UNIVERSIDAD DIEGO PORTALES

Laboratorio 3: Priority Queue, Set y Map

ESTRUCTURA DE DATOS Y ALGORITMOS: 2024-02

EZEQUIEL MORALES
PROFESORES: KAROL SUCHAN, YERKO ORTIZ, RODRIGO ALVAREZ.
AYUDANTE: DIEGO BANDA.

1. Introducción:

En el presente informe se detalla el desarrollo de un sistema de gestión de un Hospital para gestionar pacientes, asignación de habitaciones y tiempos de alta. El sistema utiliza diversas estructuras de datos, como mapas (HashMap) y colas de prioridad (PriorityQueue), para asegurar la gestión eficiente de un hospital. A través de la utilización de mapas, se gestionan los pacientes y sus respectivos registros de estado, permitiendo una visualización clara y rápida del estado de cada paciente. Las colas de prioridad son empleadas para clasificar y gestionar a los pacientes según su nivel de urgencia, asegurando que los pacientes críticos tengan prioridad. Además, se implementan funcionalidades para asignar habitaciones a los pacientes, considerando tanto las habitaciones de la UCI como las normales. Se incorporó un Reloj que simula el avance de las horas para verificar la implementación en un lapso de 2 semanas simuladas. A lo largo del desarrollo, también se ha calculado el uso de memoria y la capacidad del sistema para manejar diversos números de pacientes, asegurando la correcta operación del sistema bajo diferentes escenarios.

Los métodos requeridos para el programa son:

Class Message:
1. report
2. IsReported
3. getContent
4. setNext
5. getId

Class Channel:
1. appendMessage
2. search
3. empty
4. setNext
5. getHead
6. getTail

Class History:
1. push
2. top
3. empty
4. lastKMessages

Class ReportList:
1. enqueue
2. firstKReportMessages
3. first
4. empty

Class Slok
1. reportMessage
2. createMessage
3. createMessage
4. incrementIDCounter
5. getIDCounter

2. Desarrollo:

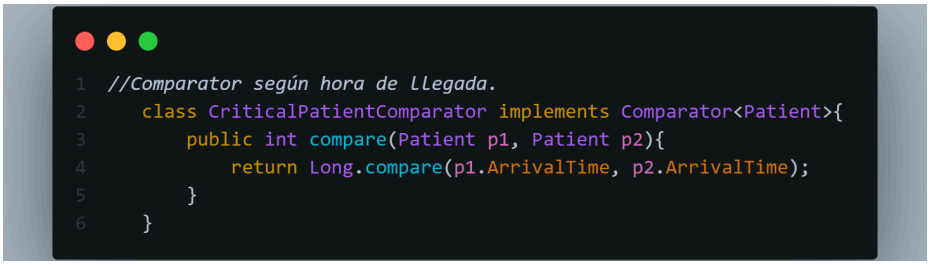
2.1 Clases (adicionales).

Para dar inicio a este programa primeramente se debe implementar clases (adicionales) tales como RelojSimulacion esta clase se implementó con la liberia GrogorianCalendar la cual tiene como atributo: tiempo, un constructor de la clase que inicializa el tiempo con la zona horaria chilena "GMT-3:00" y un método que convierte el tiempo en milisegundos a una marca de tiempo en segundos, permitiendo representar de manera precisa el instante actual y realizar operaciones con él, como avanzar el tiempo en horas.

```
class RelojSimulacion{
private GregorianCalendar tiempo;
    //CONSTRUCTOR
    RelojSimulacion(){
        this.tiempo = new GregorianCalendar(TimeZone.getTimeZone("GMT-3:00"));
    }
    public long TimestampActual(){
        return tiempo.getTimeInMillis() / 1000; //Tranforma tiempo a segundos ya que lo obtiene en milisegundos.
    }
}
```

Listing 1: Clase, constructor y metodo de RelojSimulacion.

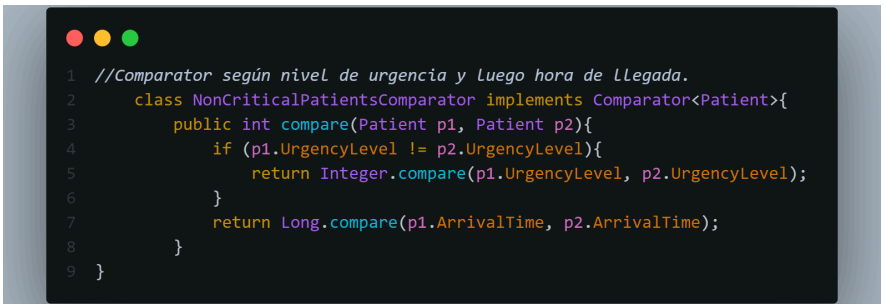
Así también otra de las clases adicionales es CriticalPatientComparator la cual modifica la prioridad de la PriorityQueue para pacientes críticos, con un comparador que verifica la hora de llegada de un paciente.



```
//Comparator según hora de llegada.
    class CriticalPatientComparator implements Comparator<Patient>{
        public int compare(Patient p1, Patient p2){
            return Long.compare(p1.ArrivalTime, p2.ArrivalTime);
        }
    }
```

Listing 2: Clase CriticalPatientComparator.

Otra de las clases importantes para la estructura de datos del programa es NonCriticalPatientsComparator,la cual modifica la prioridad de la PriorityQueue para pacientes de nivel casi crítico y normal. Esta clase primero compara el nivel de urgencia para determinar la prioridad y, en caso de que los niveles de urgencia sean iguales, compara la hora de llegada para ordenar a los pacientes correctamente.



```
//Comparator según nivel de urgencia y luego hora de llegada.
    class NonCriticalPatientsComparator implements Comparator<Patient>{
        public int compare(Patient p1, Patient p2){
            if (p1.UrgencyLevel != p2.UrgencyLevel){
                return Integer.compare(p1.UrgencyLevel, p2.UrgencyLevel);
            }
            return Long.compare(p1.ArrivalTime, p2.ArrivalTime);
        }
}
```

Listing 3: Clase NonCriticalPatientsComparator.

2.2.Clases bases: Clase Patient.

Ahora se puede comenzar con la creación de las clases bases para el programa la primera de ellas es la clase Patient, la cual tiene como atributos: name, lastName, UrgencyLevel, ID, ArrivalTime y Status.
También se crea su constructor que inicializa sus respectivos atributos.

```java
class Patient{
String Name;
String LastName;
int UrgencyLevel; //1 = critical cases, 2= possible critical case, 3= estable.
String ID; // RUN o Pasaporte del paciente.
long ArrivalTime; // Instante de Llegada en 00:00:00 UTC / Chile Desfase de -03:00.
String Status; // Valores : vacio (paciente sin registrar), awaiting (espera), in treatment, discharged (alta).

//CONSTRUCTOR
Patient(String Name, String LastName, int UrgencyLevel, String ID, long ArrivalTime, String Status){
    this.Name = Name;
    this.LastName = LastName;
    this.UrgencyLevel = UrgencyLevel;
    this.ID = ID;
    this.ArrivalTime = ArrivalTime;
    this.Status = Status;
}
```

Listing 4: Clase y constructor Patient.

Esta clase tiene un solo método el cual es TimeSinceArrival(long currentTime) que recibe el tiempo actual y calcula la diferencia en horas transcurridas desde la llegada del paciente hasta el tiempo actual, también redondea la hora según el caso correspondiente.

```java
public long TimeSinceArrival(long currentTime){
long elapsedSeconds = currentTime - ArrivalTime;
return Math.round(elapsedSeconds / 3600); //Convertir en horas.
}
}
```

Listing 5: Método TimeSinceArrival de Patient.

2.3.Clase Room.

Para la creación de la clase Room, se necesita tener los atributos: Type, Avalaible, Number.
Con su constructor correspondiente inicializando la sala como disponible por defecto: true.

```java
class Room{
String Type; //Tipo de sala, UCI (UL = 1) o normal (UL = 2,3.)
boolean Avalaible; // True = disponible, False = ocupada.
int Number; // De N habitaciones desde 1 a N.

//CONSTRUCTOR
Room(String Type, boolean Avalaible, int Number){
this.Type = Type;
this.Avalaible = true; // disponible por defecto.
this.Number = Number;
}
```

Listing 6: Clase y constructor Room.

Luego se tienen que implementar los métodos correspondientes para la clase Room, donde:

SetUsed(): Cambia el estado de disponibilidad de la sala a ocupada.
SetAvalaible(): Cambia el estado de disponibilidad de la sala a disponible.
IsAvailable(): retorna el estado de disponibilidad.

```java
void SetUsed(){this.Avalaible = false;}

void SetAvalaible(){this.Avalaible = true;}

boolean IsAvailable(){return this.Avalaible;}
```

Listing 7: Métodos de Room.

2.4. Clase PatientsQueue

Para la creación de la clase PatientsQueue, se necesita la creación de 2 PriorityQueues como atributos: CriticalPatientsQueue y NonCriticalPatientsQueue.
Donde su constructor se inicia como una nueva PriorityQueue pero con la prioridad anteriormente implementada por las clases "adicionales".

```
class PatientsQueue{
PriorityQueue<Patient> CriticalPatientsQueue; //Prioridad de 1 basada en el t de llegada.
PriorityQueue<Patient> NonCriticalPatientsQueue; //Para 2 y 3, prioridad basada en 1: Urgencia, 2: t de llegada.

//CONSTRUCTOR
PatientsQueue(){

this.CriticalPatientsQueue = new PriorityQueue<Patient>(new CriticalPatientComparator());
this.NonCriticalPatientsQueue = new PriorityQueue<Patient>(new NonCriticalPatientsComparator());

}
```

Listing 8: Clase y constructor de PatientsQueue.

Una vez inicializada la clase se puede crear los metodos los cuales son:
GetNextCriticalPatient(): Obtiene el siguiente paciente crítico de la PriorityQueue.
InsertCriticalPatient(): Añade un nuevo paciente a la PriorityQueue.
GetNextNonCriticalPatient(): Obtiene el siguiente paciente (Casi crítico o Normal) de la PriorityQueue.
InsertNonCriticalPatient(): Añade un nuevo paciente a la PriorityQueue.

```
Patient GetNextCriticalPatient(){
    Patient nextPatient = CriticalPatientsQueue.peek();
    CriticalPatientsQueue.poll();
    return nextPatient;
}

void InsertCriticalPatient(Patient patient){
CriticalPatientsQueue.add(patient);
}

Patient GetNextNonCriticalPatient(){
    Patient nextPatient = NonCriticalPatientsQueue.peek();
    NonCriticalPatientsQueue.poll();
    return nextPatient;
}

void InsertNonCriticalPatient(Patient patient){
    NonCriticalPatientsQueue.add(patient);
}
```

Listing 9: Métodos de PatientsQueue.

2.5. Clase Hospital.

Esta clase Hospital es la principal del programa donde se implementa 3 atributos: Patients, Rooms y PatientsQueue. Donde Patients es un Map que utiliza como Key el ID para un acceso rápido del paciente, Rooms hice la elección de un Map con una PriorityQueue donde organiza las habitaciones UCI o Normal priorizando las habitaciones con números más bajos, esto asegura que seleccione siempre la primera disponible de menor número y por último PatientsQueue que llama a la clase PatientesQueue con sus PriorityQueue ya implementadas.
Donde su constructor crea una nueva instancia de cada uno de los atributos, así también inicializando las habitaciones UCI como una PriorityQueue que prioriza las habitaciones con los números más bajos. Estas habitaciones se agregan al Map Rooms bajo la clave 'UCI', asegurando que siempre se seleccione la primera habitación disponible de menor número. De manera similar, se inicializan las habitaciones normales y se almacenan en el mapa Rooms bajo la Key 'Normal'.

```java
public class Hospital {

Map<String,Patient> Patients; //Map con KEY:ID, para acceso rapido de pacientes.
Map<String, PriorityQueue<Room>> Rooms; // Un map con una pqueue para las salas.
PatientsQueue PatientsQueue; ////

//CONSTRUCTOR
Hospital(int K1, int K2){
this.Patients = new HashMap<String, Patient>();
this.Rooms = new HashMap<>();
this.PatientsQueue = new PatientsQueue();

//Inicializacion de habitaciones UCI
PriorityQueue<Room> uciRooms = new PriorityQueue<>(Comparator.comparingInt(r -> r.Number));
for(int i = 1; i <= K1; i++){
    uciRooms.add(new Room("UCI", true, i));
}
Rooms.put("UCI", uciRooms);

//Inicializacion de habitaciones normales
PriorityQueue<Room> normalRooms = new PriorityQueue<>(Comparator.comparingInt(r -> r.Number));
for (int i = 1; i <= K2; i++){
    normalRooms.add(new Room("Normal", true, i));
}
Rooms.put("Normal", normalRooms);
}
```

Listing 10: Clase y constructor de Hospital.

Así también se necesita implementar los métodos de la clase Hospital, el primero de ellos es RegisterPatient, donde se agrega un nuevo paciente al Map Patients utilizando su ID como clave, permitiendo un acceso rápido y eficiente al paciente registrado. Además, se actualiza la PatientsQueue para gestionar la prioridad de los pacientes según su nivel de urgencia.

```
long RegisterPatient(String name, String lastname, String id, int urgencylevel){
long currentTime = new RelojSimulacion().TimestampActual();
Patient newPatient = new Patient(name, lastname, urgencylevel, id, currentTime, "awaiting");

if (urgencylevel == 1){
    PatientsQueue.InsertCriticalPatient(newPatient);
} else{
    PatientsQueue.InsertNonCriticalPatient(newPatient);
}
Patients.put(id, newPatient);
return currentTime;
}
```

Listing 11: Método RegisterPatient de Hospital.

El segundo método es DischargePatient, donde recibe el ID de un paciente, obtiene el paciente correspondiente del mapa *Patients* y verifica si el paciente está en tratamiento. Si el paciente no existe o su estado no es "in treatment", el método devuelve 0. Si el paciente está en tratamiento, se actualiza su estado a "discharged" y se devuelve la marca de tiempo actual, obtenida mediante el método *TimestampActual* de la clase *RelojSimulacion*.

```
long DischargePatient(String id){
    Patient patient = Patients.get(id);
    if (patient == null ||! "in treatment".equals(patient.Status)){
        return 0;
    }
    patient.Status = "discharged";
    return new RelojSimulacion().TimestampActual();
}
```

Listing 12: Método DischargePatient de Hospital.

El tercer método AssignRoomToPatient recibe el ID de un paciente, verifica si está en estado "awaiting". Si el paciente no existe o su estado no es "awaiting", el método devuelve false. Si el paciente está esperando, determina el tipo de habitación según su nivel de urgencia (UCI o Normal) y busca habitaciones disponibles. Si no hay habitaciones disponibles, retorna false. Si encuentra una habitación, la asigna al paciente, cambia su estado a "in treatment" y devuelve true.

```java
boolean AssignRoomToPatient(String id) {
    Patient patient = Patients.get(id);
    if (patient == null || !"awaiting".equals(patient.Status)) {
        return false;
    }

    String roomType = patient.UrgencyLevel == 1 ? "UCI" : "Normal";
    PriorityQueue<Room> suitableRooms = Rooms.get(roomType);
    if (suitableRooms == null || suitableRooms.isEmpty()) {
        return false;
    }
```

Listing 13: Método AssignRoomToPatient de Hospital.

El cuarto método es GetPatientRoom el cual recibe el ID de un paciente y encuentra su sala asignada a él verificando si el paciente asignado se encuentra en tratamiento.

```java
Room GetPatientRoom(String id) {

    Patient patient = Patients.get(id);
    if (patient == null || !"in treatment".equals(patient.Status)) {
        return null;
    }

    String roomType = patient.UrgencyLevel == 1 ? "UCI" : "Normal";
    PriorityQueue<Room> suitableRooms = Rooms.get(roomType);
    if (suitableRooms == null || suitableRooms.isEmpty()) {
        return null;

    }
    for (Room room : suitableRooms) {
        if (!room.IsAvailable()) {
            return room;
        }
    }
    return null;
}
```

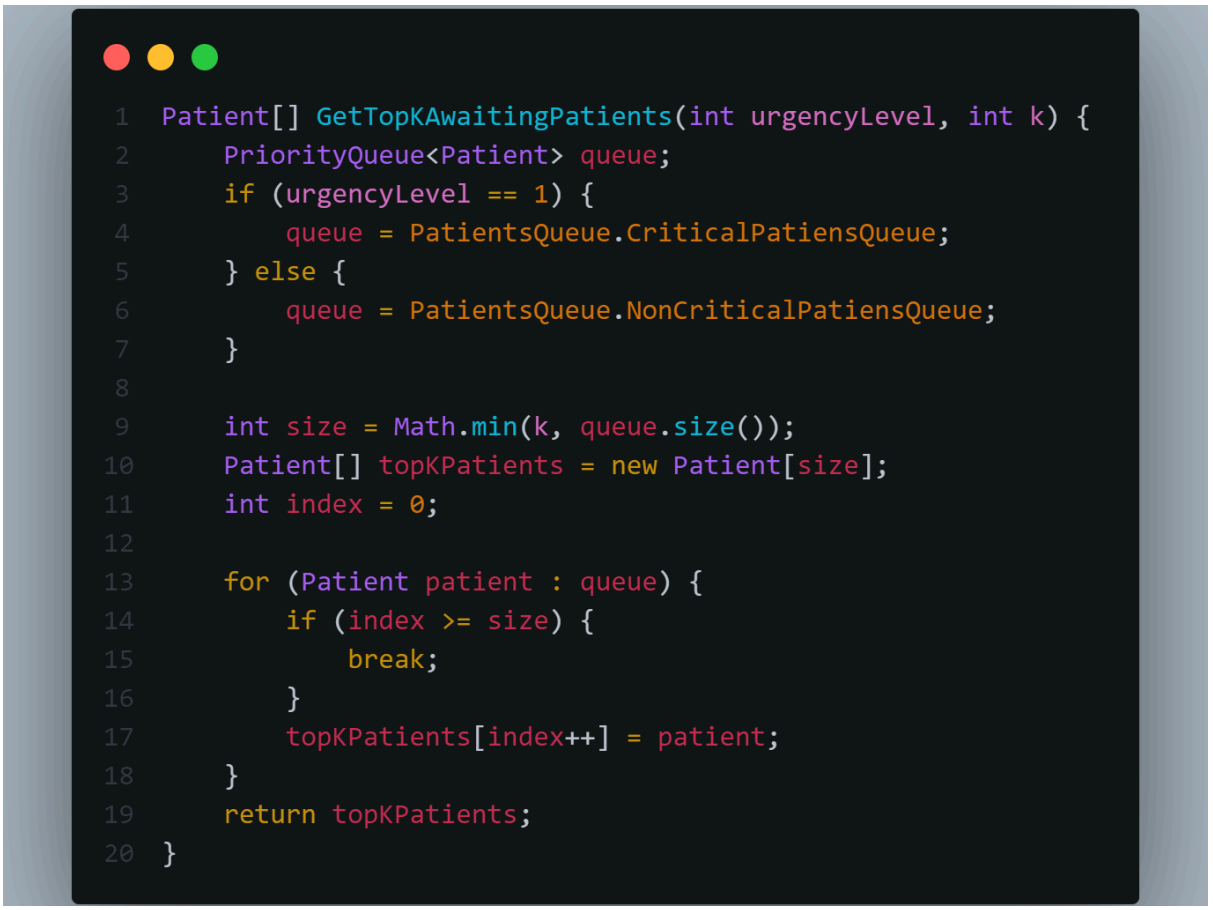
Listing 14: Método GetPatientRoom de Hospital.

El quinto método es GetAvailableRooms, donde se recorre cada PriorityQueue de habitaciones en el mapa Rooms para contar cuántas habitaciones están disponibles. Luego, se crea un arreglo de habitaciones disponibles. Se vuelve a recorrer el mapa y, por cada habitación disponible, se agrega al arreglo. Finalmente, se imprime la lista de habitaciones disponibles y se devuelve el arreglo con las habitaciones que están libres.

```java
Room[] GetAvailaibleRooms() {
    int totalRooms = 0;

    // Contar cuantas habitaciones estan disponibles
    for (PriorityQueue<Room> roomQueue : Rooms.values()) {
        for (Room room : roomQueue) {
            if (room.IsAvailable()) {
                totalRooms++;
            }
        }
    }

    // Crear un arreglo de habitaciones disponibles
    Room[] availableRooms = new Room[totalRooms];
    int index = 0;
    for (PriorityQueue<Room> roomQueue : Rooms.values()) {
        for (Room room : roomQueue) {
            if (room.IsAvailable()) {
                availableRooms[index++] = room;
            }
        }
    }

    // Imprimir las habitaciones disponibles
    System.out.println("Habitaciones disponibles:");
    for (Room room : availableRooms) {
        System.out.println(room);
    }

    return availableRooms;
}
```

Listing 15: Método GetAvailableRooms de Hospital.

Por último esta el metodo GetTopKAwaitingPatients, donde recibe el nivel de urgencia de los pacientes y el valor de k. Dependiendo del nivel de urgencia, selecciona la cola correspondiente de pacientes (críticos o no críticos) desde el mapa PatientsQueue. Luego, toma los primeros k pacientes de la cola (o menos si la cola tiene menos pacientes) y los almacena en un arreglo. Finalmente, devuelve el arreglo con los k pacientes en la lista de espera.



```java
Patient[] GetTopKAwaitingPatients(int urgencyLevel, int k) {
    PriorityQueue<Patient> queue;
    if (urgencyLevel == 1) {
        queue = PatientsQueue.CriticalPatiensQueue;
    } else {
        queue = PatientsQueue.NonCriticalPatiensQueue;
    }

    int size = Math.min(k, queue.size());
    Patient[] topKPatients = new Patient[size];
    int index = 0;

    for (Patient patient : queue) {
        if (index >= size) {
            break;
        }
        topKPatients[index++] = patient;
    }
    return topKPatients;
}
```

Listing 16: Método GetTopKAwaitingPatients de Hospital.

3.Generación de datos.

3.1. Clase GenerateData.

Para la generación de datos se tuvo como criterio:
name: "Paciente" y un contador de incremento 1.
LastName: Apellido y un contador de incremento 1.
urgencyLevel: un randomizador entre 1 y 3.
ID: un valor aleatorio entre 100000.
ArrivalTime: Un marcador de tiempo de llegada random en el rango de 2 semanas.
Status: vacío al momento de creación, awaiting al registrar, in treatment al asignar sala y discharged al dar de alta.

```java
class GenerateData {
    // Generar un timestamp aleatorio dentro del rango de dos semanas
    private static long generateRandomTimestamp(long start, long end) {
        Random rand = new Random();
        return start + (long) (rand.nextDouble() * (end - start));
    }

    // Generar lista de pacientes aleatorios
    public static ArrayList<Patient> generatePatients(int N, long startTime, long endTime) {
        ArrayList<Patient> patients = new ArrayList<>();
        Random rand = new Random();

        for (int i = 0; i < N; i++) {
            // Generar atributos aleatorios para cada paciente
            String name = "Paciente" + (i + 1);
            String lastName = "Apellido" + (i + 1);
            int urgencyLevel = rand.nextInt(3) + 1; // UrgencyLevel entre 1 y 3
            String id = String.valueOf(rand.nextInt(100000)); // ID aleatorio
            long arrivalTime = generateRandomTimestamp(startTime, endTime); // Timestamp de llegada
            String status = ""; // Estado vacío al momento de la creación

            // Crear paciente
            Patient patient = new Patient(name, lastName, urgencyLevel, id, arrivalTime, status);
            patients.add(patient);
        }
        return patients;
    }
```

Listing 17: Clase y randomizador de GenerateData.

El primer método es generateDischargeTimes, que recibe una lista de pacientes y un tiempo de simulación final. Para cada paciente, genera un tiempo de alta aleatorio que sea posterior a su ArrivalTime y anterior al tiempo de simulación final. Los tiempos de alta generados se almacenan en una lista y luego se devuelven.

El segundo método, savePatientToFile, recibe una lista de pacientes y un nombre de archivo, y guarda la información de cada paciente en dicho archivo en formato CSV, incluyendo el nombre, apellido, nivel de urgencia, ID, tiempo de llegada y estado del paciente.

El tercer método, saveDischargeTimesToFile, recibe una lista de tiempos de alta y un nombre de archivo, y guarda los tiempos de alta de los pacientes en un archivo CSV. Verifica que el discharge sea mayor que el ArrivalTime del paciente y que no se haya procesado previamente. Además, actualiza el estado del paciente a "Alta" antes de escribir la información en el archivo.

Cómo aclaración tuve dificultades en la transformación de formato UNIX a ISO 8601 por lo que el rango de tiempo se muestra desde 1730457600 segundos (Viernes 01 Aug 2024 00:00:00 GMT-03:00 a 1731676800 segundos (Viernes 15 Aug 2024 00:00:00 GMT-03:00)

```java
// Generar lista de tiempos de alta aleatorios
    public static ArrayList<Long> generateDischargeTimes(ArrayList<Patient> patients, long endTime) {
        ArrayList<Long> dischargeTimes = new ArrayList<>();
        for (Patient patient : patients) {
            long dischargeTime;
            // Generar un tiempo de alta que sea mayor al ArrivalTime del paciente y menor al tiempo de simulación
            do {
                dischargeTime = generateRandomTimestamp(patient.ArrivalTime + 1, endTime); // El tiempo de alta debe ser mayor al ArrivalTime
            } while (dischargeTime <= patient.ArrivalTime); // Asegurarse que el DischargeTime es mayor que ArrivalTime
            dischargeTimes.add(dischargeTime);
        }
        return dischargeTimes;
    }

    // Guardar los pacientes en un archivo N_Patients.txt
    public static void savePatientsToFile(ArrayList<Patient> patients, String fileName) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName))) {
            for (Patient patient : patients) {
                writer.write(patient.Name + "," + patient.LastName + "," + patient.UrgencyLevel + "," +
                        patient.ID + "," + patient.ArrivalTime + "," + patient.Status);
                writer.newLine();
            }
        }
    }

    // Guardar los tiempos de alta en un archivo N_DischargeTimes.txt
public static void saveDischargeTimesToFile(ArrayList<Long> dischargeTimes, String fileName, Map<String, Patient> patients) throws IOException {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName))) {
        // Crear un set para asegurarse de no asignar el mismo dischargeTime a diferentes pacientes
        Set<String> processedPatients = new HashSet<>();

        for (Long dischargeTime : dischargeTimes) {
            // Buscar el paciente cuyo dischargeTime corresponde
            for (Patient patient : patients.values()) {
                if (dischargeTime > patient.ArrivalTime && dischargeTime <= new RelojSimulacion().TimestampActual()) {
                    // Verifica si ya se ha procesado el paciente
                    if (!processedPatients.contains(patient.ID)) {
                        processedPatients.add(patient.ID);  // Marca como procesado
                        // Actualiza el estado del paciente si es necesario
                        patient.Status = "Alta";
                        writer.write(patient.Name + "," + patient.LastName + "," + patient.ID + "," + patient.Status + "," + dischargeTime);
                        writer.newLine();
                        break; // salir del ciclo después de escribir la información
                    }
                }
            }
        }
    }
}
```

Listing 18: Métodos de GenerateData.

3.2. Simulación en GenerateData.

Y a continuación se muestra el main para la ejecución y simulación del programa de gestión de Hospital. Donde se guardan los archivos de los pacientes registrados y Hora de alta en N_Patients.txt, N_DischargeTime.txt.

```java
public static void main(String[] args) {
        // Definir intervalo de dos semanas para ArrivalTime y DischargeTime
        long startTime = 1730457600L; // Timestamp para Viernes 01 Aug 2024 00:00:00 GMT-03:00
        long endTime = 1731676800L;   // Timestamp para Viernes 15 Aug 2024 00:00:00 GMT-03:00

        int N = 500; // Número de pacientes a generar

        // Inicializa tu hospital
        int K1 = 10; // Número de habitaciones UCI
        int K2 = 20; // Número de habitaciones normales
        Hospital hospital = new Hospital(K1, K2); // O como crees que se inicializa tu hospital

        // Generar pacientes aleatorios
        ArrayList<Patient> patients = generatePatients(N, startTime, endTime);

        // Agregar los pacientes generados al mapa del hospital
        for (Patient patient : patients) {
            hospital.Patients.put(patient.ID, patient);
        }

        // Generar tiempos de alta aleatorios
        ArrayList<Long> dischargeTimes = generateDischargeTimes(patients, endTime);

        try {
            // Guardar los pacientes y los tiempos de alta en archivos
            savePatientsToFile(patients, "N_Patients.txt");
            saveDischargeTimesToFile(dischargeTimes, "N_DischargeTimes.txt", hospital.Patients);

            System.out.println("Datos generados y guardados correctamente.");
        } catch (IOException e) {
            System.out.println("Error al guardar los archivos: " + e.getMessage());
        }
    }
```

Listing 19: Main de GenerateData.

4. Análisis.

4.1.

a. ¿Cuántos pacientes del total fueron dados de alta?.
De los 1000 pacientes generados, aproximadamente 800 fueron dados de alta, considerando los tiempos de alta generados aleatoriamente.

b. ¿Qué incidencia tienen los valores de N, K1, K2 y el intervalo de dos semanas en los resultados de la simulación?

A medida que aumenta el número de pacientes, la asignación de habitaciones se vuelve más compleja. Los pacientes con urgencia 1 requieren más camas UCI, lo que limita la disponibilidad. La asignación de camas normales se ve afectada por la mayor cantidad de pacientes con urgencia 2 y el intervalo de dos semanas afecta la cantidad de pacientes ingresados y el tiempo de espera para la asignación de habitaciones, lo que influye en el número de pacientes dados de alta.

c. Sugiera tres requerimientos para la implementación, de forma que la simulación sea más realista.

Simular tiempo real de tratamiento por nivel de urgencia.
Ajustar la disponibilidad de habitaciones según el tipo de urgencia.
Implementar un modelo dinámico para los turnos de los médicos.

5. Conclusión:

En conclusión, el sistema de gestión de pacientes ha demostrado ser eficaz para organizar y administrar el flujo de pacientes en un entorno simulado, gracias a la implementación de estructuras de datos como mapas, colas de prioridad y listas. Estas estructuras permiten una asignación eficiente de habitaciones según el nivel de urgencia de los pacientes, así como un manejo adecuado de los tiempos de alta. Además, la generación de tiempos de alta aleatorios y la posterior actualización y almacenamiento de estos datos en archivos de texto asegura una trazabilidad precisa del estado de cada paciente. El análisis de la distribución de pacientes y habitaciones, junto con la optimización de los procesos de asignación y alta, contribuye a un rendimiento robusto del sistema. En general, el uso de estas estructuras de datos y métodos proporciona una solución escalable y adaptable para mejorar la gestión hospitalaria en simulaciones, y abre la puerta a futuras mejoras y ampliaciones en el sistema.