Informe del Trabajo Práctico N°2

Aplicaciones de estructuras jerárquicas y grafos

Materia: Algoritmos y Estructuras de Datos - FIUNER

Integrantes:

Flores, Valentina

Aguilar Gonzales, Andrea Fernanda

Lell, Camila Luciana

Fecha de entrega: 6 de junio del 2025

ACTIVIDAD 1

Sala de Emergencias

En esta primera actividad nos enfrentamos al desafío de modelar un sistema de triaje para una sala de emergencias, en el que los pacientes debían ser atendidos en función de la gravedad de su estado, o como segundo criterio el orden de llegada. Para resolver esta situación, diseñamos una cola de prioridad utilizando internamente un montículo mínimo en la clase MonticuloMinimo, almacenando objetos de la clase Paciente.

Cada paciente almacena dos atributos claves: el nivel de riesgo (1 para crítico, 2 para moderado y 3 para bajo) y su timestamp de llegada, lo que nos permitió establecer un nivel de atención basada en dos criterios:

1. Prioridad principal: menor nivel de riesgo.

2. En caso de empate: atender al que llegó primero.

La simulación original, proporcionada por la cátedra, usaba una cola FIFO que atendía pacientes en orden de llegada. Modificamos el archivo main.py para reemplazar la cola FIFO por MonticuloMinimo. Cada ciclo de la simulación:

- Genera un paciente con riesgo aleatorio y lo inserta en el montículo usando insertar.
- Con 50% de probabilidad, atiende al paciente más crítico usando extraer_min.
- Muestra los pacientes en espera y el estado de la cola.

La clase MonticuloMinimo es genérica, tal como pedía el enunciado, por lo tanto acepta cualquier tipo de dato mediante un el término genérico (cmp_func). Para los pacientes, definimos la función comparar_pacientes en main.py, que compara primero por riesgo (get_riesgo) y luego por timestamp (get_timestamp). Esto separa la implementación genérica específica al triaje.

Estructura seleccionada: Cola de Prioridad basada en Montículo de Mínimos

Seleccionamos un montículo binario mínimo porque era la mejor manera de atender al paciente más crítico de manera eficaz. La estructura mantiene el elemento de mayor prioridad (menor riesgo) en la raíz, permitiendo:

- **Insertar (encolar)** Agregar un paciente al montículo y ajustar su posición hacia arriba (complejidad O(log n)), donde n es el número de pacientes, prioridad basada en:

1. Nivel de riesgo (1: crítico, 2: moderado, 3: bajo).

2. Timestamp de llegada (en caso de empate).

- **Extraer (desencolar)** siempre al paciente con mayor prioridad médica y ajustar el montículo hacia abajo (complejidad O(log n))

El montículo mínimo es eficiente en tiempo y espacio (O(n) para almacenamiento), como se indicó es genérico gracias a cmp_func, ya que separa la lógica de priorización (comparar_pacientes) de la estructura (MonticuloMinimo).

Alternativas como listas ordenadas o árboles binarios de búsqueda eran menos eficientes en tiempo de ejecución o más complejas de implementar sin beneficio adicional relevante.

Comprobación mediante simulación

Para validar la implementación de MonticuloMinimo, utilizamos la simulación dinámica proporcionada por la cátedra, colocada en main.py. En cada ciclo, se genera un paciente con un nivel de riesgo aleatorio (1: crítico, 2: moderado, 3: bajo) y se inserta en el montículo usando “insertar”. Con un 50% de probabilidad, se atiende al paciente más crítico mediante “extraer_min”, priorizando según la función “comparar_pacientes”, que evalúa el riesgo y el timestamp para desempates.

Durante las pruebas, confirmamos que los pacientes con riesgo 1 se atienden primero, seguidos por los de riesgo 2 y 3. En caso de empate en el riesgo, se respeta el orden de llegada (menor timestamp). También realizamos pruebas unitarias para verificar casos específicos, como el manejo de montículos vacíos y la correcta priorización. El rendimiento se mantuvo estable con un número creciente de pacientes, cumpliendo con las complejidades teóricas de O(log n) para inserciones y extracciones.

Observaciones generales

- La implementación de MonticuloMinimo resolvió el problema de triaje, evitando operaciones costosas como recorrer o reordenar listas (O(n)), mediante jerarquía.

- La clase es modular y genérica, ya que usa una función de comparación personalizable (cmp_func), lo que permite reutilizarla para otros tipos de datos más allá de pacientes.

- El diseño optimiza el uso de memoria, utilizando solo el montículo para almacenar pacientes (O(n)), sin necesidad de estructuras auxiliares innecesarias.

- El comportamiento observado en la simulación y las pruebas unitarias coincide con las complejidades teóricas esperadas (O(log n) para inserciones y eliminaciones).

Conclusiones

La clase MonticuloMinimo resuelve el problema de triaje de forma eficiente y genérica, cumpliendo con los requisitos del enunciado. La adaptación de la simulación y las complejidades O(log n) para inserciones y eliminaciones aseguran un sistema robusto y escalable.

Pseudocódigo para explicar la gestión de la simulación de triaje con MonticuloMinimo, incluyendo la priorización por riesgo y desempates por timestamp:

```
ALGORITMO Sistema de Triaje

    Crear montículo mínimo M con función comparar_pacientes(p1, p2):

        Si p1.riesgo ≠ p2.riesgo entonces

            Retornar p1.riesgo - p2.riesgo  // Menor riesgo = mayor prioridad

        Retornar p1.timestamp - p2.timestamp  // Desempate por llegada


    Para i desde 1 hasta 20 hacer:

        Generar paciente P con riesgo y timestamp aleatorios

        M.insertar(P)

        Si aleatorio() < 0.5 y M no está vacío entonces

            paciente_atendido = M.eliminar_mínimo()

            Mostrar("Atendiendo:", paciente_atendido)

        Mostrar("Pacientes en espera:", M.elementos)

        Esperar(1 segundo)

FIN
```