Nombre y Apellido: Emiliano Germani

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

```
23  #Subprograma que realiza la rotacion de un Nodo hacia la derecha
24  def rotateRight(B,avlnode):
25    if avlnode.parent!=None: #Caso que NO es la Raiz
26      if avlnode.parent.key>avlnode.key: #Verifica la key del padre para saber si sera hijo izquierdo o derecho
27        avlnode.parent.leftnode=avlnode.leftnode
28        avlnode.leftnode.parent=avlnode.parent
29      else:
30        avlnode.parent.rightnode=avlnode.leftnode
31        avlnode.leftnode.parent=avlnode.parent
32      avlnode.parent=avlnode.leftnode #Asigno como padre del nodo a su hijo izquierdo
33      if avlnode.parent.rightnode!=None: #Caso en el que el hijo izquierdo tenia un hijo derecho
34        avlnode.leftnode=avlnode.parent.rightnode
35      else: #caso contrario se le asigna None
36        avlnode.leftnode=None
37      avlnode.parent.rightnode=avlnode
38    else: #Caso en el que es la Raiz
39      #print("es la raiz")
40      B.root=avlnode.leftnode
41      avlnode.leftnode.parent=None
42      avlnode.parent=avlnode.leftnode
43      if avlnode.parent.rightnode!=None: #Caso en el que el hijo izquierdo tenia un hijo derecho
```

Nombre y Apellido: Emiliano Germani

```
44          avlnode.leftnode=avlnode.parent.rightnode
45        else:
46          avlnode.leftnode=None
47        avlnode.parent.rightnode=avlnode
48      return B
49
50  #Subprograma que realiza la rotacion de un Nodo hacia la izquierda
51  def rotateLeft(B,avlnode):
52    if avlnode.parent != None: #Caso que no es la Raiz
53      if avlnode.parent.key>avlnode.key: #Verifica la key del padre para saber si sera hijo
    izquierdo o derecho
54          avlnode.parent.leftnode=avlnode.rightnode
55          avlnode.rightnode.parent=avlnode.parent
56      else:
57          avlnode.parent.rightnode=avlnode.rightnode
58          avlnode.rightnode.parent=avlnode.parent
59        avlnode.parent=avlnode.rightnode #Asigno como padre del nodo a su hijo derecho
60      if avlnode.parent.leftnode!=None: #Caso en el que el hijo derecho tenia un hijo
    izquierdo
61          avlnode.rightnode=avlnode.parent.leftnode
62      else: #caso contrario se le asigna None
63          avlnode.rightnode=None
64        avlnode.parent.leftnode=avlnode
65    else: #Caso en el que es la Raiz
66        B.root=avlnode.rightnode
```

```
67        avlnode.rightnode.parent=None
68        avlnode.parent=avlnode.rightnode
69      if avlnode.parent.leftnode!=None: #Caso en el que el hijo derecho tenia un hijo
    izquierdo
70          avlnode.rightnode=avlnode.parent.leftnode
71      else:
72          avlnode.rightnode=None
73        avlnode.parent.leftnode=avlnode
74    return B
```

https://replit.com/@EmilianoGermani/Trabajo-Practico-2-Arbol-Balanceado-AVL

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

Nombre y Apellido: Emiliano Germani

```python
#Recurrencia para calcular el BalanceFactor de un Arbol
def BalanceFactor(currentNode,height):
  if currentNode.leftnode==None and currentNode.rightnode==None: #Si no tiene hijos el balanceFactor es 0
    currentNode.bf=0
    height=height+1
    return height
  else:
    if currentNode.leftnode!=None: #Devuelve la altura del hijo izquierdo
      heightLeft=BalanceFactor(currentNode.leftnode,height)
    else:
      heightLeft=0
    if currentNode.rightnode!=None: #Devuelve la altura del hijo derecho
      heightRight=BalanceFactor(currentNode.rightnode,height)
    else:
      heightRight=0
    currentNode.bf=heightLeft - heightRight #Calcula el BalanceFactor del CurrentNode
    if heightLeft>heightRight: #Retorna la altura mas alta
      return heightLeft+1
    else:
      return heightRight+1
```

```python
#Subprograma que calcular el BalanceFactor de un arbol
def calculateBalance(AVLTree):
  if AVLTree.root==None:
    return None
  else:
    BalanceFactor(AVLTree.root,0)
    return AVLTree
```

https://replit.com/@EmilianoGermani/Trabajo-Practico-2-Arbol-Balanceado-AVL

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.

Nombre y Apellido: Emiliano Germani

```python
#Subprograma recursivo que realiza el balanceo del arbol B en caso de necesitar realizar rotacion para balancearlo
def Balance(B,currentNode):
  if currentNode.parent.bf>1: #Caso que el padre este inclinado hacia IZQUIERDA
    if currentNode.bf<0: #Caso que el nodo a rotar tenga inclinacion a DERECHA
      B=rotateLeft(B,currentNode)
      cambio=currentNode.bf
      currentNode.bf=currentNode.parent.bf*(-1)
      currentNode.parent.bf=cambio*(-1)
      currentNode=currentNode.parent
    B=rotateRight(B,currentNode.parent)
    currentNode.bf=currentNode.bf-1
    if currentNode.rightnode.leftnode==None and currentNode.rightnode.rightnode!=None: #Si el nuevo nodo padre NO tenia hijo derecho
      currentNode.rightnode.bf=currentNode.rightnode.bf-2
    elif currentNode.rightnode.leftnode!=None and currentNode.rightnode.rightnode==None: #Si el nuevo nodo padre tenia hijo derecho
      currentNode.rightnode.bf=currentNode.rightnode.bf-1
    else:
      currentNode.rightnode.bf=0
  elif currentNode.parent.bf<-1: #Caso que el padre este inclinado hacia DERECHA
    if currentNode.bf>0: #Caso que el nodo a rotar tenga inclinacion a IZQUIERDA
      B=rotateRight(B,currentNode)
      cambio=currentNode.bf
      currentNode.bf=currentNode.parent.bf*(-1)
      currentNode.parent.bf=cambio*(-1)
      currentNode=currentNode.parent
    B=rotateLeft(B,currentNode.parent)
    currentNode.bf=currentNode.bf+1
    if currentNode.leftnode.rightnode==None and currentNode.leftnode.leftnode!=None: #Si el nuevo nodo padre NO tenia hijo izquierdo
      currentNode.leftnode.bf=currentNode.leftnode.bf+2
    elif currentNode.leftnode.rightnode!=None and currentNode.leftnode.leftnode==None: #Si el nuevo nodo padre tenia hijo izquierdo
      currentNode.leftnode.bf=currentNode.leftnode.bf+1
    else:
      currentNode.leftnode.bf=0

def searchNoBalance(currentNode):
  if currentNode!=None:
    if currentNode.bf>1:
      return currentNode.leftnode
    elif currentNode.bf<-1:
      return currentNode.rightnode
    else:
      searchNoBalance(currentNode.leftnode)
      searchNoBalance(currentNode.rightnode)

def reBalance(AVLTree):
  if AVLTree==None:
    return None
  elif AVLTree.root==None:
    return None
  else:
    AVLTree=calculateBalance(AVLTree)
    Node=searchNoBalance(AVLTree.root)
    if Node!=None:
      Balance(AVLTree,Node)
```

https://replit.com/@EmilianoGermani/Trabajo-Practico-2-Arbol-Balanceado-AVL

Ejercicio 4:

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

Nombre y Apellido: Emiliano Germani

```
#Subprograma que realiza los cambios del Balance Factor y las rotaciones en caso de ser necesario
def BFchange(B,currentNode):
  if currentNode.parent!=None:
    if currentNode.parent.bf==0:
      if currentNode.parent.key>currentNode.key:
        currentNode.parent.bf=1
        BFchange(B,currentNode.parent)
      elif currentNode.parent.key<currentNode.key:
        currentNode.parent.bf=-1
        BFchange(B,currentNode.parent)
    elif currentNode.parent.bf==1:
      if currentNode.parent.key>currentNode.key:
        currentNode.parent.bf=2
        Balance(B,currentNode)
      elif currentNode.parent.key<currentNode.key:
        currentNode.parent.bf=0
        return
    else:
      if currentNode.parent.key>currentNode.key:
        currentNode.parent.bf=0
        return
      elif currentNode.parent.key<currentNode.key:
        currentNode.parent.bf=-2
        Balance(B,currentNode)


#Parte recursiva del subprograma insert
def insertTree(B,currentNode,NewNode):
  if currentNode.key<NewNode.key:
    if currentNode.rightnode==None:
      currentNode.rightnode=NewNode
      NewNode.parent=currentNode
      BFchange(B,NewNode)
      return NewNode.key
    else:
      return insertTree(B,currentNode.rightnode,NewNode)
  elif currentNode.key>NewNode.key:
    if currentNode.leftnode==None:
      currentNode.leftnode=NewNode
      NewNode.parent=currentNode
      BFchange(B,NewNode)
      return NewNode.key
    else:
      return insertTree(B,currentNode.leftnode,NewNode)

#Subprograma que recibe un arbol binario, un elemento y una key, se encarga de ingresar dicho nodo con el elemento y la key ingresada dentro del arbol binario, retorna la misma key si ha logrado ingresarse
def insert(B,element,key):
  if B.root==None:
    B.root=AVLNode()
    B.root.value=element
    B.root.key=key
    B.root.bf=0
    return key
  else:
    NewNode=AVLNode()
    NewNode.value=element
    NewNode.key=key
    NewNode.bf=0
    return insertTree(B,B.root,NewNode)
```

Nombre y Apellido: Emiliano Germani

https://replit.com/@EmilianoGermani/Trabajo-Practico-2-Arbol-Balanceado-AVL

Ejercicio 5:

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

```
189  #Subprograma que busca el Menor de sus Mayores, desvinculandolo y devolviendo dicho nodo para ser
     reemplazado
190  def MenorDeMayores(currentNode):
191    if currentNode.leftnode!=None:
192      return MenorDeMayores(currentNode.leftnode)
193    if currentNode.rightnode!=None:
194      currentNode.parent.leftnode=currentNode.rightnode
195      currentNode.rightnode.parent=currentNode.parent
196    else:
197      if currentNode.parent.key<currentNode.key:
198        currentNode.parent.rightnode=None
199      else:
200        currentNode.parent.leftnode=None
201    return currentNode
202
203  #Subprograma que es la parte recursiva a la hora de buscar el nodo a eliminar para el subprograma
     DELETE y DELETEKEY
204  def deleteNodeTree(B,currentNode,key):
205    if currentNode.key==key:
206      if currentNode.rightnode==None and currentNode.leftnode==None: #Caso que el nodo a eliminar no
     tenga hijos
207        if currentNode.parent==None:
208          B.root=None
209        else:
```

```
210          if currentNode.parent.key>key:
211            currentNode.parent.leftnode=None
212            currentNode.parent=None
213          else:
214            currentNode.parent.rightnode=None
215            currentNode.parent=None
216        return key
217      elif currentNode.rightnode!=None and currentNode.leftnode==None: #Caso que el nodo a eliminar solo
     tenga hijo derecho
218        if currentNode.parent==None:
219          B.root=currentNode.rightnode
220          B.root.parent=None
221        else:
222          if currentNode.parent.key>key:
223            currentNode.parent.leftnode=currentNode.rightnode
224            currentNode.rightnode.parent=currentNode.parent
225            currentNode.parent=None
226          else:
227            currentNode.parent.rightnode=currentNode.rightnode
228            currentNode.rightnode.parent=currentNode.parent
229            currentNode.parent=None
230        return key
231      elif currentNode.rightnode==None and currentNode.leftnode!=None: #Caso que el nodo a eliminar solo
     tenga hijo izquierdo
```

Nombre y Apellido: Emiliano Germani

```
232         if currentNode.parent==None:
233           B.root=currentNode.leftnode
234           B.root.parent=None
235         else:
236           if currentNode.parent.key>key:
237             currentNode.parent.leftnode=currentNode.leftnode
238             currentNode.leftnode.parent=currentNode.parent
239             currentNode.parent=None
240           else:
241             currentNode.parent.rightnode=currentNode.leftnode
242             currentNode.leftnode.parent=currentNode.parent
243             currentNode.parent=None
244         return key
245       elif currentNode.rightnode!=None and currentNode.leftnode!=None: #Caso que el nodo a eliminar
    tenga 2 hijos
246         NewNode=MenorDeMayores(currentNode.rightnode)
247         NewNode.rightnode=currentNode.rightnode
248         if currentNode.rightnode!=None:
249           currentNode.rightnode.parent=NewNode
250         NewNode.leftnode=currentNode.leftnode
251         if currentNode.leftnode!=None:
252           currentNode.leftnode.parent=NewNode
253         if currentNode.parent==None:
254           NewNode.parent=currentNode.parent
255           B.root=NewNode
```

```
256           B.root.parent=None
257         else:
258           if currentNode.parent.key>key:
259             currentNode.parent.leftnode=NewNode
260             NewNode.parent=currentNode.parent
261             currentNode.parent=None
262           else:
263             currentNode.parent.rightnode=NewNode
264             NewNode.parent=currentNode.parent
265             currentNode.parent=None
266         return key
267
268     elif currentNode.key>key:
269       return deleteNodeTree(B,currentNode.leftnode,key)
270     else:
271       return deleteNodeTree(B,currentNode.rightnode,key)
```

```
273   #Subprograma que recibe un arbol binario y un elemento, elimina el nodo que contenga a dicho elemento
    y lo reemplaza por el Menor de su Mayores
274   def delete(B,element):
275     key=search(B,element)
276     if key!=None:
277       deleteNodeTree(B,B.root,key)
278       return reBalance(B)
279     return None
```

https://replit.com/@EmilianoGermani/Trabajo-Practico-2-Arbol-Balanceado-AVL

Parte 2

Ejercicio 6:

1. Responder V o F y justificar su respuesta:
   a. ___ En un AVL el penúltimo nivel tiene que estar completo
   b. ___ Un AVL donde todos los nodos tengan factor de balance 0 es completo
   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.
   d. ___ En todo *AVL* existe al menos un nodo con factor de balance 0.

a) Falso, porque puede tener un árbol con un lado de altura 4 y otro de altura 3

Nombre y Apellido: Emiliano Germani

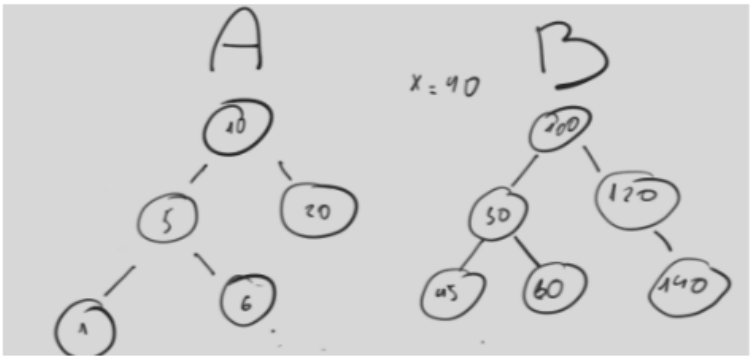
b) Verdadero, porque para que todos los nodos tengan un factor de balance 0 entonces el árbol esta balanceado

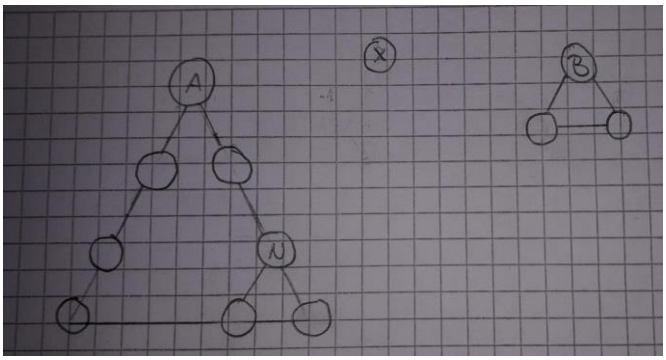
c) Falso, porque puede haber una raíz con un factor de balance 1 y todos sus hijos derechos balance 0, entonces al agregarles un nodo mas no pierde el balance en los hijos pero si en la raíz

d) Verdadero, los últimos nodos al no tener hijos estan con un factor de balance 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*.

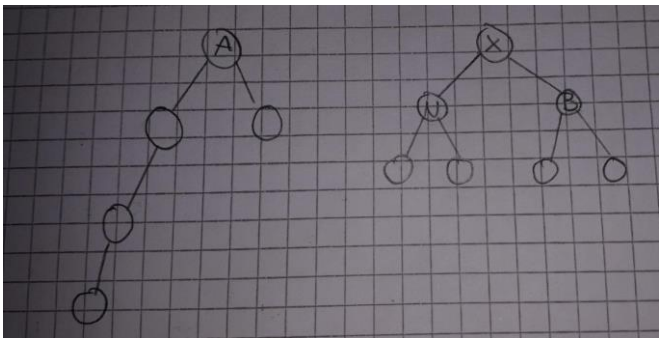




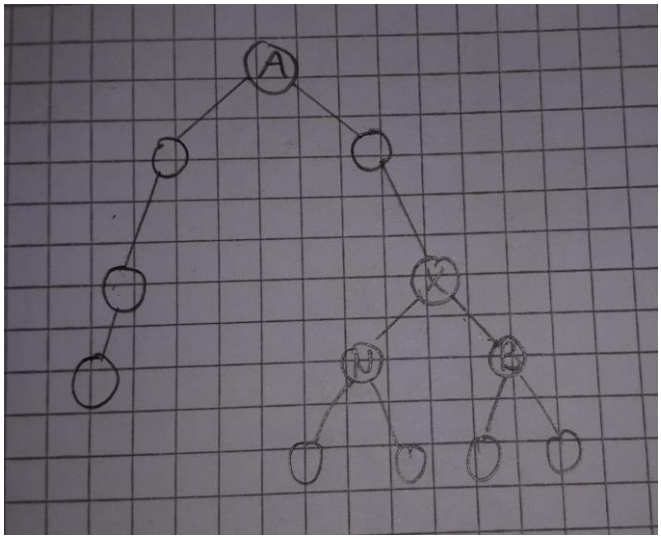
Se observa primero que teniendo dos arboles AVL y un valor X (siendo la key de X mas grande que todos los nodos de A y mas chico que todos los nodos de B)

Lo primero que se realiza es verificar la altura de ambos nodos, evaluando a partir de los BalanceFactor cual lado es el mas alto (de esta forma solo se evalua un lado del árbol siendo (Log n))

Nombre y Apellido: Emiliano Germani



Una ves que se sabe cual es el nodo de menor altura, se saca el nodo del árbol mas alto que tenga la misma altura que el árbol mas chico.



Ambos nodos se insertan como hijos del valor X y seguidamente el nuevo árbol con raíz X se inserta como hijo del árbol mas alto

Dicha operación es (Log n + Log m) porque solo se necesito evaluar la altura de un lado del árbol en ambos casos

Ejercicio 8:

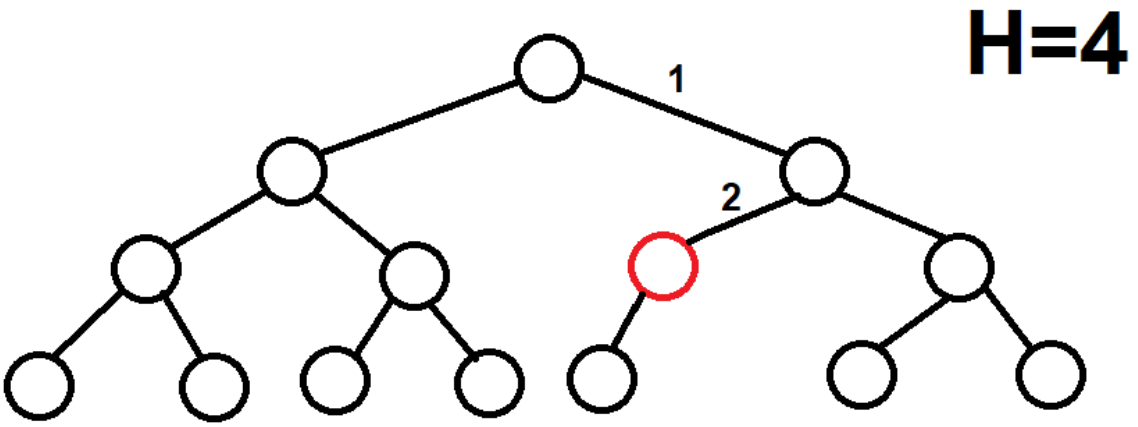
Considere una rama truncada en un AVL como un camino simple desde la raíz hacia un nodo que tenga una referencia None (que le falte algún hijo). Demuestre que la mínima longitud (cantidad de aristas) que puede tener una rama truncada en un AVL de altura h es h/2 (tomando la parte entera por abajo).

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.

Al evaluar la opcion de que el arbol esta balanceado, la única forma de que tenga una rama truncada seria posible si el penultimo nodo le faltase un hijo, ya que si algun nodo anterior al penultimo le faltara un hijo el arbol no estaría balanceado.

Entonces teniendo encuenta que a un penultimo hijo le falta un nodo, ese nodo tendría una altura de H-1, si el nodo fuese de altura h=4, entonces el nodo que le falta un hijo estaría en altura h=3, por lo que tomando la longitud como cantidad de aristas desde la raiz hasta ese nodo, seria una longitud de 2 aristas, que es el resultado de dividir h/2.

Nombre y Apellido: Emiliano Germani



Parte 3

Ejercicios

1. Si n es la cantidad de nodos en un árbol AVL, implemente la operación **`height()`** en el modulo **`avltree.py`** que determine su altura en O(log n). Justifique el por qué de dicho orden.

```
38  #Subprograma que busca los nodos con la altura mas grande
39  def treeheight(currentNode,cont):
40    if currentNode!=None:
41      #print ("Entro con el valor ",currentNode.value)
42      #print ("El contador es ",cont)
43      if currentNode.balanceFactor==1:
44        cont=cont+1
45        cont=treeheight(currentNode.leftnode,cont)
46      elif currentNode.balanceFactor==-1:
47        cont=cont+1
48        cont=treeheight(currentNode.rightnode,cont)
49      else:
50        if currentNode.leftnode==None and
    currentNode.rightnode==None:
51          return cont
52        else:
53          cont=cont+1
54          cont=treeheight(currentNode.rightnode,cont)
55    return cont
56
57  #Subprograma que calcula la altura del arbol
58  def height(AVLTree):
59    cont=0
60    cont=treeheight(AVLTree.root,cont)
```

Para hallar la altura del arbol con una complejidad de O(Log n) se evalua el BalanceFactor del nodo, si el balanceFactor es igual a 1, entonces se mueve al nodo izquierdo, ya que sabe que hay mas nodos de ese lado, que del lado derecho, y si el balanceFactor fuese -1 seria lo mismo pero moviéndose al nodo Derecho, ya que sabe que se encuentra el nodo de mayor altura de ese lado.

Nombre y Apellido: Emiliano Germani

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.

Teniendo como parámetro un intervalo de valores KEY lo primero que se hace es buscar el valor mínimo de key dentro del árbol, una ves encontrado el nodo que contiene la KEY mas chica dentro del intervalo, se realiza lo mismo para hallar la key mas grande del intervalo en el árbol. Una ves hallados el valor mínimo y máximo de key dentro del árbol, solo se busca cuantos nodos hay dentro de ellos, ignorando los nodos que se encuentren fuera del intervalo de keys.