

INFORME TALLER AVL

Juan Daniel Torres Ramirez – 2240082

- En la implementación de arboles AVL que se presentará a continuación, se agregaron los métodos de *eliminación*, *MinimumValueNode* y *recorrido Inorder*. A su vez, se corrigieron varios errores que presentaba el código base.

A continuación, se mostrará la lista de errores presentes:

ERROR	DESCRIPCIÓN
Error en la formula de factor de balance	Se tomaba el factor de balance como: $\text{AlturaSubarbolIzquierdo} - \text{SubarbolDerecho}$ cuando el orden correcto (según las diapositivas) es: $\text{AlturaSubarbolDerecho} - \text{SubarbolIzquierdo}$
No retorno de rotaciones	Las llamadas a <code>rotateLeft(node)</code> y <code>rotateRight(node)</code> no se hacían con <code>return</code> , es decir, nunca se reasignaba la nueva raíz del subárbol.
Condiciones de balance invertidas	<code>getBalance</code> estaba definido como derecha - izquierda, pero las comprobaciones (<code>if balance > 1</code> , entre otros.) seguían la lógica de izquierda - derecha.

➤ CASOS DE PRUEBA:

VALORES A INSERTAR	RESULTADOS
[10, 20, 30, 40, 50, 25]	<div>10</div> <div>20</div> <div>25</div> <div>30</div> <div>40</div> <div>50</div>

[10, 20, 30, 40, 50, 25] ELIMINANDO EL 20		10 25 30 40 50	
[10, 30, 22, 40, 51, 25, 7, 92]		7 10 22 25 30 40 51 92	
[]			

En los casos anteriores se ve como en los arboles con contenido se imprimían de menor a mayor usando el método de recorrido Inorder, por otro lado, cuando se borraba todo el contenido del árbol AVL ya no se imprimía nada como era de esperarse. En estos casos de ejemplo se introdujeron datos al azar y también se eliminaron datos para comprobar el funcionamiento del código.

CÓDIGO FINAL (CORREGIDO)

```
import sys
```

```
class Node:
```

```
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1
```

```
def getHeight(node):
```

```
    if not node:
        return 0
    return node.height
```

```

def getBalance(node):
    if not node:
        return 0
    return getHeight(node.right) - getHeight(node.left)

def updateHeight(node):
    if node:
        node.height = 1 + max(getHeight(node.left),
                                getHeight(node.right))

def rotate_right(y):
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    updateHeight(y)
    updateHeight(x)

    return x

def rotate_left(x):
    y = x.right
    T2 = y.left

    y.left = x
    x.right = T2

    updateHeight(x)
    updateHeight(y)

    return y

class AVLTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        self.root = self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if not node:
            return Node(value)

```

```

    if value < node.value:
        node.left = self._insert_recursive(node.left, value)
    elif value > node.value:
        node.right = self._insert_recursive(node.right, value)
    else:
        return node

    updateHeight(node)
    balance = getBalance(node)

    if balance > 1 and getBalance(node.right) >= 0:
        return rotate_left(node)
    if balance > 1 and getBalance(node.right) < 0:
        node.right = rotate_right(node.right)
        return rotate_left(node)

    if balance < -1 and getBalance(node.left) <= 0:
        return rotate_right(node)
    if balance < -1 and getBalance(node.left) > 0:
        node.left = rotate_left(node.left)
        return rotate_right(node)

    return node

def delete(self, val, Node):
    if Node is None:
        return Node
    elif val < Node.value:
        Node.left = self.delete(val, Node.left)
    elif val > Node.value:
        Node.right = self.delete(val, Node.right)
    else:
        if Node.left is None:
            return Node.right
        elif Node.right is None:
            return Node.left
        rgt = self.MinimumValueNode(Node.right)
        Node.value = rgt.value
        Node.right = self.delete(rgt.value, Node.right)

        Node.height = 1 + max(getHeight(Node.left),
getHeight(Node.right))
        balance = getBalance(Node)

    if balance > 1 and getBalance(Node.right) >= 0:
        return rotate_left(Node)

```

```

        if balance > 1 and getBalance(Node.right) < 0:
            Node.right = rotate_right(Node.right)
            return rotate_left(Node)

        if balance < -1 and getBalance(Node.left) <= 0:
            return rotate_right(Node)
        if balance < -1 and getBalance(Node.left) > 0:
            Node.left = rotate_left(Node.left)
            return rotate_right(Node)

    return Node

def inorder(self, root):
    if root is None:
        return
    self.inorder(root.left)
    print(root.value)
    self.inorder(root.right)

def MinimumValueNode(self, Node):
    if Node is None or Node.left is None:
        return Node
    else:
        return self.MinimumValueNode(Node.left)

avl = AVLTree()
values_to_insert = [10, 20, 30, 40, 50, 25]

print("Insertando valores:", values_to_insert)
for val in values_to_insert:
    avl.insert(val)

print("\n--- Después de inserciones ---")
avl.inorder(avl.root)
print("")

avl.delete(20, avl.root)
avl.inorder(avl.root)
print("")
avl2 = AVLTree()
values_to_insert2 = [10, 30, 22, 40, 51, 25, 7, 92]
for val in values_to_insert2:
    avl2.insert(val)
avl.inorder(avl2.root)
print("*****")
print("")

```

```
print("")

for val in [10, 30, 22, 40, 51, 25, 7, 92]:
    avl2.root = avl2.delete(val, avl2.root)
avl.inorder(avl2.root)
```