UNIVERSIDAD DIEGO PORTALES
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES

ESTRUCTURAS DE DATOS & ANÁLISIS DE ALGORITMOS

Laboratorio 4: Implementación de un Sistema de Atención de Urgencias con estructura de Datos Eficientes

---

Autores:
Cinthya Fuentealba Bravo
Joaquín Roco Fuenzalida

Profesor:
Marcos Fantoval

11 de Junio de 2025

Índice

1. Introducción

Las atenciones de salud en urgencias deben ser rápidas para poder atender de forma oportuna y eficiente al público al que se atiende. Los pacientes que llegan a los centros de salud a lo largo del país se enfrentan a situaciones en las cuales deben esperar por mucho tiempo, debido a la alta demanda que existe, para poder recibir la atención que necesitan; por ende, se requiere priorizar a algunas personas en comparación con otras. Es por este motivo que el Ministerio de Salud de Chile realizó una categorización oficial, la cual estandariza la prioridad y el tiempo máximo que se deben demorar para atender a los pacientes; esta es la siguiente:

- C1: Emergencia Vital: atención inmediata.
- C2: Urgencia / Alta Complejidad: hasta 30 minutos.
- C3: Mediana Complejidad: hasta 1 hora y 30 minutos.
- C4: Baja Complejidad: hasta 3 horas.
- C5: Atención General: Sin tiempo máximo. Depende de la demanda que existe.

En este laboratorio, como objetivo se busca poder solucionar el problema propuesto, generando que las personas esperen el mínimo tiempo posible para recibir su atención de salud, haciendo el sistema eficiente. Por ello, se implementa y diseña un sistema utilizando estructuras de datos avanzadas como colas de prioridad, pilas y mapas. Además, de combinar conceptos de las estructuras ya mencionadas, también se realiza el análisis algorítmico del código ejecutado. Todo esto es implementado en lenguaje informático de Java.

Durante el desarrollo del informe se detallará cómo se realiza el programa informático, las partes que lo componen, y sobre todo poder entender el funcionamiento de las estructuras de datos ya mencionadas, para así poder dar una solución frente a un problema que ocurre en la vida real. Además de generar una simulación para poder comprobar su funcionamiento.

En el siguiente link: [1] se encuentra el repositorio de GitHub con todos los documentos requeridos en la realización de este informe.

2. Implementación

Para comenzar a dar solución al problema planteado, y teniendo en cuenta las consideraciones que se deben tener en la simulación, se inicia la creación del código informático, para ello lo primero que se realiza en este laboratorio es la creación de las siguientes clases:

2.1. Clase Paciente

Esta clase representa a los pacientes que van a recibir la atención de salud. Se compone por los siguientes atributos:

- `String nombre`: representa el nombre del paciente que se atenderá.
- `String apellido`: representa el apellido del paciente.
- `String id`: simboliza el RUN o el pasaporte del paciente, este es único a nivel nacional.
- `int categoria`: número del nivel de urgencia en el que es clasificado el paciente, se encuentra en un rango de 1 a 5, según la clasificación C1 a C5 (aleatorio según probabilidad).
- `long tiempoLlegada`: instante de llegada del paciente en formato Unix timestamp.
- `String estado`: representa el estado de atención del paciente, es una de las tres categorías que existen. El paciente se encuentra en espera, en atención o atendido.
- `String area`: corresponde al área hospitalaria por la cual debe ser atendido el paciente; estas pueden ser una de las tres opciones que existen. Las áreas permitidas son SAPU, urgencia adulto o urgencia infantil.
- `Stack<String>historialCambios`: es una pila que se encarga de registrar cada cambio relevante en la atención o categorización del paciente.

Lo siguiente realizado en esta clase es la implementación de los siguientes métodos:

- `long tiempoEsperaActual()`: calcula el tiempo que ha transcurrido desde que el paciente es registrado en su llegada al servicio asistencial, hasta el momento actual, representado en minutos.
- `void registrarCambio (String descripcion)`: permite añadir un cambio ocurrido en el historial del paciente.
- `String obtenerUltimoCambio()`: obtiene el último cambio registrado y lo remueve.

```java
class Paciente{  12 usages
        private String nombre;  3 usages
        private String apellido;  3 usages
        private String id;  3 usages
        private int categoria;  3 usages
        private long tiempoLlegada;  4 usages
        private String estado; //puede ser en_espera, en_atencion, atendido.  3 usages
        private String area; //Un valor entre SAPU, urgencia_adulto e infantil  3 usages
        private Stack<String> historialCambios;  6 usages

        public Paciente(String nombre, String apellido, String id, int categoria,  no usages
                        long tiempoLlegada, String estado, String area) {
            this.nombre = nombre;
            this.apellido = apellido;
            this.id = id;
            this.categoria = categoria;
            this.tiempoLlegada = tiempoLlegada;
            this.estado = estado;
            this.area = area;
            historialCambios = new Stack<>();
        }
```

Figura 1: Código Java de `Clase Paciente`, atributos.

```java
        /**Metodos obligatorios*/
        public long tiempoEsperaActual(){  no usages
            return (System.currentTimeMillis()/1000-tiempoLlegada)/60;
        }

        public void registrarCambio(String descripcion){  no usages
            historialCambios.push(descripcion);
        }

        public String obtenerUltimoCambio(){  no usages
            return historialCambios.isEmpty() ? "Sin cambios" : historialCambios.pop();
        }
    }
```

Figura 2: Código Java de `Clase Paciente`, métodos.

2.2. Clase AreaAtencion

Esta clase representa una zona del hospital. Se modela como un montón *(heap)*, en donde se insertan los pacientes según la prioridad que poseen. Las áreas que existen en el servicio de atención son tres; estas son: SAPU, urgencia adulto y urgencia infantil.

Se compone de los siguientes atributos:

- `String nombre`: corresponde al nombre del área designada a la atención del paciente.
- `PriorityQueue<Paciente>pacientesHeap`: representa una cola de prioridad que mantiene un orden de los pacientes según su categorización (de 1 a 5, en orden del más urgente al menos urgente), y en caso de empate se observa el tiempo de espera que posee el paciente, dando prioridad al que lleva mayor tiempo.
- `int capacidadMaxima`: corresponde a un número que representa la cantidad máxima permitida para atender pacientes de forma simultánea.

A continuación se detallan los métodos de la clase. Estos son:

- `void ingresarPaciente(Paciente p)`:
- `Paciente atenderPaciente()`: obtiene al paciente con mayor prioridad y lo remueve.
- `boolean estaSaturada()`: retorna *true* si ya se alcanzó la capacidad máxima y retorna *false* si aún no se alcanza la capacidad máxima.
- `List<Paciente>obtenerPacientesPorHeapSort()`: devuelve los pacientes según prioridad usando Heap-Sort.

```
class AreaAtencion{ //Representa una zona del hospital  2 usages
    /** SAPU, urgencia adulto e infantil, segun prioridad*/

    private String nombre;  3 usages
    private PriorityQueue<Paciente> pacientesHeap; //Cola de prioridad  8 usages
    private int capacidadMaxima; // capacidad max de pacientes que maneja simultaneos  4 usages

    public AreaAtencion(String nombre, int capacidadMaxima){  no usages
        this.nombre = nombre;
        this.capacidadMaxima = capacidadMaxima;
        this.pacientesHeap = new PriorityQueue<>(new Comparator<Paciente>() {
            public int compare(Paciente p1, Paciente p2) {
                if (p1.getCategoria() != p2.getCategoria()) {
                    return Integer.compare(p1.getCategoria(), p2.getCategoria());
                }else{
                    return Long.compare(p1.getTiempoLlegada(), p2.getTiempoLlegada());
                }
            }
        });
    }
```

Figura 3: Código Java de `Clase AreaAtencion`, atributos.

```java
    /** Metodos Obligatorios */
    public void ingresarPaciente(Paciente p){  no usages
        // Inserta un nuevo paciente en el heap}
        if(!estaSaturada()) {
            pacientesHeap.offer(p);
        }
    }

    public Paciente atenderPaciente(){  no usages
        //obtiene y remueve al paciente con mayor prioridad.
        return pacientesHeap.poll();
    }

    public boolean estaSaturada(){  1 usage
        //retorna true si se alcanzó la capacidad máxima
        return pacientesHeap.size() >= capacidadMaxima;
    }

    public List<Paciente> obtenerPacientesPorHeapSort(){  no usages
        //devuelve los pacientes según prioridad usando HeapSort.

        List<Paciente> pacientesCopia = new ArrayList<>(pacientesHeap);
        pacientesCopia.sort(pacientesHeap.comparator());
        return pacientesCopia;
    }
}
```

Figura 4: Código Java de `Clase AreaAtencion`, métodos.

2.3. Clase Hospital

Esta clase se encarga de administrar el hospital a nivel global, esto significa que administra el ingreso de pacientes, permite conocer la atención que está recibiendo el paciente y realiza un seguimiento de ellos.

A continuación se detallan los atributos implementados en esta clase, estos son:

- `Map<String, Paciente>pacientesTotales`: corresponde a un mapa que asocia el identificador único de cada paciente (`String id`), con el `Paciente` del hospital, de esta forma se crea un acceso rápido a la información de cada paciente que se encuentra registrado en el sistema.
- `PriorityQueue<Paciente>colaAtencion`: cola central que gestiona a los pacientes que se encuentran en estado de espera.
- `Map<String, AreaAtencion>areasAtencion`: realiza asignacion de `Paciente`

a un área en específico.

- `List<Paciente>pacientesAtendidos`: genera una lista del historial de todos los pacientes que han sido atendidos en el centro asistencial.

A continuación como en clases anteriores, se detallan los métodos implementados en esta clase, siendo los siguientes:

- `void registrarPaciente(Paciente p)`: inserta al paciente en la cola general y en el mapa `pacientesTotales`. También asigna la categoría y área de atención.
- `void reasignarCategoria (String id, int nuevaCategoria)`: permite actualizar la categoría de un paciente y registrar el cambio en su historial.
- `Paciente atenderSiguiente()`: extrae de la cola general el paciente que posee la mayor prioridad y se asigna al área correspondiente.
- `List<Paciente>obtenerPacientesPorCategoria(int categoria`:corresponde a una lista de pacientes que se encuentra en espera de una determinada categoría.
- `AreaAtencion obtenerArea(String nombre)`: devuelve el área que corresponde según el nombre del paciente.

```
class Hospital{  no usages
    //Administra el ingreso, atención y seguimiento de pacientes a nivel global del hospital

    private Map<String, Paciente> pacientesTotales = new HashMap<>();  2 usages
    private PriorityQueue<Paciente> colaAtencion;  4 usages
    private Map<String, AreaAtencion> areasAtencion = new HashMap<>();  5 usages
    private List<Paciente> pacientesAtendidos = new ArrayList<>();  1 usage

    public Hospital(){  no usages
        colaAtencion = new PriorityQueue<>(new Comparator<Paciente>() {
            public int compare(Paciente p1, Paciente p2) {
                if (p1.getCategoria() != p2.getCategoria()) {
                    return Integer.compare(p1.getCategoria(), p2.getCategoria());
                } else {
                    return Long.compare(p1.getTiempoLlegada(), p2.getTiempoLlegada());
                }
            }
        });
        areasAtencion.put("SAPU", new AreaAtencion( nombre: "SAPU",  capacidadMaxima: 100));
        areasAtencion.put("Urgencia Adulto", new AreaAtencion( nombre: "Urgencia Adulto",  capacidadMaxima: 100));
        areasAtencion.put("Urgencia Infantil", new AreaAtencion( nombre: "Urgencia Intantil",  capacidadMaxima: 100));
    }
```

Figura 5: Código Java de `Clase Hospital`, atributos.

```java
/** Metodo Obligatorios */
public void registrarPaciente(Paciente p){  no usages
    //inserta al paciente en la cola general y en la estructura pacientesTotales.
    // También asigna su categoría y su área de atención.
    pacientesTotales.put(p.getId(), p);
    colaAtencion.offer(p);
}

public void reasignarCategoria(String id, int nuevaCategoria){  no usages
    //permite actualizar la categoría de un paciente y registrar el cambio en su historial.

    Paciente p = pacientesTotales.get(id);
    if(p!=null){
        p.registrarCambio( descripcion: "Cambio de categoría: " + p.getCategoria() + "a" + nuevaCategoria);
        p.setCategoria(nuevaCategoria);
    }
}
```

Figura 6: Código Java de `Clase Hospital`, métodos parte 1.

```java
    public Paciente atenderSiguiente(){  no usages
        //extrae de la cola general el paciente con mayor prioridad y lo asigna a su área.
        Paciente p = colaAtencion.poll();
        if(p!=null){
            p.setEstado("Atendido");
            pacientesAtendidos.add(p);
            areasAtencion.get(p.getArea()).ingresarPaciente(p);
        }
        return p;
    }

    public List<Paciente> obtenerPacientesPorCategoria(int categoria){  no usages
        // lista de pacientes en espera de una determinada categoría.
        return colaAtencion.stream().filter( Paciente p -> p.getCategoria() == categoria).collect(Collectors.toList());
    }

    public AreaAtencion obtenerArea(String nombre){  no usages
        //devuelve el área correspondiente por nombre.
        return areasAtencion.get(nombre);
    }
```

Figura 7: Código Java de `Clase Hospital`, métodos parte 2.

3. Experimentación

En esta sección del informe se realiza la experimentación, así como a comprobar el funcionamiento del sistema de simulación de la atención en las áreas de urgencias.

Como objetivo principal en esta sección es evaluar como se comporta el sistema creado ante la llegada de pacientes y la atención que reciben. Para comprobar el sistema es necesario tener en consideración los tiempos máximos (que se explicaron en la parte de la introducción),los cuales están definidos por el Ministerio de Salud y las prioridades clínicas de cada categoría.

Se desarrolla una simulación representando una jornada de 24 horas de duración, en la cual se generan pacientes aleatorios y se les asigna la atención correspondiente según las clasificaciones pertinentes para cada caso.

Además, se realiza una recopilación de datos empíricos como los tiempos de espera, cantidad de pacientes atendidos por categoría y cantidad de pacientes que excedieron el tiempo máximo permitido. De esta forma, se puede analizar el funcionamiento y eficiencia del sistema, para también así proponer mejoras al sistema.

A continuación se detallan las clases creadas para llevar a cabo esta sección.

3.1. Simulación

Para tener en consideración el contexto de este problema, se implementa una simulación por un periodo de 24 horas, se administra el flujo de llegada de los pacientes y también la atención recibida por los pacientes en una de las unidades de urgencias.

En esta simulación se considera que un paciente nuevo llega al servicio asistencial cada 10 minutos. Debido a que se trata de una simulación de la vida real, para hacerlo aún más similar a la vida cotidiana, se realiza una categorización de los pacientes según una distribución de probabilidades. Esto se realiza, ya que en la vida real no llega la misma cantidad de pacientes según categoría, sino que por lo general la cantidad de pacientes aumenta a medida que la complejidad de la urgencia disminuye.

Categorización	Porcentaje de Probabilidad
C1	10 %
C2	15 %
C3	18 %
C4	27 %
C5	30 %

Tabla 1: Relación que existe entre una categorización según la urgencia del paciente (esta va de 1 a 5, siendo 1 muy grave y 5 menos grave), y el porcentaje de probabilidad de ser paciente asociado a la categorización.

Además, cada 15 minutos transcurridos, se atiende a un paciente con una política de atención priorizada, basada en:

1. Categoría de urgencia.
2. Tiempo de espera acumulado desde su llegada y registro hasta el momento actual.

Otra apreciación que hay que tener en cuenta es que durante el transcurso del día la demanda va variando, siendo que si aumenta la demanda más de lo habitual se debe atender a dos pacientes que se encuentren en la cola, por cada 3 pacientes nuevos que lleguen al centro asistencial.

Por otra parte, se implementa una estructura del historial de atención por paciente, para ello se crea una pila la cual registra los cambios que se hayan generado en el paciente.

Por último, cada área del hospital es modelada por un montón independiente, esto genera que se aplique el ordenamiento por montón y así analizar la carga de atención que existe en cada área.

3.2. Clase GeneradorPacientes

Esta clase se encarga de crear pacientes de manera aleatoria y los guarda en un archivo llamado `Pacientes_24h.txt`. En esta clase se crean pacientes con:

- Nombre.
- Apellido
- Id único.
- Categoría con la probabilidad correspondiente.
- Timestamp de llegada cada 10 minutos.
- Estado inicial (en espera).
- Historial vació.

En esta clase se genera una lista de pacientes, de acuerdo con las probabilidades que existen por categoría, y creando una llegada de pacientes cada 10 minutos desde un timestamp base. Además, los datos son exportados a un archivo llamado `Pacientes_24h.txt`, esto permite reutilizar los pacientes para diferentes pruebas y así no generarlos nuevamente.

```java
class GeneradorPacientes {  no usages
    private static final String[] nombres = {"Ana", "Bastian", "Carla",  2 usages
            "David", "Elena", "Fernando", "Gabriela", "Hector", "Isabel", "Juan"};
    private static final String[] apellidos = {"Araya", "Briones", "Castillo",  2 usages
            "Díaz", "Escobar", "Figueroa", "Guzman", "Herrera", "Ibañez", "Jiménez"};
    private static final Random rand = new Random();  4 usages

    public static List<Paciente> generarPacientes(int cantidad) {  no usages
        List<Paciente> pacientes = new ArrayList<>();
        long timestampBase = System.currentTimeMillis() / 1000;
        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); //Id unico P1000, P1001, ....
            int categoria = generarCategoria();
            long llegada = timestampBase + (i * 600); //Cada 10 minutos llegada de paciente nuevo
            String area = asignarAreaAleatoria();

            //Creación de paciente
            Paciente p = new Paciente(nombre, apellido, id, categoria, llegada, area);
            pacientes.add(p);
        }
        return pacientes;
    }
```

Figura 8: Código Java de `Clase GeneradorPacientes` parte 1.

```java
    // Asignación de categoria según probabilidad
    private static int generarCategoria() {  1 usage
        int cr = rand.nextInt( bound: 100);
        {//Genera un numero entre el 0 y el 100
            if (cr < 10) {return 1;} // C1 el 10% va del 0->9
            else if (cr < 25) {return 2;} //C2 el 15% va del 10->24
            else if (cr < 43) {return 3;} //C3 el 18% va del 25->42
            else if (cr < 70) {return 4;} //C4 el 27% va del 43->69
            else if (cr < 100) {return 5;} //C5 el 30% va del 70->99
        }
        return cr;
    }
    private static String asignarAreaAleatoria() {  1 usage
        String[]areas = {"SAPU","Urgencia Adulto","Urgencia Infantil"};
        return areas[rand.nextInt(areas.length)];
    }
```

Figura 9: Código Java de `Clase GeneradorPacientes` parte 2.

```java
public static void guardarPacientesEnArchivo(List<Paciente> pacientes,   no usages
                                             String nombreArchivo)throws IOException{
    BufferedWriter bw = new BufferedWriter(new FileWriter(nombreArchivo));
    for (Paciente p : pacientes){
        bw.write( str: p.getId() + "," + p.getNombre() + ","
                + p.getApellido() + "," + p.getCategoria() + ","
                + p.getTiempoLlegada() + "," + p.getArea() + "\n");
    }
    bw.close();
}
```

Figura 10: Código Java de `Clase GeneradorPacientes` parte 3.

3.3. Clase SimuladorUrgencia

Para continuar con el desarrollo del laboratorio, se crea esta clase llamada `Clase SimuladorUrgencia`, el cual se encarga de generar una jornada de 24 horas en el hospital y realiza una gestión sobre, la llegada de pacientes cada 10 minutos, atención de pacientes cada 15 minutos y la atención adicional de los pacientes cada vez que se acumulan 3 nuevos pacientes se deben atender a 2 pacientes.

En la clase también se implementa un método por el cual se verifica si los pacientes excedían el tiempo máximo de espera, según la categoría correspondiente.

Esta clase realiza la creación de un método llamado `simular (int pacientesPorDia)`, el cual recibe el número total de pacientes del día, luego lee la lista generada desde el archivo, después simula minuto a minuto del día, lo que corresponde a 1440 minutos (24 horas X 60 minutos). Luego se lleva un control del tiempo, los pacientes que están en espera y los pacientes que han sido atendidos.

Por último durante el desarrollo se almacenan las métricas de:

- Total de pacientes atendidos.
- Cantidad atendida por categoría (C1 a C5).
- Promedio de tiempo de espera por categoría.
- Pacientes que excedieron el tiempo máximo permitido.
- Peores tiempos de espera por categoría.
- Pacientes que no alcanzaron a ser atendidos en la jornada simulada.

```java
class SimuladorUrgencia{  no usages
    private Hospital hospital = new Hospital();  2 usages
    private List<Paciente> pacientes;  3 usages
    private int nuevosAcumulados = 0;  3 usages
    private int tiempoActual = 0;  no usages
    private Map<Integer, List<Long>> tiemposEsperaPorCategoria = new HashMap<>();  3 usages
    private List<Paciente> pacietesFueraDeTiempo = new ArrayList<>();  3 usages

    public SimuladorUrgencia(List<Paciente> pacientes){  no usages
        this.pacientes = pacientes;
        for (int i =1; i<=5; i++){
            tiemposEsperaPorCategoria.put(i, new ArrayList<>());
        }
    }
```

Figura 11: Código Java de `Clase SimuladorUrgencia` parte 1.

```java
public void simular(int pacientesPorDia){  no usages
    int tiempoActual = 0;
    int indicePaciente = 0;
    int acumulados = 0;

    for (tiempoActual = 0; tiempoActual < 1440; tiempoActual++){//Llegada de pacientes cada 10 minutos
        if((tiempoActual % 10 == 0) && (indicePaciente < pacientes.size())){
            Paciente p = pacientes.get(indicePaciente);
            hospital.registrarPaciente(p);
            nuevosAcumulados++;
            indicePaciente++;
        }

        if (tiempoActual % 15 == 0){//atención cada 15 minutos
            atenderPaciente();
        }

        if(nuevosAcumulados >= 3){
            atenderPaciente();
            nuevosAcumulados = 0;
        }
    }
    imprimirResumen();
}
```

Figura 12: Código Java de `Clase SimuladorUrgencia` parte 2.

```java
private void atenderPaciente(){  2 usages
    Paciente p = hospital.atenderSiguiente();
    if(p!=null){
        long espera = (System.currentTimeMillis() / 1000) - (p.getTiempoLlegada());
        tiemposEsperaPorCategoria.get(p.getCategoria()).add(espera);
        pacietesFueraDeTiempo.add(p);

        if(excedeTiempoMaximo(p.getCategoria(), espera)){
            pacietesFueraDeTiempo.add(p);
        }
    }
}

private boolean excedeTiempoMaximo(int categoria, long espera){  1 usage
        switch(categoria){
            case 1: return espera > 0;
            case 2: return espera > 30;
            case 3: return espera > 90;
            case 4: return espera > 180;
            case 5: return false;
            default: return false;
        }
}
```

Figura 13: Código Java de `Clase SimuladorUrgencia` parte 3.

```java
private void imprimirResumen(){  1 usage
    for(int categoria = 1; categoria <= 5; categoria++){
        List<Long> tiempos = tiemposEsperaPorCategoria.get(categoria);
        double promedio = tiempos.stream().mapToLong(Long::longValue).average().orElse( other: 0);
        System.out.println("Categoría" + categoria
                + ":atendidos = " + tiempos.size() + ",espera promedio = " + promedio + "min");
    }
    System.out.println("Pacientes fuera de tiempo:" + pacietesFueraDeTiempo.size());
}
```

Figura 14: Código Java de `Clase SimuladorUrgencia` parte 4.

3.4. Pruebas

A continuación se realizan las pruebas necesarias para validar la simulación y verificar que el sistema funcione de la forma correcta.

Se solicitan y realizan las siguientes pruebas:

- **Seguimiento individual**
  Se selecciona a un paciente que pertenece a la categoría C4 (Baja complejidad), y se registra el tiempo exacto en que tarda ser atendido. Se logra verificar el

tiempo real que espera un paciente con esta categoría en ser atendido en el centro asistencial.

Por lo tanto, se selecciona el primer paciente generado de manera aleatoria de la categoría, en esta prueba fue el paciente P1007, el cual es atendido en 0 minutos, por lo tanto, fue atendido enseguida, esto se puede deber a que posiblemente había pocos pacientes al momento de su llegada.

En consecuencia, este resultado comprueba que el sistema no solo favorece a las categorías de mayor complejidad o urgencia, sino que, por el contrario, se atiende a los pacientes de otras categorías cuando existe la disponibilidad de su atención de salud.

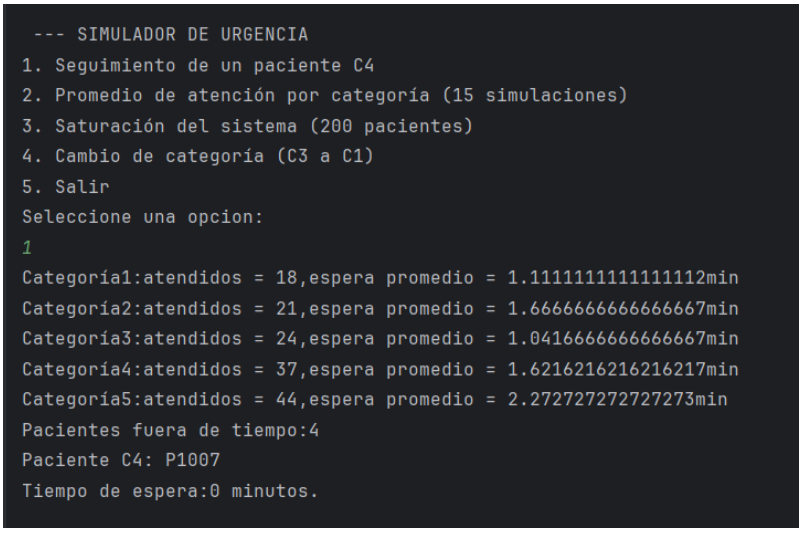

Figura 15: Captura de pantalla de opción 1 del menú, “Seguimiento de un paciente C4”, en la imagen se puede observar el paciente atendido id paciente (Paciente C4), categoría(P1007) y su tiempo de espera(0 minutos).

ya

- **Promedio por categoría**
  Se ejecuta la simulación una cantidad de 15 veces, y se calcula el tiempo promedio de atención para cada una de las categorías.

  En la prueba generada se evalúa si el sistema implementa una prioridad clínica en promedio. Se ejecutan 15 simulaciones diferentes, y se observa que por lo general que las categorías de mayor urgencia tienden a tener tiempos de esperas más bajos. La diferencia entre categorías no es tan notoria por la carga balanceada de pacientes y la capacidad de responder del sistema.

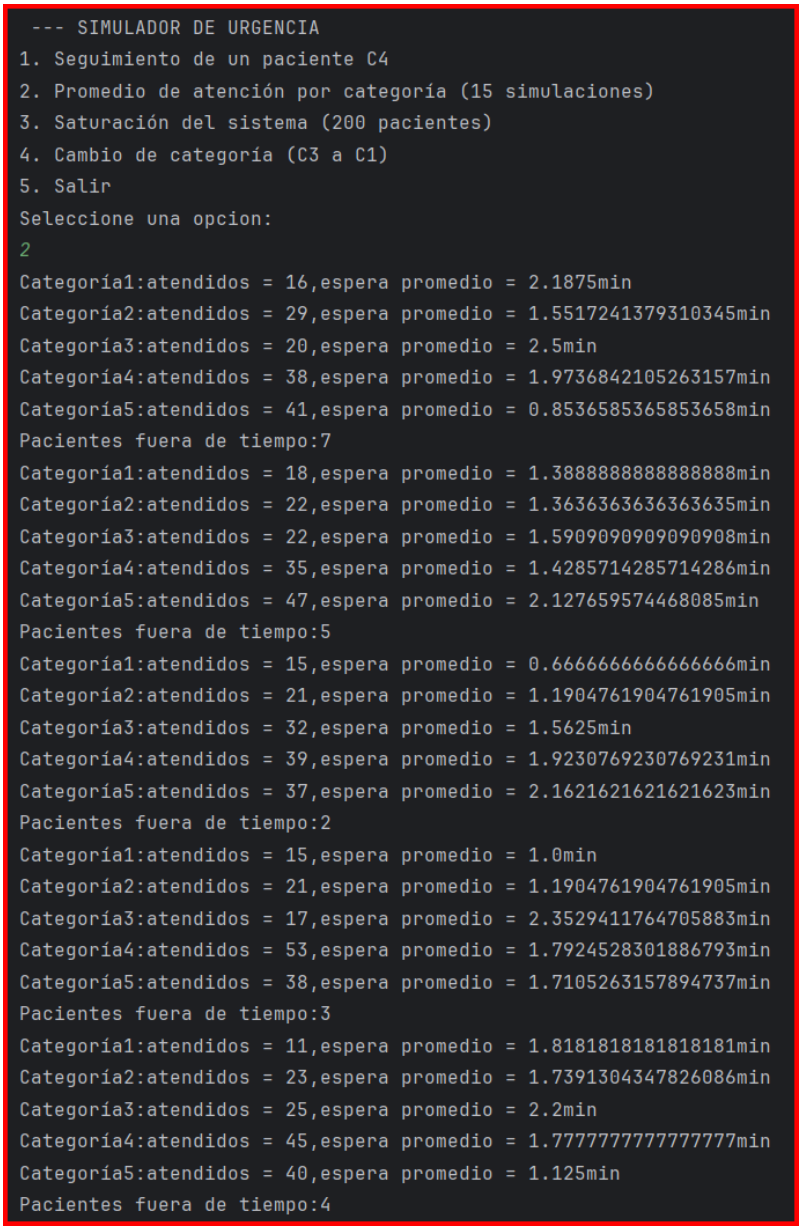

Figura 16: Captura de pantalla de opción 2, “Promedio de atención por categoría (15 simulaciones)”, simulaciones 1, 2, 3, 4 y 5.

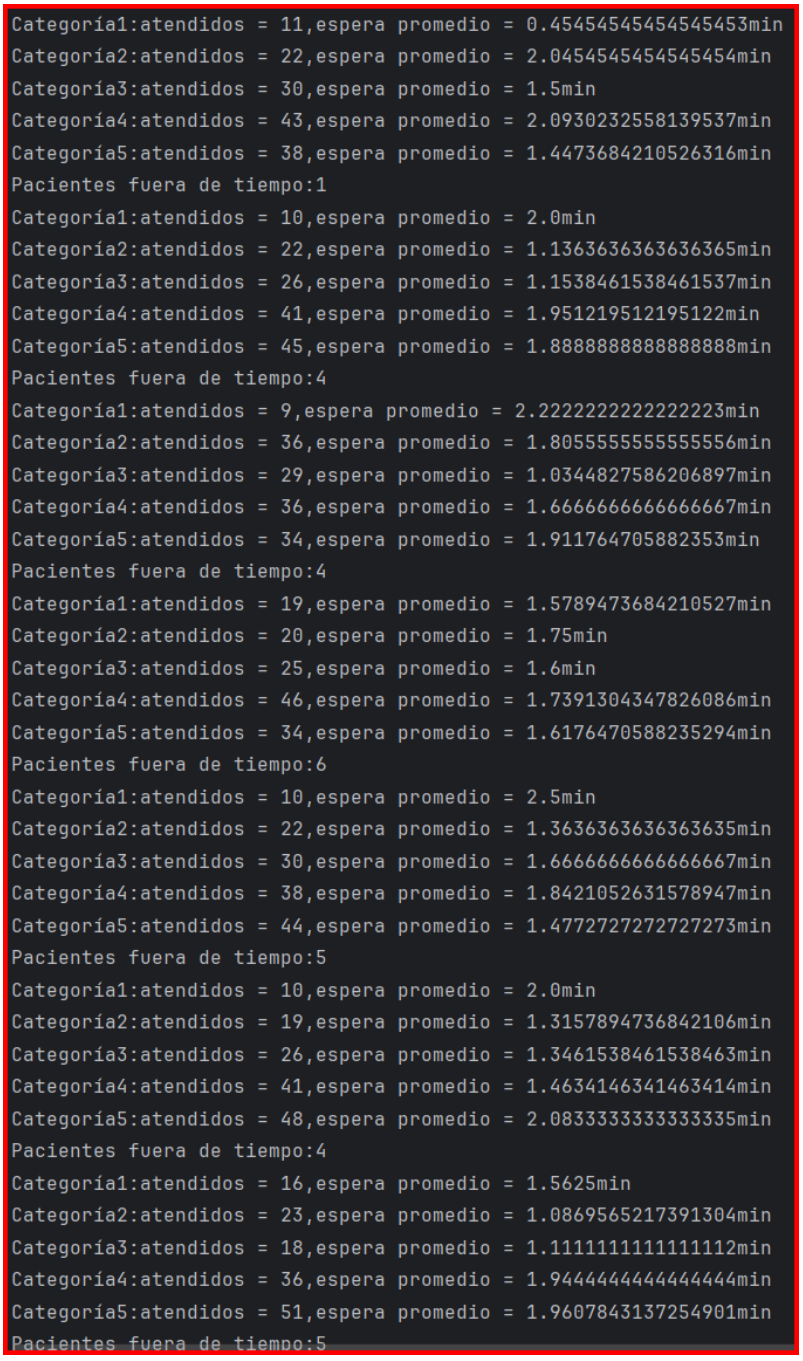

Figura 17: Captura de pantalla de opción 2, “Promedio de atención por categoría (15 simulaciones)”, simulaciones 6, 7, 8, 9, 10, 11 y 12.

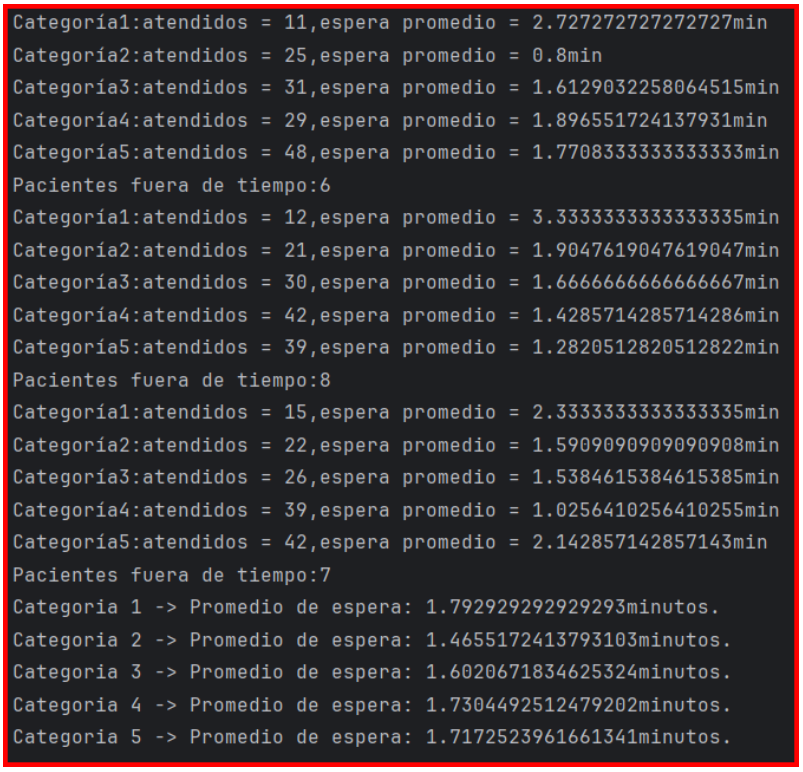

Figura 18: Captura de pantalla de opción 2, "Promedio de atención por categoría (15 simulaciones)", simulaciones 13, 14 y 15. Además se muestra el promedio que existe por categoría de todas las simulaciones.

Categoría	Promedio de espera (minutos)
C1	1.79
C2	1.47
C3	1.60
C4	1.73
C5	1.71

Tabla 2: Resultados de tiempos de espera asociados a su categoría. Tiempo promedio total de las 15 simulaciones.

- **Saturación del sistema**
  Se simula una jornada con una mayor cantidad de pacientes (200) y se registra qué categorías se ven más afectadas en el tiempo de demora. El objetivo de esta prueba es ver como se comporta el sistema ante una mayor cantidad de personas.

```
 --- SIMULADOR DE URGENCIA
1. Seguimiento de un paciente C4
2. Promedio de atención por categoría (15 simulaciones)
3. Saturación del sistema (200 pacientes)
4. Cambio de categoría (C3 a C1)
5. Salir
Seleccione una opcion:
3
Categoría1:atendidos = 21,espera promedio = 1.6666666666666667min
Categoría2:atendidos = 20,espera promedio = 1.75min
Categoría3:atendidos = 20,espera promedio = 1.25min
Categoría4:atendidos = 38,espera promedio = 2.236842105263158min
Categoría5:atendidos = 45,espera promedio = 1.3333333333333333min
Pacientes fuera de tiempo:7
Estadisticas Final:
Categoría1:atendidos = 21,promedio espera = 1.6666666666666667min, Peor espera = 5min
Categoría2:atendidos = 20,promedio espera = 1.75min, Peor espera = 5min
Categoría3:atendidos = 20,promedio espera = 1.25min, Peor espera = 5min
Categoría4:atendidos = 38,promedio espera = 2.236842105263158min, Peor espera = 5min
Categoría5:atendidos = 45,promedio espera = 1.3333333333333333min, Peor espera = 5min
Pacientes fuera de tiempo: 7
```

Figura 19: Captura de pantalla de opción 3, "Saturación del sistema (200 pacientes)", tiempos promedios de categorías y peor espera por categoría.

En la siguiente tabla se muestran los resultados obtenidos:

Categoría	Atendidos	Promedio espera(minutos)	Peor espera(minutos)
C1	21	1.67	5
C2	20	1.75	5
C3	20	1.25	5
C4	38	2.24	5
C5	45	1.33	5
Pacientes fuera de tiempo			7 pacientes

Tabla 3: Tabla que muestra los resultados encontrados, divididos por categoría, número de pacientes atendidos, promedio de espera en minutos, y tiempo de espera en el peor caso en minutos. Además, se presenta la cantidad de pacientes que se encuentran fuera de tiempo con un total de 7.

- **Cambio de categoría**
  Se simula el caso de un paciente que fue mal categorizado y se utiliza el historial para registrar la corrección de este cambio.

  En esta prueba se simula el caso de un paciente mal clasificado perteneciente a la categoría 3 y se cambia a la categoría 1. El sistema permite que se realice

el cambio con éxito, se puede ver en la Figura 20.

El cambio de categoría en la vida real ocurre debido a errores humanos o que el paciente se encuentra en una situación de mayor riesgo vital, ya que pudo haber empeorado; pero el sistema se adapta correctamente permitiendo que el paciente sea reasignado a la categoría a la cual verdaderamente pertenece.

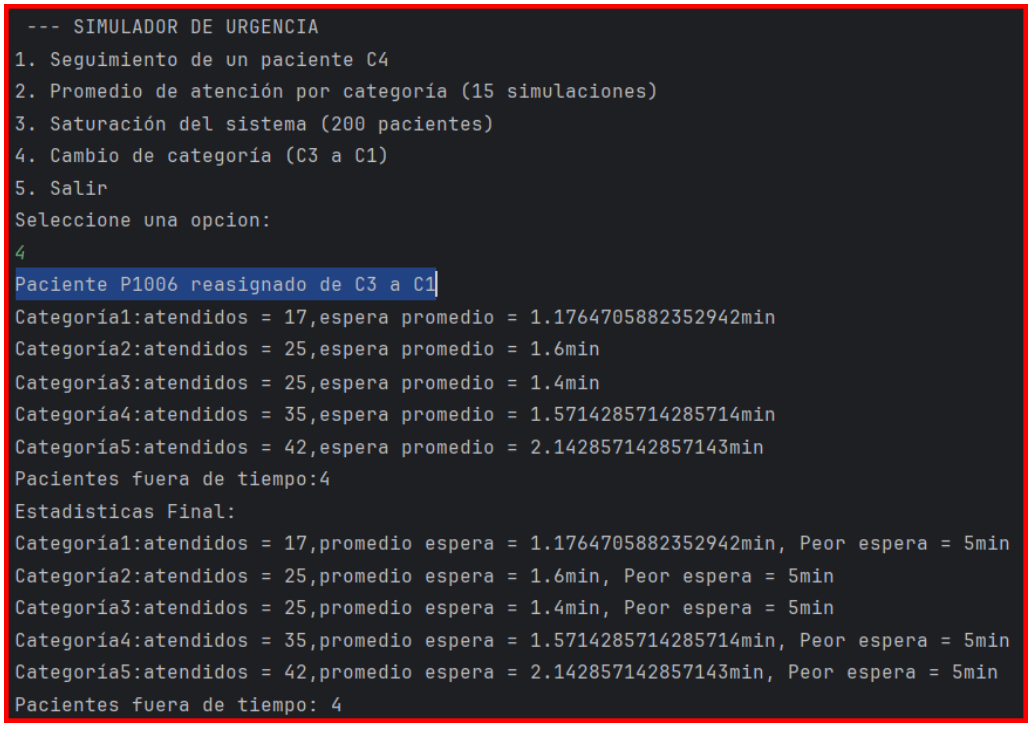

Figura 20: Captura de pantalla de opción 4, “Cambio de categoría (C3 a C1)”, se observa en pantalla que el cambio se realizó correctamente desde la categoría 3 a la 1 y que fue del paciente *P1006*.

4. Análisis

En esta sección se realiza el análisis del código propuesto, este se divide respondiendo las siguientes preguntas:

4.1. Análisis de Resultados Generales

- ¿Cuántos pacientes fueron atendidos en total al finalizar la simulación?
  En la prueba número 4 (Cambio de categoría (C3 a C1)), se suman los pacientes atendidos por categoría lo que da un total de 144 pacientes atendidos.
- ¿Cuántos pacientes por categoría (C1 a C5) fueron atendidos?
  En la siguiente tabla se muestran los resultados:

Categoría	N° de pacientes atendidos
C1	17
C2	25
C3	25
C4	35
C5	42
Total	144

Tabla 4: Tabla que asocia a total de pacientes atendidos por categoría, y al final la suma del total de todas las categorías.

- ¿Cuál fue el tiempo promedio de espera por categoría?
  En la siguiente tabla se puede apreciar el promedio por categoría:

Categoría	Promedio de tiempo de espera(minutos)
C1	1.17
C2	1.6
C3	1.4
C4	1.57
C5	2.14

Tabla 5: Tabla que asocia el tiempo total de espera de los pacientes, desde su llegada al centro asistencial, hasta su atención, estos se encuentran ordenados por categoría, y al final la suma del total de todas las categorías.

- ¿Cuantos pacientes excedieron el tiempo máximo de espera definido para su categoría? Según los resultados obtenidos, en la prueba número cuatro, el total de pacientes que encontraban fuera de tiempo fue la cantidad de: cuatro.

- ¿Cuáles fueron los peores tiempos de espera registrados por categoría? en la última prueba realizada se aprecia que en todas las categorías la peor espera en todas las categorías es de: 5 minutos.

4.2. Análisis Asintótico de Métodos Críticos

Los métodos, analizados a continuación, son métodos críticos del sistema y se examinan en notación *Big-O*.

Se logra apreciar que las operaciones más frecuentes tienen complejidad logarítmica, y estas se dedican al registro de pacientes, y la atención de estos, esto ocurre debido al uso de `PriorityQueue`. Por otro lado, que el ordenamiento de pacientes mediante *HeapSort*, tiene una complejidad O(n log n).

El análisis que se realiza en notación *Big-O*, se justifican a continuación uno por uno de ellos, siendo los siguientes métodos claves:

- `registrarPaciente` de la clase `Hospital`: este método inserta en `PriorityQueue` (cola general), esto es en tiempo logarítmico.
- `atenderSiguiente` de la clase `Hospital`: aquí se extrae del *heap*, esta operación es de tiempo logarítmico.
- `ingresarPaciente` de la clase `AreaAtencion`: inserta a paciente en *heap* de área, es de tiempo logarítmico.
- `ordenarPacientesPorHeapSort` de la clase `AreaAtencion`: este método realiza una copia, más un ordenamiento, por ende es de tiempo O(n log n).
- `heapSort` de la clase `HeapSortHospital`: en este método se realiza ordenamiento por heap, por ende es de tiempo O(n log n).
- `reasignarCategoria` de la clase `Hospital`: este método realiza un acceso directo al mapa y realiza una modificación simple, por ende es de tiempo O(1).

A continuación se presenta una tabla resumen:

Método	Clase	Complejidad
`registrarPaciente`	`Hospital`	O(log n)
`atenderSiguiente`	`Hospital`	O(log n)
`ingresarPaciente`	`AreaAtencion`	O(log n)
`ordenarPacientesPorHeapSort`	`AreaAtencion`	O(n log n)
`heapSort`	`HeapSortHospital`	O(n log n)
`reasignarCategoria`	`Hospital`	O(1)

Tabla 6: Tabla que asocia el método, con su clase y su complejidad correspondiente. En formato *Big-O*.

4.3. Decisiones de Diseño

En esta parte del informe se describen las principales decisiones tomadas para administrar pacientes, salas y prioridades, para ello se responden las siguientes preguntas:

- ¿Qué estructuras se utilizan y por qué?

  Se utilizan las siguientes estructuras:

  - **PriorityQueue** para colas de espera global y por área: garantiza inserción y extracción en O(log n) según urgencia y tiempo de llegada.
  - **Stack** en Paciente.historialCambios[1]: registro LIFO de eventos de reclasificación.
  - **Mapas (HashMap)** para acceso rápido a pacientes por ID y a áreas por nombre.

- ¿Cómo se modeló la cola central de atención?

  La cola central es modelada con un **heap** que prioriza primero la categoría (C1–C5) y, en empate, al paciente con mayor tiempo de espera.

- ¿Cómo se administró el uso de montones para áreas de atención?

  Cada área **(SAPU, urgencia adulto, urgencia infantil)** mantiene su propia cola de prioridad, lo que facilita análisis de carga y balanceo independiente.

- ¿Qué ventajas se observan en el enfoque que se dio?

  Se observan principalmente:

  - Latencia de extracción controlada (O(log n)), incluso con gran volumen de pacientes.
  - Fácil extensión: agregar nuevos criterios de prioridad o cambiar tiempos máximos sin reformular la lógica del *heap*.

- ¿Se detectó alguna limitación o ineficiencia?

  La copia completa del *heap* para ordenar (HeapSort) implica O(n log n) adicional, lo que es costoso si se invoca frecuentemente.

  La medición de tiempos usando reloj real (System.currentTimeMillis) dificulta la reproducibilidad de la simulación.

4.4. Ventajas y Desventajas del Sistema

A continuación se mencionan ventajas y desventajas que se apreciaron en la implementación.

- Ventajas

  1. **Eficiencia algorítmica:** operaciones críticas en O(log n) permiten escalar a cientos de pacientes sin cuellos de botella.
  2. **Flexibilidad de prioridades:** fácil ajustar reglas (ej., incorporar nuevos umbrales o áreas) sin reescribir estructuras de datos.

- Desventajas

  1. **Reproducibilidad reducida:** mezcla de tiempo de simulación (minuto a minuto) y reloj real puede sesgar métricas de espera.
  2. **Consumo adicional:** la función de ordenamiento completo del heap (obtenerPacientesPorHeapSort) duplica datos y aumenta uso de memoria y CPU.

4.5. Desafíos Encontrados

En esta presente área del trabajo se describen dos desafíos que se generó y se enfrentó durante la implementación y la simulación.

1. **Sincronización de tiempo de simulación:** mantener un contador de minutos independiente evitó inconsistencias, pero requirió redefinir cálculos de espera.

2. **Gestión de pacientes fuera de tiempo:** diseñar la lógica para registrar solo cuando un paciente excede su umbral evitó duplicaciones en la lista de "fuera de tiempo".

4.6. Extensión del Sistema: Turnos Médicos

Para acercar la simulación a la realidad, se podría modelar un turno rotativo de personal usando una cola circular o lista enlazada que asigne médicos a franjas horarias. Al finalizar cada turno, el médico se reencola al final y el siguiente toma el relevo, permitiendo medir carga de trabajo y tiempo de atención por profesional.

4.7. Solución del Sistema por pacientes no atendidos

Para solucionar el problema de por pacientes no atendidos se propone lo siguiente:

Al cerrar la jornada, los pacientes aún en cola se transfieren a la siguiente simulación con prioridad incrementada. Por tanto, aquellos pacientes son atendidos al comienzo de la siguiente jornada.

Alternativamente, se escala dinámicamente la capacidad máxima de cada área en función de la demanda acumulada, para reducir *backlog* y garantizar que menos pacientes queden sin atención. Este *backlog* se trata de una señal de alerta que muestra cuántas personas quedaron esperando, indicando que el hospital necesita adaptarse o mejorar para cubrir la demanda real.

5. Conclusión

Durante el desarrollo del laboratorio se logró aplicar de forma práctica conceptos importantes de estructuras de datos como lo son las colas de prioridad, montones, pilas y mapas. Se realizó bajo el contexto de simulación de la vida real, este es el de la atención de pacientes en el área hospitalaria de urgencias.

Se implementó una simulación de una jornada de 24 horas, se puede apreciar que las decisiones algorítmicas influyen en los tiempos de atención, asignación de recursos y cumplir con la categorización correcta de pacientes.

Se observó la eficiencia de usar `PriorityQueue` para ordenar pacientes por urgencias y el tiempo de espera de los pacientes. También los beneficios de uso de `Stack`, para poder buscar cambios en el historial clínico.

Finalmente, en este trabajo se demuestra que el uso de estructura de datos no son solo herramientas técnicas, sino que son necesarios para el diseño de sistemas eficientes y sostenibles. Como una mejora futura, se propone extender el sistema con gestión de turnos médicos, de esta forma se podría equiparar la carga de atención y hacer la simulación más similar a la vida cotidiana.

Referencias

[1] *Repositorio en Github*. 2025. URL: https://github.com/Cinthya982012/Informe_Lab4_S2_EDA.git.