Aida Laricchia

Legajo:13251

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

```
def rotateRight(tree,rotenode):
    new_root=rotenode.leftnode
    rotenode.leftnode=new_root.rightnode
    if new_root.rightnode != None:
        new_root.rightnode.parent=rotenode
    new_root.parent=rotenode.parent
```

```
    if rotenode.parent == None:
        tree.root=new_root
    else:
        if rotenode.parent.rightnode == rotenode:
            rotenode.parent.rightnode=new_root
        else:
            rotenode.parent.left=new_root
    new_root.rightnode=rotenode
    rotenode.parent=new_root.rightnode

def rotateLeft(tree,rotenode):
    new_root=rotenode.rightnode
    rotenode.rightnode=new_root.leftnode
    if new_root.leftnode != None:
        new_root.leftnode.parent=rotenode
    new_root.parent=rotenode.parent
    if rotenode.parent == None:
        tree.root=new_root
    else:
        if rotenode.parent.leftnode == rotenode:
            rotenode.parent.leftnode=new_root
        else:
            rotenode.parent.left=new_root
    new_root.leftnode=rotenode
    rotenode.parent=new_root.leftnode
```

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

```
#funcion que al pasarle un nodo me dice su altura eesta sera necesaria para la
f.calculateblance
#esta sera una funcion recursiva
def altura(node):
    if node ==None:
        return 0
    return (1+max(altura(node.left),altura(node.right)))
```

```
#Esta funcion calcula el bf de un solo nodo
def calculate_balance(B):
    node=B.root
    if node != None:
        calculate_balance_node(node)

#En esta funcion se recorre todo el arbol mientras se va calculado el bf de cada nodo
def calculate_balance_node(node):
    if node != None:
        node.bf= altura(node.left)-altura(node.right)
        calculate_balance_node(node.leftnode)
        calculate_balance_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.

```
#funcion que busca el nodo desbalanceado
def find_node_desbalance(AVLT):
    calculate_balance(AVLT)
    node=AVLT.root
    if node != None:
        return (find_node_desbalanceR(node))

#retorna el nodo que tiene el desbalance
def find_node_desbalanceR(node):
    if node != None:
        if node.bf != 1 and node.bf !=0 and node.bf !=-1:
            return node
        find_node_desbalanceR(node.leftnode)
        find_node_desbalanceR(node.rightnode)


def reBalance(AVLTree):
    bf_arbol=calculate_balance_node(AVLTree.root)
```

```
    if bf_arbol != 1 and bf_arbol !=0 and bf_arbol !=-1:
        node = find_node_desbalance(AVLTree)
        #Rebalanceamos el arbol
        rebalanceR(AVLTree, node)

#esta funcion recibe el nodo que esta desbalanceado
def rebalanceR(AVLTree, node):
    if node.bf < -1:
        if node.rightnode.bf == 1:
            rotateRight(AVLTree,node.rightnode)
            rotateLeft(AVLTree, node)
        else:
            rotateLeft(AVLTree, node)
    if node.bf > 1:
        if node.leftnode.bf == -1:
            rotateLeft(AVLTree, node.leftnode)
            rotateRight(AVLTree, node)
        else:
            rotateRight(AVLTree, node)
```

Ejercicio 4:

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

```
#insert binarytree modificado para AVLT
def insertw(B,newnode, currentnode):
    if newnode.key>currentnode.key:
        if currentnode.rightnode==None:
            newnode.parent=currentnode
            currentnode.rightnode=newnode
            #cuando ya insertamos tenemos que chequear que el arbol siga siendo AVL
            #tendremos que cheaquer en esa rama el bf de cada uno de sus padre si uno
            #no se encuentra en el intervalo entonces balanceamos
            node_desbalance=check_balance_parent(newnode)
            if node_desbalance != None:
                #balancear
                rebalanceR(B, node_desbalance)
            return newnode.key
        else:
            return insertw(newnode, currentnode.rightnode)

    elif newnode.key<currentnode.key:
        if currentnode.leftnode==None:
```

```
            newnode.parent=currentnode
            currentnode.leftnode=newnode
            #chequeamos solo subieron en el arbol
            node_desbalance=check_balance_parent(newnode)
            if node_desbalance != None:
                #balancear
                rebalanceR(B, node_desbalance)

            return newnode.key
        else:
            return insertw(newnode, currentnode.leftnode)
    else:
        return None


def check_balance_parent(node):
    while node != None:
        calculate_balance_node(node)
        if node.bf >1 or node.bf <-1:
            return node
        node=node.parent
    return None

#mismo inser de binarytree pero ahora pasamos el arbol como parametro
def insert(B,element,key):
    newnode=AVLNode()
    newnode.value=element
    newnode.key=key
    currentnode=B.root
    if B.root==None:
        B.root=newnode
    else:
        return insertw(B,newnode, currentnode)
```

Ejercicio 5:

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

```
#Funcion delete
def deletew(B,deletingnode):
    if deletingnode!=None:
        #Caso 1: Hoja
        if deletingnode.leftnode==None and deletingnode.rightnode==None:
            if deletingnode==B.root:
```

```
                B.root=None
                return deletingnode.key
            else:
                padre=deletingnode.parent
                #Me fijo si el nodo a eliminar es el hijo derecho o izquierdo de su
padre
                if padre.rightnode==deletingnode:

                    padre.rightnode=None
                #balancear
                    node_desbalance=check_balance_parent(padre)
                    if node_desbalance != None:
                        #balancear
                        rebalanceR(B, node_desbalance)
                    return deletingnode.key
                else:
                    padre.leftnode=None
                    #balancear
                    node_desbalance=check_balance_parent(padre)
                    if node_desbalance != None:
                        #balancear
                        rebalanceR(B, node_desbalance)
                    return deletingnode.key

        #Caso 2: Un solo hijo
        #Si es el hijo izquierdo
        elif deletingnode.rightnode==None:
            if deletingnode==B.root:
                B.root=deletingnode.leftnode
                return deletingnode.key
            else:
                padre=deletingnode.parent
                #Me fijo si el nodo a eliminar es el hijo derecho o izquierdo de su
padre
                if padre.rightnode==deletingnode:
                    padre.rightnode=deletingnode.leftnode
                    node_desbalance=check_balance_parent(padre)
                    if node_desbalance != None:
                        #balancear
                        rebalanceR(B, node_desbalance)
                    return deletingnode.key
                else:
                    padre.leftnode=deletingnode.leftnode
                    node_desbalance=check_balance_parent(padre)
```

```
                    if node_desbalance != None:
                        #balancear
                        rebalanceR(B, node_desbalance)
                    return deletingnode.key
            #Si es el hijo derecho
            elif deletingnode.leftnode==None:
                if deletingnode==B.root:
                    B.root=deletingnode.rightnode
                    return deletingnode.key
                else:
                    padre=deletingnode.parent
                    #Me fijo si el nodo a eliminar es el hijo derecho o izquierdo de su
padre
                    if padre.rightnode==deletingnode:
                        padre.rightnode=deletingnode.rightnode
                        node_desbalance=check_balance_parent(padre)
                        if node_desbalance != None:
                            #balancear
                            rebalanceR(B, node_desbalance)
                        return deletingnode.key
                    else:
                        padre.leftnode=deletingnode.rightnode
                        node_desbalance=check_balance_parent(padre)
                        if node_desbalance != None:
                            #balancear
                            rebalanceR(B, node_desbalance)
                        return deletingnode.key

            #Caso 3: Dos hijos
            else:
                aux1=deletingnode.rightnode
                aux2=deletingnode.leftnode
                #Buscamos al mayor de menores
                mayor=mayordemenores(deletingnode)
                #En el caso de que el mayor de menores tenga hijos izquierdos y no sea el
hijo izquierdo del nodo a eliminar:
                if mayor.leftnode!=None and mayor!=aux2:
                    menor=menorhijoizq(mayor)
                    #Para evitar crear un ciclo con padres=hijos
                    if menor.parent!=deletingnode.leftnode:
                        menor.leftnode=aux2
                elif mayor.leftnode==None and mayor!=aux2:
                    mayor.leftnode=aux2
                mayor.rightnode=aux1
```

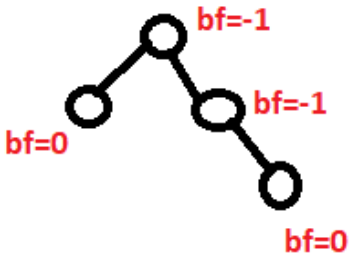
```
            if deletingnode==B.root:
                B.root=mayor
            else:
                padre=deletingnode.parent
                mayor.parent=padre
                if padre.rightnode==deletingnode:
                    padre.rightnode=mayor
                    node_desbalance=check_balance_parent(padre)
                    if node_desbalance != None:
                        #balancear
                        rebalanceR(B, node_desbalance)
                else:
                    padre.leftnode=mayor
                    node_desbalance=check_balance_parent(padre)
                    if node_desbalance != None:
                        #balancear
                        rebalanceR(B, node_desbalance)

            return deletingnode.key
    else:
        return None

#Funcion que encuentra el nodo mayor entre los hijos menores de un nodo
def mayordemenores(node):
    currentnode=node.leftnode
    while currentnode.rightnode!=None:
        currentnode=currentnode.rightnode
    currentnode.parent.rightnode=None
    return currentnode

#Funcion que encuentra el ultimo hijo menor dentro de una rama
def menorhijoizq(node):
    if node.leftnode==None:
        return node
    else:
        return menorhijoizq(node.leftnode)


def delete(B,element):
    currentnode=B.root
    deletingnode=searchR(currentnode, element)
    return deletew(B,deletingnode)
```

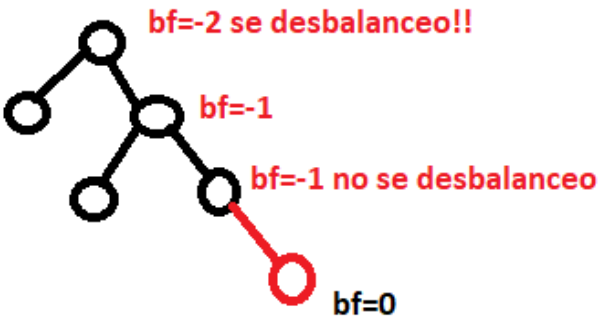
Parte 2

Ejercicio 6:

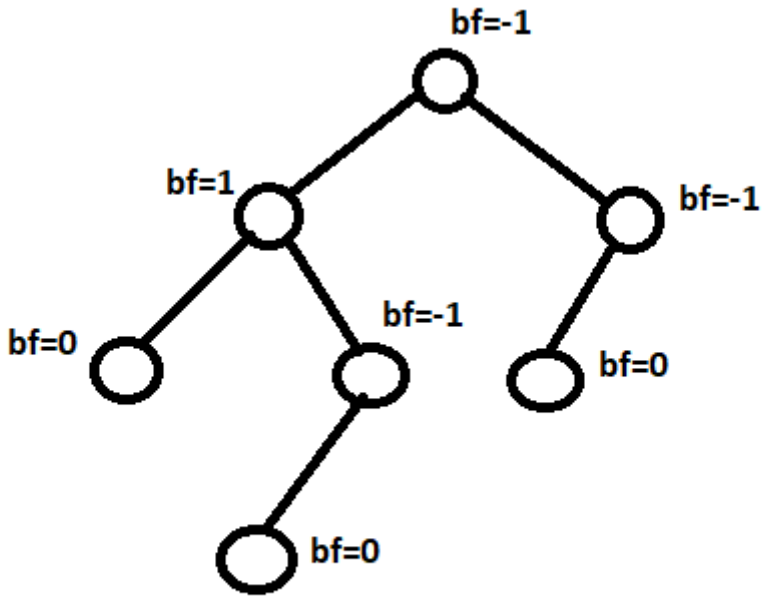
1. Responder V o F y justificar su respuesta:
   a. _F_ En un AVL el penúltimo nivel tiene que estar completo
   Suponiendo verdadero, existe un nodo x tal que tiene un hijo hacia la izquierda y ese nodo hijo tiene a su un hijo hacia la izquierda su bf es 2 o -2. Con este contra ejemplo queda demostrado que la afirmacion es falsa.

   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 entonces 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.





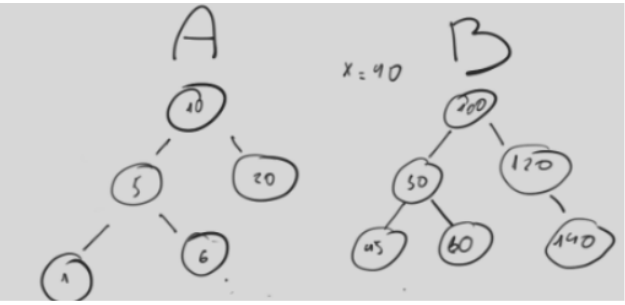
   d. __F_ En todo *AVL* existe al menos un nodo con factor de balance 0.



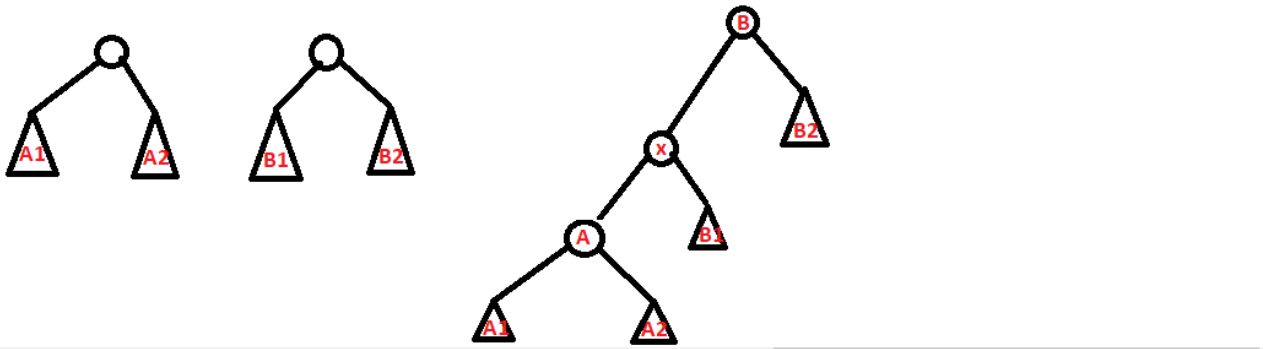
Si no tenemos en cuenta la hojas ningun nodo tiene un bf=0.

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*.



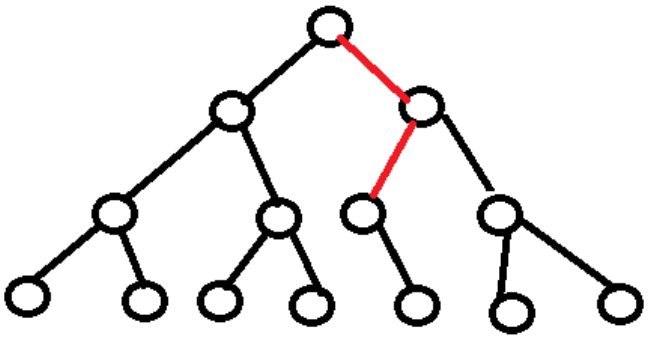
1ro: comparamos las alturas de A y B.
2do: se inserta el nodo X en el arbol B a la altura del arbol A tomando desde de B hacia arriba.
3ro: se inserta el nodo el arbol A como hijo izquierdo de X.
4to: desconecto el subarbol B1 de B e inserto el subarbol B1 como hijo derecho de X.
5to: X será la raiz de un subarbol AVL.
6to: nuestro bf difiere en 1.



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.



h=3
h/2=X
3/2=X
1,5=X
se verifica que la longitud de la rama truncada es mayor o igual a h/2
h-2.X=0
X=h/2

Cumplirá para todo h.

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.