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

```
class AVLTree:
    root = None

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

def search(A, element):
    return searchRecursivo(A.root, element)
```

```python
def search(A, element):
    return searchRecursivo(A.root, element)


def searchRecursivo(AVLNode, element):
    if (AVLNode == None):
        return None

    if (AVLNode.value == element):
        return AVLNode.key

    leftNode = searchRecursivo(AVLNode.leftnode, element)
    if (leftNode != None):
        return leftNode

    rightNode = searchRecursivo(AVLNode.rightnode, element)
    if (rightNode != None):
        return rightNode

def printInOrder(AVLNode):
    if AVLNode != None:
        printInOrder(AVLNode.leftnode)

        print(AVLNode.value)

        printInOrder(AVLNode.rightnode)
```

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

```
39  def rotateLeft(AVLTree, AVLNode):
40      newRoot = AVLNode.rightnode
41      AVLNode.rightnode = newRoot.leftnode
42      if AVLNode.rightnode is not None:
43          AVLNode.rightnode.parent = AVLNode
44      newRoot.leftnode = AVLNode
45      if AVLNode.parent is None:
46          AVLTree.root = newRoot
47      elif AVLNode.parent.leftnode == AVLNode:
48          AVLNode.parent.leftnode = newRoot
49      else:
50          AVLNode.parent.rightnode = newRoot
51      newRoot.parent = AVLNode.parent
52      AVLNode.parent = newRoot
53      return newRoot
54
55  def rotateRight(AVLTree, AVLNode):
56      newRoot = AVLNode.leftnode
57      AVLNode.leftnode = newRoot.rightnode
58      if AVLNode.parent == None:
59          AVLTree.root = newRoot
60      elif AVLNode.parent.leftnode == AVLNode:
61          AVLNode.parent.leftnode = newRoot
62          newRoot.parent = AVLNode.parent
63      else:
64          AVLNode.parent.rightnode = newRoot
65          newRoot.parent = AVLNode.parent
66
67      newRoot.rightnode = AVLNode
68      AVLNode.parent = newRoot
69      return newRoot
70
```

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

```python
def height(B):
    if (B.root == None):
        return 0
    return heightRecursivo(B.root)


def max(a, b):
    if (a >= b):
        return a
    return b


def heightRecursivo(node):
    if (node == None):
        return 0
    leftHeight = heightRecursivo(node.leftnode)
    rightHeight = heightRecursivo(node.rightnode)
    return max(leftHeight, rightHeight) + 1

def calculateBalance(AVLTree):
    calculateBalanceRecursivo(AVLTree.root)
    return AVLTree

def calculateBalanceRecursivo(AVLNode):
    if (AVLNode != None):
        calculateBalanceRecursivo(AVLNode.leftnode)

        if (AVLNode.leftnode != None and AVLNode.rightnode != None):
            AVLNode.bf = heightRecursivo(AVLNode.leftnode) - heightRecursivo(AVLNode.rightnode)
        elif(AVLNode.leftnode != None):
            AVLNode.bf = heightRecursivo(AVLNode.leftnode)
        elif(AVLNode.rightnode != None):
            AVLNode.bf = -heightRecursivo(AVLNode.rightnode)
        else:
            AVLNode.bf = 0

        calculateBalanceRecursivo(AVLNode.rightnode)
```

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.

```
111  def reBalance(AVLTree):
112      calculateBalance(AVLTree)
113      reBalanceRecursivo(AVLTree.root, AVLTree.root)
114      return AVLTree
115
116  def reBalanceRecursivo(AVLNode, root):
117      if(AVLNode.leftnode != None):
118          reBalanceRecursivo(AVLNode.leftnode, root)
119
120      if(AVLNode.bf == 0):
121          return AVLNode.key
122
123      A = AVLTree
124
125      if(AVLNode.bf < -1):
126          if(AVLNode.rightnode != None):
127              if AVLNode.rightnode.bf > 0:
128                  rotateRight(A, AVLNode.rightnode)
129                  rotateLeft(A, AVLNode)
130                  calculateBalanceRecursivo(root)
131
132              else:
133                  rotateLeft(A, AVLNode)
134                  calculateBalanceRecursivo(root)
135      elif(AVLNode.bf > 1):
136          if(AVLNode.leftnode != None):
137              if AVLNode.leftnode.bf < 0:
138                  rotateLeft(A, AVLNode.leftnode)
139                  rotateRight(A, AVLNode)
140                  calculateBalanceRecursivo(root)
141              else:
142                  rotateRight(A, AVLNode)
143                  calculateBalanceRecursivo(root)
144
145
146      if(AVLNode.rightnode != None):
147          reBalanceRecursivo(AVLNode.rightnode, root)
```

Ejercicio 4:

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

```
149  def insert(AVLTree, element, key):
150      newNode = AVLNode()
151      newNode.key = key
152      newNode.value = element
153
154      if (AVLTree.root == None):
155          AVLTree.root = newNode
156          return key
157      insertRecursivo(newNode, AVLTree.root)
158      reBalance(AVLTree)
159      return AVLNode.key
160
161  def insertRecursivo(newNode, currentNode):
162      if (currentNode.key > newNode.key):
163          if (currentNode.leftnode == None):
164              currentNode.leftnode = newNode
165              newNode.parent = currentNode
166              return newNode.key
167          else:
168              insertRecursivo(newNode, currentNode.leftnode)
169      elif (currentNode.key < newNode.key):
170          if (currentNode.rightnode == None):
171              currentNode.rightnode = newNode
172              newNode.parent = currentNode
173              return newNode.key
174          else:
175              insertRecursivo(newNode, currentNode.rightnode)
176
177      else:
178          return None
```

Ejercicio 5:

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

```
180   def delete(B, element):
181       key = search(B, element)
182       if (key == None):
183           return None
184       else:
185           node = searchRecursivoKey(B.root, key)
186           deleteCasos(B, node)
187           reBalance(B)
188           return node.key
189
190
191   def deleteKey(B, key):
192       key = searchRecursivoKey(B.root, key)
193       if (key == None):
194           return None
195       else:
196           node = searchRecursivoKey(key)
197           return deleteCasos(B, node)
198
199
200   def searchKey(B, key):
201       return searchRecursivoKey(B.root, key)
202
203
204   def searchRecursivoKey(node, key):
205       if (node == None):
206           return None
207       if (node.key == key):
208           return node
209       rightNode = searchRecursivoKey(node.rightnode, key)
210       if (rightNode != None):
211           return rightNode
212       leftNode = searchRecursivoKey(node.leftnode, key)
213       if (leftNode != None):
214           return leftNode
```

```
#Divido en casos para orientar mejor el código:
#Caso 1: el nodo a eliminar es una hoja
#Caso 2: el nodo a eliminar tiene un hijo del lado izquierdo
#Caso 3: el nodo a eliminar tiene un hijo en el lado derecho
#Caso 4: el nodo a eliminar tiene dos hijos


def deleteCasos(B, node):
    if (node == None):
        return node

    #Caso 1
    if (node.leftnode == None and node.rightnode == None):

        if (node.parent.leftnode != None and node.parent.leftnode == node):
            node.parent.leftnode = None

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

    #Caso 2
    elif (node.leftnode != None and node.rightnode == None):
        if (node.parent.leftnode != None and node.parent.leftnode == node):
            node.parent.leftnode = node.leftnode
        if (node.parent.rightnode != None and node.parent.rightnode == node):
            node.parent.rightnode = node.leftnode
    #Caso 3
    elif (node.leftnode == None and node.rightnode != None):
        if (node.parent.leftnode != None and node.parent.leftnode == node):
            node.parent.leftnode = node.rightnode
        if (node.parent.rightnode != None and node.parent.rightnode == node):
            node.parent.rightnode = node.rightnode
```

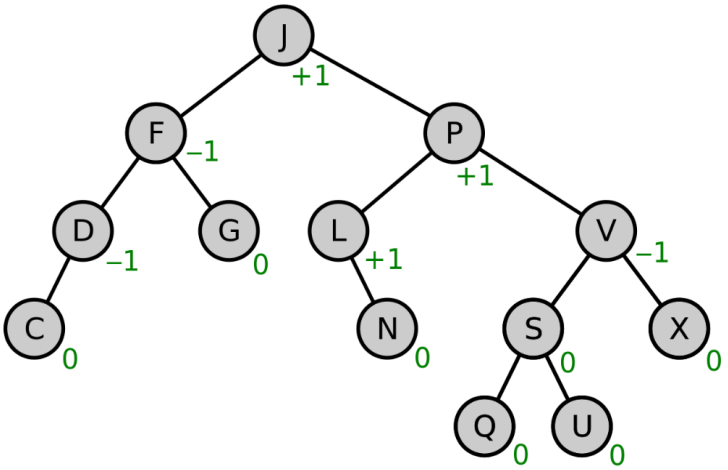
```
    #Caso 4
    #Tengo dos opciones: elegir el mayor de los nodos de la rama izquierda, o el menor de la rama derecha
    #Creo dos funciones para poder buscar estos nodos
    else:

        # biggestnode = biggestNode(node.leftnode)
        # biggestnode.parent = None
        # if(node.leftnode == biggestnode):
        #   node.leftnode = None
        # if(node.rightnode == biggestnode):
        #   node.rightnode = None
        # biggestnode.leftnode = node.leftnode
        # biggestnode.rightnode = node.rightnode
        # if(biggestnode.rightnode != None):
        #   biggestnode.rightnode.parent = biggestnode
        # if(biggestnode.leftnode != None):
        #   biggestnode.leftnode.parent = biggestnode
        # B.root = biggestnode


        smallestnode = smallestNode(node.rightnode)

        smallestnode.parent = None
        if (node.leftnode == smallestnode):
            node.leftnode = None
        if (node.rightnode == smallestnode):
            node.rightnode = None
        smallestnode.leftnode = node.leftnode
        smallestnode.rightnode = node.rightnode
        if (smallestnode.rightnode != None):
            smallestnode.rightnode.parent = smallestnode
        if (smallestnode.leftnode != None):
            smallestnode.leftnode.parent = smallestnode
        B.root = smallestnode

    return node.key

```

```
286  def biggestNode(node):
287      if (node.rightnode != None):
288          currentNode = biggestNode(node.rightnode)
289          if (currentNode != None):
290              return currentNode
291      else:
292          return node
293
294
295  def smallestNode(node):
296      if (node.leftnode != None):
297          currentNode = smallestNode(node.leftnode)
298          if (currentNode != None):
299              return currentNode
300
301      else:
302          return node
```

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



   Como podemos ver en este ejemplo, es un AVL (ya que bf = {-1, 0, 1}). Este árbol está balanceado y su penúltimo nivel no está completo.

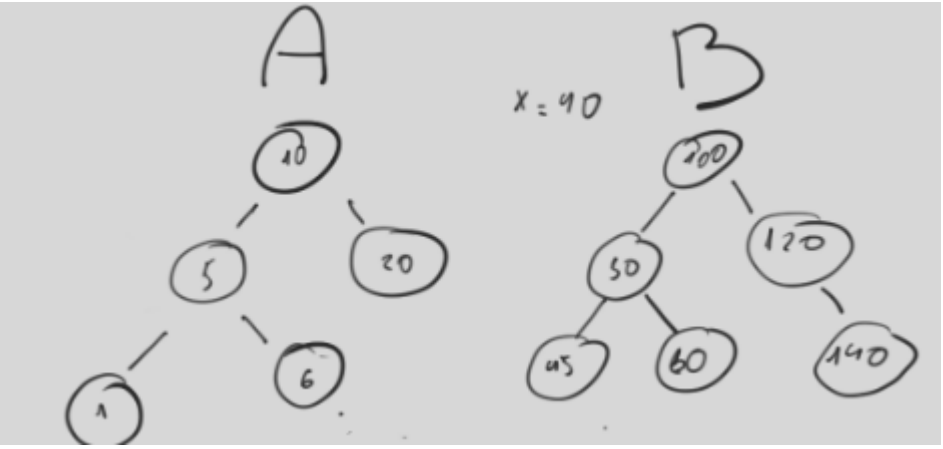
   b. **V** Un AVL donde todos los nodos tengan factor de balance 0 es completo
   Es verdadero ya que cada subárbol del árbol AVL va a tener un subárbol derecho y otro izquierdo que tengan la misma altura, caso contrario, algún nodo tendría bf != 0 y no sería completo ya que algún nodo tendría un sólo hijo.
   Suponemos que existe un AVL que tiene todos los nodos con bf = 0 y no es completo, es decir que algún nodo tiene un sólo hijo, por lo tanto su balance factor será distinto de 0, demostrando por contradicción que es V.

c. **F** En la inserción en un AVL, si al actualizar 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.

d. **F** En todo *AVL* existe al menos un nodo con factor de balance 0.
Podemos dar un contraejemplo tan simple como un AVLTree con un nodo raíz y sólamente un hijo (izquierdo o derecho, es indistinto), y el único nodo (además de la hoja) que tiene el árbol es de bf != 0 (sería -1 o 1).

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*.



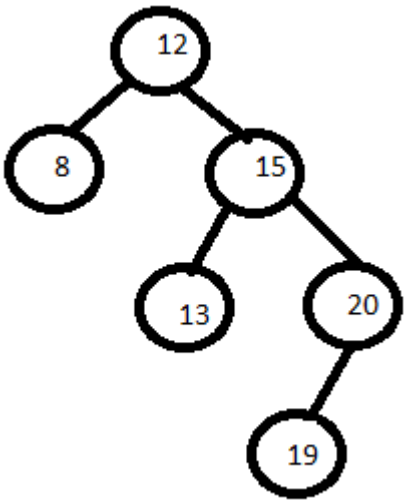
```
304  def joinAVLTreesAndKey(A, B, x):
305      newNode = AVLNode
306      newNode.key = x
307      newNode.value = x
308      alturaA = height(A)
309      alturaB = height(B)
310      if(alturaA == alturaB):
311          C = AVLTree
312          newNode.leftnode = A.root
313          newNode.rightnode = B.root
314          C.root = newNode
315          return C
316      elif(alturaA < alturaB):
317          currentNode = B.root
318          while(heightRecursivo(currentNode) != alturaA):
319              currentNode = currentNode.leftnode
320              if(heightRecursivo(currentNode) == alturaA):
321                  newNode.parent = currentNode.parent
322                  currentNode.parent = newNode
323                  newNode.leftnode = A.root
324                  newNode.rightnode = currentNode
325                  return B
326      else:
327          currentNode = A.root
328          while(heightRecursivo(currentNode) != alturaB):
329              currentNode = currentNode.rightnode
330              if(heightRecursivo(currentNode) == alturaB):
331                  newNode.parent = currentNode.parent
332                  currentNode.parent = newNode
333                  newNode.rightnode = B.root
334                  newNode.leftnode = currentNode
335                  return B
```

Ejercicio 8:

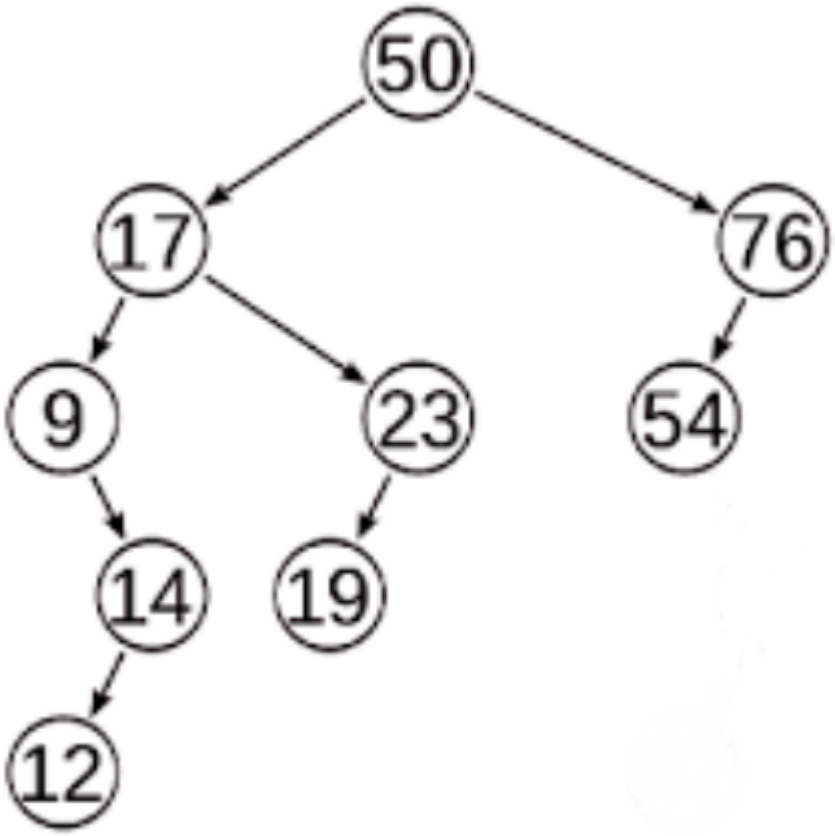
Considere una rama truncada en un AVL como un camino simple desde la raíz hacia un nodo que tenga una referencia None (que le falte algún hijo). Demuestre que la mínima longitud (cantidad de aristas) que puede tener una rama truncada en un AVL de altura h es h/2 (tomando la parte entera por abajo).

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.

Tomemos un árbol cualquiera de altura h: sabemos que, por propiedad, la diferencia de alturas entre las subramas de un árbol no puede ser mayor que 1. Suponemos que un subárbol tiene una rama truncada que tiene longitud menor a h/2, por ende el otro subárbol debe tener una altura mayor a h/2. Al ocurrir esto, la diferencia de altura será justamente mayor a 1, incumpliéndose la propiedad y por lo tanto podemos afirmar que la longitud mínima debe ser h/2.



Si tomamos este árbol, observamos que la rama izquierda está truncada, y su altura es 0 (menor a 1, que sería 3/2, tomando la parte entera inferior) y el árbol queda desbalanceado.



Y en este ejemplo la subrama derecha tiene altura 1 (menor a la mitad de h=4), y podemos observar que también se encuentra desbalanceado.

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.