TRABAJO PRÁCTICO ARBOLES AVL

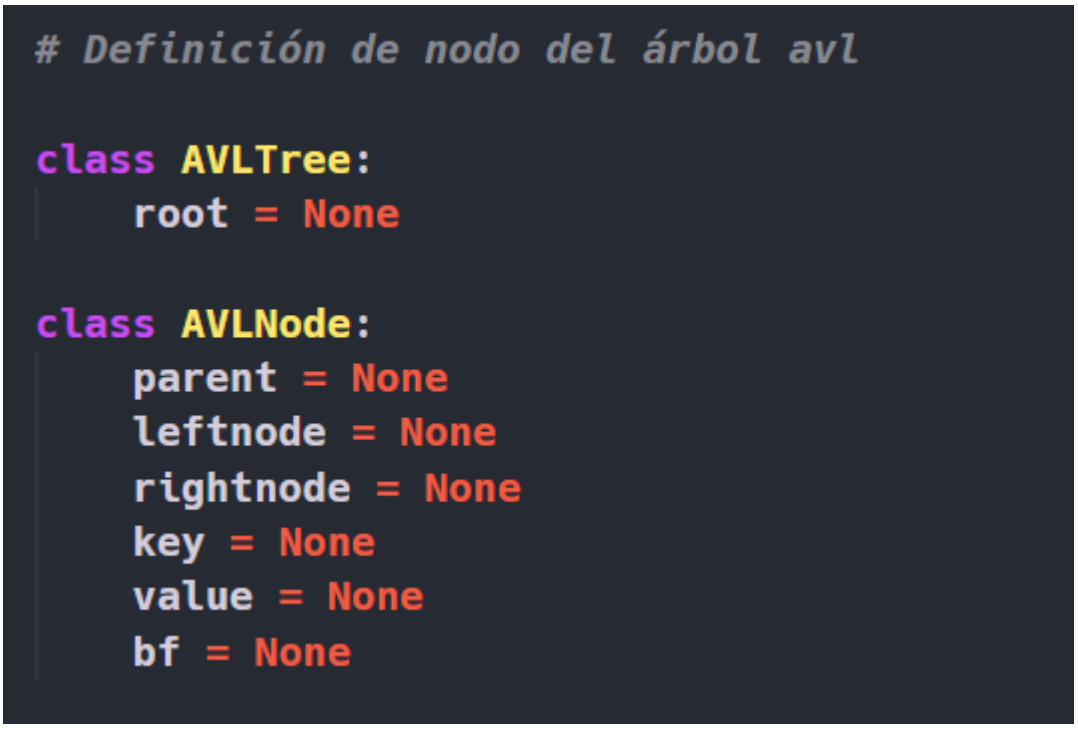
PARTE 1

INTRODUCCIÓN

A partir de estructuras definidas como :

```
class AVLTree:
      root = None


class AVLNode:
        parent = None
        leftnode = None
        rightnode = None
        key = None
        value = None
         bf = None
```



```
# Definición de nodo del árbol avl

class AVLTree:
    root = None

class AVLNode:
    parent = None
    leftnode = None
    rightnode = None
    key = None
    value = None
    bf = None
```

EJERCICIO 1

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

```python
def rotateLeft(T, x):
    """
    Rotación a la izquierda sobre el nodo x.
    T: AVLTree
    x: AVLNode
    Retorna la nueva raíz de ese subárbol
    """
    y = x.rightnode
    if y is None:
        return x  # No se puede rotar si no hay hijo derecho

    # Hacer el cambio de punteros
    x.rightnode = y.leftnode
    if y.leftnode is not None:
        y.leftnode.parent = x

    y.parent = x.parent
    if x.parent is None:
        T.root = y
    else:
        if x == x.parent.leftnode:
            x.parent.leftnode = y
        else:
            x.parent.rightnode = y

    y.leftnode = x
    x.parent = y

    return y  # Nueva raíz del subárbol
```

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(T, x):
    """
    Rotación a la derecha sobre el nodo x.
    T: AVLTree
    x: AVLNode
    Retorna la nueva raíz de ese subárbol
    """
    y = x.leftnode
    if y is None:
        return x  # No se puede rotar si no hay hijo izquierdo

    # Hacer el cambio de punteros
    x.leftnode = y.rightnode
    if y.rightnode is not None:
        y.rightnode.parent = x

    y.parent = x.parent
    if x.parent is None:
        T.root = y
    else:
        if x == x.parent.leftnode:
            x.parent.leftnode = y
        else:
            x.parent.rightnode = y

    y.rightnode = x
    x.parent = y

    return y  # Nueva raíz del subárbol
```

EJERCICIO 2

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
def calculateBalance(T):
    """
    Calcula de forma recursiva el balance factor (bf) de cada nodo del árbol.
    Entrada: AVLTree
    Salida: El mismo árbol T pero con los bf actualizados
    """

    def altura(node):
        if node is None:
            return -1  # Árbol vacío tiene altura -1
        return 1 + max(altura(node.leftnode), altura(node.rightnode))   # Altura del nodo de forma recursiva

    def calcular(node):
        if node is None:
            return
        # calcular primero en hijos
        calcular(node.leftnode)
        calcular(node.rightnode)
        # asignar balance factor
        node.bf = altura(node.leftnode) - altura(node.rightnode)

    calcular(T.root)
    return T
```

EJERCICIO 3

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
def reBalance(T):
    """
    Rebalancea el árbol AVL aplicando rotaciones según el factor de balance.
    Entrada: T (AVLTree)
    Salida: T (balanceado)
    """
    # Paso 1: recalcular balance factors
    calculateBalance(T)

    def rebalance_node(node):
        if node is None:
            return None

        # Rebalancear primero los hijos (postorden)
        rebalance_node(node.leftnode)
        rebalance_node(node.rightnode)

        # Revisar balance del nodo actual
        if node.bf > 1:  # Desbalanceado a la izquierda
            if node.leftnode is not None and node.leftnode.bf < 0:
                # Rotación doble: izquierda + derecha
                rotateLeft(T, node.leftnode)
            rotateRight(T, node)

        elif node.bf < -1:  # Desbalanceado a la derecha
            if node.rightnode is not None and node.rightnode.bf > 0:
                # Rotación doble: derecha + izquierda
                rotateRight(T, node.rightnode)
            rotateLeft(T, node)

        return node

    rebalance_node(T.root)
    # recalcular balances luego de reestructurar
    calculateBalance(T)
    return T
```

EJERCICIO 4

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

```python
def insertAVL(T, element, key):
    """
    Inserta un nodo en un árbol AVL y mantiene el balance.
    T: AVLTreegit push
    element: valor del nodo
    key: clave del nodo
    """
    # 1 Crear nodo nuevo
    nuevo = AVLNode()
    nuevo.key = key
    nuevo.value = element
    nuevo.bf = 0

    # 2 Insertar como en un BST normal
    if T.root is None:
        T.root = nuevo
        return key

    def insert_rec(node):
        if key < node.key:
            if node.leftnode is None:
                node.leftnode = nuevo
                nuevo.parent = node
                return key
            else:
                return insert_rec(node.leftnode)
        elif key > node.key:
            if node.rightnode is None:
                node.rightnode = nuevo
                nuevo.parent = node
                return key
            else:
                return insert_rec(node.rightnode)
        else:
            # Ya existe la clave
            return None

    insert_rec(T.root)

    # 3 Rebalancear todo el árbol
    reBalance(T)

    return key
```

EJERCICIO 5

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

```python
def delete(T, element):
    """
    Elimina el nodo cuyo value == element del árbol AVL T y rebalancea.
    Retorna la clave eliminada o None si no existe.
    """
    # Buscar el nodo a eliminar
    node_to_delete = _search_node(T.root, element)
    if node_to_delete is None:
        return None

    deleted_key = node_to_delete.key

    # Caso 1: hoja
    if node_to_delete.leftnode is None and node_to_delete.rightnode is None:
        _replace_node_in_parent(T, node_to_delete, None)
    # Caso 2: un solo hijo
    elif node_to_delete.leftnode is None:
        _replace_node_in_parent(T, node_to_delete, node_to_delete.rightnode)
    elif node_to_delete.rightnode is None:
        _replace_node_in_parent(T, node_to_delete, node_to_delete.leftnode)
    # Caso 3: dos hijos
    else:
        successor = _find_min(node_to_delete.rightnode)
        node_to_delete.key = successor.key
        node_to_delete.value = successor.value
        _replace_node_in_parent(T, successor, successor.rightnode)

    # Rebalancear el árbol AVL
    reBalance(T)
    return deleted_key
```

—----------

PARTE 2

—----------

EJERCICIO 6

1. En un AVL el penúltimo nivel tiene que estar completo. Falso.

En un AVL (árbol binario de búsqueda balanceado) **no necesariamente** el penúltimo nivel está completo. Lo que garantiza un AVL es que para cada nodo la diferencia de alturas entre sus subárboles sea como máximo 1, pero eso no implica que todas las posiciones del penúltimo nivel deban estar ocupadas (eso es propio de un árbol *completo* o *casi completo*).

2. Un AVL donde todos los nodos tengan factor de balance 0 es completo. Falso.

Que todos los nodos tengan factor de balance 0 significa que cada nodo tiene subárboles de igual altura. Eso lo hace un **árbol perfectamente balanceado en altura**, pero no necesariamente completo. Por ejemplo, un AVL de 3 niveles donde faltan algunos nodos en el último nivel aún puede tener FB = 0 en todos los nodos.

3. En la inserción en un AVL, si al actualizar 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. Falso.

Aunque el padre no se haya desbalanceado, su **factor de balance sí pudo haber cambiado** (por ejemplo, de 0 a 1 o de 0 a –1). Eso implica que al seguir subiendo, los ancestros pueden también cambiar sus factores de balance. Por eso, hay que seguir verificando hacia arriba hasta la raíz (o hasta que uno quede con FB = 0 y antes era ≠ 0, lo cual indica que ya no habrá más cambios).

4. En todo AVL existe al menos un nodo con factor de balance 0. Verdadero.

Siempre habrá al menos un nodo con FB = 0, ya que en la raíz (o en algún subárbol) las alturas de los subárboles pueden coincidir. De hecho, un árbol AVL de altura 1 (un solo nodo) ya tiene FB = 0 en la raíz.

—----------

EJERCICIO 7

Planteo del problema

- Tenemos **dos árboles AVL**:
  - A con m nodos
  - B con n nodos
- Además, un valor x tal que:

$$\forall a \in A,\ \forall b \in B \quad a < x < b$$

→ o sea, **todas las claves de A son menores que x y todas las claves de B son mayores que x**.

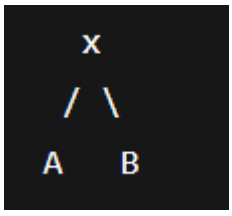
- Queremos formar un nuevo **AVL** que contenga:

$$\text{keys}(A) \cup \{x\} \cup \text{keys}(B)$$

- Restricción: el algoritmo debe ser **O(log m + log n)**.

Idea clave

Como x se ubica **entre todos los nodos de A y B**, la solución natural es:



El problema es que **las alturas de A y B pueden diferir** → necesitamos unirlos de manera balanceada.

Algoritmo (esquema)

1. **Comparar alturas**:
   Sea `hA = altura(A)` y `hB = altura(B)`.

2. **Caso 1: |hA - hB| ≤ 1**
   - Crear un nodo con clave `x`.
   - Asignar `A` como su hijo izquierdo y `B` como su hijo derecho.
   - El árbol ya es AVL.
   - Costo: `O(1)`

3. **Caso 2: hA > hB + 1**
   - Necesitamos "colocar x y B" en el subárbol derecho de `A`.
   - Desplazarnos **hA - hB - 1 niveles** hacia la derecha de `A` hasta encontrar un nodo `p` donde insertar `x`.
   - Allí `p.right` será reemplazado por un nodo `x` con:
     - `x.left = p.right`
     - `x.right = B`
   - Rebalancear hacia arriba (a lo sumo `O(log m)`).

4. **Caso 3: hB > hA + 1** (simétrico)
   - Similar al anterior, pero recorriendo hacia la izquierda en `B`.
   - Se inserta `x` entre `A` y el subárbol izquierdo de `B`.
   - Rebalancear en `O(log n)`.

Complejidad

- Solo bajamos como máximo `|hA - hB|` niveles en el árbol mayor.
- Eso cuesta `O(log m)` o `O(log n)` según el caso.
- El rebalanceo en AVL también es `O(log m)` o `O(log n)`.
- En total: **O(log(m)+log(n))**

Pseudocódigo

```python
def joinAVL(A, x, B):
    if abs(height(A) - height(B)) <= 1:
        node = Node(x)
        node.left = A
        node.right = B
        updateHeight(node)
        return node

    if height(A) > height(B) + 1:
        # bajar por el hijo derecho de A
        A.right = joinAVL(A.right, x, B)
        return rebalance(A)

    if height(B) > height(A) + 1:
        # bajar por el hijo izquierdo de B
        B.left = joinAVL(A, x, B.left)
        return rebalance(B)
```

EJERCICIO 7

Definiciones

- **Altura de un AVL = h**:
  La longitud máxima desde la raíz hasta una hoja.

- **Rama truncada**:
  Un camino desde la raíz hasta un nodo que **carece de algún hijo** (None).

Idea intuitiva

- En un AVL, la diferencia de altura entre los dos subárboles de cada nodo es como máximo 1.

- Entonces, si un nodo no está completo, significa que **uno de sus hijos tiene menor altura que el otro**.

- Para “empujar” lo más abajo posible la **primera falta de hijo**, el árbol debe estar lo más lleno posible, con la excepción mínima que provoca el desbalance permitido.

- Eso lleva a que el **camino truncado más corto** tenga al menos la mitad de la altura total.

Demostración

1. **Cota inferior de la altura mínima de un AVL con altura h**
   Se sabe que en un AVL la altura mínima de un subárbol está acotada por la **recurrencia de Fibonacci**:

   $$N(h) \geq N(h-1) + N(h-2) + 1$$

   donde N(h) es el número mínimo de nodos en un AVL de altura h.

   Eso implica que para un nodo de altura h, **el subárbol más bajo que puede tener** tiene altura al menos h−2.

2. **Dónde aparece la primera rama truncada**

- Supongamos que seguimos siempre por el lado “más corto” en cada nodo.

- Como el árbol está balanceado, ese camino “más corto” tiene altura al menos:

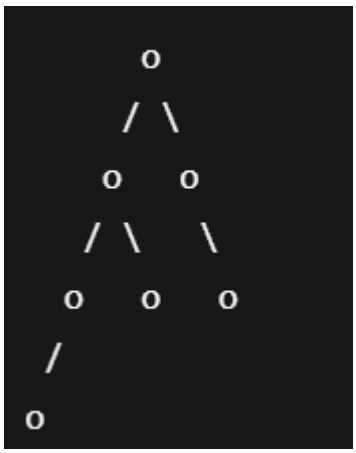
$$\left\lfloor \frac{h}{2} \right\rfloor$$

- Esto significa que para llegar a un nodo con un hijo `None`, hemos tenido que descender al menos esa cantidad de niveles.

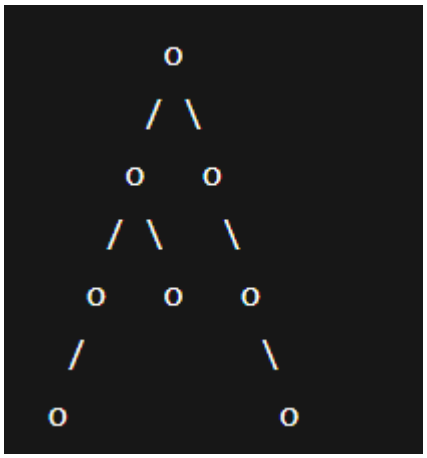
3. **Conclusión**
   Por lo tanto, cualquier rama truncada en un AVL de altura h tiene longitud mínima: $\lfloor \frac{h}{2} \rfloor$

Ejemplo gráfico

- **AVL de altura 4**:
  - Altura total: 4
  - Rama truncada más corta: longitud = 2
  - Coincide con ⌊4/2⌋ = 2.



- **AVL de altura 5**:
  - Altura total: 5
  - Rama truncada más corta: longitud = 2
  - Coincide con ⌊5/2⌋ = 2.



PARTE 3

OPCIONAL
