

Tomás Rando - 14004 - Algo 2

Punto 1)

```
479 #Ejercicio 1
480 """
481 Descripción: Implementa la operación rotación a la izquierda
482 Entrada: Un Tree junto a un AVLnode sobre el cual se va a operar la rotación a la izquierda
483 Salida: retorna la nueva raíz
484 """
485 def rotateLeft(Tree, avlnode):
486
487     #Caso si el nodo es la raíz de todo el árbol
488     if Tree.root == avlnode:
489         Tree.root = avlnode.rightrightnode
490         Tree.root.parent = None
491         if Tree.root.leftnode != None:
492             avlnode.rightrightnode = Tree.root.leftnode
493             Tree.root.leftnode.parent = avlnode
494         else:
495             avlnode.rightrightnode = None
496         Tree.root.leftnode = avlnode
497         avlnode.parent = Tree.root
498         return Tree.root
499
500     #Verifico si el nodo esta a la derecha o a la izquierda del padre
501     if avlnode.parent.rightrightnode == avlnode:
502         padreHijo = "right"
503     else:
504         padreHijo = "left"
505
506     #Inserto la nueva raíz dependiendo de PadreHijo
507     avlnode.rightrightnode.parent = avlnode.parent
508     if padreHijo == "right":
509         avlnode.parent.rightrightnode = avlnode.rightrightnode
510     else:
511         avlnode.parent.leftnode = avlnode.rightrightnode
512     #Si tiene hijo izquierdo, pasa a ser el hijo derecho de la antigua raíz. Posteriormente, la antigua raíz se inserta a la izquierda de la nueva raíz.
513     newRoot = avlnode.rightrightnode
514     if newRoot.leftnode != None:
515         avlnode.rightrightnode = newRoot.leftnode
516         newRoot.leftnode.parent = avlnode
517     else:
518         avlnode.rightrightnode = None
519     newRoot.leftnode = avlnode
520     avlnode.parent = newRoot
521     return newRoot
522 """
523
524 Descripción: Implementa la operación rotación a la derecha
525 Entrada: Un Tree junto a un AVLnode sobre el cual se va a operar la rotación a la derecha
526 Salida: retorna la nueva raíz
527 """
```

```
def rotateRight(Tree, avlnode):
    #Caso si el nodo es la raíz de todo el árbol
    if Tree.root == avlnode:
        Tree.root = avlnode.leftnode
        Tree.root.parent = None
        if Tree.root.rightrightnode != None:
            avlnode.leftnode = Tree.root.rightrightnode
            Tree.root.rightrightnode.parent = avlnode
        else:
            avlnode.leftnode = None
        Tree.root.rightrightnode = avlnode
        avlnode.parent = Tree.root
        return Tree.root

    #Verifico si el nodo esta a la derecha o a la izquierda del padre
    if avlnode.parent.rightrightnode == avlnode:
        padreHijo = "right"
    else:
        padreHijo = "left"

    #Inserto la nueva raíz dependiendo de PadreHijo
    avlnode.leftnode.parent = avlnode.parent
    if padreHijo == "right":
        avlnode.parent.rightrightnode = avlnode.leftnode
    else:
        avlnode.parent.leftnode = avlnode.leftnode
    #Si tiene hijo derecho, pasa a ser el hijo izquierdo de la antigua raíz. Posteriormente, la antigua raíz se inserta a la derecha de la nueva raíz.
    newRoot = avlnode.leftnode
    if newRoot.rightrightnode != None:
        avlnode.leftnode = newRoot.rightrightnode
        newRoot.rightrightnode.parent = avlnode
    else:
        avlnode.leftnode = None
    newRoot.rightrightnode = avlnode
    avlnode.parent = newRoot
    return newRoot
```

Punto 2)

```
#Ejercicio2
'''
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 calculateBalance(AVLTree):
    return calculateBalanceR(AVLTree, AVLTree.root)

def calculateBalanceR(AVLTree, AVLNode):
    if AVLNode == None:
        return None

    AVLNode.bf = height(AVLNode.leftnode) - height(AVLNode.rightnode)

    calculateBalanceR(AVLTree, AVLNode.leftnode)
    calculateBalanceR(AVLTree, AVLNode.rightnode)
    return AVLTree
```

```
#Función para calcular la altura del árbol
def height(currentNode):
    if currentNode == None:
        return 0
    else:
        Lheight = height(currentNode.leftnode)
        Rheight = height(currentNode.rightnode)

        if Lheight > Rheight:
            return Lheight + 1
        else:
            return Rheight + 1
```

Punto 3)

```
#Se recorre el árbol verificando los bf
def recorrerBf(AVLNode):

    if (AVLNode == None):
        return None

    if (abs(AVLNode.bf) != 1) and (AVLNode.bf != 0):
        return AVLNode

    left = recorrerBf(AVLNode.leftnode)
    if left != None:
        return left
    right = recorrerBf(AVLNode.rightnode)
    if right != None:
        return right
    return None

def reBalanceR(AVLTree, AVLNode):
    if (AVLNode.bf < -1):
        if AVLNode.rightnode != None:
            if AVLNode.rightnode.bf > 0:
                rotateRight(AVLTree, AVLNode.rightnode)
            rotateLeft(AVLTree, AVLNode)
        else:
            if (AVLNode.bf > 1):
                if AVLNode.leftnode != None:
                    if AVLNode.leftnode.bf < 0:
                        rotateLeft(AVLTree, AVLNode.leftnode)
                    rotateRight(AVLTree, AVLNode)
            return AVLTree

def reBalance(AVLTree):
    AVLTree = calculateBalance(AVLTree)
    bfVerification = recorrerBf(AVLTree.root)
    if bfVerification != None:
        reBalanceR(AVLTree, bfVerification)
    return AVLTree
```

Punto 4)

```
def updateBfAndFix(AVLTree, AVLNode):
    AVLNode.bf = height(AVLNode.leftnode) - height(AVLNode.rightnode)

    if (AVLNode.bf < -1):
        if AVLNode.rightnode != None:
            if AVLNode.rightnode.bf > 0:
                rotateRight(AVLTree, AVLNode.rightnode)
            rotateLeft(AVLTree, AVLNode)
    else:
        if (AVLNode.bf > 1):
            if AVLNode.leftnode != None:
                if AVLNode.leftnode.bf < 0:
                    rotateLeft(AVLTree, AVLNode.leftnode)
                rotateRight(AVLTree, AVLNode)

    if AVLNode == AVLTree.root:
        return AVLTree
    else:
        updateBfAndFix(AVLTree, AVLNode.parent)
    return AVLTree
```

```
def insertR(newNode, currentNode):
    if newNode.key > currentNode.key:
        if currentNode.rightnode == None:
            currentNode.rightnode = newNode
            newNode.parent = currentNode
            return newNode.key
        currentNode = currentNode.rightnode
        return insertR(newNode, currentNode)
    elif newNode.key < currentNode.key:
        if currentNode.leftnode == None:
            currentNode.leftnode = newNode
            newNode.parent = currentNode
            return newNode.key
        currentNode = currentNode.leftnode
        return insertR(newNode, currentNode)
    else:
        return None
```

```
4  def insert(AVLTree, element, key):
5      newNode = AVLNode()
6      newNode.value = element
7      newNode.key = key
8
9      if AVLTree.root != None:
10         key = insertR(newNode, AVLTree.root)
11     else:
12         AVLTree.root = newNode
13         newNode.bf = 0
14         return key
15
16     if key != None:
17         updateBfAndFix(AVLTree, newNode)
18     return key
19
```

Punto 5)

#Searchforelement y searchforminor buscan nodos utilizando recursividad que luego serán utilizados en la función delete

```
def searchforelement(currentNode, element):
```

```
    if currentNode == None:
        return None
```

```
    if currentNode.value == element:
        return currentNode
```

```
    left = (searchforelement(currentNode.leftnode, element))
    if left != None:
        return left
```

```
    right = (searchforelement(currentNode.rightnode, element))
    if right != None:
        return right
```

```
def searchforminor(currentNode):
```

```
    if currentNode.leftnode == None:
        if currentNode.rightnode != None:
            if currentNode.parent.leftnode == currentNode:
                currentNode.parent.leftnode = currentNode.rightnode
            else:
                currentNode.parent.leftnode = currentNode.rightnode
        else:
            if currentNode.parent.leftnode == currentNode:
                currentNode.parent.leftnode = None
            else:
                currentNode.parent.rightnode = None
        return currentNode
    else:
        return searchforminor(currentNode.leftnode)
```

```
def delete(AVLTree, element):
```

```
    node = searchforelement(AVLTree.root, element)
    if node != None:
```

```
        if (node.rightnode != None) and (node.leftnode != None):
            minornode = searchforminor(node.rightnode)
            if AVLTree.root == node:
```

```
                minornode.parent = None
                minornode.leftnode = AVLTree.root.leftnode
                minornode.rightnode = AVLTree.root.rightnode
                if minornode.rightnode != None:
                    minornode.rightnode.parent = minornode
                if minornode.leftnode != None:
                    minornode.leftnode.parent = minornode
                AVLTree.root = minornode
```

```
        else:
```

```
            if node.parent.rightnode == node:
                node.parent.rightnode = minornode
            if node.parent.leftnode == node:
                node.parent.leftnode = minornode
            minornode.parent = node.parent
            minornode.leftnode = node.leftnode
            minornode.rightnode = node.rightnode
            if node.leftnode != None:
                node.leftnode.parent = minornode
            if node.rightnode != None:
                node.rightnode.parent = minornode
```



```

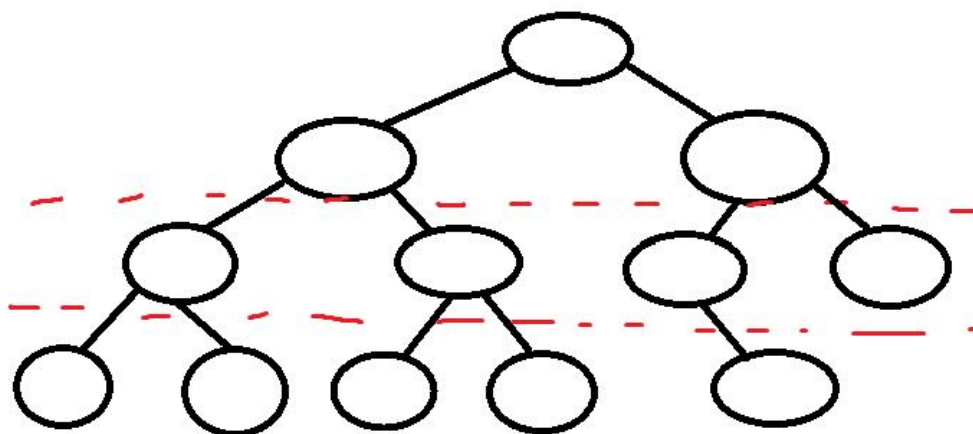
elif (node.righnode == None) and (node.leftnode == None):
    if AVLTree.root == node:
        AVLTree.root = None
    else:
        if node.parent.leftnode == node:
            node.parent.leftnode = None
        else:
            node.parent.righnode = None

elif (node.righnode == None) or (node.leftnode == None):
    if AVLTree.root == node:
        if AVLTree.root.righnode != None:
            AVLTree.root = AVLTree.root.righnode
            AVLTree.root.parent = None
        else:
            AVLTree.root = AVLTree.root.leftnode
            AVLTree.root.parent = None
    else:
        if node.parent.righnode == node:
            if node.righnode != None:
                node.parent.righnode = node.righnode
                node.parent.righnode.parent = node.parent
            else:
                node.parent.righnode = node.leftnode
                node.parent.righnode.parent = node.parent
        else:
            if node.righnode != None:
                node.parent.leftnode = node.righnode
                node.parent.leftnode.parent = node.parent
            else:
                node.parent.leftnode = node.leftnode
                node.parent.leftnode.parent = node.parent
    updateBfAndFix(AVLTree, node.parent)
    return node.key
else:
    return None

```

Punto 6)

a)

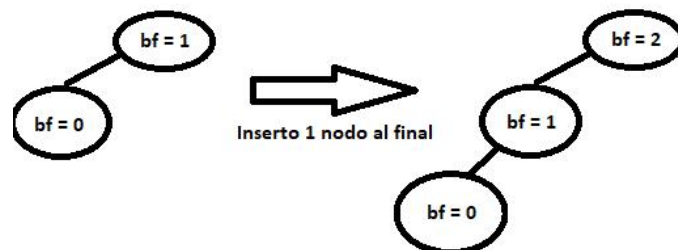


Este gráfico representa un AVL puesto que todos los bf de todos los nodos son correctos, es decir que el árbol se encuentra balanceado. Sin embargo, el penúltimo nivel no está completo pues no todos los nodos tienen dos hijos. Por ello, es falso

b)

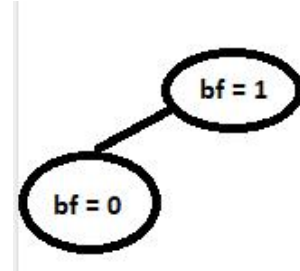
Suponemos que es falsa. Entonces suponemos que existe un AVL que tiene todos los nodos con un $bf = 0$ y no es completo. Si no es completo, significa que hay un nodo que tiene solamente un hijo. Cuando eso sucede, significa que el balance factor de ese nodo es 1 o -1. Como contradice la hipótesis, es falso.

c)



Falso, en este ejemplo, se inserta un nodo al final del AVLTree, el bf del nodo del padre cambia a 1 (No desbalanceado), sin embargo, si hay que hacer rotaciones más arriba puesto que el bf de la raíz cambia a 2 (Está desbalanceada)

d) SIN HOJAS, es falso. Ya que por ejemplo se puede tener el siguiente ejemplo (Ver imagen) en el que el bf de la raíz es 1, y, al no contar las hojas, no existe ningún nodo con $bf = 0$.



Punto 7)

Para crear el algoritmo, primero se calculan las alturas de ambos árboles (A y B). Esto se puede hacer con una complejidad de $(\log n)/(\log m)$ gracias al balanceFactor de los AVLTrees. Al tener esto, podemos contemplar 3 situaciones diferentes dependiendo de la altura calculada.

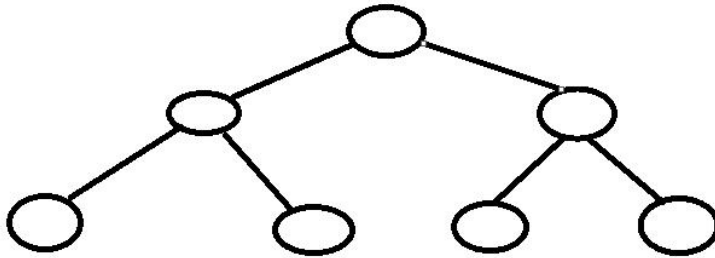
El más fácil es el caso 1, ambos árboles tienen la misma altura (o difieren en 1). En esta situación se puede usar x como la raíz del nuevo árbol, ya que los hijos derechos serán los del árbol B (Todos más grandes que este) y los izquierdos los del árbol A (Todos más chicos que x). Entonces quedará el árbol C colocando al nodo x como la raíz, a la raíz de B como el hijo derecho de x (la nueva raíz) y a la raíz de A como el hijo izquierdo de x.

En la situación 2, si la altura de B es mayor que la de A, se busca la altura de A y se va al nivel correspondiente en B. Ejemplo, si la altura de A = 3, se va al nivel 3 en B (Por los nodos más chicos). En ese nivel se coloca x como la "raíz" y el subárbol de ese nivel pasa a ser el hijo derecho de x. Y, todo A pasa a ser el hijo izquierdo de x.

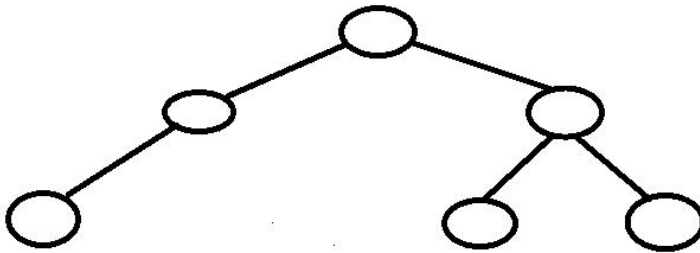
Por último, si la altura de A es mayor que la de B se hace lo mismo pero invertido, es decir, se busca la altura de B y nos dirigimos a el nivel correspondiente en A (Por los nodos más grandes). Se coloca x como raíz y ese subárbol de A pasa a estar conectado como nodo izquierdo de x. Además, se conecta a todo B con el lado derecho de la raíz.

Punto 8)

Supones que es falso. Entonces suponiendo que exista un árbol de altura h en el que la mínima ruta desde la raíz hasta el nodo con referencia none sea diferente de $h/2$, por ejemplo h . Entonces, si hacemos el esquema sería:



Sin embargo, podemos concluir que este enunciado es falso. Ya que puede existir una ruta diferente:



La cual es $h/2$. Como el enunciado negado es falso, significa que es verdadero que la mínima ruta desde la raíz hasta el nodo con referencia none en cualquier AVL es $h/2$.

Link al replit: <https://replit.com/@TomasRando/TP-1-ALGO2>

Todo el código también se encuentra en el repositorio de algoritmos2