Seguimiento Balanceado de Disponibilidad de Vehículos

Route Manager necesita un sistema robusto para monitorear la disponibilidad y capacidad de su flota de vehículos. A medida que los vehículos entran o salen de servicio, o su estado cambia, este índice debe actualizarse de manera eficiente, manteniendo un equilibrio que garantice búsquedas rápidas. Esto es crucial para la asignación de recursos en tiempo real.

Historia de usuario Nro.	1	**Título:**	Mantener Índice Balanceado de Vehículos
Descripción	COMO:	Gestor de Rutas	
	QUIERO:	Mantener un índice de vehículos disponibles, ordenado por capacidad o ID, que se actualice rápidamente y garantice búsquedas eficientes en todo momento	
	PARA:	Asignar vehículos a rutas de manera óptima y sin demoras, incluso con alta rotación de la flota.	
Criterios de aceptación	● Los vehículos se identificarán por un ID numérico único. ● Se debe poder agregar un nuevo vehículo (ID) al índice. ● Se debe poder eliminar un vehículo (ID) del índice cuando esté fuera de servicio. ● Después de cada inserción o eliminación, el árbol debe mantener su propiedad de balanceo (altura de subárboles difiere en no más de 1). ● Se debe poder buscar un vehículo por su ID y confirmar si existe. ● Si se intenta buscar o eliminar un ID no existente, se debe indicar "VEHICULO NO ENCONTRADO". ● Después de cada operación que modifique el árbol, se debe imprimir un recorrido in-order para verificar el orden y el balance (implícitamente por la estructura).		

Historia de Usuario Nro. 1: Mantener Índice Balanceado de Vehículos

La exigencia de "garantizar búsquedas eficientes en todo momento" aborda directamente la vulnerabilidad de rendimiento O(N) en el peor de los casos de un BST simple. Para un sistema operativo crítico como Route Manager, donde la disponibilidad de vehículos afecta las asignaciones de rutas en tiempo real, un rendimiento impredecible es inaceptable. Los árboles AVL, al asegurar operaciones O(log N) mediante rotaciones de auto-balanceo, proporcionan la consistencia de rendimiento necesaria. Esta decisión de diseño se basa en la necesidad de fiabilidad del sistema y garantías de rendimiento, más allá de la eficiencia promedio.

Programa para Disponibilidad de Vehículos con Árbol AVL

La estructura conceptual del programa en Java implicaría una clase AVLNode que contendría el vehicleId, la height del nodo, y referencias a los nodos left y right. Una clase VehicleAVLTree encapsularía la lógica del árbol AVL, incluyendo métodos para insert(int vehicleId), delete(int vehicleId), search(int vehicleId), y funciones auxiliares como getBalance(AVLNode node), updateHeight(AVLNode node), rotateLeft(AVLNode node), rotateRight(AVLNode node), getBalanceFactor(AVLNode node), y rebalance(AVLNode node), además de inOrderTraversal().

Diagrama de Flujo del Programa en Python:

```
A[Inicio] --> B[Inicializar Árbol AVL Vacío];
B --> C{Bucle: Leer comando hasta "END"};
C -- "Comando es ADD <ID>" --> D;
D --> E;
C -- "Comando es REMOVE <ID>" --> F;
F -- "Eliminado" --> G;
F -- "No Encontrado" --> H;
G --> I;
H --> I;
C -- "Comando es SEARCH <ID>" --> J;
J -- "Encontrado" --> K;
J -- "No Encontrado" --> L;
D --> C;
E --> C;
I --> C;
K --> C;
L --> C;
C -- "Comando es END" --> M;
M --> N[Fin];
```

Entrada	• ADD <vehicle_id>: Añade un nuevo vehículo con el ID especificado. • REMOVE <vehicle_id>: Elimina un vehículo por su ID. • SEARCH <vehicle_id>: Busca un vehículo por su ID. • END: Termina la entrada.
Salida	• Para ADD y REMOVE: Se debe imprimir el recorrido in-order del árbol después de la operación. Los IDs deben estar separados por espacios en una sola línea. • Para REMOVE: Se debe imprimir "ELIMINADO" si el ID fue eliminado con éxito, o "VEHICULO NO ENCONTRADO" si no se encontró. • Para SEARCH: Se debe imprimir "ENCONTRADO" si el ID existe, o "VEHICULO NO ENCONTRADO" en caso contrario.

Instrucciones para Calificación Automática:

- La clase principal debe llamarse ArbolAVL.
- Dentro de la clase ArbolAVL, debe existir un método llamado ejecutar.
- Únicamente se deben imprimir las salidas especificadas.

Casos de prueba (visibles)

Entradas de ejemplo 1	ADD 10 ADD 20 ADD 30 END
Salida de ejemplo 1	10 10 20 10 20 30

Entradas de ejemplo 2	ADD 30 ADD 10 ADD 20 END
Salida de ejemplo 2	30 10 30 10 20 30

Entradas de ejemplo 3	ADD 50 ADD 30 ADD 70 SEARCH 30 SEARCH 90 SEARCH 50 END
Salida de ejemplo 3	50 30 50 30 50 70 ENCONTRADO VEHICULO NO ENCONTRADO ENCONTRADO

Entradas de ejemplo 4	ADD 50 ADD 30 ADD 70 REMOVE 30 END
Salida de ejemplo 4	50 30 50 30 50 70 ELIMINADO 50 70

Entradas de ejemplo 5	ADD 50 ADD 30 ADD 70 ADD 60 REMOVE 70 END
Salida de ejemplo 5	50 30 50 30 50 70 30 50 60 70 ELIMINADO 30 50 60

Casos de prueba (ocultos)

Caso prueba	Entrada	Salida esperada
1	ADD 50 ADD 30 ADD 70 ADD 60 ADD 80 REMOVE 70 END	50 30 50 30 50 70 30 50 60 70 30 50 60 70 80 ELIMINADO 30 50 60 80
2	ADD 50 ADD 30 REMOVE 100 END	50 30 50 VEHICULO NO ENCONTRADO 30 50
3	ADD 10 ADD 20 ADD 30 REMOVE 10 ADD 40 ADD 50 ADD 25 REMOVE 40 END	10 10 20 10 20 30 ELIMINADO 20 30 20 30 40 20 30 40 50 20 25 30 40 50 ELIMINADO 20 25 30 50

Desarrollo: Implementación del Código Python

Código Python :

```python
# arbol avl

class AVLNode:

    def __init__(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1



class ArbolAVL:

    def __init__(self):

        self.root = None



    def _get_height(self, node):

        if not node:

            return 0

        return node.height



    def _update_height(self, node):

        if node:

            node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))



    def _get_balance_factor(self, node):

        if not node:

            return 0

        return self._get_height(node.left) - self._get_height(node.right)



    def _rotate_right(self, y):
```

```
        x = y.left
        T2 = x.right

        x.right = y
        y.left = T2

        self._update_height(y)
        self._update_height(x)
        return x

    def _rotate_left(self, x):
        y = x.right
        T2 = y.left

        y.left = x
        x.right = T2

        self._update_height(x)
        self._update_height(y)
        return y

    def _rebalance(self, node):
        self._update_height(node)
        balance = self._get_balance_factor(node)

        # Left Left Case
        if balance > 1 and self._get_balance_factor(node.left) >= 0:
```

```
            return self._rotate_right(node)

        # Left Right Case
        if balance > 1 and self._get_balance_factor(node.left) < 0:
            node.left = self._rotate_left(node.left)
            return self._rotate_right(node)

        # Right Right Case
        if balance < -1 and self._get_balance_factor(node.right) <= 0:
            return self._rotate_left(node)

        # Right Left Case
        if balance < -1 and self._get_balance_factor(node.right) > 0:
            node.right = self._rotate_right(node.right)
            return self._rotate_left(node)

        return node

    def _insert_recursive(self, node, key):
        if not node:
            return AVLNode(key)

        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        else: # Allow duplicate keys to go to the right, or handle as per specific requirement
            node.right = self._insert_recursive(node.right, key)
```

```
        return self._rebalance(node)

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

    def _min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

    def _delete_recursive(self, root, key):
        if not root:
            return root

        if key < root.key:
            root.left = self._delete_recursive(root.left, key)
        elif key > root.key:
            root.right = self._delete_recursive(root.right, key)
        else: # Node to be deleted found
            if not root.left or not root.right: # Node with 0 or 1 child
                temp = root.left if root.left else root.right
                root = None # Delete the node
                return temp
            else: # Node with two children
                temp = self._min_value_node(root.right)
                root.key = temp.key
```

```
                root.right = self._delete_recursive(root.right, temp.key)

        if not root:
            return root

        return self._rebalance(root)

    def delete(self, key):
        # Check if key exists before attempting deletion to conform to output requirements
        if not self.search(key):
            return False # Indicate not found
        self.root = self._delete_recursive(self.root, key)
        return True # Indicate deleted

    def _search_recursive(self, node, key):
        if not node or node.key == key:
            return node
        if key < node.key:
            return self._search_recursive(node.left, key)
        return self._search_recursive(node.right, key)

    def search(self, key):
        return self._search_recursive(self.root, key) is not None

    def _in_order_traversal_recursive(self, node, result):
        if node:
            self._in_order_traversal_recursive(node.left, result)
```

```
            result.append(str(node.key))
            self._in_order_traversal_recursive(node.right, result)
# arbol avl
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

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

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def _update_height(self, node):
        if node:
            node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

    def _get_balance_factor(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)
```

```
    def _rotate_right(self, y):
        x = y.left
        T2 = x.right

        x.right = y
        y.left = T2

        self._update_height(y)
        self._update_height(x)
        return x

    def _rotate_left(self, x):
        y = x.right
        T2 = y.left

        y.left = x
        x.right = T2

        self._update_height(x)
        self._update_height(y)
        return y

    def _rebalance(self, node):
        self._update_height(node)
        balance = self._get_balance_factor(node)
```

```
        # Left Left Case
        if balance > 1 and self._get_balance_factor(node.left) >= 0:
            return self._rotate_right(node)

        # Left Right Case
        if balance > 1 and self._get_balance_factor(node.left) < 0:
            node.left = self._rotate_left(node.left)
            return self._rotate_right(node)

        # Right Right Case
        if balance < -1 and self._get_balance_factor(node.right) <= 0:
            return self._rotate_left(node)

        # Right Left Case
        if balance < -1 and self._get_balance_factor(node.right) > 0:
            node.right = self._rotate_right(node.right)
            return self._rotate_left(node)

        return node

    def _insert_recursive(self, node, key):
        if not node:
            return AVLNode(key)

        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        else: # Allow duplicate keys to go to the right, or handle as per specific
requirement
```

```
            node.right = self._insert_recursive(node.right, key)

        return self._rebalance(node)

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

    def _min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

    def _delete_recursive(self, root, key):
        if not root:
            return root

        if key < root.key:
            root.left = self._delete_recursive(root.left, key)
        elif key > root.key:
            root.right = self._delete_recursive(root.right, key)
        else: # Node to be deleted found
            if not root.left or not root.right: # Node with 0 or 1 child
                temp = root.left if root.left else root.right
                root = None # Delete the node
                return temp
            else: # Node with two children
```

```
                temp = self._min_value_node(root.right)
                root.key = temp.key
                root.right = self._delete_recursive(root.right, temp.key)

        if not root:
            return root

        return self._rebalance(root)

    def delete(self, key):
        # Check if key exists before attempting deletion to conform to output requirements
        if not self.search(key):
            return False # Indicate not found
        self.root = self._delete_recursive(self.root, key)
        return True # Indicate deleted

    def _search_recursive(self, node, key):
        if not node or node.key == key:
            return node
        if key < node.key:
            return self._search_recursive(node.left, key)
        return self._search_recursive(node.right, key)

    def search(self, key):
        return self._search_recursive(self.root, key) is not None

    def _in_order_traversal_recursive(self, node, result):
```

```
        if node:
            self._in_order_traversal_recursive(node.left, result)
            result.append(str(node.key))
            self._in_order_traversal_recursive(node.right, result)

    def in_order_traversal(self):
        result = []
        self._in_order_traversal_recursive(self.root, result)
        return " ".join(result)

    def ejecutar(self):
        while True:
            try:
                line = input().strip()
                if line == "END":
                    break

                parts = line.split()
                command = parts[0]
                key = int(parts[1]) if len(parts) > 1 else None

                if command == "ADD":
                    self.insert(key)
                    print(self.in_order_traversal())
                elif command == "REMOVE":
                    if self.delete(key):
                        print("ELIMINADO")
```

```
                    else:
                        print("VEHICULO NO ENCONTRADO")
                    print(self.in_order_traversal())
                elif command == "SEARCH":
                    if self.search(key):
                        print("ENCONTRADO")
                    else:
                        print("VEHICULO NO ENCONTRADO")
            except EOFError:
                break
            except Exception as e:
                # print(f"Error: {e}") # For debugging
                pass # Ignore malformed input for auto-grading simplicity


if __name__ == "__main__":
    ArbolAVL().ejecutar()


    def ejecutar(self):
        while True:
            try:
                line = input().strip()
                if line == "END":
                    break

                parts = line.split()
                command = parts
                key = int(parts) if len(parts) > 1 else None
```

```
                if command == "ADD":
                    self.insert(key)
                    print(self.in_order_traversal())
                elif command == "REMOVE":
                    if self.delete(key):
                        print("ELIMINADO")
                    else:
                        print("VEHICULO NO ENCONTRADO")
                    print(self.in_order_traversal())
                elif command == "SEARCH":
                    if self.search(key):
                        print("ENCONTRADO")
                    else:
                        print("VEHICULO NO ENCONTRADO")
            except EOFError:
                break
            except Exception as e:
                # print(f"Error: {e}") # For debugging
                pass # Ignore malformed input for auto-grading simplicity

if __name__ == "__main__":
    ArbolAVL().ejecutar()
```

Desarrollo: Implementación del Código Python (Estructura Conceptual)