Despacho de Entregas Basado en Prioridad

En Route Manager, ciertas entregas o tareas pueden tener una urgencia superior, como los servicios de mensajería exprés o el transporte de suministros médicos críticos. Una cola de prioridad, implementada de manera eficiente mediante un montículo (heap), permite al "Gestor de Rutas" o al "Conductor" recuperar siempre la tarea más urgente en primer lugar.

Historia de usuario Nro.	1	**Título:**	Despachar Tareas por Prioridad
Descripción	COMO:	Gestor de Rutas o Conductor	
	QUIERO:	Poder agregar nuevas tareas de entrega con una prioridad asociada y siempre despachar la tarea más urgente primero	
	PARA:	Asegurar que los servicios críticos se atiendan a tiempo y optimizar la secuencia de entregas.	
Criterios de aceptación	● Cada tarea tendrá un ID (string) y un valor de prioridad (entero, menor valor = mayor prioridad). ● Se debe poder agregar una nueva tarea con su prioridad. ● Se debe poder extraer la tarea con la máxima prioridad (menor valor de prioridad). ● Si se intenta extraer de un montículo vacío, se debe indicar "MONTICULO VACIO". ● Después de cada operación de inserción o extracción, el montículo debe mantener su propiedad de orden (min-heap: padre menor que hijos). ● Se debe poder consultar el número actual de tareas pendientes.		

Historia de Usuario Nro. 1: Despacho de Tareas Priorizadas

Las operaciones logísticas son inherentemente dinámicas, lo que significa que las prioridades pueden cambiar. Por ejemplo, una entrega inicialmente "normal" podría volverse "urgente" debido a una queja del cliente. Aunque este ejercicio se enfoca en la adición y extracción de elementos de máxima prioridad, una implementación de montículo en un entorno real podría requerir una operación de "disminución de clave" (para aumentar la prioridad de un elemento existente), lo cual es más complejo pero vital para la re-priorización dinámica. Esto resalta la necesidad práctica de una gestión eficiente de prioridades que vaya más allá de las simples colas FIFO/LIFO.

Programa para Despacho de Prioridad con Montículo

La estructura conceptual del programa en Java implicaría una clase Task con atributos id y priority. Una clase PriorityDispatcherHeap contendría la lógica del montículo, con métodos como insertTask(Task task), extractMinTask(), y getTaskCount(). Internamente, esta clase utilizaría una implementación de min-heap basada en un arreglo.

Entrada	• ADD <task_id> <priority_value>: Añade una nueva tarea con su ID y valor de prioridad. • DISPATCH: Extrae y despacha la tarea de mayor prioridad. • COUNT: Imprime el número actual de tareas pendientes. • END: Termina la entrada.
Salida	• Para DISPATCH: Se debe imprimir "ID: • Para COUNT: Se debe imprimir el número entero de tareas. • Los demás comandos (ADD) no deben generar ninguna salida.

Instrucciones para Calificación Automática:

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

Casos de prueba (visibles)

Entradas de ejemplo 1	`ADD tarea1 5`
	`ADD tarea2 2`
	`ADD tarea3 8`
	`DISPATCH`
	`DISPATCH`
	`ADD tarea4 1`
	`DISPATCH`
	`DISPATCH`
	`END`

Salida de ejemplo 1	`ID: tarea2, Prioridad: 2`
	`ID: tarea1, Prioridad: 5`
	`ID: tarea4, Prioridad: 1`
	`ID: tarea3, Prioridad: 8`

Entradas de ejemplo 2	`DISPATCH`
	`ADD tareaA 10`
	`DISPATCH`
	`DISPATCH`
	`END`

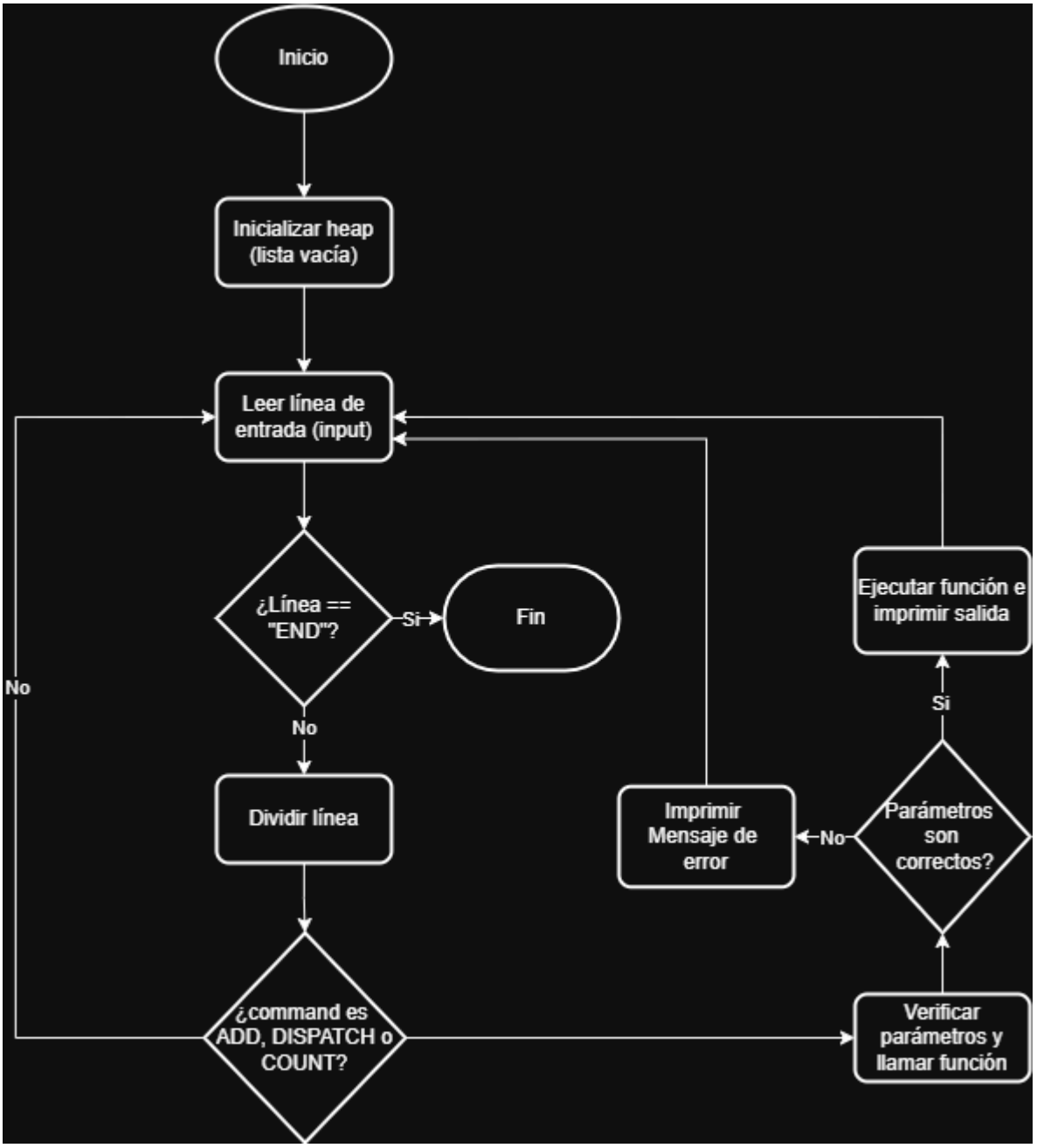
Salida de ejemplo 2	`MONTICULO VACIO`
	`ID: tareaA, Prioridad: 10`
	`MONTICULO VACIO`

Entradas de ejemplo 3	COUNT ADD t1 1 COUNT ADD t2 2 ADD t3 3 COUNT DISPATCH COUNT END
Salida de ejemplo 3	0 1 3 ID: t1, Prioridad: 1 2

Entradas de ejemplo 4	ADD tareaX 5 ADD tareaY 5 ADD tareaZ 5 DISPATCH DISPATCH DISPATCH END
Salida de ejemplo 4	ID: tareaX, Prioridad: 5 ID: tareaY, Prioridad: 5 ID: tareaZ, Prioridad: 5

Casos de prueba (ocultos)

Caso prueba	Entrada	Salida esperada
1	ADD A 5 ADD B 2 COUNT ADD C 8 DISPATCH COUNT ADD D 1 DISPATCH DISPATCH COUNT END	2 ID: B, Prioridad: 2 2 ID: D, Prioridad: 1 ID: A, Prioridad: 5 1



Desarrollo: Implementación del Código Python

Código Python :

```
import heapq


class Monticulos:

    def __init__(self):

        self.heap = [] # Usaremos la implementación de min-heap de Python (heapq)


    def insertTask(self, task_id, priority):
```

```
        # heapq es un min-heap, por lo que almacenamos (priority, task_id)
        # para que se ordene por prioridad (el menor valor de prioridad es el más alto)
        heapq.heappush(self.heap, (priority, task_id))

    def extractMinTask(self):
        if not self.heap:
            return None
        priority, task_id = heapq.heappop(self.heap)
        return (task_id, priority)

    def getTaskCount(self):
        return len(self.heap)

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

                parts = line.split(maxsplit=2)
                command = parts[0]

                if command == "ADD":
                    if len(parts) > 2:
                        try:
                            task_id = parts[1]
```

```
                            priority = int(parts[2])
                            self.insertTask(task_id, priority)
                        except ValueError:
                            pass # Ignorar líneas con formato incorrecto para ADD
                elif command == "DISPATCH":
                    result = self.extractMinTask()
                    if result:
                        task_id, priority = result
                        print(f"ID: {task_id}, Prioridad: {priority}")
                    else:
                        print("MONTICULO VACIO")
                elif command == "COUNT":
                    print(self.getTaskCount())
            except EOFError:
                break
            except Exception as e:
                # print(f"Error: {e}") # For debugging
                pass


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