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

```
14  #rotacion izq
15  def rotateLeft(Tree,avlnode):
16      new_root = avlnode.rightnode
17      avlnode.rightnode = new_root.leftnode
18      if new_root.leftnode != None:
19          new_root.leftnode.parent = avlnode
20      new_root.parent = avlnode.parent
21      if avlnode.parent == None:
22          Tree.root = new_root
23      else:
24          if avlnode.parent.leftnode == avlnode:
25              avlnode.parent.leftnode = new_root
26          else:
27              avlnode.parent.leftnode = new_root
28      new_root.leftnode = avlnode
29      avlnode.parent = new_root.leftnode
30
```

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

```
31  #rotacion derecha
32  def rotateRight(Tree,avlnode):
33      new_root = avlnode.leftnode
34      avlnode.leftnode = new_root.rightnode
35      if new_root.rightnode != None:
36          new_root.rightnode.parent = avlnode
37      new_root.parent = avlnode.parent
38      if avlnode.parent == None:
39          Tree.root = new_root
40      else:
41          if avlnode.parent.rightnode == avlnode:
42              avlnode.parent.rightnode = new_root
43          else:
44              avlnode.parent.leftnode = new_root
45      new_root.rightnode = avlnode
46      avlnode.parent = new_root.rightnode
```

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

```
48  # calcular altura del nodo
49  def altura(node):
50      if node != None:
51          return (1+max(altura(node.leftnode),altura(node.rightnode)))
52      else:
53          return 0
54
55  #calcular bf de cada nodo
56  def calculateBalance(AVLTree):
57      node = AVLTree.root
58      if node != None:
59          calculateBalance_node(node)
60      return
61
62  def calculateBalance_node(node):
63      if node != None:
64          node.bf = altura(node.leftnode)-altura(node.rightnode)
65          calculateBalance_node(node.leftnode)
66          calculateBalance_node(node.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.

```
68   #buscamos nodo desbalanceado para rebalancear el arbol
69   def find_node_desbalance(AVLTree):
70       calculateBalance_node(AVLTree)
71       node = AVLTree.root
72       if node != None:
73           return(find_node_desbalanceR(node))
74
75   def find_node_desbalanceR(node):
76       if node != None:
77           if node.bf != 1 and node.bf != 0 and node.bf != -1:
78               return node
79           find_node_desbalanceR(node.leftnode)
80           find_node_desbalanceR(node.rightnode)
```

```
82   #rebalanceamos el arbol
83   def reBalance(AVLTree):
84       bf_arbol = calculateBalance_node(AVLTree.root)
85       if bf_arbol != 1 and bf_arbol != 0 and bf_arbol != -1:
86           node = find_node_desbalance(AVLTree)
87           reBalanceR(AVLTree,node)
88
89   def reBalanceR(AVLTree,node):
90       if node.bf < -1:
91           if node.rightnode.bf == 1:
92               rotateRight(AVLTree,node.rightnode)
93               rotateLeft(AVLTree, node)
94           else:
95               rotateLeft(AVLTree, node)
96       if node.bf > 1:
97           if node.leftnode.bf == -1:
98               rotateLeft(AVLTree, node.leftnode)
99               rotateRight(AVLTree, node)
100          else:
101              rotateRight(AVLTree, node)
```

Ejercicio 4:

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

```
103  #insert binarytree modificado para AVLT
104  def insertR(B,newnode, currentnode):
105      if newnode.key>currentnode.key:
106          if currentnode.rightnode==None:
107              newnode.parent=currentnode
108              currentnode.rightnode=newnode
109              #cuando ya insertamos tenemos que chequear que el arbol siga siendo AVL
110              node_desbalance=check_balance_parent(newnode)
111              if node_desbalance != None:
112                  reBalanceR(B, node_desbalance)
113              return newnode.key
114          else:
115              return insertR(newnode, currentnode.rightnode)
116      elif newnode.key<currentnode.key:
117          if currentnode.leftnode==None:
118              newnode.parent=currentnode
119              currentnode.leftnode=newnode
120              #chequeamos solo subiendo en el arbol
121              node_desbalance=check_balance_parent(newnode)
122              if node_desbalance != None:
123                  reBalanceR(B, node_desbalance)
124
125              return newnode.key
126          else:
127              return insertR(newnode, currentnode.leftnode)
128      else:
129          return None
```

```
136  def check_balance_parent(node):
137      while node != None:
138          calculateBalance_node(node)
139          if node.bf >1 or node.bf <-1:
140              return node
141          node=node.parent
142      return None
143
144  def insert(B,element,key):
145      newnode=AVLNode()
146      newnode.value=element
147      newnode.key=key
148      currentnode=B.root
149      if B.root==None:
150          B.root=newnode
151      else:
152          return insertR(B,newnode, currentnode)
```

Ejercicio 5:

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

```
149   #Funcion delete
150   def deleteR(B,deletingnode):
151       if deletingnode!=None:
152           #Caso 1: Hoja
153           if deletingnode.leftnode==None and deletingnode.rightnode==None:
154               if deletingnode==B.root:
155                   B.root=None
156                   return deletingnode.key
157               else:
158                   padre=deletingnode.parent
159                   #Me fijo si el nodo a eliminar es el hijo derecho o izquierdo de su padre
160                   if padre.rightnode==deletingnode:
161                       padre.rightnode=None
162                       node_desbalance=check_balance_parent(padre)
163                       if node_desbalance != None:
164                           reBalanceR(B, node_desbalance)
165                       return deletingnode.key
166                   else:
167                       padre.leftnode=None
168                       node_desbalance=check_balance_parent(padre)
169                       if node_desbalance != None:
170                           reBalanceR(B, node_desbalance)
171                       return deletingnode.key
```

```
172           #Caso 2: Un solo hijo
173           #Si es el hijo izquierdo
174           elif deletingnode.rightnode==None:
175               if deletingnode==B.root:
176                   B.root=deletingnode.leftnode
177                   return deletingnode.key
178               else:
179                   padre=deletingnode.parent
180                   #Me fijo si el nodo a eliminar es el hijo derecho o izquierdo de su padre
181                   if padre.rightnode==deletingnode:
182                       padre.rightnode=deletingnode.leftnode
183                       node_desbalance=check_balance_parent(padre)
184                       if node_desbalance != None:
185                           reBalanceR(B, node_desbalance)
186                       return deletingnode.key
187                   else:
188                       padre.leftnode=deletingnode.leftnode
189                       node_desbalance=check_balance_parent(padre)
190                       if node_desbalance != None:
191                           reBalanceR(B, node_desbalance)
192                       return deletingnode.key
```

```
193             #Si es el hijo derecho
194             elif deletingnode.leftnode==None:
195                 if deletingnode==B.root:
196                     B.root=deletingnode.rightnode
197                     return deletingnode.key
198                 else:
199                     padre=deletingnode.parent
200                     #Me fijo si el nodo a eliminar es el hijo derecho o izquierdo de su padre
201                     if padre.rightnode==deletingnode:
202                         padre.rightnode=deletingnode.rightnode
203                         node_desbalance=check_balance_parent(padre)
204                         if node_desbalance != None:
205                             reBalanceR(B, node_desbalance)
206                         return deletingnode.key
207                     else:
208                         padre.leftnode=deletingnode.rightnode
209                         node_desbalance=check_balance_parent(padre)
210                         if node_desbalance != None:
211                             reBalanceR(B, node_desbalance)
212                         return deletingnode.key
```

```
213             #Caso 3: Dos hijos
214             else:
215                 aux1=deletingnode.rightnode
216                 aux2=deletingnode.leftnode
217                 #Buscamos al mayor de menores
218                 mayor=mayordemenores(deletingnode)
219                 #En el caso de que el mayor de menores tenga hijos izquierdos y no sea el hijo izquierdo del nodo a
220                 if mayor.leftnode!=None and mayor!=aux2:
221                     menor=menorhijoizq(mayor)
222                     #Para evitar crear un ciclo con padres=hijos
223                     if menor.parent!=deletingnode.leftnode:
224                         menor.leftnode=aux2
225                 elif mayor.leftnode==None and mayor!=aux2:
226                     mayor.leftnode=aux2
227                 mayor.rightnode=aux1
228
229                 if deletingnode==B.root:
230                     B.root=mayor
231                 else:
232                     padre=deletingnode.parent
233                     mayor.parent=padre
234                     if padre.rightnode==deletingnode:
235                         padre.rightnode=mayor
236                         node_desbalance=check_balance_parent(padre)
```

```
234                 if padre.rightnode==deletingnode:
235                     padre.rightnode=mayor
236                     node_desbalance=check_balance_parent(padre)
237                     if node_desbalance != None:
238                         reBalanceR(B, node_desbalance)
239                 else:
240                     padre.leftnode=mayor
241                     node_desbalance=check_balance_parent(padre)
242                     if node_desbalance != None:
243                         reBalanceR(B, node_desbalance)
244
245             return deletingnode.key
246         else:
247             return None
248
249     #Funcion que encuentra el nodo mayor entre los hijos menores de un nodo
250     def mayordemenores(node):
251         currentnode=node.leftnode
252         while currentnode.rightnode!=None:
253             currentnode=currentnode.rightnode
254         currentnode.parent.rightnode=None
255         return currentnode
```

```
257     #Funcion que encuentra el ultimo hijo menor dentro de una rama
258     def menorhijoizq(node):
259         if node.leftnode==None:
260             return node
261         else:
262             return menorhijoizq(node.leftnode)
263
264     def delete(B,element):
265         currentnode=B.root
266         deletingnode=searchR(currentnode, element)
267         return deleteR(B,deletingnode)
268
269     def searchR(currentnode,element):
270         if currentnode!=None:
271             if element==currentnode.value:
272                 return currentnode
273             else:
274                 izq= searchR(currentnode.leftnode,element)
275                 der= searchR(currentnode.rightnode,element)
276                 if izq==None and der==None:
277                     return None
278                 elif izq==None:
279                     return der
280                 elif der==None:
281                     return izq
282         else:
283             return None
```

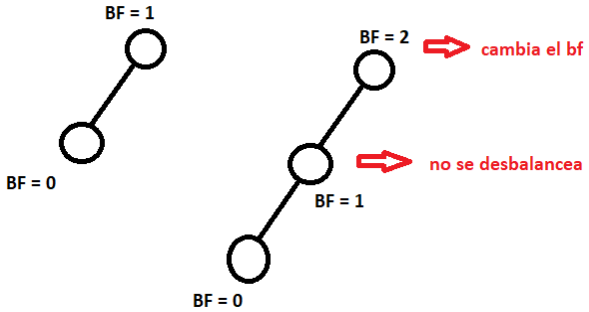
```
285    def search(AVL,element):
286        currentnode=AVL.root
287        result=searchR(currentnode, element)
288        if result!=None:
289            return result.key
290        else:
291            return None
```

Parte 2

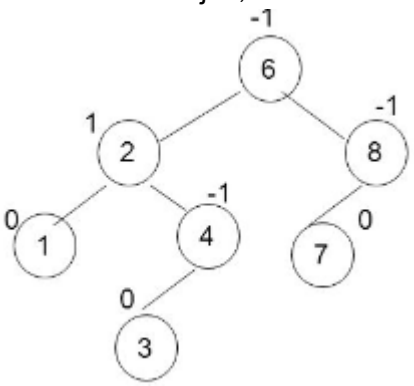
Ejercicio 6:

1. Responder V o F y justificar su respuesta:
   a. _F_ En un AVL el penúltimo nivel tiene que estar completo.
   Suponiendo que sea verdadero, existe un nodo x tal que tiene un hijo hacia la izquierda y ese nodo tiene otro hijo hacia la izquierda su bf es 2 o -2. Por este contra ejemplo se demuestra
   b. _V__ Un AVL donde todos los nodos tengan factor de balance 0 es completo.
   Suponemos que existe un AVL que tiene todos los nodos con bf = 0 y no es completo.
   Si no es completo, existe un nodo del arbol que tiene solo un hijo—> SU BF NO ES 0
   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.

      Mediante un contraejemplo:

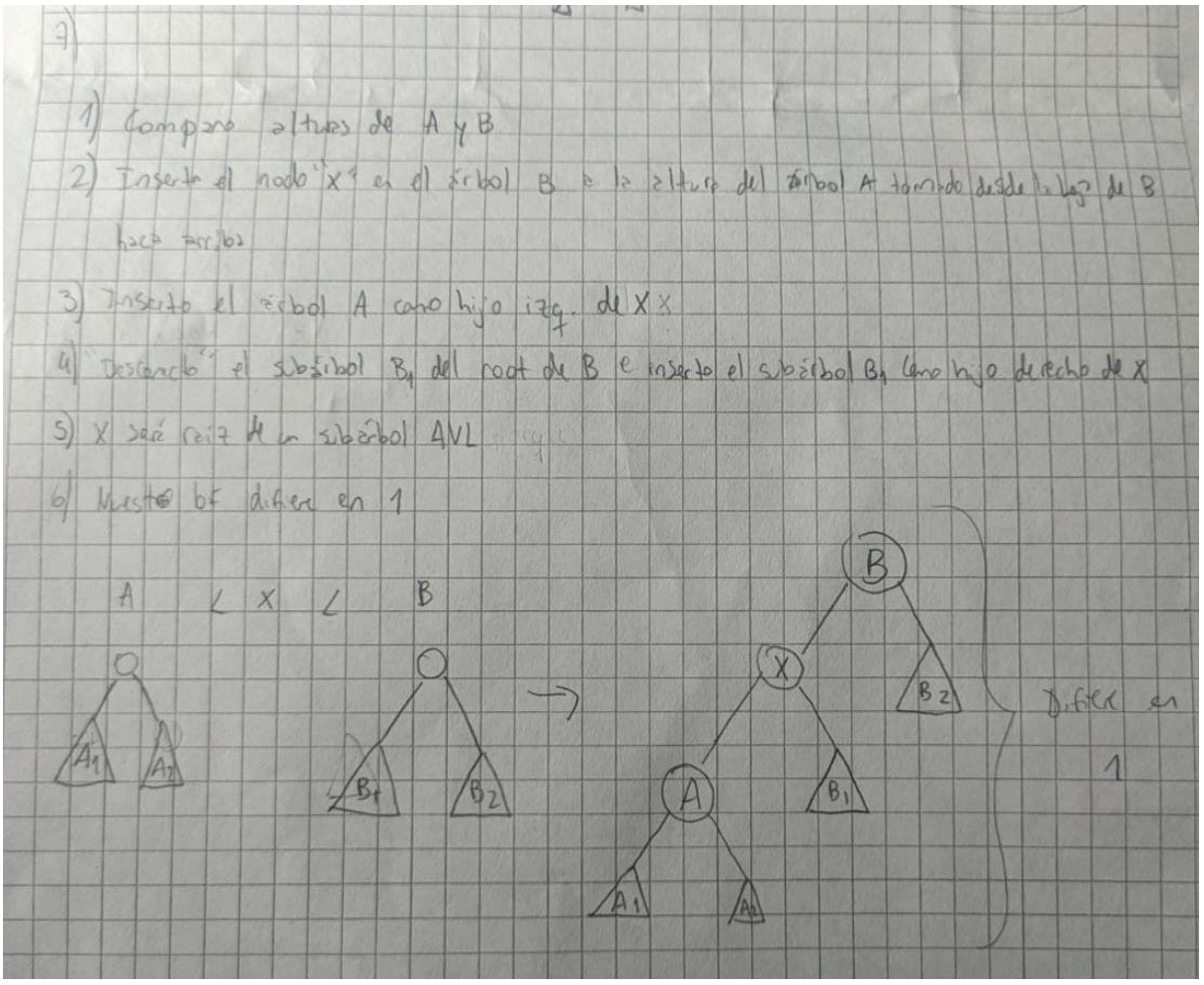


   d. __F_ En todo *AVL* existe al menos un nodo con factor de balance 0.
   Sin considerar hojas, ….



Ejercicio 7:

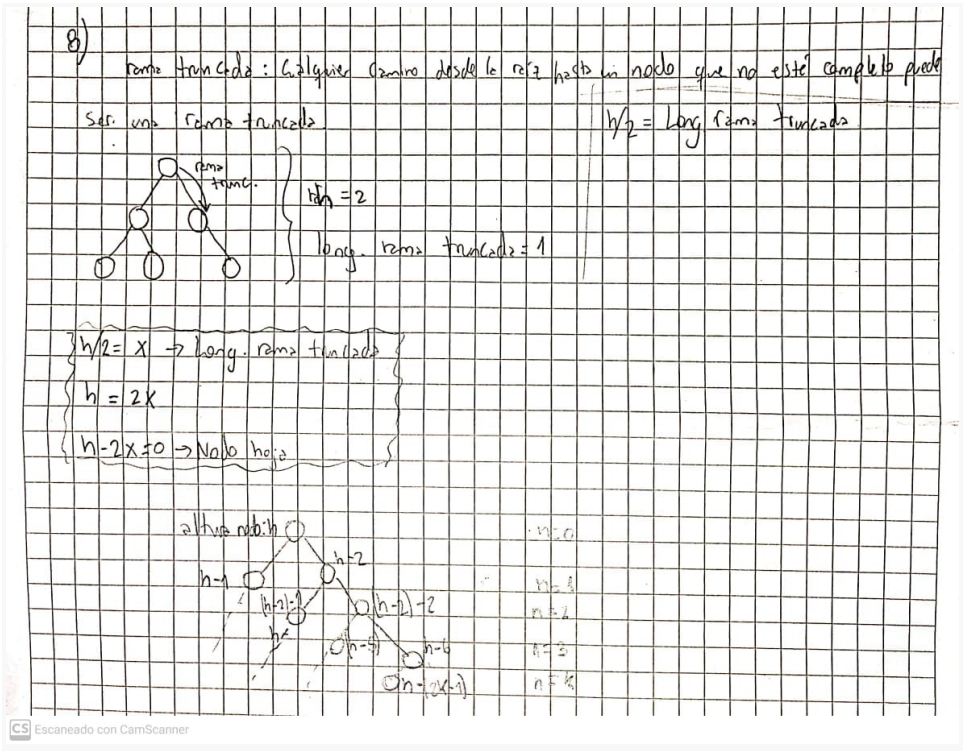
Sean *A* y *B* dos *AVL* de *m* y *n* nodos respectivamente y sea *x* un key cualquiera de forma tal que para todo key $a \in A$ y para todo key $b \in B$ se cumple que $a < x < b$. Plantear un algoritmo $O(\log n + \log m)$ que devuelva un *AVL* que contenga los key de *A*, el key *x* y los key de *B*.



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.