Gestión de Colas de Servicio Dinámicas

En el ecosistema de Route Manager SaaS, las solicitudes de servicio, que abarcan desde entregas hasta recolecciones, se generan y reciben de manera continua. Para gestionar eficazmente estas tareas pendientes antes de su asignación a rutas específicas, se requiere una colección ordenada y flexible. Los arreglos dinámicos son una solución idónea para este propósito, dado que su capacidad de crecer según sea necesario y su eficiencia en el acceso a elementos los hacen particularmente adecuados para manejar volúmenes de datos variables.

Historia de usuario Nro.	1	**Título:**	Gestionar Cola de Servicios Pendientes
Descripción	COMO:	Gestor de Rutas	
	QUIERO:	Agregar y procesar solicitudes de servicio de forma flexible, y ver la lista actual de pendientes	
	PARA:	Mantener un flujo de trabajo organizado y asegurar que ningún servicio se pierda o se retrase en su asignación	
Criterios de aceptación	● Los servicios deben poder agregarse al final de la cola (simulando nuevas solicitudes). ● Se debe poder obtener y "procesar" el servicio más antiguo (primero en entrar, primero en salir, aunque para un ArrayList es más eficiente desde el final). ● Se debe poder insertar un servicio en una posición específica si es de alta prioridad. ● Se debe poder eliminar un servicio de cualquier posición si es cancelado. ● Se debe poder consultar el número actual de servicios pendientes. ● Si se intenta procesar un servicio de una cola vacía, se debe indicar "COLA VACIA". ● Todos los servicios se identificarán por un ID único (ej. "ORDEN_123").		

Historia de Usuario Nro. 1: Gestionar Cola de Servicios Pendientes

La historia de usuario describe una necesidad que, si bien sugiere un comportamiento de cola (procesar el servicio más antiguo), también requiere la capacidad de insertar un servicio en una posición arbitraria (para alta prioridad) y eliminarlo desde cualquier punto. Esta combinación de requisitos introduce una consideración importante en el diseño. Una cola estricta (como una Queue en Java o collections.deque en Python) podría manejar eficientemente las operaciones de "primero en entrar, primero en salir". Sin embargo, la necesidad de inserciones y eliminaciones arbitrarias indica que una estructura como un arreglo dinámico (la list de Python o ArrayList de Java) es más adecuada, ya que ofrece la flexibilidad para ambas operaciones, aunque con diferentes características de rendimiento. Esta situación ilustra una tensión común en el diseño de software: equilibrar la disciplina de una cola con la flexibilidad operativa.

Programa para Cola de Servicios Dinámica

La estructura conceptual del programa en Java implicaría una clase ServiceQueueManager que encapsularía la lógica de gestión. Esta clase dispondría de métodos como addService(String serviceId) para añadir nuevas solicitudes, processNextService() para obtener y "procesar" la siguiente tarea, insertPriorityService(int index, String serviceId) para insertar un servicio en una posición específica, cancelService(String serviceId) para eliminar un servicio por su ID, y getPendingCount() para consultar el número de servicios pendientes. Internamente, esta clase utilizaría un java.util.ArrayList<String> para almacenar los IDs de los servicios.

Entrada	Cada caso de prueba consistirá en una secuencia de comandos, uno por línea, finalizada por la palabra "END": ● ADD <service_id>: Añade un servicio al final de la cola. ● PROCESS: Procesa (elimina) el siguiente servicio. ● INSERT <index> <service_id>: Inserta un servicio en un índice específico. ● CANCEL <service_id>: Elimina un servicio específico por su ID. ● COUNT: Imprime el número actual de servicios pendientes. ● END: Termina la entrada de comandos.
Salida	● Para el comando PROCESS: Se debe imprimir el ID del servicio procesado o "COLA VACIA" si la cola está vacía. ● Para el comando COUNT: Se debe imprimir el número entero de servicios pendientes. ● Los demás comandos (ADD, INSERT, CANCEL) no deben generar ninguna salida.

Instrucciones para Calificación Automática:

- Cada caso de prueba se especifica como una secuencia de comandos terminada en "END".
- El código debe procesar los comandos secuencialmente.
- La clase principal debe llamarse ArregloDinamico.
- Dentro de la clase ArregloDinamico, debe existir un método llamado ejecutar.
- No se deben utilizar mensajes al capturar las entradas.
- Únicamente se deben imprimir los resultados finales según las especificaciones de salida.

Casos de prueba (visibles)

Entradas de ejemplo 1	`ADD servicio1`
	`ADD servicio2`
	`COUNT`
	`PROCESS`

	PROCESS COUNT END
Salida de ejemplo 1	2 servicio2 servicio1 0

Entradas de ejemplo 2	ADD A ADD B ADD C INSERT 1 D COUNT CANCEL B COUNT PROCESS PROCESS END
Salida de ejemplo 2	4 3 C D

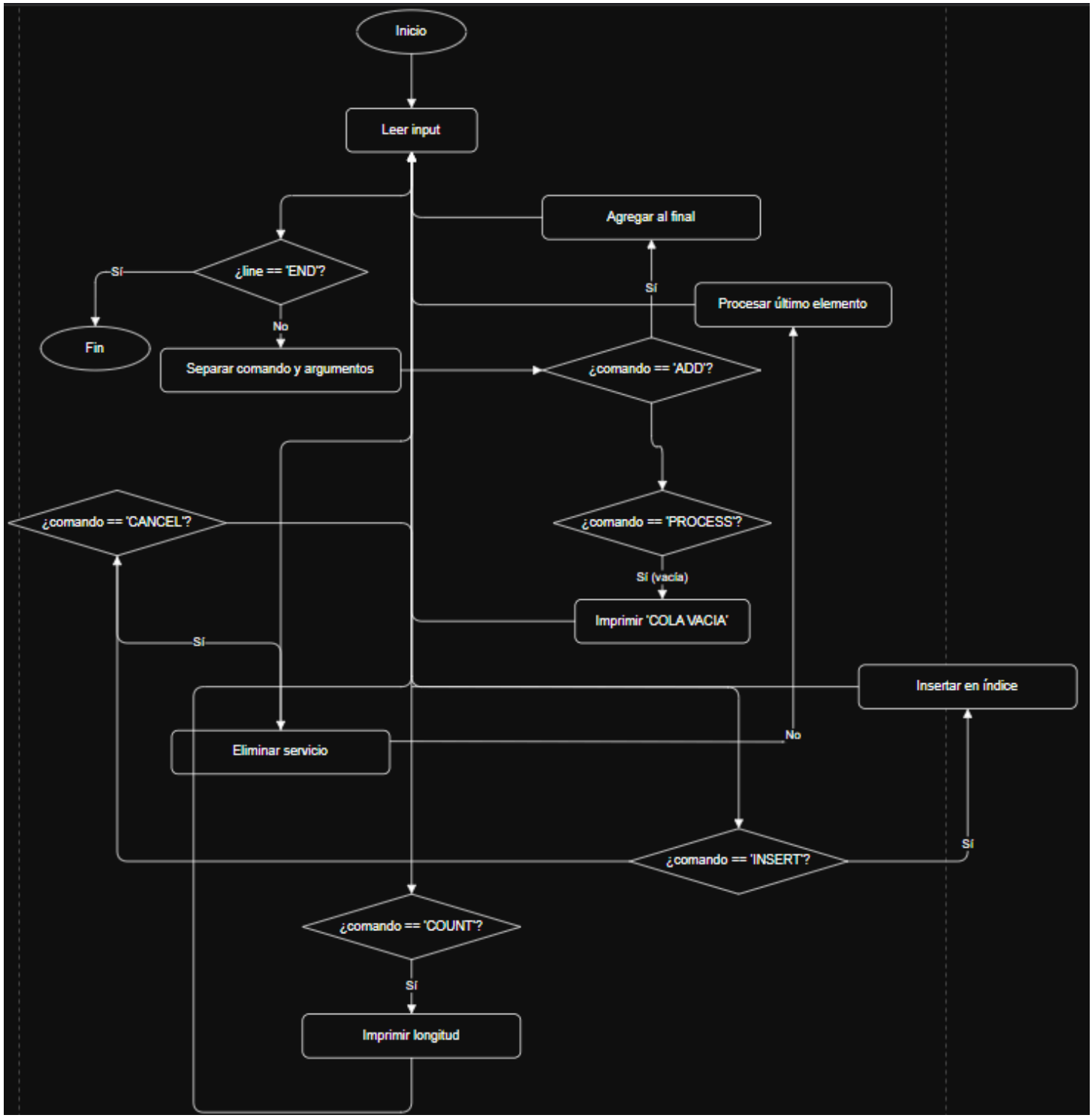
Entradas de ejemplo 3	PROCESS CANCEL inexistente ADD item PROCESS COUNT END
Salida de ejemplo 3	COLA VACIA item 0

Casos de prueba (ocultos)

Caso prueba	Entrada	Salida esperada
1	ADD uno ADD dos	4 tres dos

	INSERT 0 cero INSERT 10 tres COUNT PROCESS PROCESS PROCESS PROCESS COUNT END	uno cero 0

Diseño: Diagrama de flujo



Desarrollo: Implementación

La implementación en Python utilizará la clase list incorporada, que actúa como un arreglo dinámico.

Código Python :

```python
# Dynamycs arrays


class ArregloDinamico:

    def __init__(self):

        self.services = # La lista de Python actúa como un arreglo dinámico


    def ejecutar(self):

        while True:

            try:

                line = input().strip()

                if line == "END":

                    break


                parts = line.split(maxsplit=2)

                command = parts


                if command == "ADD":

                    service_id = parts

                    self.services.append(service_id)

                elif command == "PROCESS":

                    if self.services:

                        # Procesar desde el final para eficiencia O(1) amortizada

                        processed_id = self.services.pop()

                        print(processed_id)

                    else:

                        print("COLA VACIA")
```

```
                elif command == "INSERT":
                    index = int(parts)
                    service_id = parts
                    # La función insert de Python maneja índices fuera de rango de forma robusta
                    # inserta al final si el índice es mayor que la longitud, y al principio si es negativo.
                    self.services.insert(index, service_id)
                elif command == "CANCEL":
                    service_id = parts
                    try:
                        self.services.remove(service_id)
                    except ValueError:
                        # Servicio no encontrado, no se requiere salida según las instrucciones
                        pass
                elif command == "COUNT":
                    print(len(self.services))
            except EOFError:
                break

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