo sumo +1
o -1, se preserva la estructura AVL

Caso contario:
  Se baja b=hA-hB niveles por la derecha de A, el nodo que topa, E, se reemplaza por X, el
hijo por izquierda de X por E, y por derecha se enlaza a B. De esta forma se preserva el
órden de las llaves y las restricciones AVL, ya que tanto el subárbol de A que que tenía a
E de ráiz como el árbol B tienen la misma altura: luego el desbalanceo es de una sola
unidad.
"""


Ejercicio Opcional 1:

def heightAVL(AVL):
```

```
if AVL is None:
    return None

def heightAVLR(node):
    if node is None:
        return 0
    longest = node.rightnode if node.bf < 0 else node.leftnode
    return heightAVLR(longest) + 1

return heightAVLR(AVL.root) - 1 if AVL.root else 0
```