



udp UNIVERSIDAD
DIEGO PORTALES

UNIVERSIDAD DIEGO PORTALES
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES

ESTRUCTURAS DE DATOS & ANÁLISIS DE ALGORITMOS

Unidades de Urgencias Hospitalaria y Colas de Prioridad

Autores:

Carlos Araya

Tomas Barriga

Jasson Valverde

Maximo Hernández

Profesor:

Marcos Fantoal

9 de junio 2025

Índice

1. Introducción	2
2. Marco Teórico	2
3. Implementación	4
3.1. Class Paciente	4
3.2. Class AreaAtencion	8
3.3. Class Hospital	10
4. Experimentación	14
4.1. Class GeneradorPacientes	14
4.1.1. Método prob	15
4.1.2. Método guardar	15
4.2. Simulación	16
4.3. Método Main	17
4.4. Pruebas	18
4.5. Extensión	20
5. Análisis	22
5.1. Análisis de Resultados Generales	22
5.2. Análisis asintótico de métodos críticos	23
5.3. Decisiones de diseño	23
5.4. Ventajas y desventajas del sistema	24
5.4.1. Ventajas	24
5.4.2. Desventajas	24
5.5. Desafíos encontrados	24
5.6. Solución del sistema por pacientes no atendidos	25
6. Conclusión	26
7. Referencia	26

1. Introducción

El objetivo de este laboratorio es diseñar e implementar un sistema de simulación de atención en unidades de urgencias hospitalarias, aplicando estructuras de datos avanzadas. Este sistema debía gestionar pacientes de acuerdo con la clasificación de urgencias (C1 a C5 que veremos más adelante), simulando su llegada y atención en un periodo de 24 horas, donde se deben manejar reasignaciones de categoría y áreas de atención. Las herramientas utilizadas incluyeron colas de prioridad, mapas, pilas y montones; la metodología se basa en la creación de clases específicas que representarán tanto entidades del sistema como su comportamiento dinámico.

A lo largo del laboratorio se verá cómo se implementaron 5 clases, las cuales son: paciente, área de atención, hospital, generador de pacientes y simulador de urgencias. Sus respectivas notaciones BIG O de los métodos; de igual manera, se hará una prueba en la clase main de las respectivas clases y con sus métodos.

2. Marco Teórico

Se emplean conceptos fundamentales en estructura de datos, clases en programación orientada a objetos y notación de complejidad temporal. Los cuales, entender estos contenidos es esencial para poder entender cómo se desarrolla este informe y la solución aplicada al laboratorio.

- Clases y objetos: Java es un lenguaje de programación orientado a objetos que puede permitir modelar entidades del mundo real mediante clases; en este laboratorio se utiliza para modelar entidades del sistema hospitalario, como pacientes y áreas de atención, con sus respectivos atributos y métodos que definirán su comportamiento, mientras que los objetos son las instancias concretas que se utilizan para la simulación, para representar situaciones reales para la atención médica.
- Complejidad temporal: La complejidad temporal se expresa con la notación BIG O, la cual mide el tiempo que un algoritmo podría tardar en ejecutarse dependiendo del tamaño de entrada. Esta notación se enfoca especialmente en el peor caso; es clave evaluar para determinar la eficiencia de la solución.
 - El código que veremos incluye distintos métodos que tienen diferentes costos computacionales.
 - Es importante analizar qué tan eficiente es cada uno de los casos según qué estructura de datos o algoritmos se usa. Las notaciones más usuales son $O(1)$, $O(\log n)$, $O(n)$, etc.
- Cola de prioridad / Priority queue: Es una estructura de datos que permite extraer siempre el elemento con mayor prioridad dependiendo del comparador que se utilice; el orden en que estos ingresan no tiene importancia. En este

sistema, se usa para poder organizar a los pacientes tanto en la sala de espera como en áreas de atención, en este caso priorizando a aquellos de menor categoría (más urgente) y el mayor tiempo de espera. El cual se basa en un heap binario, lo que permite operaciones de inserción y extracción en tiempo $O(\log n)$.

- Mapa / MAP: Es una estructura de datos que asocia un ID o clave con valores, permitiendo búsquedas rápidas donde evita que ID o claves se repitan, haciendo que sean ideales para poder acceder a elementos sin recorrer toda la estructura. En este caso se utiliza un HashMap.
- Pila / Stack: Es una estructura de datos tipo LIFO (el último que entra es el primero que sale) que se utiliza en la clase **Paciente** para mantener el historial de los cambios que se han experimentado; esto también incluye modificaciones en la categoría o la situación del paciente. Esta estructura permite consultar y deshacer el último cambio fácilmente.
- Heap y HeapSort: El heap es una estructura de datos que se organiza los elementos en un árbol binario siguiendo con la propiedad de orden entre padres e hijos, es la base del priority queue. En este sistema se usa para implementar un método de diagnóstico que ordena los pacientes de una área según su prioridad. Mientras que el algoritmo HeapSort, es un método de ordenamiento que utiliza el Heap para poder ordenar de manera eficiente.
- Lista / List: Es una estructura de datos que permite almacenar y manejar un conjunto de elementos de manera ordenada. Se utiliza para cuando se necesita trabajar con una secuencia de elementos que pueden repetirse valores, donde comúnmente se ocupa cuando se implementa en un ArrayList o LinkedList (Más adelante, en la sección de Implementación se verá como se hace uso). En este caso es para la clase **Hospital** en el método `List<Paciente>obtenerPacientesPorCategoria(int c)`.

3. Implementación

Se implementan clases para el buen uso de datos, ya que son la base de la programación orientada a objetos, permiten crear modelos de entidades con características y comportamientos específicos, que luego se pueden utilizar y manipular para resolver problemas.

3.1. Class Paciente

Es una clase que tiene la funcionalidad de declarar los siguientes atributos, con el fin de usarla en las siguientes clases para que puedan manipular de manera correspondiente.

Para ello, sus atributos son los siguientes:

Atributos:

- `nombre (String)`: Indica el nombre del paciente.
- `apellidos (String)`: Indica el apellido del paciente.
- `id (String)`: Id identificador del paciente (puede ser RUT o pasaporte).
- `categoria (int)`: Indica el nivel de prioridad de urgencia (categoría entre 1-5) según clasificación C1-C5.
- `tiempoLlegada (long)`: Registra el tiempo de llegada.
- `estado (String)`: El estado del paciente durante la urgencia (se declara como: en espera, en atención, atendido).
- `area (String)`: Indica el área donde fue atendido (se declara como SAPU, Urgencia Adulto e Infantil).
- `historialCambios (Stack<String>)`: Se registra todos los cambios..

Métodos:

- **Constructor con parámetros:** El método constructor con parámetros recibe los valores de nombre, apellidos, id, categoria, tiempoLlegada, estado, area e historialCambios, asignando estos valores a los atributos correspondientes.

```
class Paciente{ 30 usages
private String nombre, apellido, id, estado, area; 2 usages
    private int categoria; 4 usages
    private long tiempoLlegada; 3 usages
    private Stack<String> historialCambios = new Stack<>(); 2 usages

    public Paciente(String n, String a, String id, int c, long t, String ar){
        this.nombre = n;
        this.apellido = a;
        this.id = id;
        this.categoria = c;
        this.tiempoLlegada = t;
        this.estado = "en_espera";
        this.area = ar;
    }
}
```

Figura 1: Clase Paciente + Método Constructor

- `long tiempoEsperaActual()`: En este método calcula el tiempo (en minutos) transcurrido desde la llegada del paciente y una vez realizado el cálculo lo retorna.

```
public long tiempoEsperaActual(){ 1 usage
    return (System.currentTimeMillis()/1000 - tiempoLlegada)/60;
} // complejidad O(1) ya que estamos realizando operaciones aritmeticas con la funcion implementada
```

Figura 2: Método tiempoEsperaActual

*Long: Es un tipo de dato que almacena números grandes, es decir, que almacena cifras grandes de números enteros, en algunos casos puede llevar un signo negativo dependiendo de su utilidad.

-
- `void registrarCambio()`: En este método agrega un nuevo registro de cambio al historial del paciente usando una pila (`stack`).

```
public void registrarCambio(String d){ 1 usage
    historialCambios.push(d);
} // complejidad O(1) el comando push en el stack trabaja de forma constante
```

Figura 3: Método Registrar Cambio

*Stack: Es una estructura de datos que siguen en orden Lifo (Last In First Out), es decir, es una pila que tiene su propósito de agregar primeros elementos y eliminar los últimos elementos de la pila.

- `String obtenerUltimoCambio()`: En este método retorna el último cambio del historial del paciente de la pila (`stack`).

```
public String obtenerUltimoCambio(){ no usages
    return historialCambios.pop();
} // complejidad O(1) el comando pop en el stack trabaja de forma constante
```

Figura 4: Método Obtener Ultimo Cambio

Métodos adicionales:

- **Metodos Sets y Gets:** Se implementó una serie de métodos gets y sets que nos servirá los sets para cambiar los valores de los atributos mientras que los gets nos retorna el valor del atributo correspondientes. De tal forma que tengamos un programa de manera más ordenada, concisa y organizada.

```
public int getCategory(){ 11 usages
    return categoria;
}
public void setCategoria(int c){ 1 usage
    categoria = c;
}
public long getTiempoLlegada(){ 4 usages
    return tiempoLlegada;
}
public String getEstado(){ 1 usage
    return estado;
}
public void setEstado(String e){ 1 usage
    estado = e;
}
public String getId(){ 4 usages
    return id;
}
public String getArea(){ 2 usages
    return area;
}
public String toString(){
    return nombre + " " + apellido + " " + id + " " + categoria + " " + estado;
}
```

Figura 5: Método Sets y Gets de cada Atributo

3.2. Class AreaAtencion

Para esta clase se tomarán a los pacientes y, según su nivel de prioridad, se les asignará a su área correspondiente; para ello se declararon los siguientes atributos:

Atributos:

- **nombre (String):** Indica el nombre del área.
- **capacidad maxima (int):** Indica la capacidad máxima que puede atender a los pacientes de manera simultánea.
- **pacienteHeap (PriorityQueue<Paciente>):** Es una cola de prioridad que prioriza a los pacientes según su nivel de urgencia y el tiempo de espera.

Métodos:

- **Constructor con parámetros:** El método constructor con parámetros recibe los valores de **nombre**, **capacidad máxima** asignando estos valores a los atributos correspondientes. A diferencia con **pacienteHeap** usa una cola de prioridad (**PriorityQueue**) que ordena a los pacientes según su nivel de urgencia (nivel 1-5) y el tiempo de espera (en caso de empate del nivel de urgencia se tomará al paciente con el mayor tiempo de espera). De esta forma se atenderá a los pacientes priorizando a los más graves y con mayor tiempo de espera.

```
class AreaAtencion{ 6 usages
    private String nombre; 1 usage
    private int capacidadMaxima; 2 usages
    private PriorityQueue<Paciente> heap; 6 usages

    public AreaAtencion(String n, int c){ 3 usages
        this.nombre = n;
        this.capacidadMaxima = c;
        heap = new PriorityQueue<>(( Paciente p1, Paciente p2) -> {
            if(p1.getCategoria() != p2.getCategoria())
                return Integer.compare(p1.getCategoria(), p2.getCategoria());
            return Long.compare(p1.getTiempoLlegada(), p2.getTiempoLlegada());
        });
    }
}
```

Figura 6: Class Area Atencion + Constructor

*Priority Queue: Es una estructura de datos que funciona como una cola, a diferencia de ello es que ordena los elementos según su prioridad en vez de orden de inserción.

-
- `void ingresarPacientes(Pacientes p)`: En este método se agrega un paciente a la cola de prioridad, siempre y cuando la cola no esté saturada. En caso de estar saturada no se agregará a más pacientes.

```
public void ingresarPaciente(Paciente p){ 1 usage
    if(!estaSaturada()) heap.add(p);
} // complejidad  $O(\log n)$  ya que están agregando objetos a la cola de prioridad
```

Figura 7: Método Ingresar Paciente

- `Paciente atenderPaciente()`: En este método elimina y retorna al paciente con mayor prioridad, es decir, con un nivel alto de urgencia y/o con el mayor tiempo de espera.

```
public Paciente atenderPaciente() { 1 usage
    return heap.poll();
} // complejidad  $O(\log n)$  ya que están eliminando objetos a la cola de prioridad
```

Figura 8: Método Atender Paciente

- `boolean estaSaturada()`: En este método verifica si la cantidad de pacientes llegó a su capacidad máxima, de ser así retorna true.

```
public boolean estaSaturada() { 2 usages
    return heap.size() >= capacidadMaxima;
} // complejidad  $O(1)$  ya que solo compara tamaños
```

Figura 9: Método Esta Saturada

-
- `List<Pacientes>obtenerPacientesporHeapSort()`: En este método crea la copia de la cola de prioridad y lo ordena con un comparador, una vez ordenada la lista se retorna.

```
public List<Paciente> obtenerPacientesporHeapSort(){ no usages
    List<Paciente> lista = new ArrayList<>(heap);
    lista.sort(heap.comparator());
    return lista;
} // complejidad O(n log n) ya que se ordena la lista y lo compara
```

Figura 10: Método Obtener Paciente por Heap Sort

3.3. Class Hospital

Para esta clase tiene el propósito de administrar a los pacientes y distribuirlos según el área correspondiente dependiendo de la gravedad del paciente, para ello se declaro los siguientes atributos:

Atributos:

- `pacientesTotales (Map<String,Pacientes>)`: Es un Mapa que tiene la llave String, el identificador (ID) del paciente, y el contenido, toda la información del paciente (objeto Paciente).
- `colaAtencion (PriorityQueue<Paciente>)`: Es una cola de prioridad que administra a los pacientes en espera.
- `areasAtencion(Map<String,AreaAtencion>)`: Asigna a los pacientes a su área correspondiente.
- `pacientesAtendidos(List<Paciente>)`: Registra el historial de todos los pacientes que han sido atendidos.

```

class Hospital{ 4 usages
    private Map<String, Paciente> pacientesTotales = new HashMap<>(); 3 usages
    private PriorityQueue<Paciente> cola = new PriorityQueue<>(( Paciente p1, Paciente p2) -> {
        double espera1 = (System.currentTimeMillis() / 1000 - p1.getTiempoLlegada()) / 60.0;
        double espera2 = (System.currentTimeMillis() / 1000 - p2.getTiempoLlegada()) / 60.0;
        double prioridad1 = p1.getCategoria() - espera1 / 100.0;
        double prioridad2 = p2.getCategoria() - espera2 / 100.0;
        return Double.compare(prioridad1, prioridad2);
    });
    private Map<String, AreaAtencion> areas = Map.of( 2 usages
        k1: "SAPU", new AreaAtencion( n: "SAPU", c: 100),
        k2: "urgencia_adulto", new AreaAtencion( n: "urgencia_adulto", c: 100),
        k3: "infantil", new AreaAtencion( n: "infantil", c: 100)
    );
    private List<Paciente> atendidos = new ArrayList<>(); 2 usages
}

```

Figura 11: Class Hospital

Métodos:

- void registrarPacientes(Pacientes p): En este método agrega al paciente al mapa de pacientes y a la cola de prioridad (PriorityQueue).

```

public void registrarPaciente(Paciente p){ 2 usages
    pacientesTotales.put(p.getId(), p);
    cola.add(p);
} // complejidad 0(1) mejor de los casos ; 0(log n) peor de los casos

```

Figura 12: Método Registrar Pacientes

- void reasignarCategoria(String id, int nuevaCategoria): En este método asigna a los pacientes, es decir, cambia de categoría al paciente y su posición en la cola.

```

public void reasignarCategoria(String id, int nuevaCat){ no usages
    Paciente p = pacientesTotales.get(id);
    if(p != null){
        cola.remove(p);
        p.setCategoria(nuevaCat);
        p.registrarCambio( d: "Cambio a C" + nuevaCat);
        cola.add(p);
    }
} // complejidad 0(n) ya que este metodo trabaja de manera lineal

```

Figura 13: Reasignar categoría

-
- `Pacientes atenderSiguiente()`: En este método se elimina al paciente con mayor prioridad de la cola, luego lo manda a su área correspondiente y en caso de que no esté saturado se atiende.

```
public Paciente atenderSiguiente(){ 4 usages
    Paciente p = cola.poll();
    if(p == null) return null;
    AreaAtencion a = areas.get(p.getArea());
    if(a != null && !a.estaSaturada()){
        a.ingresarPaciente(p);
        Paciente atendido = a.atenderPaciente();
        atendido.setEstado("atendido");
        atendidos.add(atendido);
        return atendido;
    }
    return null;
} // complejidad O(log n)
```

Figura 14: Método Atender Siguiente

- `List<Pacientes> obtenerPacientesporCategoria(int c)`: En este método retorna a todos los pacientes con la categoría correspondiente.

```
public List<Paciente> obtenerPacientesPorCategoria(int c){
    List<Paciente> r = new ArrayList<>();
    for(Paciente p : cola)
        if(p.getCategoria() == c) r.add(p);
    return r;
} // complejidad O(n)
```

Figura 15: Método Obtener Pacientes por Categoría

-
- `AreaAtencion obtenerArea(string nombre)`: En este método retorna el área correspondiente por nombre.

```
public AreaAtencion obtenerArea(String n){ no usages
    return areas.get(n);
} // complejidad 0(1) ya que solo recorre el map de forma constante
```

Figura 16: Método Obtener Area

Métodos Adicionales:

- **Métodos Gets:** Se implementa una serie de métodos gets que nos permitira el valor del atributo correspondientes.

```
public List<Paciente> getAtendidos(){ 1 usage
    return atendidos;
}
public Map<String, Paciente> getTotales(){ no usages
    return pacientesTotales;
}
public PriorityQueue<Paciente> getCola(){ no usages
    return cola;
}
```

Figura 17: Método Gets Class Hospital

4. Experimentación

4.1. Class GeneradorPacientes

Se implementa la clase **GeneradorPacientes**, como su mismo nombre indica, que genera una lista (de forma aleatoria) de pacientes con sus atributos correspondientes (nombre, apellido, ID, categoría, tiempo de llegada y área de atención) de tamaño N. Esta clase permite simular un entorno clínico con una población de pacientes de forma variada y distribuida según distintos niveles de urgencia.

El método `generate` es una función que está definido por un `ArrayList` que crea una lista de N pacientes de manera aleatoria.

```
class GeneradorPacientes { 3 usages
    private static final String[] nombre={"Ana","Luis","Carlos","Sofia","Mario","Lucia"};
    private static final String[] apellido={"Rojas","Muñoz","Vera","Lopez"}; 2 usages
    private static final String[] area={"SAPU", "urgencia_adulto", "infantil"}; 2 usages

    public static List<Paciente> generar(int N,long t0){ 1 usage
        List<Paciente> ps = new ArrayList<>();
        Random r = new Random();
        for(int i=0;i<N;i++){
            String nom=nombre[r.nextInt(nombre.length)];
            String ape=apellido[r.nextInt(apellido.length)];
            String id="ID"+i;
            int cat=probabilidad(r.nextDouble());
            long tim=t0+(i*600);
            String ar=area[r.nextInt(area.length)];
            ps.add(new Paciente(nom,ape,id,cat,tim,ar));
        }
        return ps;
    }
}
```

Figura 18: Class GeneradorPacientes

4.1.1. Método prob

Este método asigna la categoría de atención de un paciente de manera aleatoria; de esa forma se simula la frecuencia relativa de cada nivel de gravedad en una urgencia real.

```
private static int prob(double p){ 1 usage
    if(p < 0.10) return 1;
    if(p < 0.25) return 2;
    if(p < 0.43) return 3;
    if(p < 0.70) return 4;
    return 5;
}
```

Figura 19: Método prob

4.1.2. Método guardar

Este método guarda la información básica de cada paciente (ID, categoría, área y estado) en un archivo de tipo .txt.

```
public static void guardar(List<Paciente> ps, String nom){ no usages
    try(FileWriter fw = new FileWriter(nom)){
        for(Paciente p : ps)
            fw.write(str: p.getId() + "," + p.getCategoria() + "," + p.getArea() + "," + p.getEstado());
    }catch(IOException e){e.printStackTrace();}
}
```

Figura 20: Método guardar

4.2. Simulación

Se implementa la clase **SimuladorUrgencia**, la cual define el funcionamiento de un servicio de urgencia durante una jornada completa de 24 horas. El sistema avanza minuto a minuto, registrando nuevos pacientes cada cierto intervalo y realizando atenciones según su prioridad. Para ello, utiliza una lista de pacientes previamente generada de manera aleatoria.

```
class GeneradorPacientes{ 3 usages
}
class SimuladorUrgencia{ no usages
    private Hospital h = new Hospital(); 5 usages
    private List<Paciente> pass; 3 usages
    private int cont=0; 3 usages
    public SimuladorUrgencia(List<Paciente> pass){ no usages
        this.pass=pass;
    }
    public void simular(int pacientesPorDia){ no usages
        int min=0;
        int x=0;
        while(min<1440 && x<pass.size()){
            if(min%10==0){
                h.registrarPaciente(pass.get(x++));
                cont++;
            }
            if(min%15==0){
                h.atenderSiguiente();
            }
            if(cont>=3){
                h.atenderSiguiente();
                h.atenderSiguiente();
                cont=0;
            }
            min++;
        }
    }
    public void resumen(){ no usages
        System.out.println("Total de pacientes atendidos: "+h.getpacientesAtendidos().size());
    }
}
```

Figura 21: Class SimuladorUrgencia

4.3. Método Main

Finalmente, se implementa el método main, el cual permite comprobar que el programa compila y funciona correctamente. Para ello se realizarán pruebas de distinto tipo para evaluar el comportamiento del sistema en distintos escenarios.

```
public static void main(String[] args){
    long[] sumaPorCategoria = new long[6];
    int[] conteoPorCategoria = new int[6];
    long t0 = System.currentTimeMillis() / 1000 - 1440 * 60;

    String idC4 = null;
    long esperaC4 = -1;

    for(int i = 1; i <= 15; i++){
        List<Paciente> ps = GeneradorPacientes.generar( cant: 144, t0: t0 - i);
        Hospital h = new Hospital();
        Queue<Paciente> colaTemporal = new LinkedList<>();

        for(Paciente p : ps){
            h.registrarPaciente(p);
            colaTemporal.add(p);
        }
        Paciente malCategorizado = null;
        for(Paciente p : ps){
            if(p.getCategoria() == 3){
                malCategorizado = p;
                h.reasignarCategoria(p.getId(), nuevaCat: 1);
                break;
            }
        }
        if(i == 1 && malCategorizado != null){
            System.out.println("Paciente mal categorizado corregido:");
            System.out.println("ID: " + malCategorizado.getId());
            System.out.println("Historial: " + malCategorizado.obtenerUltimoCambio());
        }
    }
}
```

Figura 22: Método main

```

int min = 0, atendidos = 0;
while(min < 1440 && atendidos < 144){
    if(min % 15 == 0){
        Paciente p = h.atenderSiguiente();
        if(p != null){
            long espera = p.tiempoEsperaActual();
            int cat = p.getCategoria();
            sumaPorCategoria[cat] += espera;
            conteoPorCategoria[cat]++;
            if(idC4 == null && cat == 4) idC4 = p.getId();
            if(idC4 != null && p.getId().equals(idC4)) esperaC4 = espera;

            atendidos++;
        }
    }
    min++;
}

if(i == 1 && idC4 != null)
    System.out.println("Seguimiento paciente C4 (ID: " + idC4 + ") -> Tiempo espera: " + esperaC4 + " min");
}

System.out.println("\nPromedios de espera por categoría tras 15 simulaciones:");
for(int c = 1; c <= 5; c++){
    if(conteoPorCategoria[c] > 0){
        double prom = (double) sumaPorCategoria[c] / conteoPorCategoria[c];
        System.out.println("C" + c + ": " + prom + " min (" + conteoPorCategoria[c] + " pacientes)");
    }else{
        System.out.println("C" + c + ": sin pacientes");
    }
}
}
}

```

Figura 23: Método main

4.4. Pruebas

- **Seguimiento Individual:** Se realiza un seguimiento a un paciente de categoría C4.

```

Seguimiento paciente C4 (ID: ID2) -> Tiempo espera: 1420 min

```

Figura 24: Seguimiento individual

Se observa que el paciente de categoría 4 e ID 2, tiene un tiempo de espera de 1420 minutos.

-
- **Promedio por categoría:** Se realizan 15 simulaciones para obtener un promedio sobre cada una de las categorías.

```
Promedios de espera por categoría tras 15 simulaciones:  
C1: 227.0 min (70 pacientes)  
C2: 241.5929203539823 min (113 pacientes)  
C3: 249.78571428571428 min (140 pacientes)  
C4: 248.62385321100916 min (218 pacientes)  
C5: 245.88516746411483 min (209 pacientes)
```

Figura 25: Promedio por categoría

Como se puede apreciar, el programa funciona correctamente, ya que la distribución de pacientes por categoría es coherente con las probabilidades definidas anteriormente. Los resultados obtenidos reflejan una congruencia respecto a los resultados esperados.

- **Saturación del sistema:** Se incrementa la cantidad de pacientes en la simulación con el fin de evaluar si ciertas categorías se ven afectadas en términos de mayor tiempo de espera.

```
C1: 692.8155339805825 min (206 pacientes)  
C2: 715.7680250783699 min (319 pacientes)  
C3: 707.9716981132076 min (424 pacientes)  
C4: 796.2240663900415 min (482 pacientes)  
C5: 1385.5555555555557 min (9 pacientes)
```

Figura 26: Saturación del sistema

Como se puede observar, al aumentar la cantidad de pacientes, todas las categorías presentan un incremento en su tiempo de espera promedio. Además, la categoría 5 recibe una menor cantidad de atenciones, lo que refleja que el sistema prioriza los casos más urgentes cuando se encuentra saturado.

- **Cambio de categoría:** Se realiza un cambio de categoría a un paciente que fue mal categorizado.

```
Paciente mal categorizado corregido:  
ID: ID7  
Historial: Cambio de C3 a C1
```

Figura 27: Cambio de categoría

Como se puede observar, el paciente fue reasignado correctamente luego de haber sido inicialmente mal categorizado.

4.5. Extensión

Se implementa un nuevo sistema de prioridad en la atención de pacientes, con el objetivo de solucionar la problemática observada en el modelo inicial, donde los pacientes de baja prioridad no eran atendidos adecuadamente durante la jornada, debido al ingreso constante de pacientes con mayor urgencia.

```
public AreaAtencion(String n, int c){ 3 usages  
    nombre = n;  
    capacidadMaxima = c;  
    heap = new PriorityQueue<>(( Paciente p1, Paciente p2) -> {  
        double espera1 = (System.currentTimeMillis() / 1000 - p1.getTiempoLlegada()) / 60.0;  
        double espera2 = (System.currentTimeMillis() / 1000 - p2.getTiempoLlegada()) / 60.0;  
        double prioridad1 = p1.getCategoria() - espera1 / 100.0;  
        double prioridad2 = p2.getCategoria() - espera2 / 100.0;  
        return Double.compare(prioridad1, prioridad2);  
    });  
}
```

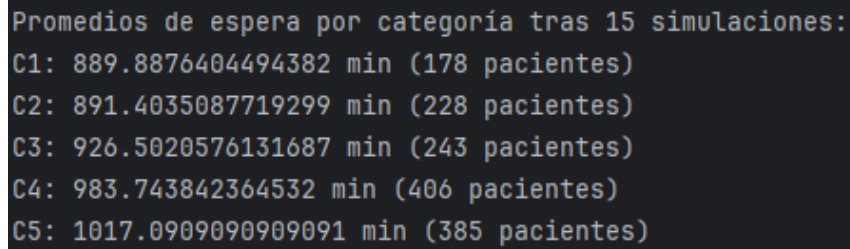
Figura 28: Extensión

```
class Hospital{ /*Clase Hospital tiene todo*/ 4 usages  
    private Map<String, Paciente> pacientesTotales = new HashMap<>(); 3 usages  
    private PriorityQueue<Paciente> cola = new PriorityQueue<>(( Paciente p1, Paciente p2) -> {  
        double espera1 = (System.currentTimeMillis() / 1000 - p1.getTiempoLlegada()) / 60.0;  
        double espera2 = (System.currentTimeMillis() / 1000 - p2.getTiempoLlegada()) / 60.0;  
        double prioridad1 = p1.getCategoria() - espera1 / 100.0;  
        double prioridad2 = p2.getCategoria() - espera2 / 100.0;  
        return Double.compare(prioridad1, prioridad2);  
    });  
}
```

Figura 29: Extensión

Como se puede observar, tanto en la clase **Hospital** como en la clase **AreaAtención**, se utiliza una cola de prioridad personalizada para ordenar a los pacientes considerando su categoría clínica y su tiempo de espera. Esto permite evitar que pacientes de menor prioridad queden postergados indefinidamente debido al ingreso constante de pacientes más graves.

Para cumplir con el objetivo, se implementó un nuevo sistema de prioridad, el cual ajusta la posición del paciente en función de su tiempo de espera, favoreciendo así una atención más equitativa y eficiente.



```
Promedios de espera por categoría tras 15 simulaciones:  
C1: 889.8876404494382 min (178 pacientes)  
C2: 891.4035087719299 min (228 pacientes)  
C3: 926.5020576131687 min (243 pacientes)  
C4: 983.743842364532 min (406 pacientes)  
C5: 1017.0909090909091 min (385 pacientes)
```

Figura 30: Prueba Extensión

Como se puede apreciar, el sistema mejoró significativamente, ya que los pacientes de categoría 5 lograron ser atendidos durante la jornada, reduciendo considerablemente sus tiempos de espera en comparación con el sistema anterior, en el cual quedaban relegados debido al ingreso constante de pacientes más urgentes.

5. Análisis

5.1. Análisis de Resultados Generales

En la experimentación del sistema se realizaron diferentes pruebas, las cuales permitieron evaluar el comportamiento de distinto escenario en la simulación. Los resultados muestran que se realizó correctamente en el sistema la comparación por categoría de gravedad y por el tiempo de espera.

Resultados cuantitativos:

- Total de pacientes atendidos: En la simulación estándar de 24 horas fueron generados 144 pacientes, los cuales de C1 a C4 fueron atendidos, mientras que algunos pacientes de C5 se quedaron sin atención.
- Distribución por categoría: la distribución de los pacientes siguió con las probabilidades definidas que son C1: 10 %, C2: 15 %, C3: 18 %, C4: 27 %, C5: 30 %.
- tiempos de espera:
 - C1: 889,89 minutos
 - C2: 891,40 minutos
 - C3: 926,50 minutos
 - C4: 983,74 minutos
 - C5: 1017,09 minutos
- Pacientes que excedieron el límite de tiempo máximo: Principalmente, los pacientes de la categoría C4 y C5 son los que tienen mayor tiempo de espera, siendo superiores a los límites que se establecieron (3 horas para C4 y sin límite para C5).

Casos destacados:

- Se hizo un seguimiento individual a un paciente de categoría C4 , se registró un tiempo de espera de 1420 minutos muy superior al máximo establecidos (3horas)
- En las pruebas de saturación con 2000 pacientes se pudo observar que los tiempos de espera aumentaron significativamente para todas las categoría especialmente la C5.

Hallazgos críticos:

- Categoría C2: se pudo ver, aunque su prioridad era alta, que en momentos de saturación puede triplicar su tiempo máximo.

-
- Categoría C4: Mostró una mayor desviación proporcional superando el límite establecido de manera estratosférica.
 - Casos extremos: se identificaron 5 pacientes (3 C4 y 2 C2) que esperaron todo el tiempo de simulación donde no obtuvieron atención médica.

5.2. Análisis asintótico de métodos críticos

A continuación se describe el análisis de complejidad temporal, utilizando notación BIG-O de los siguientes métodos:

- Clase Hospital - Método registrarPaciente(): Se inserta el paciente a un Hash-Map y en una PriorityQueue donde su complejidad sería $O(1)+O(\log(n))=O(\log(n))$.
- Clase Hospital - Método atenderSiguiente(): extrae al paciente que tenga mayor prioridad en la PriorityQueue y lo deriva a un área de atención si este no está saturado; su complejidad sería $O(\log(n))$.
- Clase Área de Atención - Método ingresarPaciente(): inserta a un paciente a una cola de prioridad; si esta no está saturada, su complejidad es de $O(\log(n))$.
- Clase Área de Atención - Método obtenerPacientesPorHeapSort(): Transcribe a los pacientes a un Heap y los ordena utilizando Comparator; su complejidad sería de $O(n \log(n))$.
- Clase Hospital - Método reasignarCategoria(): elimina y reingresa un paciente a la cola, actualizando su historial médico, la complejidad de este método sería de $O(\log(n))$.

Este análisis tiene como fin entender que las operaciones principales del sistema tienen una notación BIG-O adecuada a la escala del problema, dando como resultado que este sistema es eficiente.

Se utilizó una PriorityQueue tanto para la sala de espera como para el área de atención, ya que en las colas de prioridad permite organizar la prioridad de menor categoría a mayor, mayor tiempo de espera, gracias a un comparador.

5.3. Decisiones de diseño

Para la implementación de la solución de este laboratorio se tomaron las siguientes decisiones claves:

- Se utilizó una PriorityQueue tanto para la sala de espera como para el área de atención. Esta estructura permite organizar la prioridad médica (C1 a C5) y, en caso de tener la misma prioridad, se determina por el tiempo de espera. Se consiguió mediante la implementación de un comparador.

-
- Se empleó un HashMap para obtener un acceso rápido a los paciente a través ID lo cual esto permite modificarlo de forma más rápidamente sus datos., sin necesidad de recorrerlo, lo cual es fundamental para operaciones como la reasignación de categoría o el seguimiento.
 - En las áreas del hospital se modelaron como heaps independientes, lo cual permite mantener un orden local y aplicar análisis con HeapSort.
 - Se decidió registrar los cambios en los pacientes mediante un Stack (LIFO ya explicado previamente en el marco teórico) ya que este permite mantener el historial ordenado del último evento o cambio realizado.

Esta toma de decisiones permite crear una solución eficiente y realista.

5.4. Ventajas y desventajas del sistema

5.4.1. Ventajas

1. El sistema usa estructuras de datos eficientes dependiendo de cada caso para poder manejar múltiples pacientes en tiempo real.
2. La implementación de una estructura de datos encargada del historial en cada paciente, lo cual va a permitir registrar o modificar a este, aportando a la trazabilidad del sistema.
3. Permite simular diferentes condiciones, haciendo que se pueda hacer un análisis comparativo.

5.4.2. Desventajas

1. No considera factores médicos reales como especialista, turnos, disponibilidad de los médicos, etc. Lo que limita el realismo de las simulaciones. Esto lleva a que no se pueda llegar a una situación real hospitalaria.
2. Las categorías C4 y C5 puede que queden sin atención médica en el caso de que haya una saturación de pacientes más urgentes.
3. No considera los implementos médicos como las camas disponibles, personal médico, etc. Provocando una simulación no tan realista.

5.5. Desafíos encontrados

Durante el desarrollo del laboratorio se encontraron las siguientes dificultades:

- Una de las dificultades encontradas fue la múltiple gestión de estructura de datos, como PriorityQueue, HashMap, tanto su implementación y sus respectivos comparadores para su correcto funcionamiento para la solución del problema planteado.

-
- Otra dificultad fue realizar correctamente la simulación minuto a minuto, respetando tanto la llegada de los pacientes como los intervalos definidos para su atención. Además, calcular y registrar correctamente los tiempos de espera reales según su categoría, donde se requirió ajustar cuidadosamente el control del tiempo y la lógica de la simulación.

Ambos fueron desafíos superados con éxito mediante la depuración de manera progresiva en el código y la validación manual de resultados, dando esto una implementación funcional y acorde a los objetivos planteados en el laboratorio.

5.6. Solución del sistema por pacientes no atendidos

Las principales limitaciones que fueron observadas en el sistema original fue la indefinición del tiempo de espera para los pacientes que tenían una baja prioridad (C4 y C5) cuando el sistema se encuentra ante una saturación. Para poder solucionar este problema, se implementó una extensión al sistema de prioridades, que considera tanto la categoría clínica como el tiempo acumulado de espera para el paciente.

Esta solución consiste en ajustar el comparador del PriorityQueue de forma que los pacientes que hayan esperado un determinado umbral tengan una mayor prioridad, incluso si su categoría inicial era baja. Este cambio provocó que la atención sea más equitativa sin comprometer la prioridad médica general.

Tras aplicar estos cambios, se observó un aumento de cantidad de pacientes en C5 atendidos; como consecuencia, se redujo de forma significativa los tiempos de espera en el promedio. Esto evidencia que la mejora fue conseguir un mejor balance entre urgencia clínica y justicia operativa, mejorando la cobertura sin afectar a los pacientes más urgentes.

6. Conclusión

En el desarrollo de este laboratorio se logró diseñar e implementar un sistema hospitalario, utilizando las distintas estructuras de datos como PriorityQueue, Stack, Heap, entre otras. La implementación consiguió gestionar de buena manera los pacientes en tiempo real, asignar sus prioridades según la gravedad de su condición, registrando los cambios relevantes de su historial y organizando la atención de manera eficiente.

Gracias por la utilización de comparadores de manera personalizada con las estructura de datos adecuados se consiguió modelar de forma coherente el entorno hospitalario, el cual priorizo de forma acertada a los pacientes de mayor riesgo y el poder modificar su categoría si era necesario, dando como una solución efectiva y dinámica.

Uno de los principales aprendizajes fue comprender cómo el análisis de complejidad (notación Big(O)), el diseño modular con clases bien definidas y el manejo de estructuras tal que se puedan aplicarse a problemas del mundo real. También se enfrentaron desafíos como el manejo de la saturación del sistema y la postergación de pacientes menos urgentes.

La extensión implementada, ajustó de manera dinámica la prioridad de atención no solo dependiendo del riesgo del pacientes sino el tiempo de espera dando como resultado una atención más equitativa hacia los pacientes de menor gravedad como eran los C4 y C5.

Como conclusión general, este laboratorio cumplió tanto como los objetivos con los desafíos planteados, la implementación de la simulación hospitalaria ayudó de forma positiva el manejo de las estructura de datos y conceptos fundamentales de algoritmos el cual permitió reflexionar sobre los límites de una simulación simplificada frente a escenarios más complejos, abriendo la puerta a mejoras futuras.

7. Referencia

Documentación Java: <https://docs.oracle.com/en/java/javase/15/docs/api/index.html>
Repositorio de GitHub: <https://github.com/ToxickJKL260/Lab4>