Algoritmos 2 – Giuliano La Piana

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

```
13  #Ejercicio 1:
14  #rotateRight(Tree,avlnode)
15  #Descripción: Implementa la operación rotación a la derecha
16  #Entrada: Un Tree junto a un AVLnode sobre el cual se va a operar la rotación a
    la  derecha
17  #Salida: retorna la nueva raíz
18
19  def rotateRight(B,rotroot):
20    newroot = rotroot.leftnode
21    rotroot.leftnode = newroot.rightnode
22    #
23    if newroot.rightnode != None:
24      newroot.rightnode.parent = rotroot
25    newroot.parent = rotroot.parent
26
27    if rotroot.parent == None:
28      B.root = newroot
29    else:
30      if rotroot.parent.rightnode == rotroot:
31        rotroot.parent.rightnode = newroot
32      else:
33        rotroot.parent.leftnode = newroot
34    newroot.rightnode = rotroot
```

```
def rotateLeft(B,rotroot):
  newroot = rotroot.rightnode
  rotroot.rightnode = newroot.leftnode
  #
  if newroot.leftnode != None:
    newroot.leftnode.parent = rotroot
  newroot.parent = rotroot.parent

  if rotroot.parent == None:
    B.root = newroot
  else:
    if rotroot.parent.leftnode == rotroot:
      rotroot.parent.leftnode = newroot
    else:
      rotroot.parent.rightnode = newroot
  newroot.leftnode = rotroot
  rotroot.parent = 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.

```
#Ejercicio 2:
#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 update_bf(root,height):
  if(root.rightnode == None and root.leftnode == None):
    return height
  else:
    if(root.leftnode != None):
      leftHeight = update_bf(root.leftnode,height+1)
    else:
      leftHeight = 0
    if(root.rightnode != None):
      rightHeight = update_bf(root.rightnode,height+1)
    else:
      rightHeight = 0
    if(leftHeight > rightHeight):
      return leftHeight
    else:
      return rightHeight
```

```python
def calculateBalance(B):
  if B.root != None:
    calculateBalanceR(B.root)
  else:
    return


#Funcion recursiva que asigna el balanceFactor a cada nodo
def calculateBalanceR(newnode):
  if(newnode.leftnode != None):
    leftHeight = height(newnode.leftnode)
    calculateBalanceR(newnode.leftnode)
  else:
    leftHeight = 0
  if(newnode.rightnode != None):
    rightHeight = height(newnode.rightnode)
    calculateBalanceR(newnode.rightnode)
  else:
    rightHeight = 0
  newnode.bf = rightHeight - leftHeight
```

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.

```python
def reBalance(B):
  calculateBalance(B)
  reBalanceR(B,B.root)


#Funcion recursiva encargada de realizar las rotaciones necesaria para hacer
cumplir la condicion de AVL
def reBalanceR(B,newnode):
    if(newnode.bf <= -2):
      rotateRight(B,newnode.leftnode)
    elif(newnode.bf >= 2):
      rotateLeft(B,newnode.rightnode)
    else:
      if(newnode.leftnode != None):
        reBalanceR(B,newnode.leftnode)
      if(newnode.rightnode != None):
        reBalanceR(B,newnode.rightnode)
#Verificamos que se haya rotado correctamente, caso contrario volvemos a rota
```

Ejercicio 4:

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

```python
#Ejercicio 4:
#Implementar la operación insert() en  el módulo avltree.py garantizando que el árbol  binario resultante sea un árbol AVL

def insertR(newNode,currentNode):
  if 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:
    if currentNode.rightnode==None:
      newNode.parent=currentNode
      currentNode.rightnode=newNode
      return newNode
    else:
      return insertR(newNode,currentNode.rightnode)
  else: #newNode.key==currentNode.key es decir la clave ya existe.
    return None
```

```python
def insert(B,element,key):
  newNode=AVLNode()
  newNode.key=key
  newNode.value=element
  if B.root==None:
    B.root=newNode
    return newNode.key
  else:
    current = B.root
    newNode=insertR(newNode,B.root)
  reBalance(B)
  return newNode
```

Ejercicio 5:

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

```python
#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):
  currentNode=searchR(B.root,element)
  if currentNode==None:
    return None

  #El elemento a eliminar es una hoja.
  if currentNode.leftnode==None and currentNode.rightnode==None:
    if currentNode.key<currentNode.parent.key: #Nodo a la izquierda del padre.
      currentNode.parent.leftnode=currentNode.leftnode
    else: #Nodo a la derecha del padre.
      currentNode.parent.rightnode=currentNode.rightnode
    reBalance(B)
    return currentNode.key

  #El nodo tiene un hijo a la izquierda.
  if currentNode.leftnode!=None and currentNode.rightnode==None:
    if currentNode.key<currentNode.parent.key: #Nodo a la izquierda del padre.
      currentNode.parent.leftnode=currentNode.leftnode
    else: #Nodo a la derecha del padre.
```

```
#El nodo tiene un hijo a la derecha.
if currentNode.leftnode==None and currentNode.rightnode!=None:
  if currentNode.key<currentNode.parent.key: #Nodo a la izquierda del padre.
    currentNode.parent.leftnode=currentNode.rightnode
  else: #Nodo a la derecha del padre.
    currentNode.parent.rightnode=currentNode.rightnode
  reBalance(B)
  return currentNode.key

#El nodo tiene dos hijos.
if currentNode.leftnode!=None and currentNode.rightnode!=None:

  newNode=currentNode.rightnode #Menor de sus mayores.
  newNode=MenordesusMayores(newNode)

  oldkey=currentNode.key

  currentNode.key=newNode.key
  currentNode.value=newNode.value
  #Se desvincula el nodo menor de sus mayores.
  newNode.parent.leftnode=newNode.leftnode #Sabemos que a su izquierda no hay ningún nodo.
  reBalance(B)
  return oldkey
```

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.

6)a) V, suponiendo que en el antepenúltimo nivel existe un nodo que no es completo, es decir que tiene un hijo
b) V, suponiendo un AVL con algún nodo de bf ¡= 0 que es completo. Si un nodo tiene bf ¡= 0 implica que tiene un hijo y al tener un hijo ya es completo
c) F, ya que el desbalanceo puede ocurrir mas arrba
d)F

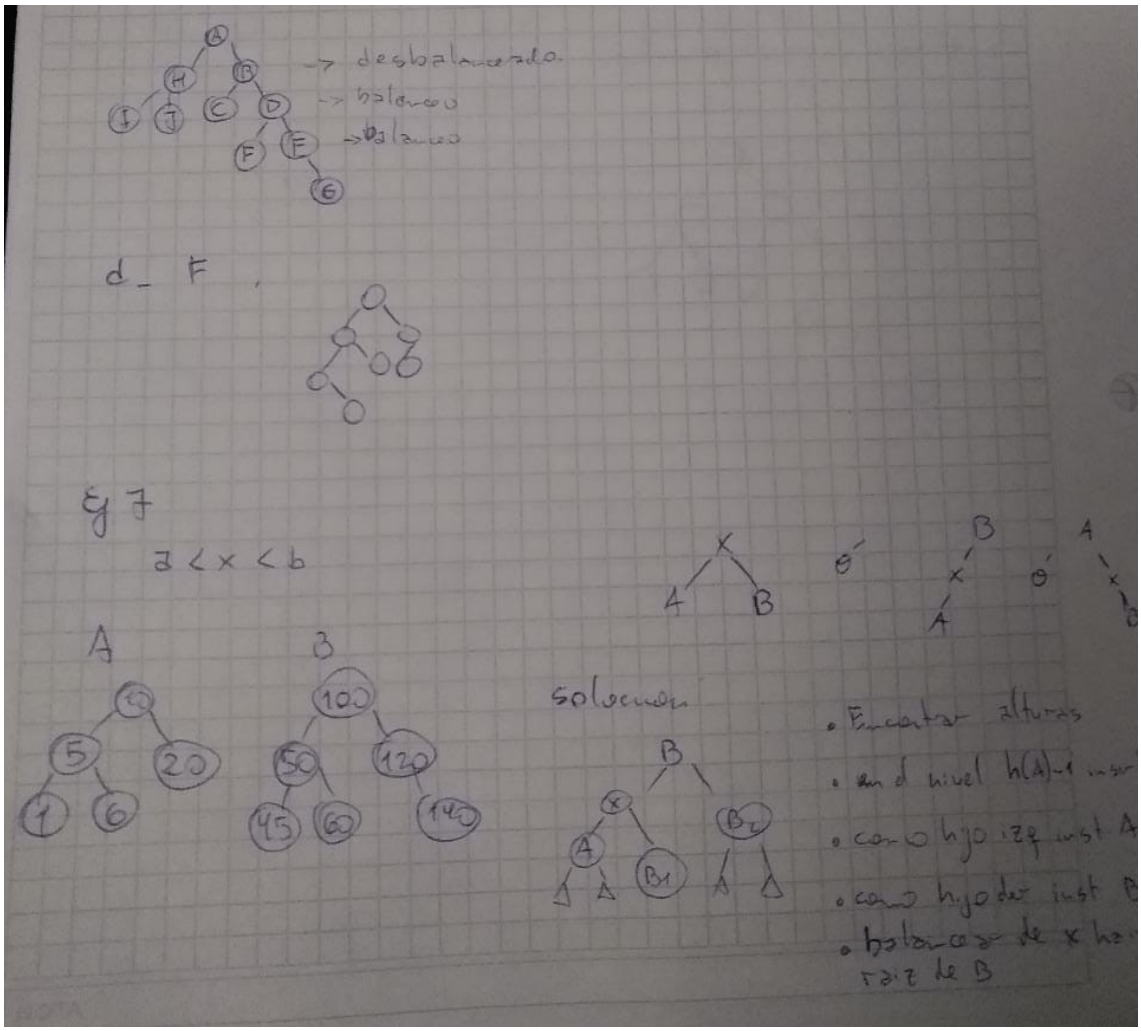
Ejercicio 7:

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.


-> desbalanceado
-> balanceo
-> balanceo
d_ F
Ej 7
a < x < b
A
B
Solución
• Encontrar alturas
• en el nivel h(A)-1
• como hijo izq inst A
• como hijo der inst B
• balancear de x hacia raiz de B