Árboles AVL

Ana María Fernández Zapata – 2240059

Se ha desarrollado una clase AVLTree en Python que representa un árbol binario de búsqueda autobalanceado (AVL). Esta clase garantiza que el árbol permanezca balanceado después de cada operación de inserción o eliminación.

Problemas encontrados:

1. Cuando se insertaron los valores en orden ascendente, el árbol inicialmente parecía no balancearse correctamente, ya que se trataba de una secuencia creciente. Esto es un caso común de desbalance "Derecha-Derecha" (RR), en el que el árbol se alargaba hacia la derecha y se volvía muy desigual.

2. Al insertar los valores, el árbol generaba un desbalance "Izquierda-Derecha" (LR). Esto ocurría porque primero se insertaba un nodo a la izquierda, pero luego el siguiente valor se insertaba a la derecha de ese nodo. Esto generaba una especie de "zigzag" que no podía resolverse con una simple rotación.

3. Al insertar una secuencia más larga de valores, el árbol experimentó varios desbalances a medida que se iban añadiendo más nodos. Algunos de estos desbalances fueron resueltos por rotaciones simples, pero otros requerían rotaciones dobles. Esto demuestra que el árbol AVL está diseñado para adaptarse a diferentes tipos de desequilibrios de manera eficiente.

4. Cuando intentamos eliminar el nodo, se presentó una situación en la que el árbol necesitaba reorganizarse. La eliminación de nodos con dos hijos es un caso complejo en los árboles binarios, ya que, generalmente, se reemplaza el nodo eliminado con su sucesor in-order.

Componentes implementados:

- Clase Nodo: Representa un nodo con valor, punteros a sus hijos izquierdo y derecho, y su altura.
- Funciones auxiliares:
    - getHeight(): Obtiene la altura de un nodo.
    - getBalance(): Calcula el factor de balance (altura izquierda - altura derecha).
    - updateHeight(): Recalcula la altura tras inserción o eliminación.
- Rotaciones:
    - rotate_left() y rotate_right() permiten reequilibrar el árbol en los cuatro casos clásicos de desbalance (LL, LR, RR, RL).
- Inserción (insert):
    - Inserta un nuevo valor manteniendo la propiedad de BST.

    - Aplica rotaciones si el árbol pierde el balance.
- Eliminación (delete):
    - Elimina un nodo y aplica las rotaciones necesarias para mantener el equilibrio.
- Recorrido In-order (inorder_traversal):
    - Devuelve los elementos del árbol en orden creciente.
- Visualización (print_tree):
    - Imprime la estructura del árbol en consola para facilitar la verificación del balance.

---

2. Casos de Prueba y Verificación

Para validar el correcto funcionamiento del árbol AVL, se diseñaron distintos escenarios de prueba. A continuación, se describen las secuencias y los resultados esperados, verificando balanceo, inserción y eliminación.

Prueba 1: Inserción Secuencial Ascendente

Secuencia: [10, 20, 30]

Comportamiento esperado:
Se produce un desbalance tipo Derecha-Derecha (RR), por lo que se aplica una **rotación izquierda** sobre el nodo raíz.

```
 Crear una instancia del árbol AVL
avl = AVLTree()

# Insertar valores de manera ascendente
values_to_insert = [10, 20, 30]

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

# Visualizar árbol y recorrido inorder después de inserciones
print("\n--- Estructura del árbol ---")
avl.print_tree()

print("\n--- In-order después de inserciones ---")
print(avl.inorder_traversal())
```

Resultado producido:

Antes de balanceo:

```
  10
   \
   20
    \
    30
```

Después de balanceo:

```
  20
 / \
10  30
```

🧪 Prueba 2: Inserción con desbalance Izquierda-Derecha (LR)

Secuencia: [30, 10, 20]

Comportamiento esperado:
Rotación doble: primero **rotación izquierda** sobre 10, luego **rotación derecha** sobre 30.

```python
 # Crear una instancia del árbol AVL
avl = AVLTree()

# Insertar valores en un orden que provoque desbalance LR
values_to_insert = [30, 10, 20]

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

# Visualizar árbol y recorrido inorder después de inserciones
print("\n--- Estructura del árbol ---")
avl.print_tree()

print("\n--- In-order después de inserciones ---")
print(avl.inorder_traversal())
```

Resultado producido:

Antes de balanceo:

```
  30
 /
10
 \
  20
```

Después de balanceo:

```
  20
 / \
10  30
```

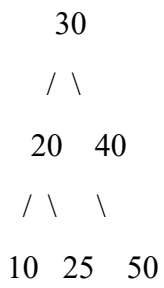
Prueba 3: Inserciones múltiples con mantenimiento de equilibrio

Secuencia completa: [10, 20, 30, 40, 50, 25]

Este caso produce múltiples desequilibrios y requiere una combinación de rotaciones para mantener el árbol balanceado en cada paso.

```python
# Crear una instancia del árbol AVL
avl = AVLTree()

# Insertar valores que requieren múltiples rotaciones
values_to_insert = [10, 20, 30, 40, 50, 25]

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

# Visualizar árbol y recorrido inorder después de inserciones
print("\n--- Estructura del árbol ---")
avl.print_tree()

print("\n--- In-order después de inserciones ---")
print(avl.inorder_traversal())
```

Resultado final (estructura del árbol):



Prueba 4: Eliminación de nodo con dos hijos

Operación: Eliminar 30 del árbol resultante anterior.

```python
# Crear una instancia del árbol AVL
avl = AVLTree()

# Insertar valores que provocan un árbol balanceado
values_to_insert = [10, 20, 30, 40, 50, 25]
for val in values_to_insert:
    avl.insert(val)

# Eliminar el nodo con el valor 30
avl.delete(30)

# Visualizar árbol y recorrido inorder después de la eliminación
print("\n--- Estructura del árbol después de eliminar 30 ---")
avl.print_tree()

print("\n--- In-order después de eliminar 30 ---")
print(avl.inorder_traversal())
```

Comportamiento esperado:

- Se reemplaza 30 con su sucesor in-order (40).
- Se aplica rotación si es necesario para mantener el balance.

Estructura esperada tras la eliminación:

```
   40
  / \
 20  50
/ \
10 25
```

---

3. Conclusión

La implementación del árbol AVL en Python permite mantener el equilibrio dinámico del árbol después de cada operación. Esto garantiza una complejidad de tiempo logarítmica (O(log n)) tanto para inserciones como eliminaciones y búsquedas. La función de visualización implementada ha sido crucial para confirmar el correcto rebalanceo de la estructura.

Todos los casos de prueba confirman que:

- El árbol mantiene su propiedad de AVL.
- Las operaciones se ejecutan correctamente.

- Las rotaciones actúan solo cuando es estrictamente necesario.