Indexación Eficiente de Órdenes

El "Gestor de Rutas" en Route Manager necesita una capacidad ágil para registrar, localizar y eliminar órdenes específicas, generalmente identificadas por un order_id único. Un Árbol Binario de Búsqueda (BST) es una estructura de datos fundamental que mantiene los elementos en un orden clasificado, lo que permite operaciones eficientes de búsqueda, inserción y eliminación. Esta característica es vital para mantener un inventario de solicitudes de servicio actualizado y rápidamente accesible.

<table>
<tr><td>Historia de usuario Nro.</td><td>1</td><td>Título:</td><td>Gestionar Órdenes por ID</td></tr>
<tr><td rowspan="3">Descripción</td><td>COMO:</td><td colspan="2">Gestor de Rutas</td></tr>
<tr><td>QUIERO:</td><td colspan="2">Poder registrar nuevas órdenes, buscar órdenes existentes y eliminar órdenes canceladas o completadas por su ID único</td></tr>
<tr><td>PARA:</td><td colspan="2">Mantener un inventario actualizado y accesible de todas las solicitudes de servicio.</td></tr>
<tr><td>Criterios de aceptación</td><td colspan="3">● Las órdenes se identificarán por un ID numérico único.
● Se debe poder insertar una nueva orden (ID) en el sistema.
● Se debe poder buscar una orden por su ID y confirmar si existe.
● Se debe poder eliminar una orden por su ID.
● Si se intenta buscar o eliminar un ID no existente, se debe indicar "ID NO ENCONTRADO".
● Después de cada operación de inserción, eliminación o búsqueda, se debe realizar un recorrido in-order del árbol e imprimir los IDs para verificar el orden.</td></tr>
</table>

Historia de Usuario Nro. 1: Gestión de Órdenes

Los BSTs ofrecen una complejidad de tiempo promedio de O(log N) para operaciones de búsqueda, inserción y eliminación, lo cual es altamente eficiente para grandes volúmenes de datos. Sin embargo, existe una consideración importante: en el peor de los casos, como cuando los datos se insertan ya ordenados, un BST puede degenerar en una estructura similar a una lista enlazada, lo que resultaría en un rendimiento de O(N). Para Route Manager, donde los IDs de las órdenes podrían llegar de forma secuencial o semi-ordenada (por ejemplo, IDs basados en el tiempo), esta degeneración es un riesgo real. Este ejercicio, si bien demuestra la funcionalidad de un BST, subraya implícitamente la necesidad de estructuras que mitiguen esta vulnerabilidad de rendimiento, preparando el terreno para la introducción de árboles AVL en el siguiente ejercicio.

Programa para Indexación de Órdenes con BST

La estructura conceptual del programa en Java implicaría una clase BSTNode para representar cada nodo del árbol, conteniendo el orderId, y referencias a los nodos left y right. Una clase OrderManagerBST encapsularía la lógica del BST, con métodos para insert(int orderId), search(int orderId), delete(int orderId), y inOrderTraversal().

Entrada	Cada caso de prueba consistirá en una secuencia de comandos, uno por línea, finalizada por la palabra "END": ● INSERT <order_id>: Inserta una nueva orden con el ID especificado. ● SEARCH <order_id>: Busca una orden por su ID. ● DELETE <order_id>: Elimina una orden por su ID. ● END: Termina la entrada.
Salida	● Para INSERT y DELETE: 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 SEARCH: Se debe imprimir "ENCONTRADO" si el ID existe, o "ID NO ENCONTRADO" en caso contrario. ● Para DELETE: Se debe imprimir "ELIMINADO" si el ID fue eliminado con éxito, o "ID NO ENCONTRADO" si no se encontró.

Instrucciones para Calificación Automática:

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

Casos de prueba (visibles)

Entradas de ejemplo 1	INSERT 50 INSERT 30 INSERT 70 INSERT 20 INSERT 40 INSERT 60 INSERT 80 END
Salida de ejemplo 1	50 30 50 30 50 70 20 30 50 70 20 30 40 50 70 20 30 40 50 60 70 20 30 40 50 60 70 80

Entradas de ejemplo 2	INSERT 50 INSERT 30 INSERT 70 SEARCH 30 SEARCH 90

	SEARCH 50 END
Salida de ejemplo 2	50 30 50 30 50 70 ENCONTRADO ID NO ENCONTRADO ENCONTRADO

Entradas de ejemplo 3	INSERT 50 INSERT 30 INSERT 70 INSERT 20 INSERT 40 DELETE 20 ELIMINADO 20 30 40 50 70 20 30 40 50 70 DELETE 40 ELIMINADO 30 50 70 30 50 70 DELETE 70 ELIMINADO 30 50 30 50 END
Salida de ejemplo 3	50 30 50 30 50 70 20 30 50 70 20 30 40 50 70 ELIMINADO 30 40 50 70 ELIMINADO 30 50 70 ELIMINADO 30 50

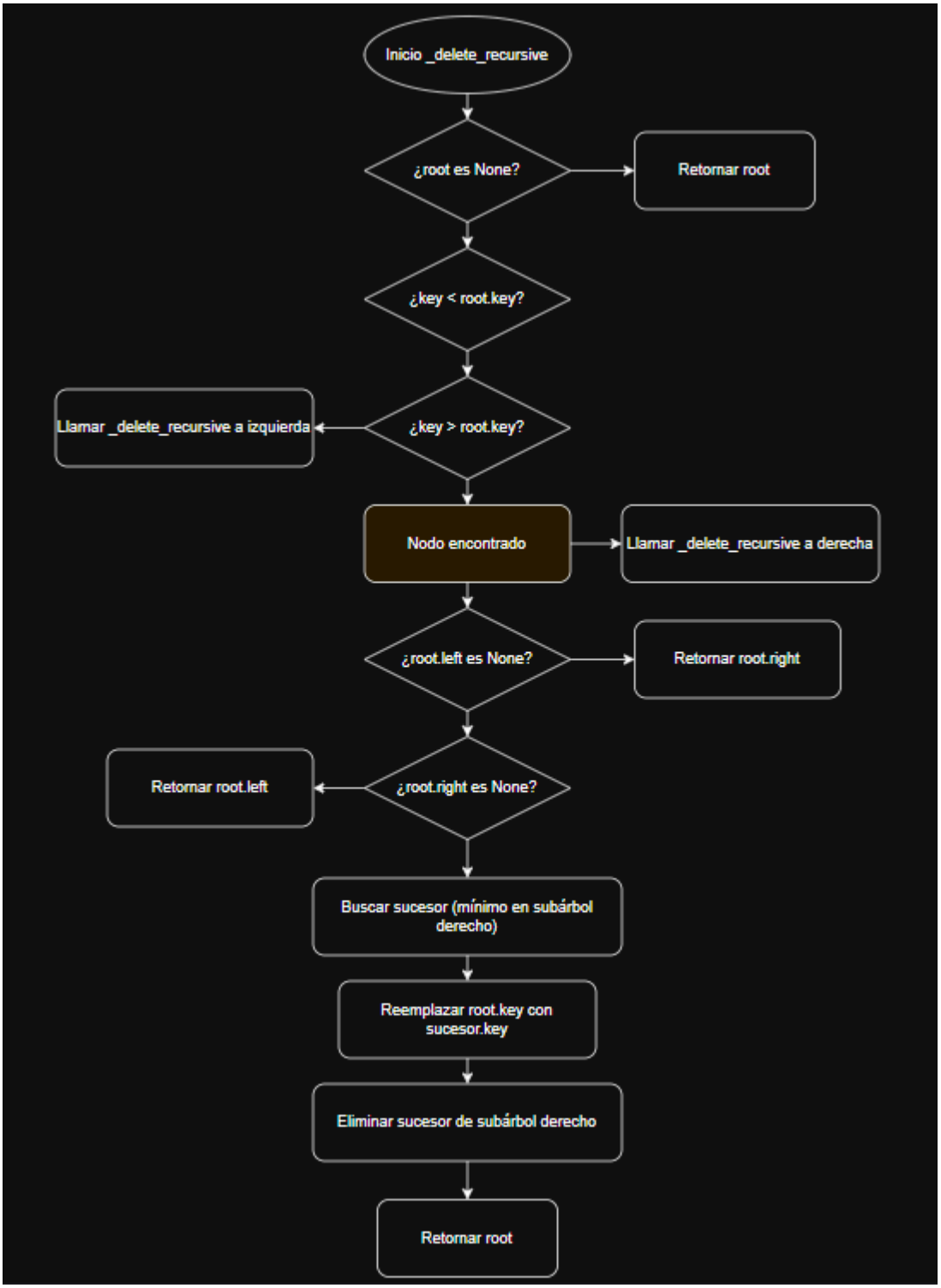
Entradas de ejemplo 4	INSERT 50 INSERT 30 INSERT 70 INSERT 60 INSERT 80 DELETE 70 ELIMINADO

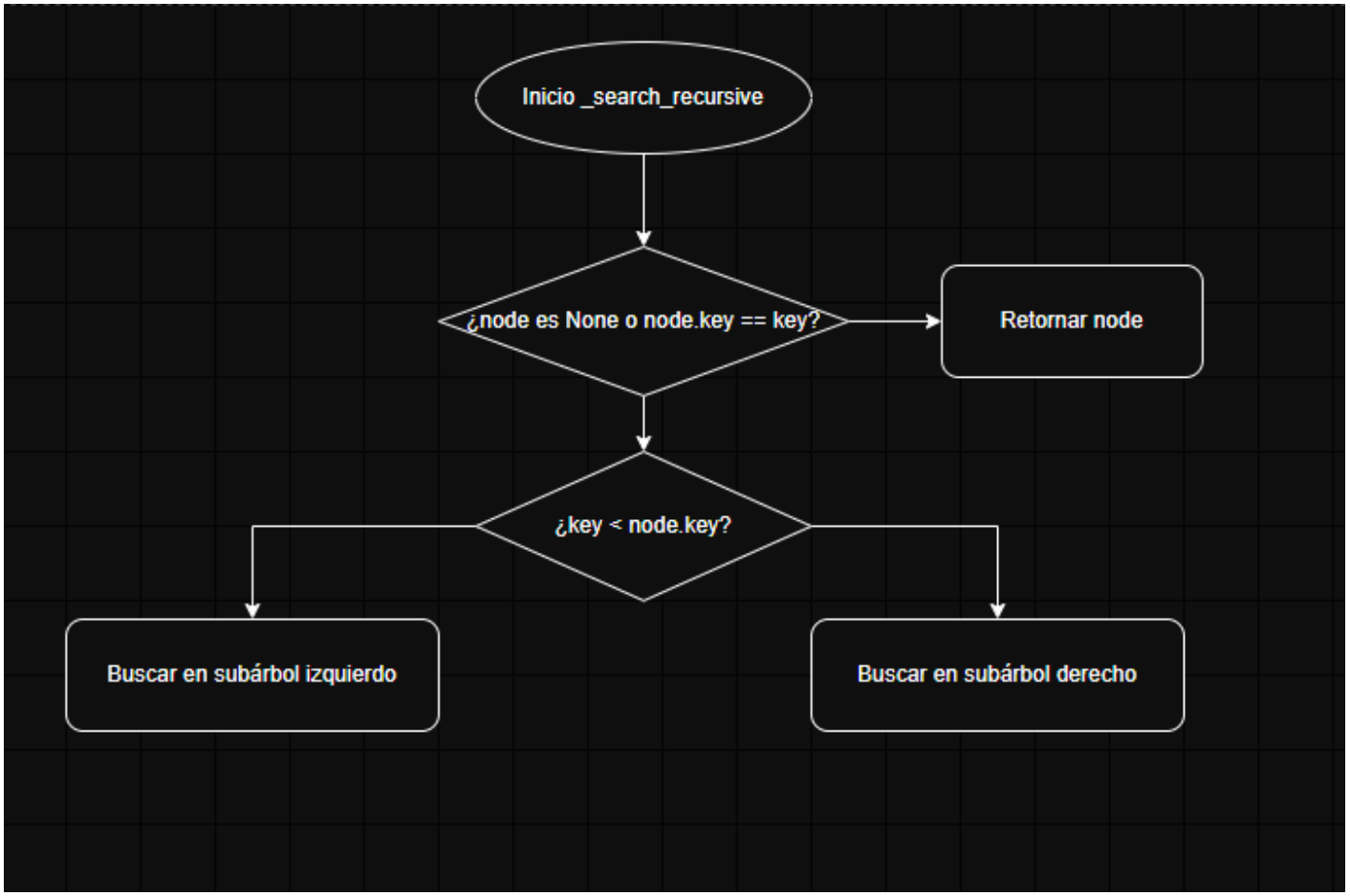
	30 50 60 80 30 50 60 80 END
Salida de ejemplo 4	50 30 50 30 50 70 30 50 60 70 30 50 60 70 80 ELIMINADO 30 50 60 80

Casos de prueba (ocultos)

Caso prueba	Entrada	Salida esperada
1	INSERT 50 DELETE 50 ELIMINADO INSERT 50 INSERT 30 DELETE 50 ELIMINADO 30 INSERT 50 INSERT 70 DELETE 50 ELIMINADO 70 INSERT 50 INSERT 30 INSERT 70 DELETE 50 ELIMINADO 30 70 END	50 ELIMINADO 50 30 50 ELIMINADO 30 50 50 70 ELIMINADO 70 50 30 50 30 50 70 ELIMINADO 30 70
2	INSERT 50 INSERT 30 DELETE 100 ID NO ENCONTRADO 30 50 END	50 30 50 ID NO ENCONTRADO 30 50

Diagramas de flujo: diseño





Desarrollo: Implementación del Código Python

Código Python :

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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

    def _insert_recursive(self, node, key):
        if node is None:
```

```
            return Node(key)
        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        else:
            node.right = self._insert_recursive(node.right, key)
        return node

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

    def _search_recursive(self, node, key):
        if node is None 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 _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 root is None:
            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:
            if root.left is None:
                temp = root.right
                root = None
                return temp
            elif root.right is None:
                temp = root.left
                root = None
                return temp

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

    def delete(self, key):
        initial_root = self.root
        self.root = self._delete_recursive(self.root, key)
        return initial_root is not None and self.root!= initial_root or (self.root is None and initial_root is not None) # Simple check if root changed or was deleted
```

```
    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)
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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

    def _insert_recursive(self, node, key):
        if node is None:
            return Node(key)
        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        else:
            node.right = self._insert_recursive(node.right, key)
        return node

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

```
    def _search_recursive(self, node, key):
        if node is None 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 _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 root is None:
            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:
            if root.left is None:
```

```
                temp = root.right
                root = None
                return temp
            elif root.right is None:
                temp = root.left
                root = None
                return temp

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

    def delete(self, key):
        initial_root = self.root
        self.root = self._delete_recursive(self.root, key)
        return initial_root is not None and self.root!= initial_root or (self.root is None and initial_root is not None) # Simple check if root changed or was deleted

    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 == "INSERT":
                    self.insert(key)
                    print(self.in_order_traversal())
                elif command == "SEARCH":
                    if self.search(key):
                        print("ENCONTRADO")
                    else:
                        print("ID NO ENCONTRADO")
                elif command == "DELETE":
                    if self.search(key): # Check existence before attempting
delete
                        self.delete(key)
                        print("ELIMINADO")
                        print(self.in_order_traversal())
```

```
                    else:
                        print("ID NO ENCONTRADO")
                        print(self.in_order_traversal()) # Still print traversal
as per requirements

            except EOFError:
                break
            except Exception as e:
                # Handle other potential errors like invalid key format
                pass

if __name__ == "__main__":
    ArbolBinarioBusqueda().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 == "INSERT":
                    self.insert(key)
```

```
                print(self.in_order_traversal())
            elif command == "SEARCH":
                if self.search(key):
                    print("ENCONTRADO")
                else:
                    print("ID NO ENCONTRADO")
            elif command == "DELETE":
                if self.search(key): # Check existence before attempting
delete
                    self.delete(key)
                    print("ELIMINADO")
                    print(self.in_order_traversal())
                else:
                    print("ID NO ENCONTRADO")
                    print(self.in_order_traversal()) # Still print traversal
as per requirements

        except EOFError:
            break
        except Exception as e:
            # Handle other potential errors like invalid key format
            pass

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