Universidad Diego Portales
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES

ESTRUCTURAS DE DATOS &
ANÁLISIS DE ALGORITMOS

Laboratorio 4: Unidades de urgencia hospitalaria y colas de prioridad.

Autores:
Valentina Martínez
José Pablo Peña

Profesor:
Marcos Fantoval

11 de junio de 2025

Índice

1. Introducción

Este laboratorio tiene como objetivo aplicar estructuras de datos en un contexto hospitalario, una situación realista y dinámica que requiere tanto análisis algorítmico como evaluaciones empíricas del sistema desarrollado.

Para llevar esto a cabo se utilizará el sistema de categorización oficial del Ministerio de Salud de Chile, que clasifica a los pacientes en cinco niveles de atención, desde C1 hasta C5, según la gravedad de su condición y el tiempo máximo de respuesta esperado. C1 corresponde a atención inmediata, mientras que C5 se refiere a atención general.

La implementación de este sistema se realizará utilizando estructuras de datos, como pilas (stack), colas de prioridad (priority queue), mapas (map), entre otras, además de métodos de ordenamiento como HeapSort. Todo esto es para desarrollar clases que simulen el funcionamiento de un sistema de urgencias hospitalarias. Este debe permitir la gestión de pacientes según su nivel de prioridad, el registro y modificación de atenciones, la asignación de áreas de atención, y la simulación en tiempo real de ingresos y egresos de pacientes.

2. Implementación

Para llevar a cabo este laboratorio se hizo uso del lenguaje de programación Java, el cual al ser orientado a objeto, nos permite crear distintas clases. El código completo se encuentra en: https://github.com/valemiauz/Lab-4

2.1. Clase Paciente

La primera clase es llamada Paciente, que representa a una persona que requiere atención médica, para esto se le asignaron variados atributos tales como nombre, aprellido, id (RUN o pasaporte), categoría (nivel de urgencia asginada aleatoriamente), tiempo de llegada, estado (en espera, en atención o atendido), área (SAPU, urgencia adulto e infantil) y por último una pila llamada historial cambios, que registra cada cambio relevante en la atención o categorización del paciente.

2.1.1. Métodos

- tiempoEsperaActual: calcula el tiempo de espera del paciente, desde su llegada hasta el momento actual en minutos. Tiene complejidad O(1).

```
1      public long tiempoEsperaActual(){
2          long tiempoActual = System.currentTimeMillis() / 1000;
3          long diferencia = tiempoActual-this.tiempoLlegada;
4          return diferencia / 60;
5      }
```

- registrarCambio(String descripcion): registra cambios en el historial hospitalario del paciente, para ello se agrega un elemento String a la pila de historial del paciente. Tiene complejidad O(1).

```
    public void registrarCambio(String descripcion){
        historialCambios.push(descripcion);
    }
```

- obtenerUltimoCambio: Método que retorna el último cambio registrado por el paciente y lo remueve de la pila del historial hospitalario. Para ello, se utiliza el método pop() de la clase Stack, que permite eliminar y retornar el último elemento agregado. Tiene complejidad O(1).

```
     public String obtenerUltimoCambio(){
        if(historialCambios.empty()){
            return "No hay cambios.";
        } else {
            return historialCambios.pop();
        }
    }
```

2.2. Clase Área atención

Esta clase representa una zona del hospital, siendo estas SAPU, urgencia adulto e infantil. Cuenta con atributos como su nombre y su capacidad máxima de pacientes. También, contará con una cola de prioridad de pacientes, los cuales mantendrá ordenados por nivel de urgencia y su tiempo de espera.

2.2.1. Métodos

- ingresarPaciente(Paciente p): método que ingresa al paciente a la cola de prioridad. Tiene complejidad O(log n).

```
     public void ingresarPaciente (Paciente p){
        pacientesHeap.add(p);
    }
```

- Paciente atenderPaciente: método encargado de atender un paciente sacándolo de la cola de prioridad. Tiene complejidad O(log n).

```
    public Paciente atenderPaciente(){
        return pacientesHeap.poll();
    }
```

- estaSaturada: método que revisa si el área alcanzó la capacidad máxima, este retornara true si es así y false en el caso contrario. Tiene complejidad O(1).

```
     public boolean estaSaturada() {
        return pacientesHeap.size() >= capacidadMaxima;   }
```

- obtenerPacientesPorHeapSort: método que devuelve una lista de pacientes en orden de prioridad usando Heapsort. Para esto crea una copia de la cola pacientesHeap. Luego se inicializa una lista llamada ordenados que almacena los pacientes en orden. Después se recorre la copia con un while que se ejecuta mientras la cola no esté vacía, luego de cada iteración se extrae el paciente con mayor prioridad usando poll() y se agrega a la lista ordenados, después de que vaciada la cola, se retorna ordenados conteniendo los pacientes ordenados según prioridad. Tiene complejidad O(n log n).

```
1     public List<Paciente> obtenerPacientesPorHeapSort() {
2         PriorityQueue<Paciente> copia = new
              PriorityQueue<>(pacientesHeap);
3         List<Paciente> ordenados = new ArrayList<>();
4         while (!copia.isEmpty()) {
5             ordenados.add(copia.poll());
6         }
7         return ordenados;
8     }
```

2.3. Clase Hospital

Esta clase administra el ingreso y atención de pacientes en todo el hospital. Para realizar esto se tienen los siguientes atributos; un mapa de pacientes que asocia el id de cada paciente al hospital, para así tener un acceso rápido a la información de este, una cola de atención que gestiona a los pacientes que ingresen al hospital, o sea, los pacientes en espera, un mapa de áreas de atención para asignarle a cada paciente, una lista de atendidos para tener registro de cada paciente que se haya atendido en el hospital y por último un método que sirve para agregar nuevas áreas de atención al hospital.

2.3.1. Métodos

- registrarPaciente(Paciente p): registra a un paciente en el sistema hospitalario. Primero, almacena al paciente en el mapa pacientesTotales utilizando su ID como clave, lo que permite acceder fácilmente a sus datos. También lo agrega a la cola de atención (colaAtencion). Finalmente, se registra un mensaje en el historial del paciente mediante el método registrarCambio, indicando su categoría y el área asignada. Tiene complejidad O(log n).

```
1     public void registrarPaciente(Paciente p) {
2         pacientesTotales.put(p.getId(), p);
3         colaAtencion.add(p);
4         p.registrarCambio("Paciente registrado - Categoria: "
              + p.getCategoria() + ", Area: " + p.getArea());
5     }
```

- reasignarCategoria(String id, int nuevacategoria): este método tiene como objetivo cambiar la categoría de un paciente según su ID. Primero, busca al paciente, si lo encuentra, guarda su categoría anterior, le asigna la nueva, y deja un registro del cambio. Luego, lo saca de la cola de atención y lo vuelve a meter, para que quede ubicado según su nueva categoría y pueda ser atendido en el orden adecuado. Tiene complejidad O(n).

```
1     public void reasignarCategoria(String id, int
          nuevaCategoria) {
2         Paciente paciente = pacientesTotales.get(id);
3
4         if (paciente != null) {
5             int categoriaAnterior = paciente.getCategoria();
6             paciente.setCategoria(nuevaCategoria);
7             paciente.registrarCambio("Categoria cambiada de "
                  + categoriaAnterior + " a " + nuevaCategoria);
8
9             colaAtencion.remove(paciente);
10            colaAtencion.add(paciente);
11        }
12    }
```

- paciente atenderSiguiente: se atiene al próximo paciente en la cola de atención. Primero, saca al paciente que está primero en la cola y busca el área correspondiente según lo que tenga el paciente. Si el área también existe, lo añade. Después actualiza el estado del paciente a .En atención", registra ese cambio en su historial,luego lo agrega a la lista de pacientes que ya fueron atendidos. Al final, devuelve ese paciente. Tiene complejidad O(log n).

```
1         public Paciente atenderSiguiente() {
2         Paciente paciente = colaAtencion.poll();
3
4         if (paciente != null) {
5             AreaAtencion area =
                  areasAtencion.get(paciente.getArea());
6
7             if (area != null) {
8                 area.ingresarPaciente(paciente);
9             }
10
11            paciente.setEstado("En atencion");
12            paciente.registrarCambio("Paciente asignado al
                  area: " + paciente.getArea());
13            pacientesAtendidos.add(paciente);
14        }
15
16        return paciente;
17    }
```

- obetenerPacientesPorCategoria(int categoria): crea una lista vacía y luego re-

corre todos los pacientes que están en la cola de atención y revisa si su categoría coincide con la que se pasó como parámetro. Si es así, lo agrega a la lista. Luego devuelve la lista con los pacientes que tienen esa categoría. Tiene complejidad O(n).

```
        public List<Paciente> obtenerPacientesPorCategoria(int
            categoria) {
        List<Paciente> pacientesPorCategoria = new
            ArrayList<>();

        for (Paciente paciente : colaAtencion) {
            if (paciente.getCategoria() == categoria) {
                pacientesPorCategoria.add(paciente);
            }
        }

        return pacientesPorCategoria;
    }
```

- AreaAtencion obtenerArea(String nombre): este método busca en el mapa de áreas de atención un área que tenga el nombre entregado y la devuelve. Si no existe devuelve null. Tiene complejidad O(1).

```
    public AreaAtencion obtenerArea(String nombre) {
        return areasAtencion.get(nombre);
    }
```

- agregarArea(String nombre, int capacidadMaxima): este método sirve para agregar nuevas áreas de atención al hospital, sin embargo en este caso no fue necesario usarla ya que solo existen 3 áreas: SAPU, urgencia adulto e infantil. Tiene complejidad O(1).

```
            public void agregarArea(String nombre, int
                capacidadMaxima) {
        AreaAtencion area = new AreaAtencion(nombre,
            capacidadMaxima);
        areasAtencion.put(nombre, area);
    }
```

2.4. Clase GeneradorPacientes

La experimentación se basa en la simulación del funcionamiento de una sala de urgencias hospitalaria durante un período de 24 horas, considerando tanto la llegada de pacientes como su atención priorizada. Para esto, se implementará una clase llamada GeneradorPacientes, la cual se encargará de generar una lista aleatoria de pacientes (`List<Paciente>`) de tamaño N, con atributos asignados aleatoriamente pero con situaciones reales de un entorno hospitalario. Esta generación de datos permite simular el registro de pacientes, su atención según políticas de prioridad médica, y observar el comportamiento del sistema bajo distintas cargas operativas.

2.4.1. Métodos

- generarIdUnico(): Crea un ID único a partir de un número de 4 dígitos. Tiene complejidad O(1).

```
        private String generarIdUnico() {
            return String.format("PAC%04d", contadorId++);
        }
```

- generarCategoria(): Genera las categorías pedidas anteriormente (de 1 a 5). Tiene complejidad O(1).

```
        private int generarCategoria() {
        double probabilidad = random.nextDouble() * 100;

        if (probabilidad < 10) {
            return 1;
        } else if (probabilidad < 25) {
            return 2;
        } else if (probabilidad < 43) {
            return 3;
        } else if (probabilidad < 70) {
            return 4;
        } else {
            return 5;
        }
        }
```

- guardarPacientesEnArchivo(): Este método guarda en una lista de pacientes en un archivo txt, incluyendo un resumen con la cantidad y porcentaje por categoría. Su complejidad es O(n).

```
1            public void
                 guardarPacientesEnArchivo(List<Paciente>
                 pacientes, String nombreArchivo) {
2        try (PrintWriter writer = new PrintWriter(new
             FileWriter(nombreArchivo))) {
3            writer.println("=== PACIENTES GENERADOS PARA
                 SIMULACION 24H ===");
4            writer.println("Total de pacientes: " +
                 pacientes.size());
5            writer.println("Formato: ID | Nombre | Apellido |
                 Categoria | Tiempo Llegada | Estado | Area");
6            writer.println("============");
7            for (Paciente paciente : pacientes) {
8                writer.printf("%s | %s | %s | C%d | %d | %s |
                     %s%n",
9                        paciente.getId(),
10                       paciente.getNombre(),
11                       paciente.getApellido(),
12                       paciente.getCategoria(),
13                       paciente.getTiempoLlegada(),
14                       paciente.getEstado(),
15                       paciente.getArea()
16               );
17           }
18           writer.println("============");
19           writer.println("Distribucion por categorias:");
20           Map<Integer, Integer> contadorCategorias = new
                 HashMap<>();
21           for (Paciente p : pacientes) {
22           contadorCategorias.put(p.getCategoria(),
23           contadorCategorias.getOrDefault(p.getCategoria(),
                 0) + 1);
24           }
25           for (int i = 1; i <= 5; i++) {
26               int cantidad =
                     contadorCategorias.getOrDefault(i, 0);
27               double porcentaje = (cantidad * 100.0) /
                     pacientes.size();
28               writer.printf("C%d: %d pacientes (%.1f%%)%n",
                     i, cantidad, porcentaje);
29           }
30           System.out.println("Archivo '" + nombreArchivo +
                 "' creado exitosamente.");
31       } catch (IOException e) {
32           System.err.println("Error al escribir el archivo:
                 " + e.getMessage());
33       }
34   }
```

- main(String[] args): El main simula la generación de 144 pacientes en un periodo de 24 horas con un paciente cada 10 minutos. Tiene complejidad O(n).

```
        static void main(String[] args) {
        GeneradorPacientes generador = new
            GeneradorPacientes();
        long timestampInicio = System.currentTimeMillis() /
            1000;
        int pacientesPor24h = 144;
        System.out.println("=== GENERADOR DE PACIENTES
            HOSPITALARIOS ===");
        System.out.println("Generando " + pacientesPor24h + "
            pacientes para simulacion de 24 horas...");
        System.out.println("Intervalo: 1 paciente cada 10
            minutos");
        List<Paciente> pacientes =
            generador.generarPacientes(pacientesPor24h,
            timestampInicio);
        System.out.println("    " + pacientes.size() + "
            pacientes generados exitosamente");
        generador.guardarPacientesEnArchivo(pacientes,
            "Pacientes_24h.txt");
        System.out.println("\n=== ESTADISTICAS DE GENERACION
            ===");
        Map<Integer, Integer> stats = new HashMap<>();
        for (Paciente p : pacientes) {
            stats.put(p.getCategoria(),
                stats.getOrDefault(p.getCategoria(), 0) + 1);}
        System.out.println("Distribucion por categorias:");
        for (int i = 1; i <= 5; i++) {
            int cantidad = stats.getOrDefault(i, 0);
            double porcentaje = (cantidad * 100.0) /
                pacientes.size();
            String nombreCategoria = obtenerNombreCategoria(i);
            System.out.printf("C%d (%s): %d pacientes
                (%.1f%%)%n",
                    i, nombreCategoria, cantidad, porcentaje);
        }
        System.out.println("\n=== PRIMEROS 5 PACIENTES
            GENERADOS ===");
        for (int i = 0; i < Math.min(5, pacientes.size());
            i++) {
            Paciente p = pacientes.get(i);
            System.out.printf("%d. %s %s (ID: %s, C%d, Area:
                %s)%n",
                    i + 1, p.getNombre(), p.getApellido(),
                        p.getId(),
                    p.getCategoria(), p.getArea());
        }
        System.out.println("\ n   Generacion completada.
            Revisar archivo 'Pacientes_24h.txt'");
    }
```

2.5. Clase SimuladorUrgencia

Esta clase simula una urgencia hospitalaria durante un día completo (24 horas, 1440 minutos). Administra la llegada y atención de pacientes según su categoría de urgencia (C1 a C5), registra estadísticas como tiempos de espera, pacientes que superaron el tiempo máximo permitido, y cantidad de pacientes atendidos por categoría. Utiliza objetos de otras clases (Hospital, Paciente, GeneradorPacientes) y áreas como SAPU, Urgencia adulto e Infantil.

2.5.1. Métodos

- SimuladorUrgencia(): Inicializa el hospital y configura las áreas de atención. Complejidad O(1).

```
1         public SimuladorUrgencia() {
2         this.hospital = new Hospital();
3         this.generador = new GeneradorPacientes();
4         this.indicePacienteActual = 0;
5         this.contadorPacientesAcumulados = 0;
6         this.tiempoSimulacion = 0;
7
8         this.tiemposEsperaPorCategoria = new HashMap<>();
9         this.pacientesAtendidosPorCategoria = new HashMap<>();
10        this.pacientesExcedidos = new ArrayList<>();
11        this.tiemposAtencionPorPaciente = new HashMap<>();
12
13        for (int i = 1; i <= 5; i++) {
14            tiemposEsperaPorCategoria.put(i, new
                  ArrayList<>());
15            pacientesAtendidosPorCategoria.put(i, 0);
16        }
17
18        hospital.agregarArea("SAPU", 10);
19        hospital.agregarArea("Urgencia_adulto", 15);
20        hospital.agregarArea("Infantil", 8);
21    }
```

- simular (int pacientesPorDia): Este método corre la simulación por 1440 minutos (24 horas). Cada 10 minutos llega un nuevo paciente y cada 15 minutos se atiende uno. Si hay 3 o más pacientes acumulados, se atienden 2 extra. Al final se atienden los restantes. Complejidad O(n+m) donde n son los minutos y m los pacientes por día.

```
1        public void simular(int pacientesPorDia) {
2            System.out.println("INICIANDO SIMULACION DE
                 URGENCIA HOSPITALARIA");
3            System.out.println("DuraciOn: 24 horas (1440
                 minutos)");
4            System.out.println("Pacientes esperados: " +
                 pacientesPorDia);
5            System.out.println("==================\n");
6
7            long timestampInicio = System.currentTimeMillis()
                 / 1000;
8            pacientesGenerados =
                 generador.generarPacientes(pacientesPorDia,
                 timestampInicio);
9
10           for (tiempoSimulacion = 0; tiempoSimulacion <
                 1440; tiempoSimulacion++) {
11
12               if (tiempoSimulacion % 10 == 0 &&
                     indicePacienteActual <
                     pacientesGenerados.size()) {
13                   llegadaNuevoPaciente();
14               }
15
16               if (tiempoSimulacion % 15 == 0) {
17                   atenderPacienteRegular();
18               }
19
20               verificarPacientesExcedidos();
21
22               if (contadorPacientesAcumulados >= 3) {
23                   atenderPacientesAcumulados();
24                   contadorPacientesAcumulados = 0;
25               }
26
27               if (tiempoSimulacion % 60 == 0 &&
                     tiempoSimulacion > 0) {
28                   mostrarProgresoHorario();
29               }
30           }
31
32           procesarPacientesRestantes();
33
34           mostrarResultadosFinales();
35       }
```

- llegadaNuevoPaciente():Agrega un nuevo paciente al hospital cada 10 minutos, anota su tiempo de llegada, lo registra en el hospital y actualiza contadores. Tiene complejidad O(1).

```
        private void llegadaNuevoPaciente() {
        if (indicePacienteActual < pacientesGenerados.size()) {
            Paciente paciente =
               pacientesGenerados.get(indicePacienteActual);

            long timestampActual = System.currentTimeMillis()
               / 1000 + (tiempoSimulacion * 60);
            paciente.setTiempoLlegada(timestampActual);

            hospital.registrarPaciente(paciente);
            contadorPacientesAcumulados++;
            indicePacienteActual++;

            System.out.printf("[Min %d] Llegada: %s %s (ID:
               %s, C%d, Area: %s)\n",
                  tiempoSimulacion, paciente.getNombre(),
                     paciente.getApellido(),
                  paciente.getId(), paciente.getCategoria(),
                     paciente.getArea());
        }
    }
```

- atenderPacienteRegular(): Cada 15 minutos atiende un paciente. Extrae el siguiente paciente del hospital y procesa su atención. Tiene complejidad O(log n).

```
        private void atenderPacienteRegular() {
            Paciente paciente = hospital.atenderSiguiente();
            if (paciente != null) {
                procesarAtencionPaciente(paciente, "Atencion
                   regular");
            }
        }
```

- atenderPacientesAcumulados(): Cuando hay 3 o más pacientes acumulados, atiende 2 de inmediato. Complejidad O(log n).

```
        private void atenderPacientesAcumulados() {
        System.out.printf("[Min %d] ATENCION MASIVA:
            Atendiendo 2 pacientes por acumulacion\n",
            tiempoSimulacion);

        for (int i = 0; i < 2; i++) {
            Paciente paciente = hospital.atenderSiguiente();
            if (paciente != null) {
                procesarAtencionPaciente(paciente, "Atencion
                    por acumulacion");
            }
        }
    }
```

- procesarAtencionPaciente(Paciente paciente, String tipoAtencion): Registra estadísticas al atender un paciente. Calcula su tiempo de espera, actualiza contadores, listas y muestra el mensaje, y finalmente verifica si se excedió el tiempo máximo. Complejidad O(1).

```
    private void procesarAtencionPaciente(Paciente paciente,
        String tipoAtencion) {
        long tiempoEspera = calcularTiempoEspera(paciente);
        tiemposEsperaPorCategoria.get(paciente.getCategoria()).
        add(tiempoEspera);
        pacientesAtendidosPorCategoria
        .put(paciente.getCategoria(),
        pacientesAtendidosPorCategoria
        .get(paciente.getCategoria()) + 1);
        tiemposAtencionPorPaciente.put(paciente.getId(),
            tiempoEspera);
        int tiempoMaximo =
            TIEMPOS_MAXIMOS.get(paciente.getCategoria());
        if (tiempoEspera > tiempoMaximo) {
            pacientesExcedidos.add(paciente);
            paciente.registrarCambio("EXCEDIO tiempo maximo: "
                + tiempoEspera + " min (max: " + tiempoMaximo +
                " min)");
        }
        paciente.setEstado("Atendido");
        paciente.registrarCambio("Atendido despues de " +
            tiempoEspera + " minutos de espera");
        System.out.printf("[Min %d] ATENDIDO: %s %s (C%d) -
            Espera: %d min (%s)\n",
                tiempoSimulacion, paciente.getNombre(),
                    paciente.getApellido(),
                paciente.getCategoria(), tiempoEspera,
                    tipoAtencion);
    }
```

- calcularTiempoEspera(Paciente paciente): Calcula cuántos minutos han pasado desde que el paciente llegó hasta que fue atendido. O(1).

```
1        private long calcularTiempoEspera(Paciente paciente) {
2            long tiempoActual = System.currentTimeMillis() /
                 1000 + (tiempoSimulacion * 60);
3            return (tiempoActual -
                 paciente.getTiempoLlegada()) / 60;
4        }
```

- procesarPacientesRestantes(): Al final de la simulación, atiende a todos los pacientes que siguen esperando. Complejidad O(r log n) donde r es el número restante de pacientes.

```
1        private void procesarPacientesRestantes() {
2            System.out.println("\n=== PROCESANDO PACIENTES
                 RESTANTES ===");
3
4            Paciente paciente;
5            while ((paciente = hospital.atenderSiguiente()) !=
                 null) {
6                procesarAtencionPaciente(paciente,
                     "Procesamiento final");
7            }
8        }
```

- mostrarProgresoHorario(): Cada hora imprime el número de pacientes atendidos y los que excedieron el tiempo. Complejidad: O(1).

```
1        private void mostrarProgresoHorario() {
2            int hora = (int) (tiempoSimulacion / 60);
3            int totalAtendidos =
                 pacientesAtendidosPorCategoria.values().stream()
4                     .mapToInt(Integer::intValue).sum();
5
6            System.out.printf("\n--- PROGRESO HORA %d ---\n",
                 hora);
7            System.out.printf("Pacientes atendidos hasta
                 ahora: %d\n", totalAtendidos);
8            System.out.printf("Pacientes que excedieron
                 tiempo: %d\n", pacientesExcedidos.size());
9        }
```

- mostrarResultadosFinales(): Imprime el total de pacientes; el tiempo mínimo, máximo y promedio de espera por categoría; la lista de pacientes que se atendieron fuera de tiempo. Complejidad: O(c) donde c es la cantidad total de categorías.

```
        private void mostrarResultadosFinales() {
        System.out.println("\n" + "=".repeat(60));
        System.out.println("              RESULTADOS FINALES
            DE SIMULACION");
        System.out.println("=".repeat(60));
        int totalAtendidos =
            pacientesAtendidosPorCategoria.values().stream()
                 .mapToInt(Integer::intValue).sum();
        System.out.println("RESUMEN GENERAL:");
        System.out.println("- Total pacientes generados: " +
            pacientesGenerados.size());
        System.out.println("- Total pacientes atendidos: " +
            totalAtendidos);
        System.out.println("- Pacientes que excedieron tiempo
            maximo: " + pacientesExcedidos.size());
        System.out.println("\nATENCION POR CATEGORIA:");
        for (int categoria = 1; categoria <= 5; categoria++) {
            int atendidos =
                pacientesAtendidosPorCategoria.get(categoria);
            List<Long> tiempos =
                tiemposEsperaPorCategoria.get(categoria);
            if (!tiempos.isEmpty()) {
                double promedio =
                    tiempos.stream().mapToLong(Long::longValue)
                .average().orElse(0.0);
                long maximo =
                    tiempos.stream().mapToLong(Long::longValue)
                .max().orElse(0);
                long minimo =
                    tiempos.stream().mapToLong(Long::longValue)
                .min().orElse(0);
                System.out.printf("C%d (%s):\n", categoria,
                    obtenerNombreCategoria(categoria));
                System.out.printf("  - Atendidos: %d
                    pacientes\n", atendidos);
                System.out.printf("  - Tiempo promedio: %.1f
                    minutos\n", promedio);
                System.out.printf("  - Tiempo minimo: %d
                    minutos\n", minimo);
                System.out.printf("  - Tiempo maximo: %d
                    minutos\n", maximo);
                System.out.printf("  - Tiempo maximo
                    permitido: %d minutos\n",
                    TIEMPOS_MAXIMOS.get(categoria));
            }
        }
        if (!pacientesExcedidos.isEmpty()) {
```

```
31              System.out.println("\nPACIENTES QUE EXCEDIERON
                    TIEMPO MAXIMO:");
32              for (Paciente p : pacientesExcedidos) {
33                  long tiempoEspera =
                        tiemposAtencionPorPaciente.get(p.getId());
34                  System.out.printf("- %s %s (C%d): %d min (max:
                        %d min)\n",
35                      p.getNombre(), p.getApellido(),
                            p.getCategoria(),
36                      tiempoEspera,
                            TIEMPOS_MAXIMOS.get(p.getCategoria()));
37              }
38          }
39      }
```

- Seguimiento paciente C4(): Muestra el tiempo de espera de un paciente categoría 4 y si fue atendido a tiempo o no. Complejidad O(m).

```
        public void seguimientoPacienteC4() {
        System.out.println("\n=== SEGUIMIENTO INDIVIDUAL PACIENTE C4 ===");

        Paciente pacienteC4 = null;
        for (Paciente p : pacientesGenerados) {
            if (p.getCategoria() == 4) {
                pacienteC4 = p;
                break;
            }
        }
        if (pacienteC4 != null) {
            System.out.println("Paciente seleccionado: " + pacienteC4.getNombre() + " " +
                    pacienteC4.getApellido() + " (ID: " + pacienteC4.getId() + ")");
            Long tiempoEspera = tiemposAtencionPorPaciente
            .get(pacienteC4.getId());
            if (tiempoEspera != null) {
                System.out.println("Tiempo total de atencion: " + tiempoEspera + " minutos");
                System.out.println("Tiempo maximo permitido C4: " + TIEMPOS_MAXIMOS.get(4) + " minutos");
                if (tiempoEspera > TIEMPOS_MAXIMOS.get(4)) {
                    System.out.println("EXCEDIO el tiempo maximo por " +
                            (tiempoEspera - TIEMPOS_MAXIMOS.get(4)) + " minutos");
                } else {
                    System.out.println("Atendido dentro del tiempo permitido");
                }

                System.out.println("\nHistorial de cambios:");
                String cambio;
                while (!(cambio = pacienteC4.obtenerUltimoCambio()).equals("No hay cambios.")) {
                    System.out.println("- " + cambio);
                }
            }
        }
    }
```

3. Experimentación

Para este experimento se implementó una clase GeneradorPacientes encargada de crear una lista de pacientes. Se generó un total de 144 pacientes para un período de 24 horas. Los atributos de los pacientes (nombre, apellido, ID, categoría, tiempo de llegada, estado, área) fueron asignados aleatoriamente, simulando situaciones reales de un entorno hospitalario. La categoría de urgencia se asigna aleatoriamente de C1 a C5, siguiendo una distribución de probabilidad específica, desde C1 a C5 según su nivel de urgencia.

Los pacientes generados se guardaron en un archivo de texto llamado "Pacientes24h.txt", que incluye un resumen con la cantidad y el porcentaje de pacientes por categoría. Los tiempos de llegada y atención fueron los siguientes:

- La simulación se corre por 1440 minutos (24 horas).
- Un nuevo paciente llega cada 10 minutos.
- Un paciente es atendido regularmente cada 15 minutos
- Además, si se acumulan 3 o más pacientes en la cola de atención, se atienden 2 pacientes adicionales de inmediato.

Pruebas específicas realizadas: El simulador registra estadísticas como tiempos de espera, cantidad de pacientes que superaron el tiempo máximo permitido, y el número de pacientes atendidos por categoría. Se calculó el tiempo promedio de atención por categoría, así como los tiempos mínimo y máximo de espera. Se identificó la cantidad de pacientes que excedieron el tiempo máximo de espera definido para su categoría. También se realizó una prueba de seguimiento individual para un paciente de categoría C4, mostrando su tiempo de espera y si fue atendido dentro del tiempo permitido.

4. Ánalisis

4.1. Ánalisis de resultados generales

- ¿Cuántos pacientes fueron atendidos en total al finalizar la simulación? Se atendieron 144 pacientes.
- ¿Cuántos pacientes por categoría (C1 a C5) fueron atendidos?
  - C1: 16
  - C2: 13
  - C3: 29
  - C4: 41

    - C5: 45
- ¿Cuál fue el tiempo promedio de espera por categoría?
    - C1: 2.3 minutos
    - C2: 2.5 minutos
    - C3: 1.4 minutos
    - C4: 2.1 minutos
    - C5: 1.4 minutos
- ¿Cuántos pacientes excedieron el tiempo máximo de espera definido para su categoría? 6 pacientes se excedieron del tiempo máximo permitido.
- ¿Cuáles fueron los peores tiempos de espera registrados por categoría?
    - C1: 3.1 minutos
    - C2: 4.5 minutos
    - C3: 3.9 minutos
    - C4: 4.2 minutos
    - C5: 4.1 minutos

4.2. Análisis asintótico de métodos críticos

Se analiza el tiempo de ejecución teórico (notación Big-O) de los siguientes métodos:

- registrarPaciente de la clase Hospital: tiene una complejidad temporal O(log n), porque está la operación de inserción en la cola de prioridad (colaAtencion.add(p)). Las otras operaciones tienen complejidad O(1). Por lo tanto, la complejidad total del método está determinada por el insertar en la cola.
- atenderSiguiente de la clase Hospital: El método atenderSiguiente devuelve y elimina con PriorityQueue, la única operación que presenta es (poll), operación la cual tiene complejidad O(log n).
- ingresarPaciente de la clase AreaAtencion: Este método utiliza (add) que agrega un objeto Paciente a un PriorityQueue, presenta una complejidad O(log n), en la cual n se refiere a la cantidad de pacientes en pacientesHeap.
- ordenarPacientsPorHeapSort de la clase AreaAtencion: Se utiliza PriorityQueue, y se extraen elementos de la copia, y luego extrae todos los elementos n veces con poll() y los agrega a una lista. Esto resulta en una complejidad (O(n log n).

- reasignarCategoria de la clase Hospital: tiene una complejidad de O(n) debido a que la operación colaAtencion.remove(paciente) debe buscar el paciente de manera lineal en el peor caso, lo cual domina sobre las demás operaciones de tiempo constante. Por lo tanto, este método depende principalmente del número de pacientes en la cola (n).

4.3. Decisiones de diseño

- ¿Qué estructuras utilizó y por qué? Las estructuras utilizadas fueron Stack, PriorityQueue, Heap, HashMap y List. Las tres estructuras principales son la PriorityQueue para la cola central de atención (permite extraer siempre al paciente con mayor prioridad), un HashMap para acceder a los pacientes por su id, y Heaps por cada área de atención, para el análisis con HeapSort. Estas estructuras que fueron solicitadas ayudan a la eficiencia en operaciones como inserción, extracción y búsqueda, lo cual es aún más óptimo para el entorno en el cual se desarrolla el laboratorio.
- ¿Cómo modeló la cola central de atención? En primer lugar la cola central utiliza los niveles de urgencia para su prioridad siendo de C1 a C5, teniendo que C1 es la de mayor urgencia, y en segundo lugar, el tiempo total de espera que le da prioridad a quien lleva más tiempo en caso de estar en empate. Esto ayuda a que las emergencias de mayor urgencia sean atendidas primero, y permite que los pacientes con menor urgencia, pero con esperas prolongadas, vayan teniendo más prioridad, y así no hay pacientes que queden desatendidos por tiempo indefinido.
- ¿Cómo administró el uso de montones para áreas de atención? Cada área (SAPU, urgencia adulto e infantil) gestionó a sus pacientes con su heap, lo que les permite funcionar cada uno por su cuenta y por esto se pudo aplicar HeapSort para obtener listas ordenadas por prioridad sin afectar a la cola activa.
- ¿Qué ventajas observó en su enfoque? Las más grandes ventajas son en primer lugar la rapidez para atender a los pacientes, ya que el sistema siempre da prioridad a los casos más urgentes de la manera más eficiente. La segunda ventaja es la capacidad de gestión que se tiene frente a la la problemática de los pacientes entre el nivel de urgencia y el tiempo de espera, y este sistema logra tener un equilibrio para estos dos factores principales.
- ¿Detectó alguna limitación o ineficiencia? Una de las limitaciones es que los pacientes con menor prioridad (C4 y C5) queden sin ser atendidos durante momentos de mayor congestión, ya que los casos más urgentes siempre pasan primero, lo que podría dejarlos esperando indefinidamente.

4.4. Ventajas y desventajas del sistema

- Ventajas: una de las ventajas es que el sistema permite simular un entorno realista en el que hay mucho que gestionar según su categoría de urgencia y más factores, entonces esto facilita el análisis de escenarios realistas y la organización de atención bajo diferentes condiciones. Otra ventaja es que utiliza estructuras de datos avanzadas, como priorityqueue y heap, para gestionar y organizar a los pacientes en tiempo real, haciendo que los casos más urgentes sean atendidos primero y optimizando el uso de recursos.
- Desventajas: se tiene la desventaja de que si bien el sistema va en torno a un hospital, aún así faltan otros roles tales como médicos o turnos, entonces por este motivo la capacidad para simular la atención médica se ve limitada de ser lo más realista posible. También está la desventaja de que, como se mencionó anteriormente, existe dificultad para la igualdad de las áreas de atención cuando hay una gran cantidad de pacientes de baja prioridad (C4 y C5), lo que puede generar demoras prolongadas para estos casos y saturación en ciertas áreas del hospital.

4.5. Desafíos encontrados

Respecto a desafíos se tiene lo siguiente: el sistema debía priorizar a los pacientes por la categoría de urgencia (C1 a C5), y también por su tiempo de espera acumulado. Entonces esto significa que en casos de empate en la categoría, el paciente con mayor tiempo de espera debía ser atendido primero. Entones la implementación requerida fue una priorityqueue que retornaba una lista ordenada de los pacientes. Otra alternativa que puede ser utilizada es el uso dos priorityqueue distintas y así una es para la categoría y otra para el tiempo de espera. Otro desafío es el sincronizar correctamente el tiempo de llegada, atención y reasignación, esto se debe a que el funcionamiento de tiempos es el siguiente: la simulación avanzaba en intervalos de 1 minuto, pero los pacientes llegaban cada 10 minutos y la atención cada 15 minutos y que cada 3 llegadas debían de haber 2 atenciones inmediatas. Entonces la gestión del procedimiento en el tiempo adecuado es el más grande desafío del laboratorio. Para la solución de estó se tiene un ciclo principal por minutos: La simulación avanza minuto a minuto, verificando en cada iteración y también se tiene un ajuste dinámico de timestamps en el que los pacientes generados al inicio se les actualiza el tiempoLlegada durante la simulación para reflejar el tiempo real. Otra alternativa fue usar una cola de eventos ordenados por tiempo pero se descartó por mayor complejidad para gestionar la simulación.

4.6. Extensión del sistema: turnos médicos

Para llevar a cabo el sistema de manera más realista y elevar la complejidad de la simulación, se propone una extensión que incorpore médicos con sus respectivas

especialidades y turnos de trabajo. Esta extensión permitiría un análisis mayor de la capacidad operativa y la gestión de personal en el hospital.

4.6.1. Clase Médico

- **Atributos:**
  - String id: identificador único para cada médico.
  - String nombre, String apellido: información de cada médico.
  - String especialidad: esta especialidad es para separar a que pacientes puede atender el médico.
  - `Set<Integer>categoriasAtendibles`: un conjunto de las categorías de urgencia (C1 a C5) que el médico puede atender.
  - String estado: esto indica si el médico está disponible, ocupado o en descanso.
  - long tiempoLiberacion: minuto en el que el médico que está ocupado finalizará su tarea y volverá a estar disponible.
  - int duracionMediaAtencion: un valor fijo de tiempo por médico para atender.
- **Métodos Esenciales:**
  - Medico(String id, String nombre, String apellido, String especialidad): constructor para inicializar al médico.
  - void iniciarAtencion(Paciente p, long tiempoSimulacionActual): deja al médico ocupado y calcula su tiempoLiberacion basándose en la duracionMediaAtencion y el tipo de paciente.
  - void finalizarAtencion(long tiempoSimulacionActual): Una vez que tiempoSimulacionActual supera o iguala a tiempoLiberacion, marcando al médico como disponible.
  - boolean estaDisponible(long tiempoSimulacionActual): da true si el médico está disponible o si ya debería estarlo basándose en tiempoLiberacion.
  - boolean puedeAtender(Paciente p): revisa si la categoría del paciente está contenida en el categoriasAtendibles del médico.
- **Relación en áreas de atención**

  En área de atención aparte de tener una capacidadMaxima de pacientes, se tendría una `List<Medico>medicosAsignados` que contendría los médicos que trabajan en esa área específica.

La lógica del método atenderPaciente() en AreaAtencion se modificaría de la siguiente manera: inicialmente, se extrae el paciente con mayor prioridad de la cola mediante pacientesHeap.poll(). Después el sistema recorre la lista de medicosAsignados en busca de un médico que esté estaDisponible() y que puedeAtender(paciente). Si se encuentra un médico adecuado, se llama al médico, iniciarAtencion(paciente, tiempoSimulacionActual), lo que cambia el estado del paciente a en atención, aunque este no será dado de alta del sistema hasta que termine su atención. Si no hay un médico disponible en el área que pueda atender al paciente, deberá ser reasignado temporalmente a la cola principal hasta que un médico esté disponible.

5. Conclusión

El desarrollo de este laboratorio ha sido una experiencia para aprender sobre estructuras de datos y algoritmos en un entorno realista e importante como es la gestión de un hospital. Se ha programado un sistema de simulación haciendo principal uso de priorityqueue, map y stack, con esto se logra gestionar el flujo de pacientes, la priorización de la atención según categorización y la interacción con áreas de atención que tienen capacidad limitada.

Un aprendizaje fundamental en cuanto a las estructuras de datos es sobre las colas de prioridad que demostraron ser importantes para mantener el orden de atención según la gravedad y el tiempo de espera. Los mapas por otro lado tienen un acceso a la información más importante a la hora de ser un paciente en este caso. Estas estructuras son herramientas cuya correcta programación y organización son fundamentales para llevar a cabo sistemas útiles de la realidad como también lo es en este caso el hospital.

Aunque se han logrado muchos avances con el laboratorio, también han surgido oportunidades para mejorar y seguir desarrollando el proyecto. Por ejemplo, en este no se tiene en cuenta de forma detallada al personal médico, como los médicos y enfermeras, ni sus especialidades o turnos de trabajo. Esta simplificación hace que el análisis no sea tan preciso como podría ser así como también su realismo. Es por estos motivos que se tiene la idea de incluir la gestión de los turnos médicos, lo cual permitiría representar mejor la realidad.

En resumen, este laboratorio ayudó a ver que las estructuras de datos son herramientas muy útiles para resolver problemas reales. El poder programar y simular para comprender un sistema como este, conlleva un aprendizaje en general y sobre todo en la mejora de procesos. Por último, el realizar estos sistemas más difíciles ayuda en el desarrollo de sistemas eficientes y de alto rendimiento.