Francisco Alba

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(AVLT, node):
  # Guarda referencia a la nueva raíz
  newRoot = node.rightNode
  # Si la nueva raíz ya tiene un hijo en la rama izquierda, se mueve a la rama derecha de la raíz vieja
  node.rightNode = newRoot.leftNode
  # Se enlaza el nodo viejo a la rama izquierda de la nueva raíz
  newRoot.leftNode = node
  # Y finalmente se ajustan los parientes y el nodo raíz del árbol según corresponda
  node.rightNode.parent = node
  if node.parent == None:
    AVLT.root = newRoot
  else:
    if node.parent.leftNode == node:
      node.parent.leftNode = newRoot
    else:
      node.parent.rightNode = newRoot
  newRoot.parent = node.parent
  node.parent = newRoot
```

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

```python
def rotateRight(AVLT, node):
  # Guarda referencia a la nueva raíz
  newRoot = node.leftNode
  # Si la nueva raíz ya tiene un hijo en la rama derecha, se mueve a la rama derecha de la raíz vieja
  node.leftNode = newRoot.rightNode
  # Se enlaza el nodo viejo a la rama derecha de la nueva raíz
  newRoot.rightNode = node
  # Y finalmente se ajustan los parientes y el nodo raíz del árbol según corresponda
  node.leftNode.parent = node
  if node.parent == None:
    AVLT.root = newRoot
  else:
    if node.parent.leftNode == node:
      node.parent.leftNode = newRoot
    else:
      node.parent.rightNode = newRoot
  newRoot.parent = node.parent
  node.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.
Salida: El árbol AVL con el valor de balanceFactor para cada subarbol

```python
def calculateHeight(node):
  if node == None:
    return 0
  leftH = 1 + calculateHeight(node.leftNode)
  rightH = 1 + calculateHeight(node.rightNode)
  return max(leftH, rightH)-1

def calculateBFR(AVLTnode):
  if AVLTnode == None:
    return
  AVLTnode.bf = calculateHeight(AVLTnode.leftNode) - calculateHeight(AVLTnode.rightNode)
  calculateBFR(AVLTnode.leftNode)
  calculateBFR(AVLTnode.rightNode)

def calculateBF(AVLT):
  calculateBFR(AVLT.root)
```

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 reBalanceR(AVLT, AVLTnode):
  if AVLTnode == None:
    return
  if AVLTnode.bf < -1 or AVLTnode.bf > 1:
    if AVLTnode.bf > 0:
      if AVLTnode.leftNode.rightNode != None: # if AVLTnode.leftNode.bf < 0
        rotateLeft(AVLT, AVLTnode.leftNode)
        rotateRight(AVLT, AVLTnode)
      else:
        rotateRight(AVLT, AVLTnode)
    elif AVLTnode.bf < 0:
      if AVLTnode.rightNode.leftNode != None: # if AVLTnode.rightNode.bf > 0
        rotateRight(AVLT, AVLTnode.rightNode)
        rotateLeft(AVLT, AVLTnode)
      else:
        rotateLeft(AVLT, AVLTnode)
    calculateBF(AVLT)
  reBalanceR(AVLT, AVLTnode.leftNode)
  reBalanceR(AVLT, AVLTnode.rightNode)

def reBalance(AVLT, AVLTnode):
  if AVLTnode.bf > 0:
    if AVLTnode.leftNode.rightNode != None: # if AVLTnode.leftNode.bf < 0
      rotateLeft(AVLT, AVLTnode.leftNode)
      rotateRight(AVLT, AVLTnode)
    else:
      rotateRight(AVLT, AVLTnode)
  elif AVLTnode.bf < 0:
    if AVLTnode.rightNode.leftNode != None: # if AVLTnode.rightNode.bf > 0
      rotateRight(AVLT, AVLTnode.rightNode)
      rotateLeft(AVLT, AVLTnode)
    else:
      rotateLeft(AVLT, AVLTnode)
```

Ejercicio 4:

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

```python
def calcHeightAndBF(AVLT, AVLTnode):
  if AVLTnode != None:
    if AVLTnode.leftNode != None and AVLTnode.rightNode != None:
      AVLTnode.h = max(AVLTnode.leftNode.h, AVLTnode.rightNode.h) + 1
      AVLTnode.bf = AVLTnode.leftNode.h - AVLTnode.rightNode.h
    elif AVLTnode.leftNode != None:
      AVLTnode.h = AVLTnode.leftNode.h + 1
      AVLTnode.bf = AVLTnode.h
    elif AVLTnode.rightNode != None:
      AVLTnode.h = AVLTnode.rightNode.h + 1
      AVLTnode.bf = -AVLTnode.h
    else: #nodo hoja
      AVLTnode.h = 0
      AVLTnode.bf = 0
  if AVLTnode.bf < -1 or AVLTnode.bf > 1:
    reBalance(AVLT, AVLTnode)
  else:
    calcHeightAndBF(AVLT, AVLTnode.parent)

def insertR(newNode, currNode, AVLT):
  if newNode.key > currNode.key:
    if currNode.rightNode == None:
      newNode.parent = currNode
      currNode.rightNode = newNode
      calcHeightAndBF(AVLT, newNode)
      return newNode.key
    else:
      insertR(newNode, currNode.rightNode)
  elif newNode.key < currNode.key:
    if currNode.leftNode == None:
      newNode.parent = currNode
      currNode.leftNode = newNode
      calcHeightAndBF(AVLT, newNode)
      return newNode.key
    else:
      insertR(newNode, currNode.leftNode)
  else:
    return None

def insert(AVLT, elem, key):
  newNode = AVLNode()
  newNode.value = elem
  newNode.key = key
  if AVLT.root == None:
    AVLT.root = newNode
    return key
  else:
    insertR(newNode, AVLT.root, AVLT)
```

Ejercicio 5:

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

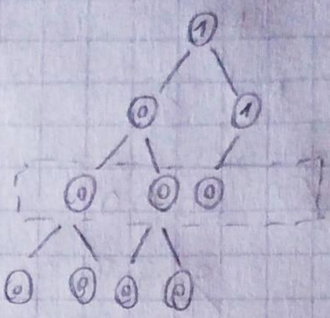
```python
def remove(node):
  if node.leftNode == None and node.rightNode == None:
    if node.parent.leftNode == node:
      node.parent.leftNode = None
    else:
      node.parent.rightNode = None
    node.parent == None
  elif (node.leftNode == None and node.rightNode != None) or (node.leftNode != None and node.rightNode == None):
    if node.leftNode != None:
      if node.parent.leftNode == node:
        node.parent.leftNode = node.leftNode
      else:
        node.parent.rightNode = node.leftNode
    else:
      if node.parent.leftNode == node:
        node.parent.leftNode = node.rightNode
      else:
        node.parent.rightNode = node.rightNode
  elif node.leftNode != None and node.rightNode != None:
    lowestFromRight = node.rightNode
    while lowestFromRight.leftNode != None:
      lowestFromRight = lowestFromRight.leftNode
    node.key = lowestFromRight.key
    node.value = lowestFromRight.value
    remove(lowestFromRight)

def deleteR(AVLT, AVLTnode, elem):
  if AVLTnode == None:
    return
  if AVLTnode.value == elem:
    remove(AVLTnode)
    calculateBF(AVLT)
    reBalanceR(AVLT)
    return AVLTnode.key
  deleteR(AVLT, AVLTnode.leftNode, elem)
  deleteR(AVLT, AVLTnode.rightNode, elem)
  return None

def delete(AVLT, elem):
  deleteR(AVLT, AVLT.root, elem)
```

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
   b. _V_ Un AVL donde todos los nodos tengan factor de balance 0 es completo
   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.
   d. _F_ En todo *AVL* existe al menos un nodo con factor de balance 0.

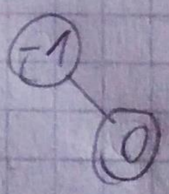
6. a) F. Lo demostramos con un contraejemplo:



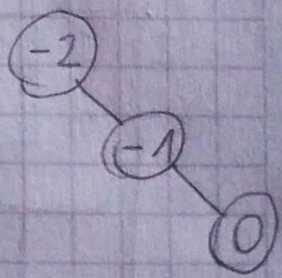
En el árbol ilustrado, el penúltimo nivel está incompleto pero el árbol es un AVL. Luego la afirmación es Falsa

b) V. Para probar que es Falso, deberíamos poder construir un árbol AVL donde todos los nodos tengan FB 0 y que a su vez sea un árbol incompleto. Sin embargo, dado que el Factor de balance es la diferencia en la altura entre el árbol izquierdo y el derecho de cada nodo, al pedir que esta diferencia sea igual a cero para todos los nodos, estamos pidiendo también que todos los nodos tengan 2 hijos (uno por rama) salvo en el último nivel, lo cual coincide a su vez con la definición de árbol completo. Luego la afirmación tiene que ser verdadera.

c) F. Contraejemplo:



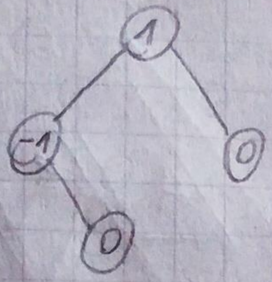
(A)



(B)

A ilustra un AVL y B ilustra la inserción de un nodo en el nodo hoja de (A). Como puede observarse, el BF del padre del nodo hoja en (B), −1, no está desbalanceado, pero sí el del ancestro de ese nodo padre, que es −2

d) F. Para probar que la afirmación es Falsa, buscamos un árbol AVL que no tenga nodos con bF = 0, a excepción de los nodos hoja.

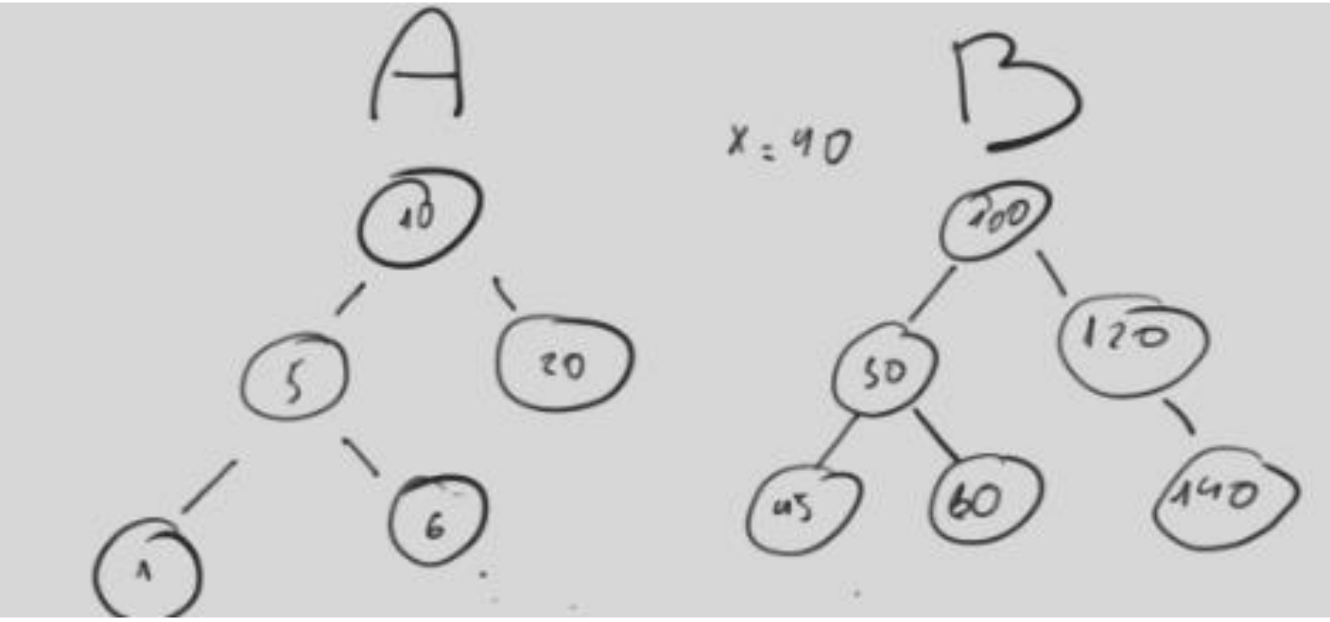


(A)

Podemos observar que el ejemplo ilustrado en (A) cumple con lo propuesto. Luego la afirmación es Falsa.

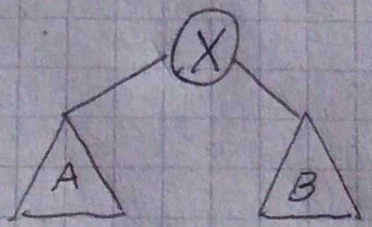
Ejercicio 7:

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*.

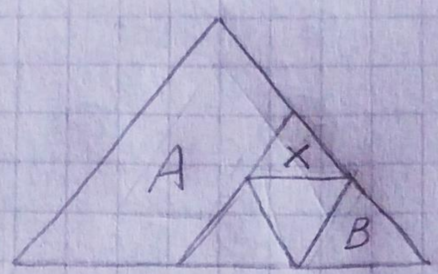


7. Para resolver el problema, planteamos 3 casos:

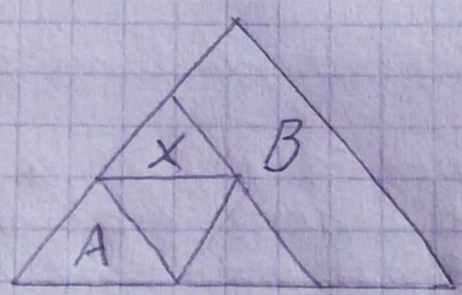
a) La altura de los árboles A y B es la misma. Entonces usamos a X como raíz, colocamos a A en la rama izquierda de X y a B en la rama derecha. No es necesario rebalancear en este caso porque A y B ya estaban balanceados, y agregar un nivel no produce desbalance:



b) A es más alto que B. En este caso, buscamos en A un subárbol en la rama derecha que tenga la misma altura que B. En lugar de ese subárbol, insertamos X y al subárbol extraído lo colocamos como rama izquierda de X. Podemos ahora entonces insertar B como rama derecha de X y así finalmente reducimos al mínimo el número de rebalanceos, ya que sólo deberíamos rebalancear desde X hacia arriba



c) B es más alto que A. Repetimos el procedimiento anterior, pero buscando en el subárbol izquierdo de B un árbol de la misma altura que A.



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.

> Suponemos que la mínima longitud de una rama truncada en un AVL de altura h es k > h/2 y buscamos llegar a una contradicción.
> Para que el AVL sea de altura h, tiene que haber una rama completa de longitud h que se conecta a la raíz.
> Como la rama truncada tiene longitud k > h/2, para no exceder la altura total del árbol h (lo cual sería una contradicción al supuesto de que el árbol es de altura h) debemos conectar la rama truncada a otra rama que tenga longitud h - k < h/2. Pero esto significa que la rama completa a la cual estamos intentando conectar la rama truncada es más corta que la rama truncada en sí, lo cual no puede ser cierto en un AVL balanceado, ya que todas las ramas de un AVL deben tener una diferencia de altura de como máximo 1.
> Por lo tanto, la suposición de que la mínima longitud de una rama truncada en un AVL de altura h es mayor que h/2 es falsa y así la mínima longitud que puede tener 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.