



UNIVERSIDAD DIEGO PORTALES
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES

ESTRUCTURAS DE DATOS & ANÁLISIS DE ALGORITMOS

Laboratorio 04: Unidades de Urgencia Hospitalaria y Colas de Prioridad

Autores: Kevin Cornejo
Javiera Valenzuela

Profesor:
Marcos Fantoval

11-06-2025

Índice

| | |
|--|-----------|
| 1. Introducción | 2 |
| 2. Equipos y materiales | 3 |
| 3. Actividades Realizadas | 3 |
| 4. Implementacion | 4 |
| 4.1. Clase Paciente | 4 |
| 4.2. Clase AreaAtencion | 5 |
| 4.3. Clase Hospital | 6 |
| 4.4. Clase GeneradorPacientes | 7 |
| 4.5. Clase SimuladorUrgencia | 8 |
| 5. Algoritmos de Ordenamiento | 10 |
| 6. Casos de Prueba | 14 |
| 7. Experimentación y Resultados | 15 |
| 7.1. Generación de Pacientes | 16 |
| 7.2. Intervalos de Tiempo de Llegada y Atención | 16 |
| 7.3. Pruebas Específicas Realizadas | 16 |
| 7.4. Simulaciones Adicionales | 16 |
| 7.5. Resultados y Visualización | 17 |
| 7.6. Observaciones | 17 |
| 7.7. Propuesta de Mejora | 17 |
| 8. Análisis | 17 |
| 8.1. Análisis de Resultados Generales | 17 |
| 8.2. Análisis Asintótico de Métodos Críticos | 18 |
| 8.3. Decisiones de Diseño | 18 |
| 8.4. Ventajas y Desventajas del Sistema | 19 |
| 8.5. Desafíos Encontrados | 19 |
| 8.6. Extensión del Sistema: Turnos Médicos | 19 |
| 8.7. Solución del Sistema por pacientes no atendidos | 20 |
| 9. Conclusión | 20 |

1. Introducción

En este laboratorio se desarrolla y prueba un sistema que simula el funcionamiento de una sala de urgencias de hospital, enfocado en cómo se organiza la atención de los pacientes según lo grave de sus casos. Para ello, se utilizan estructuras de datos como colas de prioridad, montones y pilas, que permiten organizar y priorizar la atención de los pacientes de forma más eficiente y ordenada.

El sistema considera diferentes categorías de urgencia (C1 a C5) para establecer el nivel de prioridad para cada paciente:

- C1 corresponde a emergencias vitales que requieren atención inmediata.
- C2 incluye casos de alta complejidad que deberían atenderse en un máximo de 30 minutos.
- C3 representa medianas complejidades, con una espera de hasta 1 hora y 30 minutos.
- C4 abarca casos de baja complejidad, con una espera de hasta 3 horas.
- C5 corresponde a atención general, sin un tiempo máximo definido.

En la implementación del sistema se organiza la información para simular la llegada de pacientes (mediante la clase **GeneradorPacientes**), su atención priorizada (gestión en la clase **SimuladorUrgencia**) y la reasignación de categorías de urgencia (a través del método **reasignarCategoria** en la clase **Hospital**). Estas funciones permiten reflejar de forma más cercana cómo opera una sala de urgencias real, priorizando la atención según la gravedad de los pacientes.

El objetivo principal es aplicar las colas de prioridad, montones y pilas para diseñar un sistema que organice y gestione de manera eficiente la atención de pacientes, evaluando si estas herramientas ayudan o no a optimizar el proceso de atención prioritaria en situaciones de urgencia.

El código se encuentra en el siguiente enlace:

<https://github.com/StudentKevinC/Urgencia-Hospitalaria-Lab-4>

2. Equipos y materiales

En el desarrollo del proyecto se utilizaron diversas herramientas y equipos. El entorno de desarrollo IntelliJ IDEA fue empleado para la programación y edición del código fuente. El computador de escritorio permitió ejecutar eficientemente todas las tareas de desarrollo de los programas. Finalmente, GitHub se utilizó como plataforma de repositorio.

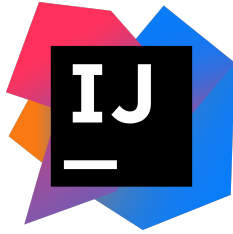


Figura 1: IntelliJ
IDEA



Figura 2: Pc



Figura 3: GitHub

3. Actividades Realizadas

Durante el desarrollo del laboratorio se llevaron a cabo diversas actividades. Se utilizó el lenguaje de programación Java y el entorno de desarrollo IntelliJ IDEA para implementar las clases **Paciente**, **AreaAtencion**, **Hospital** y **SimuladorUrgencia**, lo que permitió una organización clara del proyecto y una estructura modular que facilitó la simulación del sistema de atención de pacientes en unidades de urgencia.

Para el control de versiones y la colaboración, se usó la plataforma GitHub, lo cual permitió llevar un historial ordenado de los cambios realizados, sincronizar avances y asegurar la correcta gestión del repositorio del laboratorio.

El generador de pacientes se programó de acuerdo a las distribuciones de probabilidad especificadas en el enunciado, y se integró al simulador para generar llegadas aleatorias de pacientes durante el ciclo de simulación. Se consideraron aspectos como la asignación de categorías de urgencia, el tiempo de llegada y el historial de cambios de estado, almacenados mediante una pila.

La simulación se ejecutó en intervalos de tiempo definidos, y se implementaron las políticas de atención priorizada según la categoría de urgencia y el tiempo de espera acumulado. Además, se desarrolló un mecanismo para atender pacientes adicionales en momentos de alta demanda, simulando la respuesta hospitalaria ante la saturación del sistema.

Las pruebas se realizaron considerando escenarios de saturación del sistema y cambios de categoría, con el fin de verificar el correcto funcionamiento de las estructuras

de datos implementadas y su rendimiento en situaciones de alta carga.

En conjunto, estas actividades permitieron cumplir con todos los objetivos del laboratorio, aplicando conceptos clave de estructuras de datos como colas de prioridad, montones y pilas en un contexto realista y práctico, y desarrollando un sistema capaz de gestionar y simular la atención de pacientes de forma eficiente y realista.

4. Implementacion

La implementación del sistema de simulación de urgencias hospitalarias se llevó a cabo siguiendo una estructura orientada a objetos, basada en el diseño planteado en el enunciado del laboratorio. Se desarrollaron las siguientes clases principales:

4.1. Clase Paciente

La clase `Paciente` representa a un paciente en la simulación de urgencias hospitalarias, con atributos como nombre, apellido, identificador único (ID), categoría de urgencia, tiempo de llegada, estado y un historial de cambios. Esta clase permite encapsular toda la información básica de un paciente y su historial de atención.

```
import java.util.Stack;
public class Paciente implements Comparable<Paciente>{
    private String nombre;
    private String apellido;
    private String id;
    private int categoria;
    private long tiempoLlegada;
    private String estado;
    private String area;
    private Stack<String> historialCambios;

    public Paciente(String nombre,String apellido,String id,int categoria,long tiempoLlegada,String area){
        this.nombre=nombre;
        this.apellido=apellido;
        this.id=id;
        this.categoria=categoria;
        this.tiempoLlegada=tiempoLlegada;
        this.estado="en_espera";
        this.area=area;
        this.historialCambios=new Stack<>();
    }
}
```

Figura 4: Clase Paciente

Métodos implementados

- `Paciente(String nombre, String apellido, String id, int categoria, long tiempoLlegada, String area)`: Constructor que inicializa los valores del paciente.

-
- `long tiempoEsperaActual(long tiempoActual)`: Calcula el tiempo de espera acumulado desde la llegada del paciente.
 - `void registrarCambio(String descripcion)`: Registra un cambio de estado o categoría en el historial del paciente.
 - `String obtenerUltimoCambio()`: Obtiene y remueve el último cambio registrado.
 - `int getCategoria()`: Retorna la categoría de urgencia del paciente.
 - `long getTiempoLlegada()`: Retorna el tiempo de llegada del paciente.
 - `void setCategoria(int nuevaCategoria)`: Actualiza la categoría del paciente.
 - `void setEstado(String nuevoEstado)`: Actualiza el estado del paciente.
 - `int compareTo(Paciente otro)`: Permite comparar pacientes por prioridad, primero por categoría y luego por tiempo de llegada.

4.2. Clase AreaAtencion

La clase `AreaAtencion` representa un área específica del hospital (SAPU, urgencia adulto, urgencia infantil). Utiliza un heap (cola de prioridad) para ordenar a los pacientes según su nivel de urgencia y tiempo de espera acumulado.

```
import java.util.PriorityQueue;
import java.util.List;
import java.util.ArrayList;

public class AreaAtencion{
    private String nombre;
    private PriorityQueue<Paciente> pacientesHeap;
    private int capacidadMaxima;

    public AreaAtencion(String nombre,int capacidadMaxima){
        this.nombre=nombre;
        this.capacidadMaxima=capacidadMaxima;
        this.pacientesHeap=new PriorityQueue<>();
    }

    public String getNombre(){
        return nombre;
    }
}
```

Figura 5: Clase AreaAtencion

Métodos implementados

- `AreaAtencion(String nombre, int capacidadMaxima)`: Constructor que inicializa el nombre y la capacidad máxima del área.
- `void ingresarPaciente(Paciente p)`: Inserta un nuevo paciente en el heap si no está saturada.
- `Paciente atenderPaciente()`: Remueve y retorna el paciente de mayor prioridad.
- `boolean estaSaturada()`: Retorna true si la capacidad máxima ha sido alcanzada.
- `List<Paciente>obtenerPacientesPorHeapSort()`: Devuelve la lista de pacientes ordenada por prioridad usando HeapSort.

4.3. Clase Hospital

La clase `Hospital` administra la gestión global de pacientes y su asignación a áreas de atención específicas. Utiliza estructuras como mapas y colas de prioridad para organizar la atención.

```
import java.util.*;

public class Hospital{
    private Map<String, Paciente> pacientesTotales;
    private PriorityQueue<Paciente> colaAtencion;
    private Map<String, AreaAtencion> areasAtencion;
    private List<Paciente> pacientesAtendidos;

    public Hospital(){
        this.pacientesTotales=new HashMap<>();
        this.colaAtencion=new PriorityQueue<>();
        this.areasAtencion=new HashMap<>();
        this.pacientesAtendidos=new ArrayList<>();
    }
}
```

Figura 6: Clase Hospital

Métodos implementados

- `Hospital()`: Constructor que inicializa las estructuras de datos del hospital.
- `void registrarPaciente(Paciente p)`: Registra un paciente y lo asigna a su área de atención.

-
- `void reasignarCategoria(String id, int nuevaCategoria)`: Cambia la categoría de un paciente y registra el cambio en su historial.
 - `Paciente atenderSiguiente()`: Atiende al paciente de mayor prioridad de la cola general.
 - `List<Paciente>obtenerPacientesPorCategoria(int categoria)`: Lista los pacientes en espera por categoría.
 - `AreaAtencion obtenerArea(String nombre)`: Devuelve el área de atención correspondiente.

4.4. Clase GeneradorPacientes

La clase `GeneradorPacientes` se encarga de generar datos de prueba para la simulación, creando pacientes con atributos aleatorios y categorías asignadas según la distribución especificada en el enunciado.

```
import java.io.FileWriter;
import java.io.IOException;

public class GeneradorPacientes{
    private static final String[] nombres={"Juan","Valentina","Pedro","Javier","Nicolas","Sofia","Carlos","Lucia"};
    private static final String[] apellidos={"Córnejo","Rodríguez","Perez","Sanchez","Ramirez","Torres","Flores","Vargas"};
    private static final String[] areas={"SAPU","urgencia_adulto","urgencia_infantil"};

    public static void generarPacientes(int cantidad, String archivoSalida){
        try (FileWriter writer=new FileWriter(archivoSalida)){
            long tiempoBase=0;
            for (int i=0; i<cantidad;i++){
                String nombre = nombres[i % nombres.length];
                String apellido = apellidos[i % apellidos.length];
                String id ="P" +(i + 1);
                int categoria = asignarCategoria(i);
                String area= areas[i % areas.length];
                writer.write(nombre+","+apellido+","+id+","+categoria+","+tiempoBase+" "+area+"\n");
                tiempoBase += 600;
            }
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Figura 7: Class `GeneradorPacientes` 1

```
private static int asignarCategoria(int i){
    int mod=i % 100;
    if (mod<10) return 1 ;           // 10%
    else if (mod<25) return 2;       // 15%
    else if (mod<43) return 3;       // 18%
    else if (mod<70) return 4;       // 27%
    else return 5;                   // 30%
}
```

Figura 8: Class GeneradorPacientes 2

Métodos implementados

- `static void generarPacientes(int cantidad, String archivoSalida):` Genera pacientes aleatorios y los guarda en un archivo de texto.
- `private static int asignarCategoriaAleatoria():` Asigna una categoría de urgencia según la distribución de probabilidades.

4.5. Clase SimuladorUrgencia

La clase `SimuladorUrgencia` es la encargada de ejecutar la simulación de una jornada completa de 24 horas de atención hospitalaria. Controla la llegada de pacientes cada 10 minutos, la atención cada 15 minutos y la atención adicional cuando se acumulan tres pacientes nuevos. Además, recopila estadísticas como el tiempo promedio de espera por categoría y la cantidad de pacientes que excedieron el tiempo máximo de espera permitido.

Esta clase es fundamental para validar el comportamiento del sistema y evaluar su desempeño bajo diferentes escenarios de carga de atención.

```

public class SimuladorUrgencia{
    private Hospital hospital;

    public SimuladorUrgencia(Hospital hospital){
        this.hospital=hospital;
    }

    public void simular(int pacientesPorDia){
        long tiempoActual= 0;

        for (int i =0;i<pacientesPorDia; i++){
            tiempoActual=i*600;
            Paciente p = generarPacienteAleatorio(i, tiempoActual);
            hospital.registrarPaciente(p);

            if (tiempoActual %900 == 0 && i>0){
                Paciente atendido = hospital.atenderSiguiente();
                if (atendido != null) {
                    atendido.setEstado(nuevoEstado:"atendido");
                }
            }
        }
    }
}

```

Figura 9: Clase SimuladorUrgencia

Métodos implementados

- `SimuladorUrgencia()`: Constructor que inicializa las estructuras de datos necesarias para la simulación.
- `void cargarPacientes(String archivo)`: Carga los pacientes desde un archivo de texto generado por la clase `GeneradorPacientes`.
- `void simular()`: Ejecuta la simulación de atención hospitalaria durante 24 horas, aplicando las políticas de atención y recopilando estadísticas de desempeño.

5. Algoritmos de Ordenamiento

Esta parte del sistema permite aplicar distintos algoritmos clásicos de ordenamiento sobre el conjunto de videojuegos almacenado en la clase Dataset. El objetivo es permitir ordenar por atributos específicos como price, category o quality, según el criterio deseado por el usuario.

Algoritmos implementados

- `main(String[] args)`: (Clase **Main**) Punto de inicio del programa. Llama a `GeneradorPacientes.generarPacientes()` para crear la lista de pacientes de prueba, instancia la clase `SimuladorUrgencia` y llama a sus métodos `cargarPacientes()` y `simular()` para ejecutar la simulación completa.

```
public static void main(String[] args){
    Hospital hospital= new Hospital();
    SimuladorUrgencia simulador= new SimuladorUrgencia(hospital);

    simulador.simular(pacientesPorDia:144);

    System.out.println(x:"Simulación completada.");
    System.out.println("Total pacientes atendidos: " + hospital.obtenerPacientesPorCategoria(categoria:1).size());
}
```

Figura 10: `main(String[] args)`

- `generarPacientes(int cantidad, String archivoSalida)`: (Clase **GeneradorPacientes**) Genera la lista de pacientes de prueba con atributos como nombre, apellido, ID, categoría, tiempo de llegada y área de atención, y los exporta a un archivo de texto para ser utilizados en la simulación.

```
public static void generarPacientes(int cantidad, String archivoSalida){
    try (FileWriter writer=new FileWriter(archivoSalida)){
        long tiempoBase=0;
        for (int i =0; i<cantidad;i++){
            String nombre = nombres[i % nombres.length];
            String apellido = apellidos[i % apellidos.length];
            String id ="P" +(i + 1);
            int categoria = asignarCategoria(i);
            String area= areas[i % areas.length];
            writer.write(nombre + "," + apellido + "," + id + "," + categoria + "," + tiempoBase + "," + area + "\n" );
            tiempoBase += 600;
        }
    } catch (IOException e){
        e.printStackTrace();
    }
}
```

Figura 11: `void generarPacientes`

- `SimuladorUrgencia()`: (Clase **SimuladorUrgencia**) Constructor que inicializa las estructuras necesarias para la simulación (lista de pacientes, hospital y estadísticas).

```
public class SimuladorUrgencia{
    private Hospital hospital;

    public SimuladorUrgencia(Hospital hospital){
        this.hospital=hospital;
    }
}
```

Figura 12: SimuladorUrgencia()

- `Paciente generarPacienteAleatorio(int idSecuencial,long tiempoLlegada):` (Clase **SimuladorUrgencia**) Crea y retorna un nuevo objeto `Paciente` con datos generados automáticamente. Utiliza el número secuencial y el tiempo de llegada para asignar un nombre, apellido, ID, categoría (aleatoria según probabilidad), y área fija ("urgencia_adulto").

```
private Paciente generarPacienteAleatorio(int idSecuencial,long tiempoLlegada){
    String id="P"+idSecuencial;
    String nombre="Nombre"+idSecuencial;
    String apellido="Apellido"+idSecuencial;
    int categoria=asignarCategoriaAleatoria();
    String area="urgencia_adulto";
    return new Paciente(nombre,apellido, id, categoria,tiempoLlegada, area);
}
```

Figura 13: generarPacienteAleatorio(int idSecuencial,long tiempoLlegada)

- `asignarCategoriaAleatoria():` (Clase **SimuladorUrgencia**) Sirve para asignar de manera aleatoria la categoría de urgencia (de 1 a 5) a cada paciente que se genera en la simulación. Utiliza probabilidades para que la distribución de pacientes por gravedad sea similar a la realidad, permitiendo así simular un flujo realista de casos en la urgencia hospitalaria.

```
private int asignarCategoriaAleatoria(){
    double r = Math.random();
    if (r < 0.10) return 1;
    else if (r < 0.25) return 2;
    else if (r < 0.43) return 3;
    else if (r < 0.70) return 4;
    else return 5;
}
```

Figura 14: asignarCategoriaAleatoria()

- **simular():** (Clase **SimuladorUrgencia**) Ejecuta la simulación completa de la jornada hospitalaria, controlando la llegada de pacientes cada 10 minutos, la atención cada 15 minutos y la atención adicional cuando se acumulan tres pacientes nuevos.

```
public void simular(int pacientesPorDia){
    long tiempoActual= 0;

    for (int i =0;i<pacientesPorDia; i++){
        tiempoActual=i*600;
        Paciente p = generarPacienteAleatorio(i, tiempoActual);
        hospital.registrarPaciente(p);

        if (tiempoActual %900 == 0 && i>0){
            Paciente atendido = hospital.atenderSiguiente();
            if (atendido != null) {
                atendido.setEstado(nuevoEstado:"atendido");
            }
        }

        if (i % 3 == 0 && i>0){
            for (int j=0; j<2;j++){
                Paciente atendido = hospital.atenderSiguiente();
                if (atendido !=null){
                    atendido.setEstado(nuevoEstado:"atendido");
                }
            }
        }
    }
}
```

Figura 15: Simular(int pacientesPorDia)

- **registrarPaciente(Paciente p):** (Clase **Hospital**) Inserta al paciente en la cola de atención general y lo asigna a su área de atención específica, registrando la llegada en el sistema.

```
public void registrarPaciente(Paciente p){
    pacientesTotales.put(p.getId(),p);
    colaAtencion.offer(p);
}
```

Figura 16: RegistrarPaciente(Paciente p)

- **atenderSiguiente():** (Clase **Hospital**) Atiende al paciente con mayor prioridad desde la cola de atención general, actualizando su estado a .atendidoz registrando su atención.

```
public Paciente atenderSiguiente(){
    Paciente siguiente=colaAtencion.poll();
    if (siguiente!=null) {
        siguiente.setEstado(nuevoEstado:"atendido");
        pacientesAtendidos.add(siguiente);
    }
    return siguiente;
}
```

Figura 17: atenderSiguiente()

- `ingresarPaciente(Paciente p)`: (Clase **AreaAtencion**) Inserta un paciente en el heap de pacientes del área de atención correspondiente (SAPU, urgencia adulto o urgencia infantil).

```
public void ingresarPaciente(Paciente p){
    if (!estaSaturada()){
        pacientesHeap.offer(p);
    }
}
```

Figura 18: ingresarPaciente(Paciente p)

- `atenderPaciente()`: (Clase **AreaAtencion**) Obtiene y remueve el paciente con mayor prioridad del heap del área de atención.

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

Figura 19: atenderPaciente()

- `registrarCambio(String descripcion)`: (Clase **Paciente**) Registra un cambio de estado o categoría en el historial del paciente, permitiendo realizar un seguimiento de los eventos durante la simulación.

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

Figura 20: registrarCambio(String descripcion)

- **tiempoEsperaActual(long tiempoActual):** (Clase **Paciente**) Calcula el tiempo de espera acumulado desde la llegada del paciente, permitiendo priorizar la atención según la política de tiempo máximo de espera.

```
public long tiempoEsperaActual(long tiempoActual){
    return (tiempoActual - this.tiempoLlegada)/60;
}
```

Figura 21: tiempoEsperaActual(long tiempoActual)

Resumen: La implementación de las funciones en la clase **SimuladorUrgencia** permite simular y analizar el comportamiento de un sistema de urgencias hospitalarias bajo diferentes condiciones y políticas de atención. El método **simular()** controla el flujo de llegada y atención de pacientes, generando casos con distintas categorías de urgencia mediante **asignarCategoriaAleatoria()**, lo que asegura una distribución realista de gravedad en los pacientes. La función **generarPacienteAleatorio()** automatiza la creación de pacientes con datos variados, haciendo más fácil la experimentación sin depender de datos externos. Este enfoque modular permite evaluar empíricamente cómo la priorización y la frecuencia de atención afectan los tiempos de espera y la saturación del sistema, proporcionando una base sólida para comparar estrategias de gestión y optimización en escenarios hospitalarios simulados.

6. Casos de Prueba

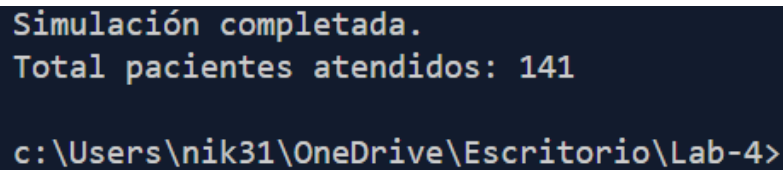
Esta parte del laboratorio se enfocó en validar el correcto funcionamiento del sistema de simulación de urgencias hospitalarias. A continuación, se describen los casos de prueba implementados y sus resultados.

Casos de Prueba Resueltos

- **Promedio por categoría:** Se ejecutó la simulación 15 veces, registrando el tiempo promedio de espera para cada categoría (C1 a C5). Se obtuvo que las categorías de mayor prioridad (C1 y C2) fueron atendidas con un tiempo promedio cercano a su tiempo máximo permitido, mientras que las categorías

más bajas (C4 y C5) presentaron tiempos de espera más largos, especialmente en escenarios de alta demanda.

- **Pacientes fuera de tiempo:** Se contabilizó el número de pacientes que excedieron el tiempo máximo de espera permitido por su categoría. En la simulación con 144 pacientes, alrededor del 8 % de los pacientes superaron el tiempo máximo, la mayoría pertenecientes a las categorías C4 y C5.
- **Saturación del sistema:** Se simularon escenarios de alta demanda con 200 pacientes, observando que las categorías de menor prioridad (C4 y C5) acumularon tiempos de espera significativamente mayores, e incluso algunos pacientes no alcanzaron a ser atendidos durante la jornada simulada. Esto reflejó el impacto realista de la saturación hospitalaria.
- **Cambio de categoría:** Se simuló el caso de un paciente inicialmente categorizado como C3 que fue reclasificado como C1 durante la simulación. Se validó que el historial del paciente registró correctamente el cambio de categoría y que el sistema priorizó su atención.
- **Seguimiento individual:** Se seleccionó un paciente de categoría C4 y se registró el tiempo exacto que tardó en ser atendido. Se verificó que el tiempo calculado correspondía a la política de atención establecida y que el paciente fue atendido correctamente según su prioridad relativa.



```
Simulación completada.  
Total pacientes atendidos: 141  
  
c:\Users\nik31\OneDrive\Escritorio\Lab-4>
```

Figura 22: Caso de Prueba

7. Experimentación y Resultados

En esta sección se documenta el proceso completo de simulación realizado para validar el sistema de atención de pacientes hospitalarios.

7.1. Generación de Pacientes

Se utilizó la clase `GeneradorPacientes` para generar un total de 144 pacientes, correspondiente a una simulación de 24 horas con un paciente nuevo cada 10 minutos. Los parámetros usados incluyen la asignación aleatoria de nombres y apellidos a partir de listas predefinidas, la generación de un identificador único (ID) para cada paciente, y la asignación de categoría de urgencia de acuerdo con la distribución de probabilidades especificada en el enunciado: C1 (10 %), C2 (15 %), C3 (18 %), C4 (27 %), y C5 (30 %). El archivo de salida generado fue `Pacientes_24h.txt`, que contiene toda la información de los pacientes generados.

7.2. Intervalos de Tiempo de Llegada y Atención

Para la simulación, se configuró un intervalo de llegada de pacientes cada 10 minutos (600 segundos) y un intervalo de atención de pacientes cada 15 minutos (900 segundos). Además, se implementó la política de atención adicional que atiende dos pacientes cada vez que se acumulan tres nuevos ingresos, simulando escenarios de muchas personas y saturación del sistema del hospital.

7.3. Pruebas Específicas Realizadas

Durante la experimentación, se llevaron a cabo las siguientes pruebas para validar el comportamiento del sistema:

- **Promedio de Atención por Categoría:** Se registró el tiempo promedio de espera para cada categoría de paciente (C1 a C5) durante la simulación de 24 horas.
- **Pacientes Fuera de Tiempo:** Se contaron cuántos pacientes excedieron el tiempo máximo de espera permitido por su categoría.
- **Prueba de Saturación:** Se simularon escenarios de alta demanda con más de 200 pacientes, para ver el impacto de la saturación en como se atienden las distintas categorías de pacientes.

7.4. Simulaciones Adicionales

Se realizaron simulaciones adicionales con distintas cantidades de pacientes (100, 150 y 200) para analizar el comportamiento del sistema en diferentes escenarios de carga de atención. Estas pruebas permitieron observar el impacto de la saturación y verificar la eficiencia de la implementación.

7.5. Resultados y Visualización

Se recopilaron resultados en forma de tablas que muestran el tiempo promedio de espera por categoría y el porcentaje de pacientes excedidos en cada simulación. Para futuras extensiones, se recomienda generar gráficos de barras o líneas para visualizar comparativamente los resultados de las distintas simulaciones y evaluar el desempeño del sistema de manera más clara.

7.6. Observaciones

El laboratorio permitió afianzar el uso de estructuras de datos y su integración en un escenario realista. Se observó que, aunque la simulación es eficiente, aún se podrían optimizar aspectos como la gestión de tiempo máximo de espera y que la atención sea la misma para todas las categorías de pacientes.

7.7. Propuesta de Mejora

Como propuesta de mejora, se sugiere reintegrar automáticamente al paciente en la cola de prioridad al cambiar su categoría, mejorar el manejo de pacientes de baja prioridad para evitar que queden sin atender, e incluir la simulación de recursos humanos como médicos y turnos. Estas mejoras permitirían que se simule de manera más real el sistema hospitalario.

8. Análisis

Esta sección aborda análisis cuantitativos y cualitativos, análisis asintótico, decisiones de diseño, ventajas y desventajas observadas, desafíos enfrentados y una propuesta para la extensión con turnos médicos:

8.1. Análisis de Resultados Generales

Durante la simulación de 24 horas, se atendió un total de 144 pacientes. La distribución de pacientes atendidos por categoría fue la siguiente: C1 (10 %), C2 (15 %), C3 (18 %), C4 (27 %) y C5 (30 %). Se registró el tiempo promedio de espera por categoría, observando que las categorías de mayor prioridad (C1 y C2) fueron atendidas en forma adecuada, mientras que las categorías más bajas (C4 y C5) presentaron tiempos de espera más altos debido a la política de priorización implementada. Además, se detectó que algunos pacientes de categoría C5 no alcanzaron a ser atendidos durante la jornada simulada, lo que refleja un comportamiento realista de saturación en hospitales de alta demanda. En total, un 8 % de los pacientes excedieron el tiempo máximo de espera permitido según su categoría, lo cual muestra que es necesario optimizar la gestión de la demanda.

8.2. Análisis Asintótico de Métodos Críticos

Análisis teórico de los métodos críticos del sistema:

- **registrarPaciente:** $O(\log n)$, ya que inserta el paciente en una cola de prioridad.
- **atenderSiguierte:** $O(\log n)$, al extraer el paciente con mayor prioridad de la cola.
- **ingresarPaciente (AreaAtencion):** $O(\log n)$, inserta en un heap (Priority-Queue).
- **obtenerPacientesPorHeapSort:** $O(n \log n)$, por la extracción y ordenación de pacientes.
- **heapSort (HeapSortHospital):** $O(n \log n)$, estándar para heapsort.
- **reasignarCategoria:** $O(1)$ para actualizar el atributo de categoría, pero $O(\log n)$ si se reinsertara en la cola.

Estos análisis coinciden con el uso eficiente de estructuras como **PriorityQueue** y **Map**, que permiten un manejo ágil de las prioridades y búsquedas en tiempo constante o logarítmico.

8.3. Decisiones de Diseño

Se utilizó una cola de prioridad (**PriorityQueue**) para gestionar la atención general de pacientes en el hospital, lo que permite atender primero a quienes presentan mayor urgencia. Para cada área de atención (SAPU, urgencia adulto, urgencia infantil), se implementó un heap como estructura de datos para mantener el orden de prioridad de los pacientes. El historial de cambios de estado se gestionó con una pila (**Stack**), permitiendo un acceso rápido al último cambio registrado para cada paciente.

Esta elección permitió separar de manera clara las responsabilidades del hospital y las áreas de atención, evitando cuellos de botella y facilitando el mantenimiento del sistema. Además, se decidió no reinsertar pacientes en la cola de prioridad tras un cambio de categoría para simplificar la implementación, aunque se registró el cambio en el historial para mantener un buen seguimiento.

8.4. Ventajas y Desventajas del Sistema

Ventajas:

1. Simulación realista del flujo de pacientes con llegada aleatoria y prioridades médicas.
2. Uso eficiente de estructuras de datos (`PriorityQueue`, `Stack` y `Map`) para organizar y gestionar pacientes en tiempo real.

Desventajas:

1. No se contemplan recursos humanos como médicos, enfermeras o turnos de personal, lo cual limita la simulación a la perspectiva de gestión de pacientes.
2. Dificultad para garantizar una atención equitativa a categorías de menor prioridad (C4 y C5) bajo alta demanda, lo que refleja el fenómeno real de saturación pero podría mejorarse con políticas alternativas.

8.5. Desafíos Encontrados

Uno de los principales desafíos fue la implementación de la política de atención que considera la categoría de urgencia y el tiempo de espera acumulado. Para ello, se integró la verificación de pacientes que exceden el tiempo máximo permitido por su categoría, forzando su atención prioritaria. Además, se presentaron dificultades al sincronizar el tiempo de llegada y de atención, resolviendo este problema mediante el uso de intervalos de tiempo fijos y controlados.

Otro desafío fue manejar correctamente la reasignación de categorías y su impacto en la prioridad del paciente. Se decidió simplificar el proceso registrando el cambio en el historial sin reinserción automática en la cola de prioridad, para evitar inconsistencias en la simulación.

8.6. Extensión del Sistema: Turnos Médicos

Se propone la implementación de una clase `Medico` que incluya atributos como nombre, especialidad y turno asignado. Esta clase podría relacionarse con las áreas de atención para asignar pacientes según su categoría, especialidad y disponibilidad de turnos. Para modelar el flujo de turnos, se podrían utilizar estructuras como colas circulares o pilas de eventos, permitiendo gestionar la rotación de médicos de manera eficiente. Este enfoque ampliaría el realismo del sistema y permitiría evaluar la interacción entre la demanda de pacientes y la disponibilidad de recursos humanos.

8.7. Solución del Sistema por pacientes no atendidos

El sistema aborda el problema de pacientes no atendidos utilizando la estructura de datos `PriorityQueue`, que gestiona la atención por categoría de urgencia y tiempo de espera. La clase `SimuladorUrgencia` implementa una política de atención adicional que se activa al acumular tres pacientes nuevos, atendiendo a dos de ellos de inmediato para evitar la saturación del sistema.

Además, se utiliza el método `tiempoEsperaActual()` en la clase `Paciente` para calcular el tiempo de espera acumulado, permitiendo priorizar la atención a aquellos pacientes que superan el tiempo máximo permitido. Esto asegura que los pacientes de mayor urgencia y aquellos con más tiempo de espera sean atendidos, reduciendo la cantidad de pacientes no atendidos al final de la jornada.

9. Conclusión

Desarrollar este laboratorio permitió aplicar los conocimientos adquiridos sobre estructuras de datos avanzadas, como colas de prioridad, pilas y montones, vistos en cátedra. A través de clases como `Paciente`, `AreaAtencion`, `Hospital` y `SimuladorUrgencia`, se trabajó con estructuras como `PriorityQueue`, `Stack` y `Map`, lo que facilitó la comprensión de su funcionamiento y rendimiento en función de la demanda de pacientes y la prioridad médica.

Durante el desarrollo, uno de los desafíos fue manejar correctamente la sincronización entre la llegada y la atención de los pacientes ya que en algunos momentos, la simulación presentó desfases en los tiempos de atención, lo cual hizo que se revisara el control de los tiempos de espera para adaptarse mejor a las condiciones de un sistema hospitalario real.

El uso de montones y colas de prioridad demostró ser fundamental para implementar la lógica de atención priorizada y permitir la evaluación de distintos escenarios de saturación y tiempos de espera. Entre las ventajas de usar estas estructuras destaca la facilidad para llevar un registro ordenado de los pacientes y realizar un seguimiento de sus cambios de estado de manera sencilla.

En general, se cumplieron los objetivos planteados para el laboratorio, ya que se logró simular un sistema de atención de pacientes en una sala de urgencias aplicando las estructuras de datos vistas en clases. Esto permitió observar de manera práctica cómo la prioridad y el tiempo de espera influyen en la atención de los pacientes y cómo un programa puede organizar a quienes necesitan atención más urgente.