Heduan Patricio

Algoritmos y Estructuras de datos II
TP2

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 rotateLeft(Tree, node):
   rootAux = node.rightNode
   if rootAux.leftNode == None:
       node.rightNode = None
       rootAux.leftNode = node
       if node.parent == None:
           Tree.root = rootAux
           rootAux.parent = None
       else:
           rootAux.parent = node.parent
           linkParent(node, rootAux)
       node.parent = rootAux

   else:
       nodeAux = rootAux.leftNode
       node.rightNode = nodeAux
       nodeAux.parent = node
       rootAux.leftNode = node
       if node.parent == None:
           Tree.root = rootAux
           rootAux.parent = None
       else:
```

```
            rootAux.parent = node.parent
            linkParent(node, rootAux)

        node.parent = rootAux
    return rootAux


def rotateRight(Tree, node):
    rootAux = node.leftNode
    if rootAux.rightNode == None:
        node.leftNode = None
        rootAux.rightNode = node
        if node.parent == None:
            Tree.root = rootAux
            rootAux.parent = None
        else:
            rootAux.parent = node.parent
            linkParent(node, rootAux)
        node.parent = rootAux
    else:
        nodeAux = rootAux.rightNode
        node.leftNode = nodeAux
        nodeAux.parent = node
        rootAux.rightNode = node
        if node.parent == None:
            Tree.root = rootAux
            rootAux.parent = None
        else:
            rootAux.parent = node.parent
            linkParent(node, rootAux)
        node.parent = rootAux
    return rootAux
```

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

```
ef calculateBalance(Tree):

   calculateBalanceRec(Tree, Tree.root)

def calculateBalanceRec(Tree, node):

   profder = 0

   profizq = 0

   if node.rightNode != None:

       profder = calculateBalanceRec(Tree, node.rightNode)

   if node.leftNode != None:

       profizq = calculateBalanceRec(Tree, node.leftNode)

   if node.rightNode == None and node.leftNode == None:

       node.balanceFactor = 0

       return 1

   node.balanceFactor = profizq - profder



   if profizq <= profder:

       return profder + 1

   else:

       return profizq + 1
```

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.

```
def reBalance(Tree):
   node = Tree.root
   if node == None:
       return None
   else:
       if node.leftNode == None and node.rightNode == None:
           return
       else:
           reBalanceR(Tree, node)
def reBalanceR(Tree, node):
   if node.rightNode != None:
       reBalanceR(Tree, node.rightNode)
   if node.leftNode != None:
       reBalanceR(Tree, node.leftNode)
   elif node.balanceFactor < -1:
       if node.rightNode.balanceFactor > 0:
           rotateRight(Tree, node.rightNode)
           rotateLeft(Tree, node)
       else:
           rotateLeft(Tree, node)
   if node.balanceFactor > 1:
       if node.leftNode.balanceFactor < 0:
           rotateLeft(Tree, node.leftNode)
           rotateRight(Tree, node)
       else:
           rotateRight(Tree, node)
   calculateBalance(Tree)
   return
```

Ejercicio 4:

Implementar la operacion **`insert()`** en el modulo **`avltree.py`** garantizando que el arbol binario resultante sea un arbol AVL.

```
def avl_insert(B, element, key):
   currentnode = B.root
   newNode = AVLNode()
   newNode.value = element
   newNode.key = key
   newNode.balanceFactor = None
   if B.root == None:
       B.root = newNode
       KEY = B.root.key
   else:
       KEY = InsertR(currentnode, newNode)
       newNode.key
   calculateBalance(B)
   reBalance(B)
   return KEY
```

Ejercicio 5:

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

```
def avl_delete(B, element):

   key = bt_search(B, element)

   if key == None:

       return None

   else:

       bt_deletekey(B, key)

       calculateBalance(B)

       reBalance(B)

       return key
```

Ejercicio 6:

1. Responder V o F y justificar su respuesta:
    a. _F_ En un AVL el penúltimo nivel tiene que estar completo
    - Puede no estar completo y puede estar balanceado con factores de balance -1 o 1
    b. _V_ Un AVL donde todos los nodos tengan factor de balance 0 es completo
    - Cada subRaiz va a tener 2 hijos balanceados para que sea factor 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.
    - Puede haberse desbalanceado comparado con otras ramas del árbol.
    d. _V_ En todo *AVL* existe al menos un nodo con factor de balance 0.
    - Siendo estos nodos, las hojas del árbol

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*.

```
def getDepth(Tree, node):

   if node == None:

       return 0

   else:

       leftLength = getDepth(Tree, node.leftNode)

       rightLength = getDepth(Tree, node.rightNode)

       if leftLength > rightLength:

           return leftLength + 1

       return rightLength + 1

#function that returns a node pointer to the node who has the profsearch below it

def checkLeftProf(Tree, node,prof):

   if node == None:

       return None

   else:

       depht = getDepth(Tree, node)

       if depht != prof:

           return checkLeftProf(Tree,node.leftNode, prof)

       elif depht == prof:

           return node
```

```
def putXandA(Tree, node, Xkey, Atree):

   xNode = AVLNode()

   xNode.key = Xkey

   node.Parent.leftNode= xNode

   node.Parent = xNode

   if node.Parent ==Tree.root:

       Tree.root = xNode

   xNode.leftNode = Atree.root

   xNode.rightNode = node
```

```
def A_X_B(Atree, Xkey, Btree):

   maxADepth = getDepth(Atree,Atree.root)

   maxBDepth = getDepth(Btree,Btree.root)

   print(maxADepth)

   print(maxBDepth)

   if maxADepth >= maxBDepth:

       NewTree = AVLTree()

       avl_insert(NewTree, Xkey, Xkey)

       NewTree.root.leftNode = Atree.root

       NewTree.root.rightNode = Btree.root

       Atree.root.parent = NewTree.root

       Btree.root.parent = NewTree.root

       Atree.root = None

       Btree.root = None

       return NewTree

   nodePivotB = checkLeftProf(Btree, Btree.root, maxADepth)

   print(nodePivotB.key)

   if nodePivotB == None:

       return None

   putXandA(Btree,nodePivotB,Xkey,Atree)

   return Btree
```

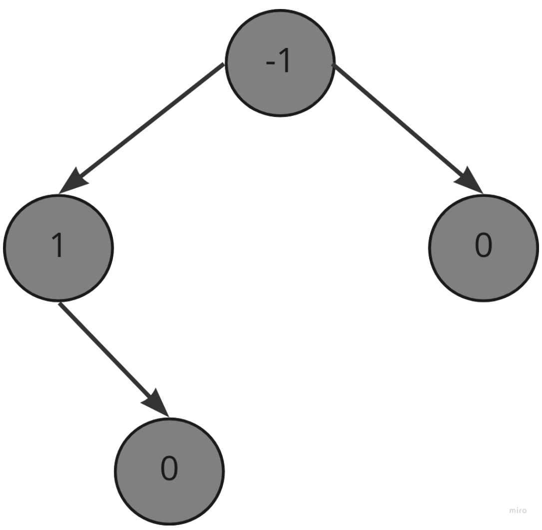
Ejercicio 8:

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.

Ejemplos (siendo los numeros de nodos los balance factors)

<u>Ejemplo 1:</u>

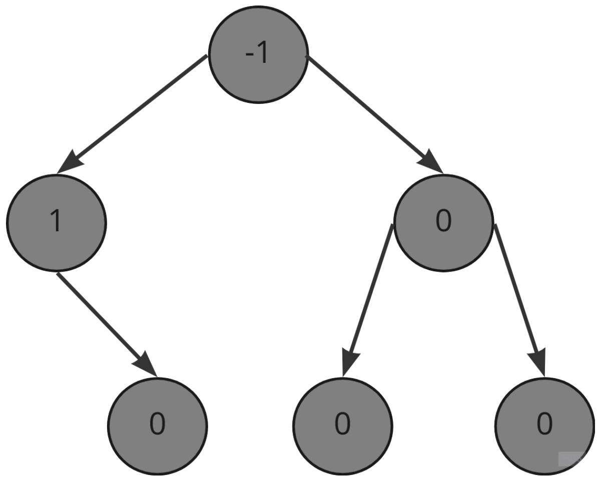


h = 2

[h/2]=1

La mínima altura a tomar, tiene longitud 1

Ejemplo 2:

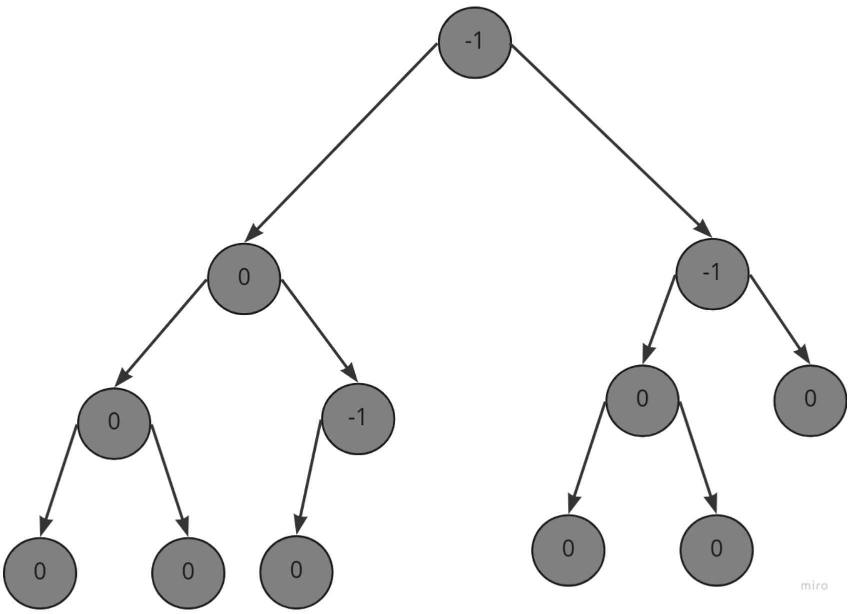


h = 2

[h/2]=1

La mínima altura a tomar, tiene longitud 1 (se repite el primer ejemplo)

Ejemplo 3:



h=3

[h/2] = 1.5 ≡ 1

Como al fin y al cabo el árbol está balanceado, y su altura es igual a 3, no puede haber un camino menor a 2 de longitud.
Con ejemplos de alturas mayores, esto último se repite al árbol está balanceado