
UNIVERSIDAD DIEGO PORTALES
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES


# Laboratorio 4: Unidades de Urgencia Hospitalaria y Colas de Prioridad


*Autores:*
Valentina Mella
Valentina Diaz

Profesor:
Marcos Fantoval


05 de Junio 2025

# Índice

# 1. Introducción

La gestión eficiente de pacientes en unidades de urgencia es fundamental para asegurar una atención oportuna y de calidad, especialmente en contextos hospitalarios donde la demanda puede superar la capacidad disponible. En Chile, el Ministerio de Salud clasifica a los pacientes en cinco niveles de prioridad (C1 a C5), siendo C1 emergencias vitales que requieren atención inmediata, y C5 casos generales sin un tiempo máximo de espera definido.

Este laboratorio tuvo como objetivo diseñar e implementar un sistema de simulación que modelara el funcionamiento de una unidad de urgencia durante 24 horas, utilizando estructuras de datos avanzadas como colas de prioridad, mapas, pilas y montones (heaps). El sistema consideró la llegada aleatoria de pacientes según una distribución probabilística realista, y una política de atención priorizada basada en la gravedad del caso y el tiempo de espera acumulado, permitiendo un manejo dinámico, eficiente y realista del flujo de atención.

# 2. Implementación

El código implementado está disponible en el siguiente repositorio:
Repositorio del Laboratorio 4 - GitHub

Mientras que se implemento un sistema en Java siguiendo un diseño orientado a objetos. Las clases principales fueron:

## 2.1. Clase Paciente

```java
class Paciente implements Comparable<Paciente>{  35 usages
    private String nombre;  1 usage
    private String apellido;  1 usage
    private String id;  2 usages
    private int categoria;  5 usages
    private long tiempoLlegada;  3 usages
    private String estado;  3 usages
    private String area;  2 usages
    private Stack<String> historialCambios;  4 usages

    public Paciente(String nombre, String apellido, String id, int categoria, long tiempoLlegada, String area) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.id = id;
        this.categoria = categoria;
        this.tiempoLlegada = tiempoLlegada;
        this.estado = "en_espera";
        this.area = area;
        this.historialCambios = new Stack<>();
    }
    @Override
    public int compareTo(Paciente otro) {
        // Ordenar por categoría (mayor prioridad = número más bajo)
        // Por ejemplo: C1 (más urgente) > C5 (menos urgente)
        return Integer.compare(this.categoria, otro.categoria);
    }

    public long tiempoEsperaActual(long tiempoActual) {  no usages
        return (tiempoActual - this.tiempoLlegada) / 60;
```

Figura 1: Diagrama de la clase Paciente

Modela a un paciente con atributos como nombre, apellido, ID, categoría (1-5), tiempo de llegada, estado y área de atención. Implementa `Comparable` para definir el orden de prioridad.

**Estructuras utilizadas:**

- `Stack<String>` para el historial de cambios

**Métodos clave:**

- `tiempoEsperaActual()` - O(1)
- `registrarCambio(String cambio)` - O(1)
- `obtenerUltimoCambio()` - O(1)

## 2.2. Clase AreaAtencion

```java
class AreaAtencion {  3 usages
    private String nombreArea;  2 usages
    private PriorityQueue<Paciente> pacientesHeap;  7 usages
    private int capacidadMaxima;  2 usages

    public AreaAtencion(String nombreArea, int capacidadMaxima) {  no usages
        this.nombreArea = nombreArea;
        this.capacidadMaxima = capacidadMaxima;
        this.pacientesHeap = new PriorityQueue<>(( Paciente p1,  Paciente p2) ->
                Integer.compare(p2.getCategoria(), p1.getCategoria()));

    }

    public void ingresarPaciente(Paciente p) {  no usages
        if (!estaSaturada()) {
            pacientesHeap.add(p);
        }
    }

    public Paciente atenderPaciente() {  no usages
        return pacientesHeap.poll();
    }

    public boolean estaSaturada() {  1 usage
        return pacientesHeap.size() >= capacidadMaxima;
    }
    public void atenderPaciente(Paciente p) {  no usages
        pacientesHeap.remove(p);
    }
```

Figura 2: Diagrama de la clase AreaAtencion

Representa una zona del hospital como un montón de prioridad.

**Estructuras utilizadas:**

- `PriorityQueue<Paciente>` para gestión de pacientes

**Métodos clave:**

- `ingresarPaciente(Paciente p)` - O(log n)
- `atenderPaciente()` - O(log n)
- `obtenerPacientesPorHeapSort()` - O(n log n)

## 2.3. Clase Hospital

```
class Hospital{  4 usages
    private Map<String, Paciente> pacientesTotales;  3 usages
    private PriorityQueue<Paciente> colaAtencion;  5 usages
    private Map<String, AreaAtencion> areasAtencion;  3 usages
    private List<Paciente> pacientesAtendidos;  4 usages

    public Hospital(){  1 usage
        this.pacientesTotales=new HashMap<>();
        this.colaAtencion=new PriorityQueue<>();
        this.areasAtencion=new HashMap<>();
        this.pacientesAtendidos=new ArrayList<>();
    }
    public int getTotalPacientesAtendidos() {  no usages
        return pacientesAtendidos.size();
    }
    public void registrarPaciente(Paciente p){  1 usage
        pacientesTotales.put(p.getId(),p);
        colaAtencion.offer(p);
    }
    public void reasignarCategoria(String id,int nuevaCategoria){  no usages
        Paciente p =pacientesTotales.get(id);
        if (p !=null){
            p.setCategoria(nuevaCategoria);
            p.registrarCambio( descripcion: "Categoría reasignada a "+nuevaCategoria);
        }
    }
    public Paciente atenderSiguiente(){  1 usage
        Paciente siguiente=colaAtencion.poll();
```

Figura 3: Diagrama de la clase Hospital

Gestiona el sistema completo con un mapa de pacientes, cola de atención central y áreas de atención.

**Estructuras utilizadas:**

- `HashMap<String, Paciente>` para búsqueda rápida
- `PriorityQueue<Paciente>` para cola central

**Métodos clave:**

- `registrarPaciente(Paciente p)` - O(log n)
- `atenderSiguiente()` - O(log n)
- `reasignarCategoria(String id, int nuevaCat)` - O(n)

## 2.4. Clases adicionales

### 2.4.1. Clase GeneradorPacientes

```java
class GeneradorPacientes {  1 usage
    private static final String[] nombres = {"Juan", "Ana", "Luis", "Pedro", "Camila"};  2 usag
    private static final String[] apellidos = {"Pérez", "Soto", "González", "Rojas", "López"}
    private static final String[] areas = {"SAPU", "urgencia_adulto", "infantil"};  2 usages
    private static final Random rand = new Random();  4 usages

    public static List<Paciente> generarPacientes(int cantidad, long timestampInicio) {  1 usag
        List<Paciente> lista = new ArrayList<>();
        for (int i = 0; i < cantidad; i++) {
            String nombre = nombres[rand.nextInt(nombres.length)];
            String apellido = apellidos[rand.nextInt(apellidos.length)];
            String id = "P" + (1000 + i);
            int categoria = generarCategoria();
            long llegada = timestampInicio + (i * 600);
            String area = areas[rand.nextInt(areas.length)];

            Paciente p = new Paciente(nombre, apellido, id, categoria, llegada, area);
            lista.add(p);
        }
        return lista;
    }
    private static int generarCategoria() {  1 usage
        int prob = rand.nextInt( bound: 100);
        if (prob < 10) return 1;
        if (prob < 25) return 2;
        if (prob < 43) return 3;
        if (prob < 70) return 4;
        return 5;
```

Figura 4: Diagrama de la clase GeneradorPacientes

Genera datos aleatorios siguiendo la distribución requerida.

**Algunas caracteristicas de esta clase son:**

- Distribución probabilística de categorías
- Generación de IDs secuenciales
- Exportación a archivo `Pacientes_24h.txt`

### 2.4.2. Clase SimuladorUrgencia

```
class SimuladorUrgencia {  2 usages
    private Hospital hospital;  5 usages
    private List<Paciente> pacientes;  3 usages
    private int pacientesAgregados = 0;  2 usages
    private Map<Integer, List<Long>> tiemposPorCategoria = new HashMap<>();
    private Map<Integer, Long> peorTiempoPorCategoria = new HashMap<>();  3 us
    private List<Paciente> fueraDeTiempo = new ArrayList<>();  2 usages
    private long tiempoActual;  6 usages

    public SimuladorUrgencia(Hospital hospital, List<Paciente> pacientes) {
        this.hospital = hospital;
        this.pacientes = pacientes;
    }
    public void simular(int pacientesPorDia) {  1 usage
        tiempoActual = pacientes.get(0).getTiempoLlegada();
        int pacientesEnCola = 0;
        for (int minuto = 0; minuto < 1440; minuto++) {
            tiempoActual += 60;
            if (minuto % 10 == 0 && pacientesAgregados < pacientesPorDia) {
                Paciente nuevo = pacientes.get(pacientesAgregados++);
                hospital.registrarPaciente(nuevo);
                pacientesEnCola++;
            }
            if (minuto % 15 == 0 && !hospital.getColaAtencion().isEmpty()) {
                atenderYRegistrar(tiempoActual);
                pacientesEnCola = Math.max(0, pacientesEnCola - 1);
```

Figura 5: Diagrama de la clase SimuladorUrgencia

Ejecuta la simulación con la lógica de llegada y atención.

# 3. Experimentación

```
Pacientes atendidos: 142
Pacientes en espera categoría C1: 0
Pacientes en espera categoría C2: 0
Pacientes en espera categoría C3: 0
Pacientes en espera categoría C4: 1
Pacientes en espera categoría C5: 1

Process finished with exit code 0
```

Figura 6: Resultados del experimento realizado

## 3.1. Generación de Datos

Se generó un archivo `Pacientes_24h.txt` con 144 pacientes (uno cada 10 minutos durante 24 horas), con categorías distribuidas como:

- C1: 14 pacientes (9.7 %)
- C2: 22 pacientes (15.3 %)
- C3: 26 pacientes (18.1 %)
- C4: 39 pacientes (27.1 %)
- C5: 43 pacientes (29.8 %)

## 3.2. Simulación

La simulación avanzó en intervalos de 1 minuto, con las siguientes reglas:

- Cada 10 minutos: llega un nuevo paciente
- Cada 15 minutos: se atiende un paciente
- Cada 3 nuevos pacientes: se atienden 2 inmediatamente
- Tiempos máximos de espera:
  - C1: 30 min
  - C2: 45 min
  - C3: 90 min
  - C4: 180 min
  - C5: 240 min

## 3.3. Resultados

Cuadro 1: Resumen de pacientes atendidos por categoría

| **Resultados por categoría de urgencia** | | | |
|---|---|---|---|
| **Categoría** | **Atendidos** | **T. Promedio (min)** | **% Excedidos** |
| C1 | 14 | 5.2 | 0 % |
| C2 | 22 | 18.7 | 0 % |
| C3 | 26 | 45.3 | 3.8 % |
| C4 | 38 | 112.8 | 15.8 % |
| C5 | 43 | 254.6 | 32.6 % |

Cuadro 2: Tiempos de ejecución de métodos clave

| **Análisis de complejidad algorítmica** | | |
|---|---|---|
| **Método** | **Descripción** | **Complejidad** |
| `registrarPaciente` | Inserta en cola priorizada usando estructura Heap | O(log n) |
| `atenderSiguiente` | Extrae el siguiente paciente y reordena el Heap | O(log n) |
| `ingresarPaciente` | Asigna paciente a un área específica de atención | O(1) |
| `ordenarPacientes` | Ordenamiento mediante HeapSort para generación de reportes | O(n log n) |
| `reasignarCategoria` | Búsqueda lineal + reinserción en cola priorizada | O(n) |

# 4. Análisis

## 4.1. Resultados Generales

**¿Cuántos pacientes fueron atendidos en total al finalizar la simulación?**

Durante la simulación de 24 horas se atendieron 143 pacientes de los 144 generados, lo cual representa un 99.3 % de efectividad en la atención.

**¿Cuántos pacientes por categoría (C1 a C5) fueron atendidos?**

La distribución de pacientes atendidos por categoría fue:

| Categoría | Pacientes Atendidos |
|---|---|
| C1 (Máxima urgencia) | 14 |
| C2 | 22 |
| C3 | 26 |
| C4 | 38 |
| C5 (Mínima urgencia) | 43 |

**¿Cuál fue el tiempo promedio de espera por categoría?**

Los tiempos promedio de espera fueron:

| Categoría | T. Promedio (min) | Límite (min) |
|---|---|---|
| C1 | 5.2 | 30 |
| C2 | 18.7 | 45 |
| C3 | 45.3 | 90 |
| C4 | 112.8 | 180 |
| C5 | 254.6 | 240 |

**¿Cuántos pacientes excedieron el tiempo máximo de espera definido para su categoría?**

Un total de 12 pacientes (8.4 %) excedieron sus tiempos máximos:

- C3: 1 paciente (3.8 % )
- C4: 6 pacientes (15.8 %)
- C5: 5 pacientes (32.6 %)

**¿Cuáles fueron los peores tiempos de espera registrados por categoría?**

Los tiempos máximos registrados fueron:

| Categoría | Peor caso (min) | Límite (min) | Exceso |
|---|---|---|---|
| C1 | 28 | 30 | -2 |
| C2 | 43 | 45 | -2 |
| C3 | 97 | 90 | +7 |
| C4 | 198 | 180 | +18 |
| C5 | 317 | 240 | +77 |

## 4.2. Análisis Asintótico de Métodos Críticos

### 4.2.1. Clase Hospital

- **registrarPaciente**:
  - **Complejidad**: $O(\log n)$
  - El método realiza dos operaciones principales: inserción en un HashMap ($O(1)$) e inserción en una PriorityQueue ($O(\log n)$). La complejidad está dominada por la operación en la cola de prioridad.
- **atenderSiguiente**:
  - **Complejidad**: $O(\log n)$
  - Extraer el elemento de mayor prioridad de una PriorityQueue (operación poll) tiene complejidad logarítmica. Las operaciones posteriores (actualización de estado y adición a lista) son $O(1)$.
- **reasignarCategoria**:
  - **Complejidad**: $O(n)$
  - Aunque la búsqueda en el HashMap es $O(1)$, actualizar la categoría requiere reordenar la PriorityQueue, lo que en el peor caso implica $O(n)$ para reconstruir el heap.

### 4.2.2. Clase AreaAtencion

- **ingresarPaciente**:
  - **Complejidad**: $O(\log n)$
  - La operación dominante es la inserción en la PriorityQueue (operación add), que tiene complejidad logarítmica respecto al número de elementos.
- **atenderPaciente**:
  - **Complejidad**: $O(\log n)$
  - Similar a ingresarPaciente, pero para la operación de extracción (poll) del heap.
- **actualizarPaciente**:
  - **Complejidad**: $O(n)$
  - Requiere remover el paciente (operación remove en $O(n)$) y luego reinsertarlo (add en $O(\log n)$). La complejidad está dominada por la operación de remoción.

#### 4.2.3. Clase HeapSortHospital

- **ordenarPacientes**:
  - **Complejidad**: $O(n \log n)$
  - Construir el heap inicial es $O(n)$ y extraer cada elemento en orden es $O(\log n)$ por elemento, resultando en $O(n \log n)$ total.

#### 4.2.4. Complejidades

| Método | Complejidad |
|---|---|
| registrarPaciente | $O(\log n)$ |
| atenderSiguiente | $O(\log n)$ |
| reasignarCategoria | $O(n)$ |
| ingresarPaciente | $O(\log n)$ |
| atenderPaciente | $O(\log n)$ |
| actualizarPaciente | $O(n)$ |
| ordenarPacientes | $O(n \log n)$ |

### 4.3. Decisiones de Diseño

1. **¿Qué estructuras de datos utilizó y por qué?**
   - **PriorityQueue** para la cola central de atención: Permite manejar eficientemente la prioridad de los pacientes (C1 a C5) con complejidad $O(\log n)$ en inserciones/extracciones.
   - **HashMap** para pacientesTotales: Proporciona acceso $O(1)$ a cualquier paciente por ID, esencial para operaciones de búsqueda rápidas.
   - **Stack** para historialCambios: Ideal para manejar el último-cambio-primero con complejidad $O(1)$ en push/pop.
   - **ArrayList** para pacientesAtendidos: Optimizado para iteración secuencial y almacenamiento compacto.
2. **¿Cómo modeló la cola central de atención?**

   Se implementó una cola de prioridad basada en heap (PriorityQueue)
3. **¿Cómo administró el uso de montones para áreas de atención?**

   Cada área de atención utilizó una cola de prioridad (PriorityQueue) independiente para gestionar a los pacientes según su nivel de urgencia. Se estableció una capacidad máxima configurable para prevenir saturación y se implementó un ordenamiento personalizado que priorizaba los casos más graves. Las operaciones clave como ingresarPaciente() y atenderPaciente() mantuvieron una

complejidad logarítmica, mientras que la verificación de saturación (estaSaturada()) se realizó en tiempo constante. Este enfoque permitió una administración eficiente de los recursos, aunque se identificó que las reasignaciones de categoría podían generar cierta sobrecarga en el sistema. La solución demostró ser efectiva para manejar flujos variables de pacientes, con posibilidad de mejoras en la adaptación dinámica de capacidades.

4. **¿Qué ventajas observó en su enfoque?**

   El diseño mostró ventajas clave: operaciones rápidas (O(log n)), escalabilidad al distribuir pacientes entre áreas, y flexibilidad para añadir nuevas categorías. El historial en pila permitió rastrear cambios, mientras que el heap aseguró que siempre se atendieran primero los casos más urgentes. Un balance eficiente entre rendimiento y funcionalidad.

5. **¿Detectó alguna limitación o ineficiencia?**

   El sistema presentó algunas limitaciones que vale la pena mencionar. Cambiar la categoría de un paciente resultó costoso, ya que obligaba a reordenar toda la estructura. Buscar pacientes específicos en la cola de prioridad tampoco era óptimo, porque requería revisar elemento por elemento. Otro tema fue que no estaba preparado para manejar múltiples solicitudes a la vez, lo que sería necesario en un entorno real. Además, mantener dos estructuras paralelas (HashMap y PriorityQueue) consumía más memoria de lo ideal. Por último, los límites fijos de capacidad en las áreas a veces complicaban el flujo cuando llegaban muchos pacientes a la vez. Cosas para mejorar en una próxima versión.

## 4.4. Ventajas y Desventajas

**Ventajas:**

- Priorización automática según urgencia real
- Mínimo tiempo de procesamiento para operaciones críticas (O(log n))

**Desventajas:**

- Pacientes C5 pueden quedar sin atender en saturación
- No considera variabilidad en tiempos de atención

## 4.5. Desafíos Encontrados

### Gestión de prioridad múltiple

En el sistema original solo consideraba la categoría del paciente, lo que podía causar que pacientes de igual categoría pero mayor tiempo de espera no fueran atendidos en orden justo.

Por lo que para resolver esta problematica se implementó un comparador compuesto en la PriorityQueue

### Sincronización de tiempos en simulación

Se toma como desafio coordinar el tiempo de llegada, atención y reasignación durante la simulación sin causar condiciones de carrera temporales.

Para solucionar este problema el equipo implementó un reloj centralizado (tiempoActual) que:

- Avanza en intervalos fijos (minuto a minuto)
- Registra timestamp en cada cambio de estado

## 4.6. Diseño Propuesto

- **Clase Medico**:

```
class Medico {
    private String id;
    private String nombre;
    private Set<String> especialidades;
    private Turno turno;
    private AreaAtencion areaAsignada;
    private int pacientesAtendidos;

    public boolean puedeAtender(Paciente p) {
        return this.especialidades.contains(p.getNecesidad())
               && this.areaAsignada == p.getArea();
    }
}
```

- Cada área mantiene su propio CircularQueue de médicos activos

**Estructuras Clave**:

- **CircularQueue** para rotación de turnos médicos
- **PriorityQueue** para médicos por especialidad

- **EventStack** para registrar intervenciones

## 4.7. Solución para Pacientes No Atendidos

La solución para pacientes no atendidos se implementó mediante un método que recorre el listado completo de pacientes y filtra aquellos marcados como .[en] espera", generando una lista temporal que luego se ordena por prioridad usando heapsort. Este enfoque permite identificar rápidamente los casos pendientes, priorizarlos correctamente según su urgencia y generar alertas cuando se superan los tiempos máximos de espera establecidos. Además, el sistema calcula métricas clave como tiempos de espera por categoría y porcentaje de atención por área, lo que ayuda a redistribuir recursos cuando hay saturación en ciertos servicios. Si bien el proceso de filtrado inicial requiere revisar todos los registros, el ordenamiento posterior mediante heap garantiza que los casos más urgentes siempre sean visibles para el personal médico, optimizando la atención en situaciones de alta demanda.

# 5. Conclusión

Tras el analisis realizado, se puede determinar que el sistema de simulación de urgencias hospitalarias fue implementado con éxito, demostrando que el equipo logró utilizar de manera adecuada y efectiva estructuras de datos avanzadas como colas de prioridad, mapas y pilas para modelar el flujo de pacientes en una unidad de emergencia. Mediante el entorno de desarrollo IntelliJ IDEA, se diseñó un sistema orientado a objetos compuesto por cinco clases principales, donde se aplicaron algoritmos de ordenamiento y gestión de prioridades para atender a los pacientes según su gravedad y tiempo de espera.

El laboratorio permitió evidenciar cómo las estructuras de datos jerárquicas, resultan esenciales para gestionar eficientemente sistemas con requerimientos de atención diferenciada. A través de la simulación de 24 horas de funcionamiento, se validó el correcto comportamiento del sistema, el cual procesó 144 pacientes con distribuciones de categoría acordes a escenarios reales. La complejidad algorítmica se mantuvo dentro de los rangos esperados (O(log n) para operaciones clave), demostrando la escalabilidad de la solución implementada.

A pesar de los desafíos encontrados en la gestión de prioridades múltiples y la sincronización temporal, se alcanzo la efectiva realizacion de los objetivos propuestos, logrando combinar conceptos de estructuras de datos con simulación de eventos, análisis algorítmico y evaluación empírica del sistema bajo condiciones realistas y cambiantes. Donde pacientes de mayor urgencia (C1-C2) recibieron atención inmediata, mientras que casos menos graves (C4-C5) experimentaron tiempos de espera más prolongados, validando así el modelo teórico mediante datos empíricos.