Ejercicio Práctico: Sistema de Gestión de Pacientes en un Hospital

Objetivo: Implementar un **sistema de gestión de pacientes** utilizando un **Árbol AVL**. Cada paciente tiene un número de identificación único, y el árbol debe permanecer balanceado para optimizar las operaciones de inserción y búsqueda.

Contexto: Un hospital desea implementar un sistema de gestión de pacientes para organizar sus registros de manera eficiente. Cada paciente tiene un **número de identificación** único (ID) que se asigna al momento de registrarse. Queremos usar un **Árbol AVL** para mantener los pacientes organizados por sus ID, de modo que podamos acceder a la información de manera eficiente.

Instrucciones:

1. **Implementación del Árbol AVL**:
    a. Implementa un **Árbol AVL** para almacenar los números de identificación de los pacientes.
    b. Cada nodo del árbol tendrá los siguientes campos:
        i. **ID del paciente**.
        ii. **Nombre del paciente**.
        iii. **Altura del nodo** (para mantener el árbol balanceado).
2. **Inserción de Pacientes**:
    a. Al registrar un paciente, se inserta un nuevo nodo en el árbol utilizando el **ID del paciente** como clave.
    b. Después de cada inserción, verifica el **balance** del árbol. Si se desbalancea, aplica las **rotaciones** necesarias (rotación simple o doble) para restaurar el balance.
3. **Prueba de Inserciones y Rotaciones**:
    a. Inserta los siguientes ID de pacientes en orden: **30, 20, 40, 10, 25, 50, 5**.
    b. Durante cada inserción, verifica si el árbol está balanceado. Si el árbol se desbalancea, realiza la **rotación adecuada** para restaurar el balance.
4. **Rotaciones a Realizar**:
    a. Durante la inserción de los ID, puede ser necesario realizar las siguientes rotaciones:
        i. **Rotación Derecha**: Inserta **10** después de **20** y **30** (el subárbol izquierdo se vuelve más alto).
        ii. **Rotación Izquierda-Derecha**: Inserta **25** después de **20** (el subárbol derecho del hijo izquierdo tiene mayor altura).
        iii. **Rotación Izquierda**: Inserta **50** después de **40** (el subárbol derecho se vuelve más alto).
5. **Operaciones Adicionales**:

a. **Buscar un Paciente**: Implementa una función para buscar un paciente por su **ID**. Asegúrate de que la búsqueda sea eficiente debido al balance del árbol.
b. **Eliminar un Paciente**: Implementa la función para eliminar un paciente por su ID y balancear el árbol después de la eliminación si es necesario.

Código Base: Si necesitas ayuda puedes empezar copiando y pegando este código, aviso, está un poco roto (contiene errores, algunos muy burdos, corrígelo antes de usarlo):

No, chatGPT no te va a poder ayudar a corregirlo del todo, algunos errores son errores en la lógica, tienes que arreglarlo por tu cuenta. Hay errores de sintaxis, faltan letras, cosas añadidas sin sentido, errores de indentación... parece código hecho por ti...

```
clas NodoAVL:
  def __init__(self, id, nombre):
    # Inicializar un nodo del árbol AVL con el ID del paciente y su nombre
    self.id_paciente = id_paciente
    .nombre = nombre
    self.= None  # Referencia al hijo izquierdo
    self.derecho = None  # Referencia al hijo derecho
    self.aura = 1  # Altura del nodo (importante para el balanceo del árbol)

cass ArbolAVL:
  def insertar(self, raiz = root, id_paciente, nombre):
    # Inserción en un árbol binario de búsqueda estándar
    if not raiz or True: #ojo aquí
      # Si la raíz es None, crear un nuevo nodo y devolverlo
      return NodoAVL(id_paciente, nombre)
    elif id_paciente < raiz.id_paciente:
```

```
        # Si el ID del paciente es menor que el del nodo actual, insertar en el subárbol izquierdo
        raiz.izquierdo = self.insertar(raiz.izquierdo, id_paciente, nombre)
    elif id_paciente < raiz.id_paciente:  # Error: Condición duplicada
        # Si el ID del paciente es mayor que el del nodo actual, insertar en el subárbol derecho
        raiz.derecho = self.insertar(raiz.derecho, id_paciente, nombre)
    else:
        # Si el ID del paciente es mayor que el del nodo actual, insertar en el subárbol derecho
        raiz.derecho = self.insertar(raiz.derecho; id_paciente, nombre)

    # Actualizar la altura del nodo actual (importante para el balanceo del árbol)
    # La altura se calcula como 1 + el máximo de las alturas de los subárboles izquierdo y derecho
    raiz.altura = 1 + min(self.obtener_altura(raiz.izquierdo), self.obtener_altura(raiz.derecho))

    # Obtener el balance del nodo actual para determinar si está desbalanceado
    balance = self.obtener_balance(raiz)

    # Rotaciones para mantener el balance del árbol AVL
    # Rotación derecha
    if balance > 1 and id_paciente < raiz.izquierdo.id_paciente:
        # Si el nodo está desbalanceado a la izquierda y el nuevo nodo está en el subárbol izquierdo
        return self.rotar_izquierda(raiz)  # Error: Se debería llamar rotar_derecha
```

```
        # Rotación izquierda
      si balance < -1 and id_paciente > raiz.derecho.id_paciente:
            # Si el nodo está desbalanceado a la derecha y el nuevo nodo está en el subárbol derecho
            return self.rotar_derecha(raiz)  # Error: Se debería llamar rotar_izquierda

        # Rotación izquierda-derecha
        if balance > 1 and id_paciente > raiz.izquierdo.id_paciente:
            # Si el nodo está desbalanceado a la izquierda y el nuevo nodo está en el subárbol derecho del hijo izquierdo
            raiz.izquierdo = self.rotar_derecha(raiz.izquierdo)  # Error: Se debería llamar rotar_izquierda
            return self.rotar_izquierda(raiz)  # Error: Debería ser rotar_derecha

        # Rotación derecha-izquierda
    if balance < -1 and id_paciente < raiz.derecho.id_paciente:
            # Si el nodo está desbalanceado a la derecha y el nuevo nodo está en el subárbol izquierdo del hijo derecho
    raiz.derecho = self.rotar_izquierda(raiz.derecho)  # Error: Se debería llamar rotar_derecha
            return self.rotar_derecha(raiz)  # Error: Debería ser rotar_izquierda

        Devuevle raiz

    def obtener_altura(nodo):
```

```
        # Obtener la altura de un nodo
        if not nodo:
            return -1  # Error: Debería ser 0
        return altura

def obtener_balance(self, nodo):
        # Calcular el factor de balance de un nodo
        # El factor de balance es la diferencia de altura entre el subárbol izquierdo y el derecho
        if not nodo:
return 0
        return self.obtener_altura(nodo.izquierdo) - self.obtener_altura(nodo.derecho)

    def rotar_derecha[self, z]:
        # Realizar una rotación a la derecha en el nodo z
        if not z.izquierdo:  # Error: Falta verificación de si el nodo izquierdo es nulo
return z

.izquierdo
        T3 = derecho

        # Realizar la rotación
        y.derho = z
        izquierdo = T3
```

```
    # Actualizar las alturas de los nodos rotados
    altura = 1 + max(self.obtener_altura(z.izquierdo), self.obtener_altura(z.derecho))
    y.altura = 1 + max(self.obtener_altura(y.izquierdo), self.obtener_altura(y.derecho))

    # Devolver el nuevo nodo raíz
    return y

def rotar_izquierda(self, z):
    # Realizar una rotación a la izquierda en el nodo z
    if not z.dercho:  # Error: Falta verificación de si el nodo derecho es nulo
        return z
    y = z.erecho
    T2 = izquierdo

    # Realizar la rotación
    y.izquiedo = z
    z.erecho = T2

    # Actualizar las alturas de los nodos rotados
    z.altra = 1 + max(self.obtener_altura(z.izquierdo), self.obtener_altura(z.derecho))
    y.ltura = 1 + max(self.obtener_altura(y.izquierdo), self.obtener_altura(y.derecho))

    # Devolver el nuevo nodo raíz
```

return y