Parte 1

Importante: Los ejercicios de esta primera parte tienen como objetivo codificar las diferentes funciones básicas necesarias para la implementar un árbol AVL.

A partir de estructuras definidas como :

```
class AVLTree:
      root = None

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

Copiar y adaptar todas las operaciones del **binarytree.py** (i.e insert(), delete(), search(),etc) al nuevo módulo **avltree.py.** Notar que estos luego deberán ser implementados para cumplir que la propiedad de un árbol AVL

Ejercicio 1

Crear un modulo de nombre **avltree.py** Implementar las siguientes funciones:

rotateLeft(Tree,avlnode)
Descripción: Implementa la operación rotación a la izquierda
Entrada: Un Tree junto a un AVLnode sobre el cual se va a operar la rotación a la izquierda
Salida: retorna la nueva raíz

```python
def rotateLeft(Tree,avlnode):
    if avlnode != None:
        previusRoot = avlnode
        if avlnode.rightnode.rightnode != None:
            nodeRightRight = avlnode.rightnode.rightnode
        avlNewRoot = avlnode.rightnode
        if avlnode.rightnode.leftnode == None:
            """Caso Normal"""
            avlNewRoot.parent = avlNewRoot
            avlNewRoot.rightnode = nodeRightRight
            avlNewRoot.leftnode = previusRoot
        else:
            """Caso Especial"""
            nodeRightLeft = avlNewRoot.leftnode
            avlNewRoot.parent = avlNewRoot
            avlNewRoot.leftnode = previusRoot
            avlNewRoot.rightnode = nodeRightRight
            previusRoot.rightnode = nodeRightLeft
        return avlNewRoot.key
    else:
        return
```

rotateRight(Tree,avlnode)
Descripción: Implementa la operación rotación a la derecha
Entrada: Un Tree junto a un AVLnode sobre el cual se va a operar la

rotación a la derecha
Salida: retorna la nueva raíz

```
def rotateRight(Tree,avlnode):
    if avlnode != None:
        previusRoot = avlnode
        if avlnode.leftnode.leftnode != None:
            nodeLeftLeft = avlnode.leftnode.leftnode
        avlNewRoot = avlnode.leftnode
        if avlNewRoot.rightnode == None:
            """Caso Normal"""
            avlNewRoot.parent = avlNewRoot
            avlNewRoot.leftnode = nodeLeftLeft
            avlNewRoot.rightnode = previusRoot
        else:
            """Caso Especial"""
            nodeLeftRight = avlNewRoot.rightnode
            avlNewRoot.parent = avlNewRoot
            avlNewRoot.rightnode = previusRoot
            avlNewRoot.leftnode = nodeLeftLeft
            previusRoot.leftnode = nodeLeftRight
        return avlNewRoot.key
    else:
        return
"""-----------------------------------------------
```

Ejercicio 2

Implementar una función recursiva que calcule el elemento `balanceFactor` de cada subárbol siguiendo la siguiente especificación:

`calculateBalance(AVLTree)`
Descripción: Calcula el factor de balanceo de un árbol binario de búsqueda.
Entrada: El árbol AVL sobre el cual se quiere operar.

Salida: El árbol AVL con el valor de balanceFactor para cada subarbol

```
def calculateBalance(AVLTree):
    currentNode = AVLTree.root
    if currentNode != None:
        if currentNode.rightnode == None and currentNode.leftnode == None:
            currentNode.bf = 0
        else:
            calculateBalanceR(currentNode)
    else:
        return

def calculateBalanceR(currentNode):
    if currentNode == None:
        return 0
    heightLeft = calculateBalanceR(currentNode.leftnode)
    heightRight = calculateBalanceR(currentNode.rightnode)
    currentNode.bf = heightLeft - heightRight
    if heightLeft >= heightRight:
        return heightLeft + 1
    else:
        return heightRight + 1
```

Ejercicio 3

Implementar una funcion en el modulo avltree.py de acuerdo a las siguientes especifcaciones:

reBalance(AVLTree)

Descripción: balancea un árbol binario de búsqueda. Para esto se deberá primero calcular el **balanceFactor** del árbol y luego en función de esto aplicar la estrategia de rotación que corresponda.
Entrada: El árbol binario de tipo AVL sobre el cual se quiere operar.
Salida: Un árbol binario de búsqueda balanceado. Es decir luego de esta operación se cumple que la altura (h) de su subárbol derecho e izquierdo difieren a lo sumo en una unidad.

```
def reBalance(AVLTree):
    currentNode = AVLTree.root
    if currentNode != None:
        calculateBalance(AVLTree)
        if AVLTree.root.leftnode != None or AVLTree.root.rightnode != None:
            return reBalanceR(AVLTree, currentNode)
    else:
        return

def reBalanceR(AVLTree, currentNode):
    if currentNode.leftnode != None:
        reBalanceR(AVLTree, currentNode.leftnode)
    if currentNode.rightnode != None:
        reBalanceR(AVLTree, currentNode.rightnode)

    if currentNode.bf > 1:
        if currentNode.leftnode.bf == -1:
            rotateLeft(AVLTree, currentNode.rightnode)
            rotateRight(AVLTree, currentNode)
        else:
            rotateRight(AVLTree, currentNode)
    elif currentNode.bf < -1:
        if currentNode.rightnode.bf == 1:
            rotateRight(AVLTree, currentNode.leftnode)
            rotateLeft(AVLTree, currentNode)
        else:
            rotateLeft(AVLTree, currentNode)
    return
"""
```

Ejercicio 4:

Implementar la operación **insert()** en el módulo **avltree.py** garantizando que el árbol binario resultante sea un árbol AVL.

```python
#Insert
def insert(AVLTree, element, key):
    newNode = AVLNode()
    newNode.value = element
    newNode.key = key
    nodoAVL = insertR(AVLTree, newNode, AVLTree.root)
    """Recalcular los bf del AVLTree"""
    update_bf(AVLTree, nodoAVL)
    return newNode.key

def insertR(B, newNode, currentNode):
    if B.root == None:
        B.root = newNode
        return newNode.key
    if currentNode.key != newNode.key:
        newNode.parent = currentNode
    if newNode.key > currentNode.key:
        if currentNode.rightNode == None:
            currentNode.rightNode = newNode
            return newNode
        else:
            insertR(B, newNode, currentNode.rightNode)
    else:
        if currentNode.leftNode == None:
            currentNode.leftNode = newNode
            return newNode
        else:
            insertR(B, newNode, currentNode.leftNode)
"""
```

```python
def update_bf(AVLTree, nodoAVL):
    while nodoAVL != None:
        calculateBalanceR(nodoAVL)
        nodoAVL = nodoAVL.parent
    reBalance(AVLTree)
```

Ejercicio 5:

Implementar la operación **`delete()`** en el módulo **`avltree.py`** garantizando que el árbol binario resultante sea un árbol AVL.

```python
def deleteKey(B,key):
    if key != None:
        currentNode = recorrer_tree2(B.root, key)
        keynode = deleteKeyR(B, currentNode, key)
        reBalance(B)
        return keynode
    else:
        return

def deleteKeyR(B, currentNode, key):
    if currentNode == None:
        return None
    #Delete root
    if B.root.key == key:
        if currentNode.rightNode != None:
            keyNode = B.root.key
            B.root = currentNode.rightNode
            return keyNode
        else:
            keyNode = B.root.key
            B.root = currentNode.leftNode
            return keyNode
    #Delete hoja
    if currentNode.rightNode == None and currentNode.leftNode == None:
        if currentNode.parent.rightNode.key == key:
            keyNode = currentNode.key
            currentNode.parent.rightNode = None
            return keyNode
        else:
            keyNode = currentNode.key
            currentNode.parent.leftNode = None
```

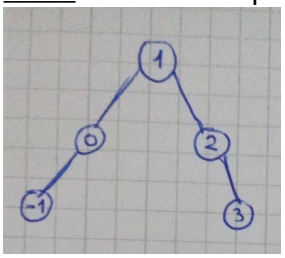
```
        return keyNode
    #Delete una rama(con subramas y hojas)
    keyNode = buscar_menorOmayorKey(currentNode, key)
    if keyNode != None:
        return keyNode
    #Delete una rama(con al menos un hijo)
    if currentNode.parent.rightNode.key == key:
        if currentNode.rightNode != None:
            keyNode = currentNode.key
            currentNode.parent.rightNode = currentNode.rightNode
            return keyNode
        else:
            keyNode = currentNode.key
            currentNode.parent.rightNode = currentNode.leftNode
            return keyNode
    else:
        if currentNode.rightNode != None:
            keyNode = currentNode.key
            currentNode.parent.leftNode = currentNode.rightNode
            return keyNode
        else:
            keyNode = currentNode.key
            currentNode.parent.leftNode = currentNode.leftNode
            return keyNode
"""-----------------------------------------------"""
```

```
#Recorrer tree2
def recorrer_tree2(currentNode, keyNode):
    if currentNode.key == keyNode:
        return currentNode
    if currentNode.key > keyNode:
        return recorrer_tree2(currentNode.leftNode, keyNode)
    else:
        return recorrer_tree2(currentNode.rightNode, keyNode)
"""-----------------------------------------------"""
#Delete una rama(con subramas y hojas)
def buscar_menorOmayorKey(currentNode, key):
    if currentNode == None:
        return
    if currentNode.rightNode.leftNode != None and currentNode.leftNode.rightNode != None:
        if currentNode.parent.rightNode.key == key:
            keyNode = currentNode.key
            currentNode.parent.rightNode = currentNode.rightNode.leftNode
            return keyNode
        else:
            keyNode = currentNode.key
            currentNode.parent.leftNode = currentNode.leftNode.rightNode
            return keyNode
```

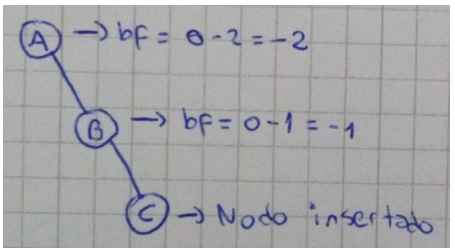
Parte 2

Ejercicio 6:

1. Responder V o F y justificar su respuesta:
   a. F En un AVL el penúltimo nivel tiene que estar completo.



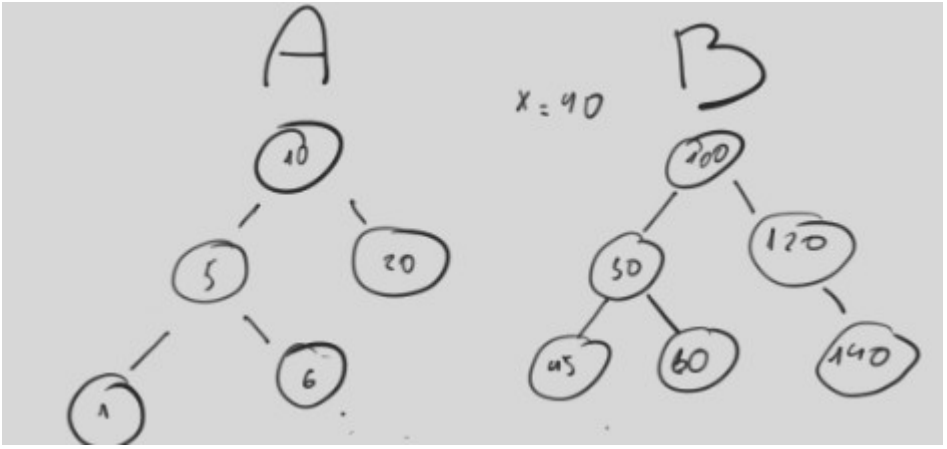
   b. V Un AVL donde todos los nodos tengan factor de balance 0 es completo.
   Es verdadero ya que si todos los nodos tienen un factor de balance de 0, entonces es porque tienen 0 o 2 hijos o los subárboles por izquierda y derecha son equivalentes en altura.
   c. F En la inserción en un AVL, si al actualizarle el factor de balance al padre del nodo insertado éste no se desbalanceó, entonces no hay que seguir verificando hacia arriba porque no hay cambios en los factores de balance.
   Es falso, supongamos que el factor de balance del padre del nodo es 1,-1 o 0. Esto no nos asegura que los ancestros de este nodo tengan un factor de balance de 1,-1 o 0. Contraejemplo:



d. __F__ En todo *AVL* existe al menos un nodo con factor de balance 0.
Esto es falso siempre y cuando no se consideren los nodos hojas.

Ejercicio 7:

Sean *A* y *B* dos *AVL* de *m* y *n* nodos respectivamente y sea *x* un key cualquiera de forma tal que para todo key $a \in A$ y para todo key $b \in B$ se cumple que $a < x < b$. Plantear un algoritmo $O(\log n + \log m)$ que devuelva un *AVL* que contenga los key de *A*, el key *x* y los key de *B*.



1er caso:
Lo primero que puede pasar es que los a∈A sean menores a x y los b∈B sean mayores a x, entonces tendríamos a un árbol con raíz x y subárbol izquierdo A y derecho B.

2do y 3er caso:
Podríamos tener a un árbol A o B talque la altura de uno sea menor a la del otro, entonces por ejemplo tomando que el árbol B es mayor en altura que A, tendríamos que colocar al árbol A dentro del árbol B como un subárbol a izquierda para no perder el balance, luego insertar x como raíz de A y un subárbol de B, viceversa para A y un subárbol B.

Ejercicio 8:

Considere una rama truncada en un AVL como un camino simple desde la raíz hacia un nodo que tenga una referencia None (que le falte algún hijo). Demuestre que la mínima longitud (cantidad de aristas) que puede tener una rama truncada en un AVL de altura h es h/2 (tomando la parte entera por abajo).

Cualquier camino desde la raíz hasta un nodo que no esté completo puede ser una rama truncada según la definición del ejercicio. Dicho nodo puede no ser necesariamente un nodo hoja.

Para calcular la mínima longitud tendríamos que ver el balance de la raíz y ver que sea bf<-1 o bf>1 ya que si es 0 podríamos asegurar que la mínima longitud = h. Si es bf<-1 quiere decir que esta desbalanceado a la izquierda por lo tanto el camino más corto estaría por ese subárbol y lo mismo pasa si es bf>1 la mínima longitud estaría a la derecha. No he podido deducir una formula para poder calcular esto ya que la cantidad de árboles y opciones es muy extensa.

Parte 3

Ejercicios Opcionales

1. Si n es la cantidad de nodos en un árbol AVL, implemente la operación **`height()`** en el módulo **`avltree.py`** que determine su altura en O(log n). Justifique el por qué de dicho orden.
2. Considere una modificación en el módulo **`avltree.py`** donde a cada nodo se le ha agregado el campo **`count`** que almacena el número de nodos que hay en el subárbol en el que él es raíz. Programe un algoritmo *O*(log *n*) que determine la cantidad de nodos en el árbol cuyo valor del key se encuentra en un intervalo [*a*, *b*] dado como parámetro. Explique brevemente por qué el algoritmo programado por usted tiene dicho orden.

A tener en cuenta:

1. **Usen lápiz y papel primero**

2. ~~**No se puede utilizar otra Biblioteca mas alla de algo1.py y las bibliotecas desarrolladas durante Algoritmos y Estructuras de Datos I.**~~

Bibliografia:

[1] Guido Tagliavini Ponce, Balanceo de arboles y arboles AVL (Universidad de Buenos Aires)
[2] Brad Miller and David Ranum, Luther College, Problem Solving with Algorithms and Data Structures using Python.