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

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

```python
def rotateLeft(T,avlnode):
    newRoot = avlnode.rightnode #asigno la nueva raiz
    newRoot.leftnode = avlnode

    if avlnode.rightnode.leftnode != None: #si la nueva raiz tiene un hijo izq:
        newRoot.leftnode.rightnode = avlnode.rightnode.leftnode  #hijo der de la vieja raiz
        newRoot.leftnode.rightnode.parent = avlnode #actualizo parent

#actualizo parent de la nueva raiz
    if avlnode.parent == None: #si la raiz vieja es la raiz
        T.root = newRoot
    else: #sino pregunto si la vieja raiz esta a la der o izq del padre
        if avlnode.parent.rightnode != avlnode:
            avlnode.parent.rightnode = newRoot
        else:
            avlnode.parent.leftnode = newRoot

    avlnode.parent = newRoot

def rotateRight(T,avlnode):
    newRoot = avlnode.leftnode  #asigno la nueva raiz
    newRoot.rightnode = avlnode

    if avlnode.leftnode.rightnode != None: #si la nueva raiz tiene un hijo derecho:
        newRoot.rightnode.leftnode = avlnode.leftnode.rightnode  #es el hijo izquierdo de la vieja raiz
        newRoot.rightnode.leftnode.parent = avlnode #actualizo parent
#actualizo parent de la nueva raiz
    if avlnode.parent == None: #si la raiz vieja es la raiz
        T.root = newRoot
    else: #sino pregunto si la vieja raiz esta a la der o izq del padre
        if avlnode.parent.rightnode == avlnode:
            avlnode.parent.rightnode = newRoot
        else:
            avlnode.parent.leftnode = newRoot
```

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 subárbol

```python
def calculateHeight(current):
    if current == None:   #CASO
        return 0          #BASE
    else:
        bf = calculateHeight(current.leftnode)-calculateHeight(current.rightnode)+1
        left = calculateHeight(current.leftnode)
        right = calculateHeight(current.rightnode)
        return bf

def calculateBalanceR(current):
    if current == None:
        return
    else:
        current.bf = calculateHeight(current.leftnode)-calculateHeight(current.leftnode)
        calculateBalanceR(current.rightnode)
        calculateBalanceR(current.leftnode)
        return

def calculateBalance(AVLTree):
    calculateBalanceR(AVLTree.root)
    return
```

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.

Ejercicio 4:

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

```python

def insertR(newNode, currentNode):
  if newNode.key > currentNode.key:
    if currentNode.rightnode == None:
      newNode.parent = currentNode
      currentNode.rightnode = newNode
      return newNode
    else:
      return insertR(newNode, currentNode.rightnode)
  elif newNode.key < currentNode.key:
    if currentNode.leftnode == None:
      newNode.parent = currentNode
      currentNode.leftnode = newNode
      return newNode
    else:
      return insertR(newNode, currentNode.leftnode)
  elif newNode.key == currentNode.key:
    return None

def insert(B,element,key):
  newNode = AVLNode()
  newNode.value = element
  newNode.key = key
  newNode.h = 0
  newNode.bf = 0
  if B.root == None:
    B.root = newNode
    return B
  else:
    node = insertR(newNode, B.root)
    if node != None:
      update_bf(B,node.parent)
  return B
```

```python
def update_bf(B,node):
    if node != None:
    #caso 1) nodo padre con un solo hijo (a la der o izq):
        if (node.rightnode != None and node.leftnode == None):
            node.h += 1
            node.bf = (node.h)
        elif (node.leftnode != None and node.rightnode == None):
            node.h += 1
            node.bf = -(node.h)
        #caso 2) nodo padre con dos hijos:
        elif (node.rightnode != None and node.leftnode != None):
            node.h = max(node.rightnode.h, node.leftnode.h)+1
            node.bf = node.leftnode.h - node.rightnode.h
        #caso 3) no existe ningun hijo:
        elif (node.rightnode == None and node.leftnode == None):
            node.h = 0
            node.bf = 0

        #verificamos si el nodo es ladeado a la derecha o izquierda
        if (node.bf < -1) or (node.bf > 1):
            reBalance(B)
        #llamo a la funcion recursiva
        update_bf(B,node.parent)
    else:
        return B
```

Ejercicio 5:

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

```python

    #DELETE
def searchKeyR(current, key):
    if (current == None):
        return None
    if (current.key == key):
        return current
    rightNode = searchKeyR(current.rightnode, key)
    if (rightNode != None):
        return rightNode
    leftNode = searchKeyR(current.leftnode, key)
    if (leftNode != None):
        return leftNode
def delete(B, element):

    key = search(B, element)
    if (key == None):
        return None
    else:
        node = searchKeyR(B.root, key)
        return deleteR(B, node)


def deleteKey(B, key):
    key = searchKeyR(B.root, key)
    if (key == None):
        return None
    else:
        node = searchKeyR(key)
        return deleteR(B,node)

```

```python
#CASO 1: el nodo a eliminar es una hoja
#CASO 2a: el nodo a eliminar tiene un hijo del lado izquierdo
#CASO 2b: el nodo a eliminar tiene un hijo en el lado derecho
#CASO 3: el nodo a eliminar tiene dos hijos

def deleteR(B, current):
    if (current == None):
        return current
    #Caso 1
    if (current.leftnode == None and current.rightnode == None):
        if (current.parent.leftnode != None and current.parent.leftnode == current):
            node = current.parent
            current.parent.leftnode = None

        if (current.parent.rightnode != None and current.parent.rightnode == current):
            node = current.parent
            current.parent.rightnode = None

        #Caso 2a
    elif (current.leftnode != None and current.rightnode == None):
        node = current.leftnode
        if (current.parent.leftnode != None and current.parent.leftnode == current):
            current.parent.leftnode = current.leftnode
        if (current.parent.rightnode != None and current.parent.rightnode == current):
            current.parent.rightnode = current.leftnode

        #Caso 2b
    elif (current.leftnode == None and current.rightnode != None):
        node = current.rightnode
        if (current.parent.leftnode != None and current.parent.leftnode == current):
            current.parent.leftnode = current.rightnode
        if (current.parent.rightnode != None and current.parent.rightnode == current):
            current.parent.rightnode = current.rightnode

        #Caso 3
        #Puedo elegir el mayor de los nodos menores o el menor de los nodos mayores
    else:
```

```python
                    """
                smallestnode = smallestNode(current.rightnode)
                node = smallestnode
                smallestnode.parent = None
                if (current.leftnode == smallestnode):
                    current.leftnode = None
                if (current.rightnode == smallestnode):
                    current.rightnode = None
                smallestnode.leftnode = current.leftnode
                smallestnode.rightnode = current.rightnode
                if (smallestnode.rightnode != None):
                    smallestnode.rightnode.parent = smallestnode
                if (smallestnode.leftnode != None):
                    smallestnode.leftnode.parent = smallestnode
                B.root = smallestnode

    #con la variable node voy guardando los nodos segun la condición para luego actualizar el bf del AVL
    #una vez eliminado el nodo, llamo a la funcion update_bf
    update_bf(B,node)  #actualizo height y bf
    return B #retorno el arbol balaceado


def biggestNode(current):
    if (current.rightnode != None):
        currentNode = biggestNode(current.rightnode)
        if (currentNode != None):
            return currentNode
    else:
        return current

def smallestNode(current):
    if (current.leftnode != None):
        currentNode = smallestNode(current.leftnode)
        if (currentNode != None):
            return currentNode
    else:
        return current
```

Lín. 287, col. 59

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

Supongamos que exitse un avl que no es completo y es un avl. Entonces existe un x que es el antepenultimo nivel y tiene un solo hijo. Entonces su bf será o 2 o -2. Por lo tanto, es una contradiccion y es F

   b. _**V**_ Un AVL donde todos los nodos tengan factor de balance 0 es completo

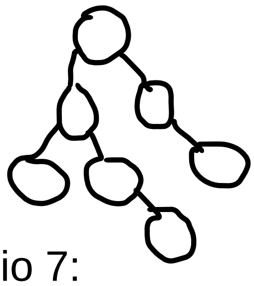
Hip: suponemos que existe un AVL donde todos los nodos tienen bf = 0 y no es completo. Por lo tanto existe un nodo que tiene un solo hijo, su bf != 0 por lo tanto es una contradicción.

   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.

En este caso se ve que el padre del nodo agregado no se ha desbalanceado pero la raiz si. Por lo tanto, si hay que seguir verificando.

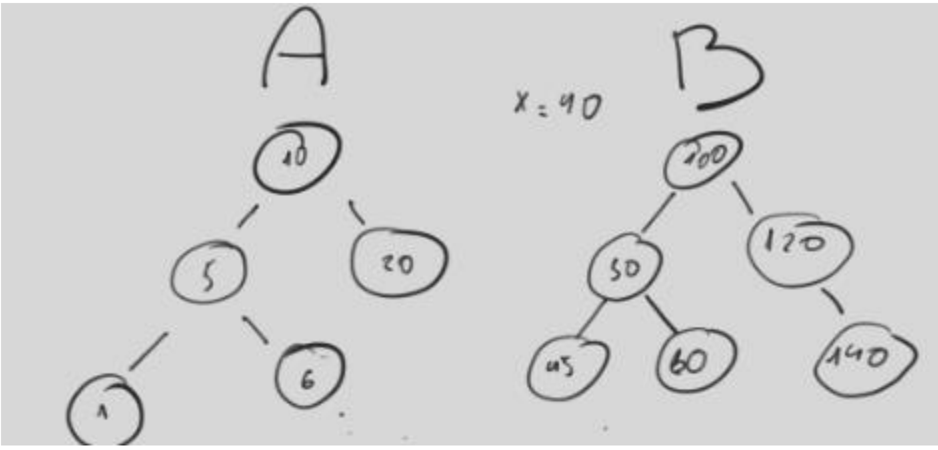
   d. _**V**_ En todo *AVL* existe al menos un nodo con factor de balance 0.

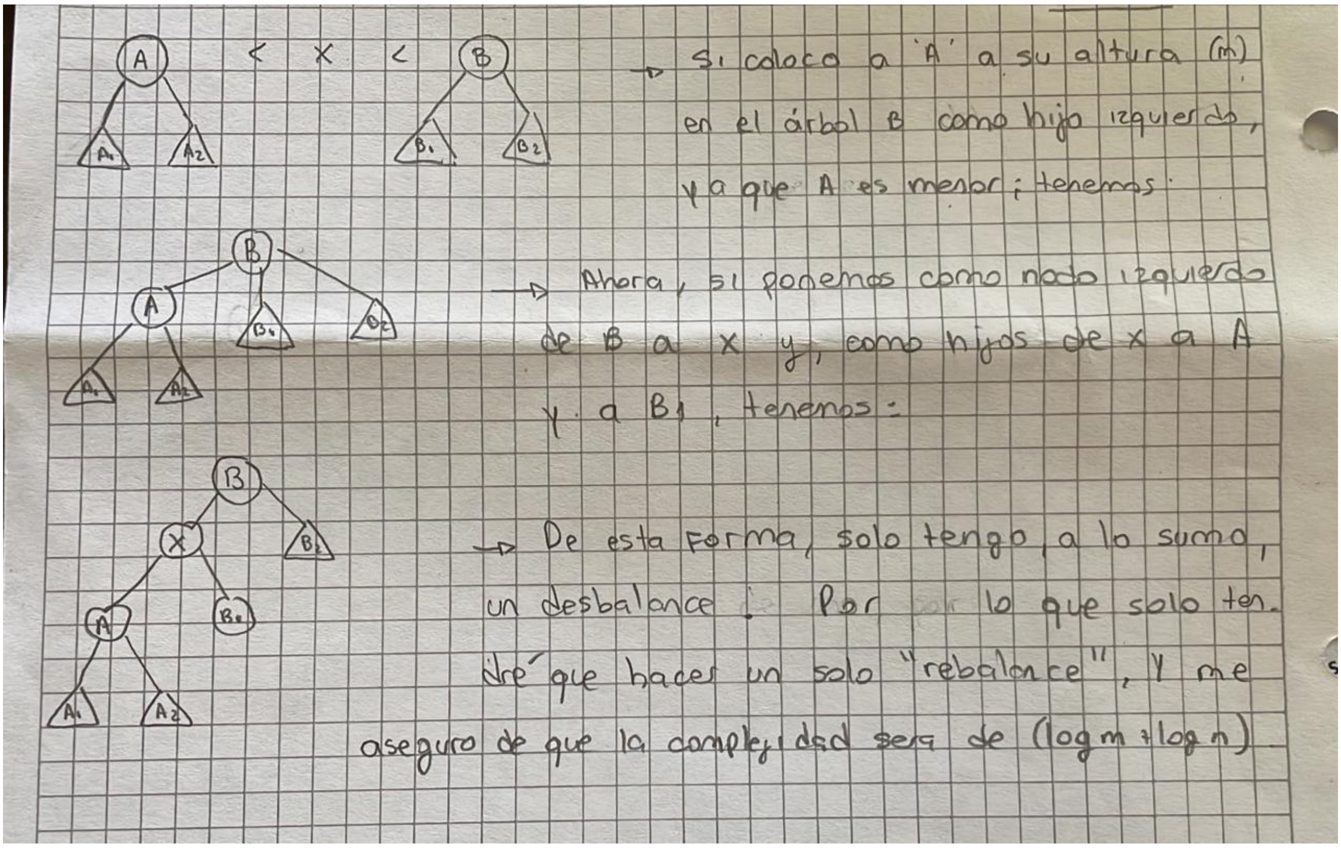
Los nodos hoja siempre tendrán bf = 0



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*.





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.

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.