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

```
def rotateLeft(Tree,avlnode):
# Función para rotar el árbol hacia la izquierda.
new_root = avlnode.rightnode
avlnode.right = new_root.leftnode
if new_root.leftnode:
new_root.leftnode.parent = avlnode
new_root.parent = avlnode.parent
if not avlnode.parent:
Tree.root = new_root
elif avlnode == avlnode.parent.leftnode:
avlnode.parent.leftnode = new_root
else:
avlnode.parent.rightnode = new_root
```

```
new_root.leftnode = avlnode
avlnode.parent = new_root
return new_root
```

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):
# Función para rotar el árbol hacia la Derecha.
new_root = avlnode.leftnode
avlnode.leftnode = new_root.rightnode
if new_root.rightnode:
new_root.rightnode.parent = avlnode
new_root.parent = avlnode.parent
if not avlnode.parent:
Tree.root = new_root
elif avlnode == avlnode.parent.rightnode:
avlnode.parent.rightnode = new_root
else:
avlnode.parent.leftnode = new_root
new_root.rightnode = avlnode
avlnode.parent = new_root
return new_root
```

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): # Implementación para calcular el balance de los nodos del árbol...
if AVLTree.root!=None:
calculate_bf(AVLTree.root)
return AVLTree

def calculate_bf(avlnode):
if avlnode !=None: # Trabaja desde el ultimo nodo hacia lariz para cada nodo hoja
avlnode.bf = bf(avlnode)
calculate_bf(avlnode.leftnode)
calculate_bf(avlnode.rightnode)
```

```
def bf(avlnode):
left_height = avlnode.leftnode.height if avlnode.leftnode else 0 # Utiliza la haltura del nodo Izquierdo
right_height = avlnode.rightnode.height if avlnode.rightnode else 0 # Utiliza la haltura del nodo Derecho
return left_height - right_height # Retorna el calculo del bf
```

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):
if AVLTree.root!=None:
recalculate_fb(AVLTree,AVLTree.root)
return AVLTree
# La funcion recalcula el Balans Factor de cada nodo empezando desde el nodo hoja hasta la raiz para cada hohja
def recalculate_fb(AVLTree,avlnode):
if avlnode !=None:
recalculate_fb(AVLTree,avlnode.leftnode)
recalculate_fb(AVLTree,avlnode.rightnode) #
avlnode.bf = bf(avlnode) # Funcion que calcula el Balans Factor
if avlnode.bf < -1 or 1 < avlnode.bf :
if avlnode.bf < 0 :
if avlnode.rightnode.bf > 0:
rotateRight(AVLTree,avlnode.rightnode)
update_height(avlnode.rightnode.rightnode)
update_count(avlnode.rightnode.rightnode)
new_root = rotateLeft(AVLTree,avlnode)
elif avlnode.bf > 0 :
if avlnode.leftnode.bf < 0:
rotateLeft(AVLTree,avlnode.leftnode)
update_count(avlnode.leftnode.leftnode)
update_height(avlnode.leftnode.leftnode)
new_root = rotateRight(AVLTree,avlnode)
update_height(avlnode)
update_count(avlnode)
recalculate_fb(AVLTree,new_root)
```

Ejercicio 4:

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

```
def insert(B,element,key):
avlnode = AVLNode()
avlnode.value = element
avlnode.key = key
avlnode.height = 0
avlnode.count = 0
avlnode.bf = 0
if B.root==None:
B.root=avlnode
return key
node = Add_Node(B.root,avlnode)
update_height(avlnode)
update_count(avlnode)
update_bf(B,avlnode)
return node.key

def Add_Node(Current,avlnode):
if Current.key > avlnode.key:
Current.count +=1
if Current.leftnode==None:
Current.leftnode=avlnode
avlnode.parent=Current
else:
return Add_Node(Current.leftnode,avlnode)
elif Current.key < avlnode.key :
Current.count +=1
if Current.rightnode==None:
Current.rightnode=avlnode
avlnode.parent=Current
else:
return Add_Node(Current.rightnode,avlnode)
return avlnode
'------------------------------------------------------------'
# Utilizado en la funciones de insercion y eliminacion y ReBalanceo
def update_bf(B, avlnode): # Esta funcio actualiza los nodos insertados y eliminados hasta la raiz
if avlnode is not None:
avlnode.bf = bf(avlnode)
if avlnode.bf < -1 or 1 < avlnode.bf:
if avlnode.bf < 0:
if avlnode.rightnode.bf > 0:
avlnode.rightnode = rotateRight(B, avlnode.rightnode)
update_height(avlnode.rightnode.rightnode)
update_count(avlnode.rightnode.rightnode)
avlnode = rotateLeft(B, avlnode)
elif avlnode.bf > 0:
if avlnode.leftnode.bf < 0:
avlnode.leftnode = rotateLeft(B, avlnode.leftnode)
update_height(avlnode.leftnode.leftnode)
```

```
update_count(avlnode.leftnode.leftnode)
avlnode = rotateRight(B, avlnode)
update_height(avlnode)
update_count(avlnode)
update_bf(B, avlnode.parent)

def update_height(node): # Esta funcion actualiza la haltura de forma recursiva hasta la rais
if node is not None:
left_height = node.leftnode.height if node.leftnode else 0
right_height = node.rightnode.height if node.rightnode else 0
node.height = max(left_height, right_height) + 1
update_height(node.parent)

def update_count(node): # La funcion actualiza la cantidad de nodos que tiene como sub arbol hasta la raiz
if node is not None:
left_count = node.leftnode.count if node.leftnode else 0
right_count = node.rightnode.count if node.rightnode else 0
node.count = left_count + right_count
update_height(node.parent)
```

Ejercicio 5:

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

```
def delete(B,element):
key=search(B,element)
if key!=None:
return deleteKey(B,key)
return None
#
def deleteKey(B, key):
if access(B, key) is not None:
B.root = delete_node(B.root, key)
return key
return None

def delete_node(node, key):
if node is None:
return node
if key < node.key:
node.leftnode = delete_node(node.leftnode, key)
elif key > node.key:
node.rightnode = delete_node(node.rightnode, key)
else:
if node.leftnode is None:
temp = node.rightnode
```

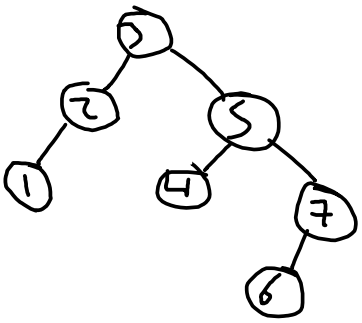
```
node = None
return temp
elif node.rightnode is None:
temp = node.leftnode
node = None
return temp
temp = min_node(node.rightnode)
node.key = temp.key
node.rightnode = delete_node(node.rightnode,temp.key)
update_height(node)
update_count(node)
update_bf(node)
return node

def min_node(node):
current = node
while current.leftnode is not None:
current = current.leftnode
return current
```

Parte 2

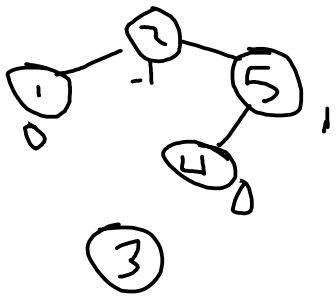
Ejercicio 6:

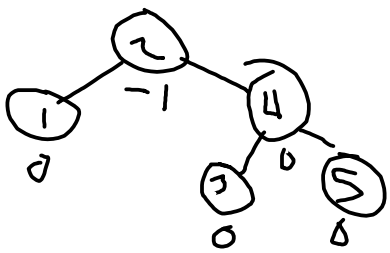
1. Responder V o F y justificar su respuesta:
   a. ___ En un AVL el penúltimo nivel tiene que estar completo
   FALSO un AVL no es necesario que el penúltimo nivel sea completo, un ejemplo contrarecicropo seria



   Si miramos en el nodo 2 que esta ante penúltimo nivel no esta completo
   b. ___ Un AVL donde todos los nodos tengan factor de balance 0 es completo
   VERDADERO Si tanto los hijos izquierdo como derecho tienen la misma altura como sub árbol entonces su bf es 0 ya que el nodo tiene 0 o 2 hijos
   c. ___ 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.
   FALSO El bf de los ancestro del nodo cambia siguiendo un camino hasta el nodo raíz
   Un contra ejemplo seria

   Insert(AVL,3)





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

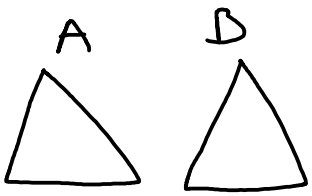
VERDADERO Todos los nodos hoja tendrá bf = 0 ya que al no tener hijos su bf siempre será 0 hasta una nueva incersion

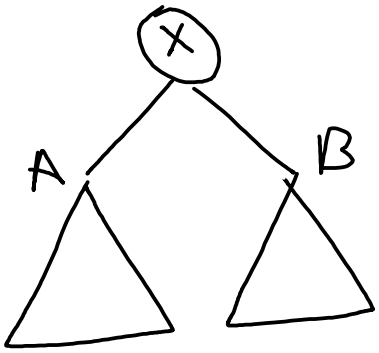
Ejercicio 7:

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*.

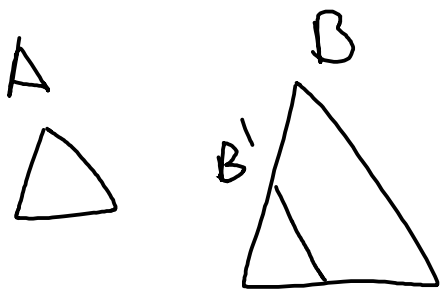
Paso 1: se calcula la hartura de los árboles A y B que tiene como complejidad O(log n) y O(log m)

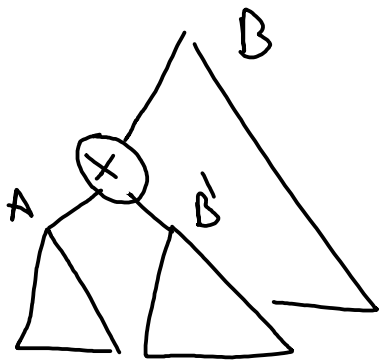
Caso 1: una vez calculado se compara si la diferencia es es de 1 entonces se coloca a x como nodo raiz y haci tener bf de 1, 0, -1



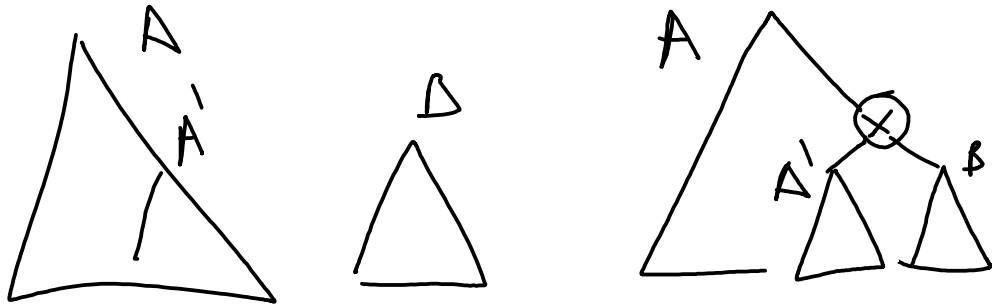


Caso 2: si la altura de B es mayor que A entonces se toma el árbol B y recorriendo a izquierda bajando desde el nodo raíz hasta la misma altura de A que llamamos B' que es menor, inserto a x teniendo como rama izquierda a A y rama derecho al sub árbol de B'





Caso 3: Si El arbola A tiene mayor altura que B entonces se desde la raíz de A se recorre a Derecha hasta llegar ha una altura equilibrada de B que llamaremos A'



Entonces dependiendo del caso, la complejidad del algoritmo solo será O(log n + log m) que se utiliza para encontrar la altura de los arboles .

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).

Para demostrar que la mínima longitud de una rama truncada en un AVL de altura h es ⌊h/2⌋, podemos usar inducción
Caso base: Cuando h=0 (es decir, el árbol está vacío), la longitud mínima de una rama truncada es 0,lo cual coincide con ⌊h/2⌋=0.

Hipótesis inductiva: Supongamos que para un árbol AVL de altura h, la mínima longitud de una rama truncada es ⌊h/2⌋.

Paso inductivo : Ahora, consideremos un árbol AVL de altura h+1. La rama truncada mas corta puede ocurrir en dos casos.

1. El hijo izquierdo esta truncado: En ese caso, La altura del hijo derecho puede ser h o h-1, ya que el árbol AVL debe estar balanceado. Si el hijo derecho tiene altura h, entonces la longitud mínima de la rama truncada es ⌊h/2⌋ + 1, ya que la rama truncada izquierdo contribuye con 1 la longitud total. En ambos casos la longitud mínima de la rama truncada es ⌊h/2⌋ +1.
2. El hijo derecho esta truncado: En este caso, la altura del hijo izquierdo es h y la longitud mínima de la rama truncada es ⌊h/2⌋ +1, ya que la rama truncada derecha contribuye con 1 a la longitud total.

   Entonces, en ambos casos, la longitud mínima de la rama en un árbol AVL de altura h+1 es ⌊h/2⌋ +1, lo que demuestra que la mínima longitud de una rama truncada en un AVL de altura h es ⌊h/2⌋

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.