UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

ALGORITMOS Y ESTRUCTURAS DE DATOS II

Árboles AVL

Yacante Daniel

Leg: 10341

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

```python
"""
Ambas funciones rotate funcionan de una forma análoga a la otra
Primero se obtienen las referencias a los nodos hijos(derecho o izquierdo)
como asi también las del nodo padre.
Se desvincula el nodo hijo con el cual se va a hacer el intercambio
Vemos si el nuevo nodo raíz tiene un hijo del lado que se hará el intercambio y
de ser asi se lo coloca como hijo del nodo raíz anterior.
Luego se actualizan las referencias correspondientes de parentesco, tanto del nodo padre
como del nodo que hemos movido
"""
def rotateLeft(tree:AVLTree,avlnode:AVLNode):
    nodoActual=avlnode
    hijoDer=nodoActual.rightnode
    padre=nodoActual.parent
    nodoActual.rightnode=None
    if hijoDer.leftnode!=None:
        nodoActual.rightnode=hijoDer.leftnode
    hijoDer.leftnode=nodoActual
    if padre!=None:
        if padre.rightnode==nodoActual:
            padre.rightnode=hijoDer
        else:
            padre.leftnode=hijoDer
    elif tree.root==avlnode:
        tree.root=hijoDer
    hijoDer.parent=nodoActual.parent
    nodoActual.parent=hijoDer
    return tree.root


def rotateRight(tree:AVLTree,avlnode:AVLNode):
    nodoActual=avlnode
    hijoIzq=nodoActual.leftnode
    padre=nodoActual.parent
    nodoActual.leftnode=None
    if hijoIzq.rightnode!=None:
        nodoActual.leftnode=hijoIzq.rightnode
    hijoIzq.rightnode=nodoActual
    if padre!=None:
        if padre.rightnode==nodoActual:
            padre.rightnode=hijoIzq
        else:
            padre.leftnode=hijoIzq
    elif tree.root==avlnode:
        tree.root=hijoIzq
    hijoIzq.parent=nodoActual.parent
    nodoActual.parent=hijoIzq
    return tree.root
```

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
"""
La función calculateBalance es un wrapper de la función __calcularAltura que recorre el árbol
desde las hojas hacia la raíz calculando siempre el bf como también retornando el valor mas grande
de las alturas de los hijos de cada nodo, es decir si un nodo tiene a su izquierda una rama de altura 3
y a su derecha una de altura 1, calcula el bf como 3-1=2 y regresa la suma entre 1 y el mayor de 3 y 1...
Es decir retornara 4, el valor de 1 se suma ya que al subir un nivel tenemos una arista mas del árbol.
"""
def calculateBalance(tree:AVLTree):
    __calcularAltura(tree.root)
    return

def __calcularAltura(avlnode:AVLNode):
    if avlnode.leftnode==None and avlnode.rightnode==None:
        avlnode.bf=0
        return 1
    elif avlnode.leftnode==None:
        hd=__calcularAltura(avlnode.rightnode)
        avlnode.bf=0-hd
        return 1+hd
    elif avlnode.rightnode==None:
        hi=__calcularAltura(avlnode.leftnode)
        avlnode.bf=hi-0
        return 1+hi
    else:
        hi=__calcularAltura(avlnode.leftnode)
        hd=__calcularAltura(avlnode.rightnode)
        avlnode.bf=hi-hd
        return 1+max(hd,hi)
    return
```

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
"""
La función de reBalance primero manda a actualizar/calcular los bf's del árbol
y mediante la función __recorreInOrder se obtiene una lista con los nodos del árbol
Para cada nodo se comprueba si el bf esta fuera de los limites para que sea un árbol balanceado
Y de ser asi, comprueba que no se tenga un caso especial, de ser asi primero se hace el giro
contrario al que se deberia hacer, y luego el que corresponde
"""
def reBalance(tree:AVLTree):
    nodosArbol=[]
    calculateBalance(tree)
    __recorreInOrder(tree.root,nodosArbol)
    for nodo in nodosArbol:
        if nodo.bf<-1:
            if nodo.rightnode.bf>0:
                rotateRight(tree,nodo.rightnode)
            rotateLeft(tree,nodo)
        elif nodo.bf>1:
            if nodo.leftnode.bf<0:
                rotateLeft(tree,nodo.leftnode)
            rotateRight(tree,nodo)
    calculateBalance(tree)
    return tree
```

Ejercicio 4:

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

```python
def __recorreInsert(avlnode:AVLNode,nod:AVLNode):
    if nod.key==avlnode.key:                                #Al insertar un nodo primero verificamos que no haya key's duplicadas
        return None
    elif nod.key<avlnode.key:                               #si la key del nodo a insertar es menor a la del nodo actual
        if avlnode.leftnode==None:                          #y el nodo actual no tiene elemento en su rama izquierda
            nod.parent=avlnode                              #se inserta el nodo a la izquierda
            avlnode.leftnode=nod
            return nod.key
        else:                                               #sino, se llama de forma recursiva a la propia función,
            return __recorreInsert(avlnode.leftnode,nod)    #con el nodo de la izq
    else:
        if avlnode.rightnode==None:                         #si la key, no es ni igual ni menor, solo queda que es mayor, y se
            nod.parent=avlnode                              #procede de forma similar al caso anterior.
            avlnode.rightnode=nod
            return nod.key
        else:
            return __recorreInsert(avlnode.rightnode,nod)

def insert(tree:AVLTree,elem, key):
    nodo=AVLNode()                                          #creo el nodo a insertar con su respectivo key y value
    nodo.value=elem
    nodo.key=key
    if tree.root==None:                                     #si el árbol esta vacío, el elemento a insertar queda como raíz del árbol
        tree.root=nodo                                      #sino, se llama a la función recursiva.
        return nodo.key
    else:
        res=__recorreInsert(tree.root,nodo)                 #Si se inserto exitosamente, se llama a un rebalanceo del árbol
        if res!=None:
            reBalance(tree)
        return res
```

Ejercicio 5:

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

```python
def delete(tree:AVLTree,elem):
    if tree.root!=None:                                     #Si el árbol no esta vacío, se llama a la función recursiva con el nodo raíz
        resp= __recorreDelete(tree.root,elem)               #como primer nodo
        if resp!=None:
            reBalance(tree)                                 #Si se elimino con éxito el nodo, se llama a un rebalanceo del árbol
        return resp
    else:
        return None
```

```python
def __recorreDelete(avlnode:AVLNode, elem):
    if avlnode.value==elem:                                              #Una vez que encuentro el nodo que tiene al elemento que busco eliminar
        if avlnode.leftnode==None and avlnode.rightnode==None:           #debo verificar 4 posibles casos, que el nodo no tenga nodos hijos, tenga
            if avlnode.parent.leftnode.value==elem:                      #un solo hijo a la izq, que tenga un solo hijo a la der, o que tenga
                avlnode.parent.leftnode=None                             #nodos hijos a ambos lados
            else:                                                        #Para los primeros 3 casos, corroboro a que lado del nodo padre se encuentra
                avlnode.parent.rightnode=None                            #el nodo a eliminar y procedo a actualizar los vínculos
        elif avlnode.leftnode==None and avlnode.rightnode!=None:
            if avlnode.parent.leftnode.value==elem:
                avlnode.parent.leftnode=avlnode.rightnode
            else:
                avlnode.parent.rightnode=avlnode.rightnode
            avlnode.rightnode.parent=avlnode.parent
        elif avlnode.rightnode==None:
            if avlnode.parent.leftnode.value==elem:
                avlnode.parent.leftnode=avlnode.leftnode
            else:
                avlnode.parent.rightnode=avlnode.leftnode
            avlnode.leftnode.parent=avlnode.parent
        elif avlnode.leftnode!=None and avlnode.rightnode!=None:         #Para el caso de que el nodo a eliminar tenga hijos a ambos lados
            nodoAct=AVLNode()
            nodoTemp=AVLNode()
            nodoAct=avlnode.rightnode                                    #se buscara el menor de los mayores, asi que me muevo a la derecha
            NHF=True                                                     #y en bucle busco el que sea el ultimo en tener componente a la izq
            while NHF:
                if nodoAct.leftnode==None:
                    NHF=False
                else:
                    nodoAct=nodoAct.leftnode
            nodoTemp=nodoAct.rightnode                                   #Se guarda en un nodo temporal la rama derecha del nodo que remplazara
            nodoAct.leftnode=avlnode.leftnode                            #Al no existir nada mas chico, se copia la rama izquierda
            if avlnode.rightnode!=nodoAct:                               #Se comprueba que el hijo a la derecha no sea el nodo que lo remplazara
                nodoAct.rightnode=avlnode.rightnode                      #si no lo es, se copia la rama derecha
            nodoAct.parent.leftnode=None                                 #se desvincula el nodo del padre
            if nodoAct.leftnode!=None:                                   #se actualizan las referencias a los padres de las ramas si es que existen
                nodoAct.leftnode.parent=nodoAct
            if nodoAct.rightnode!=None:
                nodoAct.rightnode.parent=nodoAct
            nodoAct.parent=avlnode.parent                                #se actualiza la propia referencia al padre
            if avlnode.parent.leftnode.value==elem:                      #se ve de que parte es hijo el nodo a eliminar para actualizar referencias
                avlnode.parent.leftnode=nodoAct
            else:
                avlnode.parent.rightnode=nodoAct
            if nodoTemp!=None:                                           #si existe una rama derecha del nodo que remplazara
                print(nodoAct.value,"/",nodoTemp.value)                  #se lo inserta en el árbol nuevamente
                __recorreInsert(nodoAct,nodoTemp)
        return avlnode.key
    else:                                                                #si no es el nodo que se busca, se ve si tiene nodos hijos para seguir la bú
        resp=None
        if avlnode.leftnode!=None:
            resp=__recorreDelete(avlnode.leftnode,elem)
        if resp!=None:
            return resp
        if avlnode.rightnode!=None:
            resp=__recorreDelete(avlnode.rightnode,elem)
        return resp
```

El resto de las funciones como Search, Update y las de recorrido no las he agregado al pdf ya que no eran parte del tp, pero se encuentran implementadas en el módulo.

Ejercicio 6:

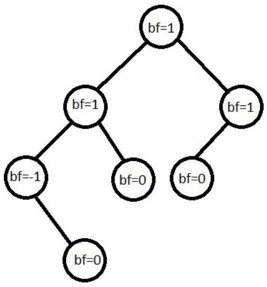
1. Responder V o F y justificar su respuesta:
   a. ___ En un AVL el penúltimo nivel tiene que estar completo
   b. ___ Un AVL donde todos los nodos tengan factor de balance 0 es completo
   c. ___ En la inserción en un AVL, si al actualizarle el factor de balance al padre del nodo insertado éste no 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. V, ya que si suponemos un AVL cuyo penúltimo nivel se encuentra incompleto, es decir que no tiene todos los nodos correspondientes al nivel que responden a la formula 2^n, donde n es el nivel del árbol en el que se esté, implicaría que en su antepenúltimo nivel hay un nodo con solo 1 hijo, y este nodo tendría como mínimo un bf de -2 o 2, ya que se ha dicho que el penúltimo nivel esta incompleto. Por lo tanto, al haber un nodo con un bf mínimo de -2 o 2 contradice a lo que hemos supuesto de que es un AVL, por lo tanto, un AVL tiene que tener su penúltimo nivel completo.

b. V, Si suponemos un AVL completo pero que tiene algún nodo con un bf distinto a 0, implicaría que ese nodo o en alguna de las ramas de sus hijos falta algún otro nodo, por lo tanto no estaría completo. Por lo tanto, si es un AVL completo todos sus nodos tienen bf=0.

c. F, ya que el desbalanceo puede ocurrir en cualquiera de los nodos superiores que se encuentran entre ese nodo insertado y la raíz.

d. F, descartando las hojas se puede construir un AVL como el de la figura que demuestra que no todos los AVL si o si tienen por lo menos un nodo con 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*.

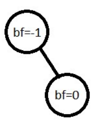
1) Encontrar las alturas de los árboles A y B y determinar cual es el de mayor tamaño.
2) En el árbol de mayor tamaño encontrar el nivel del cual el subárbol del nodo de la izquierda tenga la misma altura que el árbol de menor tamaño.
3) En el nivel anterior encontrado en el paso anterior insertar x como hijo izquierdo.
4) Como hijo izquierdo de x se inserta la raíz del árbol pequeño, y como hijo derecho de x se inserta el nodo que estaba originalmente como hijo del nodo padre de x.
5) Corroborar desde x hasta la raíz del árbol si se han producido desbalanceo, de ser así proceder con el balanceo correspondiente

Ejercicio 8:

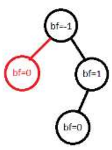
Considere una rama truncada en un AVL como un camino simple desde la raíz hacia un nodo que tenga una referencia None (que le falte algún hijo). Demuestre que la mínima longitud (cantidad de aristas) que puede tener una rama truncada en un AVL de altura h es h/2 (tomando la parte entera por abajo).

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.

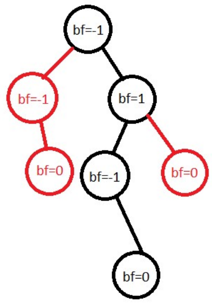
Si tenemos un árbol con un solo nodo en la raíz, el camino hacia la rama truncada es 0, ya que el propio nodo no tiene hijos. Si la altura es 0 se cumple que $\lfloor 0/2 \rfloor$ da el camino mas corto de rama truncada.



Si al nodo anterior le añadimos un hijo, ya sea a derecha o izquierda, el camino de la rama truncada es 0, ya que el nodo raíz sigue con un lugar disponible para otro hijo. Si la altura es 1 se cumple que $\lfloor 1/2 \rfloor$ da el camino más corto de rama truncada, que es de longitud 0.



Si fuera a agregar un hijo en la misma rama que he agregado uno anteriormente, no se cumpliría que este balanceado. Por lo tanto, tengo que agregar un nodo al nodo raíz para tener un árbol balanceado de altura 2.



Al hacer esto el camino de la rama truncada es de longitud 1. La altura ahora es 2, por lo que $\lfloor 2/2 \rfloor$ = 1, nuevamente se cumple que el camino es la función suelo de la altura sobre 2.

Si fuera a seguir agregando nodos en altura primero debería de completar o balancear el árbol, en los niveles anteriores.

De esta forma se puede ver que para que el camino hacia una rama truncada aumente en 1 unidad, se necesitan que se agreguen nodos en 2 niveles más. Esto se puede expresar con la función piso de la altura sobre 2. Longitud del camino de la rama truncada es igual a $\lfloor h/2 \rfloor$.