Universidad Diego Portales
Escuela de Informática &
Telecomunicaciones

ESTRUCTURAS DE DATOS & ALGORITMOS

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

Autores:

19 de noviembre de 2025

Índice

1. Introducción

A lo largo del país existen diversos centros de atención en salud, los cuales tienen diferentes áreas de atención y en cada una de ellas un sistema diferente.

En el servicio de urgencias se necesita de un sistema que sea eficiente, para determinar a que paciente se debe atender primero, según su estado de gravedad, como se logra organizar la sala de espera, y también como mantener un historial de los pacientes que ya fueron atendidos.

En este laboratorio se genera un sistema de simulación de una urgencia hospitalaria. En él se modela el flujo completo de un paciente como lo es su ingreso al sistema, su categorización, su permanencia en la sala de espera y el alta del paciente.

Para poder lograr su ejecución, es necesario que se implementen desde cero diferentes estructuras de datos avanzados en el lenguaje informático de Java. Para este trabajo se hará uso de las siguientes herramientas:

- **Montón mínimo:** una cola de prioridad, el cual representa a una cola de atención.
- Tablas de hash: para representar la sala de espera del servicio de urgencias.
- **Árbol binario de búsqueda (BST):** para expresar la creación de un archivo con las atenciones históricas de los pacientes del área de urgencias.
- **Pila (Stack):** usado para registrar los cambios en el estado de atención de un paciente.

El sistema creado para la categorización de los pacientes, se basa en la categorización oficial del Ministerio de Salud de Chile, esta permite estandarizar la prioridad y el tiempo máximo que el que pueden esperar los pacientes según su gravedad, estas son las siguientes:

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

Posteriormente de la categorización se simulan diferentes escenarios de carga en el servicio de urgencias, los cuales se analiza su comportamiento y el efecto en la prioridad de los pacientes de menor gravedad.

En el siguiente repositorio de GitHub se encuentra la información completa de este laboratorio: https://github.com/Cinthya982012/Laboratorio_4_EDA_Cinthya_

e_Ignacia/tree/main. En el enlace se encuentra, el código .Java y el archivo LaTeX para su análisis.

2. Objetivos

El objetivo principal de este laboratorio es implementar y evaluar el rendimiento de estructuras de datos avanzadas en el contexto de una simulación de urgencia hospitalaria.

Se busca como objetivo:

- Diseñar e implementar un montón mínimo para gestionar la cola de atención, priorizando a los pacientes más graves.
- Implementar dos variantes de tablas de hash (encadenamiento y sondeo lineal) para modelar la sala de espera y analizar empíricamente el impacto de las colisiones.
- Implementar un árbol binario de búsqueda (BST) para almacenar un archivo histórico de pacientes dados de alta.
- Comparar mediante experimentos las complejidades teóricas de inserción y búsqueda con los tiempos medidos en la práctica.
- Simular un día completo de funcionamiento de la urgencia bajo distintos niveles de carga, estudiando el comportamiento de los tiempos de espera por categoría.

De esta forma, el laboratorio permite conectar los conceptos teóricos de estructuras de datos y algoritmos, con un problema aplicado en un escenario real.

3. Implementación

Este laboratorio fue realizado en lenguaje Java y se implementaron diferentes clases, las cuales se muestran a continuación en este informe.

3.1. Clase *Paciente*

Esta clase representa a cada persona que ingresa a la urgencia.

Los atributos de esta clase son:

- `String id`: representa el identificador único del paciente, es decir, su rut.
- `String nombre`: corresponde al no,bre del paciente.
- `int categoria`: corresponde al nivel de urgencia desde 1 a 5 (de mayor a menor prioridad).
- `long tiempoLlegada`: timestamp de ingreso (tiempo de ingreso del paciente), en minutos para la simulación.
- `Stack<String>historialCambios`: pila que registra los cambios de estado de un paciente.

La clase también esta formada por los siguientes métodos:

- `public Paciente(String id, String nombre, int categoria, long tiempoLlegada, Stack<String>historialCambios)`: corresponde al constructor que inicializa los atributos del paciente.
- `Setters y Getters`: métodos encargados de obtener y asignar cada atributo.
- `public int compareTo(Paciente p)`: utilizado para comparar por categoria a los pacientes, si estos tuviesen la misma categoria se debe priorizar por el tiempo de llegada.
- `public void registrarCambio(String descripcion)`: registra los cambios en el historial del paciente.

3.2. Clase *ColaPrioridadPacientes* (Min-Montón)

Esta clase implementa una cola de prioridad usando un montículo binario (min-heap) sobre un arreglo. El objetivo de esta clase es que sea rápido obtener al paciente que tiene mayor prioridad y si existe un empate, se debe priorizar al que llego antes. Contiene los siguientes atributos:

- `Paciente[]pacientes`: arreglo que almacena a los pacientes que se deben atender del servicio de urgencias.
- `int tamano`: corresponde ala cantidad actual que existe de paciente en espera.

- `int capacidad`: corresponde al tamaño máximo que puede tener el arreglo de pacientes.

Esta clase contiene los siguientes métodos:

- `public ColaPrioridadPacientes()`: corresponde a un constructor el cual comienza con capacidad de 100 y tamaño cero(según la indicación de la guía.
- `public void insertar(Paciente p)`: es un método que permite insertar a un paciente en la cola de prioridad, pero si la capacidad esta en su maximo debe aumentar el tamaño.*
- `public Paciente obtenerMin()`: corresponde a un método que retorna al paciente con mayor prioridad sin la necesidad de ser sacado.
- `public Paciente extraerMin()`: método por el cual se retorna al paciente de mayor prioridad y lo saca, ya que fue atendido.*
- `private void cambiarTamano()`: este método aumenta la capacidad del arreglo al doble, si es que este se encuentra a máxima capacidad, si esto ocurre, se copian los pacientes del original al nuevo arreglo con mayor capacidad.
- `private int padre(int i)`: retorna el indice del padre, de la posición i en el montón.
- `private int hijoIzquierdo(int i)`: retorna el indice del hijo izquierdo, de la posición i en el montón.
- `private int hijoDerecho(int i)`: retorna el indice del hijo derecho, de la posición i en el montón.
- `private void subirNodo(int i)`: método que sube el nodo en el montón hasta dejarlo en la posición que corresponde, es decir, mientras el nodo hijo tenga más prioridad que su padre, se intercambian y el nodo del hijo sube, ya que tiene mayor prioridad.
- `private void bajarNodo(int i)`: método que baja el nodo en el montón hasta dejarlo en su posición correspondiente, es decir, compara el nodo padre con los nodos hijos, si los hijos tienen mayor prioridad, el padre baja.
- `private void swap(int i, int j)`: realiza el intercambio de dos pacientes en el arreglo.
- `public boolean estaVacia()`: verifica si el arreglo se encuentra vació o no, por lo tanto si esta vació, no hay pacientes.*

*-Métodos con *: Son métodos obligatorios según guía.*

3.3. Tablas de Hash (Directorio de pacientes)

En este laboratorio se implementaron dos versiones de tablas de hash, la primera de ellas en con encadenamiento separado y ola segunda con direccionamiento abierto.

Las dos versiones cumplen el mismo objetivo, pero se diferencian en la forma en que se resuelve las colisiones, por ende eso termina afectando el rendimiento de los experimentos.

A continuación vamos a describir ambas versiones:

3.3.1. Clase *HashPacientesEncadenado*

(Encadenamiento separado) Esta implementación utiliza un arreglo de listas enlazadas para almacenar los pares, cada posición del arreglo es un bucket y dentro de ellos se almacenan los pacientes con el mismo indice de hash. En otras palabras esta clase implementa un mapa (`Map<String, Paciente>`) usando una tabla de hash con ***Encadenamiento***. Por ende, cada posición del arreglo es una lista enlazada de nodos (clave -¿valor). La clave utilizada es el RUT del paciente y el valor a un objeto de tipo Paciente.

Esta clase contiene los siguientes atributos:

- `privateLinkedList<Nodo>[]`: arreglo de lista enlazadas de tamaño fijo M.
- `private int size`: indica la cantidad de elementos guardados.
- `private int capacidad`: corresponde a la capacidad que tiene el arreglo.

Y posee los siguientes métodos:

- `private static class Nodo`: se crea el nodo que guardara clave y valor asociados.
- `public HashPacientesEncadenado()`: constructor vació.
- `public HashPacientesEncadenado(int capacidad)`: constructor que permite indicar la capacidad inicial de la tabla.
- `private int hash(String clave)`: convierte una clave (RUT) en un índice válido del arreglo. Usa `hashCode()` del String(RUT) y luego lo ajusta al tamaño de la tabla.*
- `public Paciente put(String clave, Paciente valor)`: método que inserta a un paciente en la tabla o actualiza su valor si la clave ya existe. La clave corresponde al RUT del paciente y el valor corresponde a un Objeto de tipo Paciente.*
- `public Paciente get(Object clave)`: encargado de obtener el paciente asociado a una clave (RUT), si el paciente no existe se retorna un `null`, pero si se encuentra se retorna el valor de ese nodo.*

- `public Paciente remove(Object clave)`: reliza la eliminación del paciente asociado a una clave (RUT) y retorna el valor eliminado, si no se encuentram retorna `null`.*
- `public int size()`: retorna la cantidad de elementos guardados en el hash.
- `public boolean isEmpty()`: verifica que la lista este o no vacía. Retorna true si el hash no tiene ningún paciente guardado y false en caso contrario.
- `public boolean containsKey(Object clave)`: revisa si existe una clave (RUT) en la tabla. Internamente reutiliza el método get. Si hay un valor asociado retorna true a esa clave.
- `public boolean containsValue(Object value)`: revisa si existe un objeto Paciente en la tabla (comparado por equals). Valor a buscar es un Paciente, retorna true si algun nodo tiene al paciente como valor, en caso contrario retorna false.
- `public void clear()`: encargado de borrar todo el contenido de la tabla de hash, y deja todas las listas vacias y el tamaño en cero.
- `public void putAll(Map m)`: inserta todos los elementos de otro mapa en este hash. Recorre las entradas del mapa y las inserta usando put().
- `public Set<String>keySet()`: retorna un Set con todas las claves (RUT) presentes en la tabla.
- `public Collection<Paciente>values()`:retorna una colección con todos los pacientes almacenados en la tabla.
- `public Set<Entry<String, Paciente≫ entrySet()`: retorna un set con entradas clave y valor.

*-Métodos con *: Son métodos obligatorios según guía.*

3.3.2. Clase *HashPacientesSondeoLineal*

(Direccionamiento Abierto) En esta segunda tabla de hash, se implementa un `Map<String, Paciente>`, por ende usa un direccionamiento abierto con sondeo lineal para resolver colisiones. En esta otra versión no existe listas enlazadas y se almacena en tres arreglos paralelos (estados, claves y valores). Por ende, cuando dos claves calculan el mismo indice hash, el sistema busca la siguiente posición libre, es decir, avanza hasta encontrar una casilla vacía(insertar) o una casilla con la misma clave (actualizar).

Esta clase tiene los siguientes atributos:

- `private int[] estados`: arreglo que guarda el estado de cada casilla (0 = vacia, 1 = ocupada o 2 = borrada).
- `private String[] claves`: arreglo con las claves (RUT de los pacientes).

- `private Paciente[] valores`: arreglo con los valores (objetos Paciente).
- `private int capacidad`: capacidad total de la tabla (tamaño de los arreglos).
- `private int size`:cantidad de elementos actualmente almacenados.

Y tiene los siguientes métodos:

- `public HashPacientesSondeoLineal(int capacidad)`: Constructor que inicializa la tabla con una capacidad fija.
- `private int hash(String clave)`: convierte una clave en un índice del arreglo. Usa hashCode() y lo ajusta al rango [0, capacidad - 1].
- `private int buscarIndiceExistente(String clave)`: busca el indice de una clave que ya existe en la tabla y si no se retorna -1.
- `private int buscarIndiceParaInsertar(String clave)`: busca un indice para poder insertar una clave nueva.
- `public int size()`: retorna el tamaño.
- `public boolean isEmpty()`: verifica si se encuentra vacia o no.
- `public boolean containsKey(Object key)`: revisa si existe una clave en el hash.
- `public boolean containsValue(Object value)`: revisa si existe un Paciente en la tabla(compara con `equals()`).
- `public Paciente get(Object key)`: se obtiene el Paciente asociado a una clave (RUT).
- `public Paciente put(String key, Paciente value)`: inserta o actualiza un paciente asociado a una clave (RUT).
- `public Paciente remove(Object key)`: elimina la clave (RUT) y devuelve el paciente eliminado.
- `public void putAll(Map<? extends String, ? extends Paciente>m)`: inserta todos los pares clave-valor de otro mapa en este hash.
- `public void clear()`: limpia completamente la tabla, dejando todo vacío.
- `public Set<String>keySet()`: retorna un conjunto con todas las claves guardadas.
- `public Collection<Paciente>values()`: retorna una colección con todos los pacientes almacenados.
- `public Set<Map.Entry<String, Paciente≫ entrySet()`: retorna un set con las entradas clave-valor.

3.4. Clase *ArbolPacientes*

Esta clase implementa un árbol de búsqueda binaria, también conocido como Binary Search Tree (BST). Esta formado por nodos y cada uno de ellos guarda a un Paciente y el árbol se ordena por el id (RUT) del paciente.

Primero se debe crear el nodo, con lo siguiente:

- `private static class NodoArbol`: corresponde a la creación del nodo interno con sus parametros (dato, hijo izquierdo, hijo derecho).

Esta clase esta formada por los siguientes atributos:

- `private NodoArbol raiz`: nodo raiz del árbol.

Y por los siguientes métodos:

- `public ArbolPacientes()`: corresponde a un constructor que crea un árbol vació(sin pacientes).
- `public void insertar(Paciente p)`: método para insertar a un nuevo paciente.*
- `private NodoArbol insertarRec(NodoArbol actual, Paciente p)`: inserción recursiva en el árbol binario de búsqueda.
- `public Paciente buscar(String id)`: método para buscar por el RUT del paciente.*
- `private NodoArbol buscarRec(NodoArbol actual, String id)`: búsqueda recursiva en el árbol.
- `public List<Paciente>obtenerPacientesEnOrden()`: se retorna una lista de pacientes que se encuentran ordenados por el id (RUT). Se utiliza el recorrido In-Order de un BST, es decir, recorre el hijo izquierdo, luego el nodo actual y luego el hijo derecho. Elementos quedan ordenados de menor a mayor.*
- `private void inOrder(NodoArbol actual, List<Paciente>lista)`: recorrido In-Order recursivo.

*-Métodos con *: Son métodos obligatorios según guía.*

3.5. Clase *Hospital*

Esta clase coordina las diferentes estructuras de datos de la simulación, como lo son:

- Sala de espera: HashTable.
- Historico: BST.
- Cola de atención: Min-Monton(min-heap)

Esta formada por siguientes atributos:

- `private Map<String, Paciente>salaEspera`: corresponde a un mapa que asocia el RUT del paciente con el Objeto Paciente.
- `private ArbolPacientes historicoPacientes`: es un BST que almacena el historico de pacientes que fueron dados de alta. Se almacena por RUT.
- `private ColaPrioridadPacientes colaAtencion`: representa a una cola de atención, en donde el mínimo es el paciente con mayor prioridad, por ende el mas grave, si hay un empate se elige al que llego antes.

Y por los siguientes métodos:

- `public Hospital(Map<String, Paciente>implSalaEspera)`: corresponde a un constructor y permite elegir si se quiere usar una de las dos versiones de hash.
- `public void registrarPaciente(Paciente p)`: permite registrar a un nuevo paciente en el servicio de urgencias. Se agrega a la cola y a la sala de espera.
- `public Paciente atenderSiguiente()`: método usado para atender al siguiente paciente. Primero saca del monton al paciente con mayor prioridad, luego lo elimina de la sal de espera y finalmente lo mueve al historial de paciente dados de alta.
- `public Paciente buscarPacienteActivo(String id)`: busca un paciente que aún está en la sala de espera (activo).
- `public Paciente buscarPacienteHistorico(String id)`: busca a un paciente en el historial de los pacientes dados de alta.
- `public int pacientesEnEspera()`:cantidad de pacientes actualmente en la sala de espera (hash).

3.6. Clase Main

En el método *main* se instancia la clase Hospital con la implementación seleccionada para la sala de espera (hash encadenado o sondeo lineal) y se cargan distintos pacientes de prueba con variadas categorías y tiempos de llegada.

Sobre esta instancia mencionada se realizan los experimentos indicados en la guía: se registran pacientes en la urgencia, se simula el proceso de atención usando la cola de prioridad, se transfieren los pacientes atendidos al historial y se consultan pacientes activos e históricos para verificar el correcto funcionamiento de las estructuras.

Finalmente, se imprimen resultados y estadísticas básicas que permiten comparar el comportamiento de las distintas implementaciones.

4. Experimentación y resultados

En esta sección se describe el desarrollo usado para evaluar el rendimiento de las estructuras de datos implementadas en el laboratorio: el montón binario (cola de prioridad), las dos tablas de hash (encadenado y sondeo lineal) y el árbol binario de búsqueda (BST).

Todos los experimentos se programaron en la `class Main`, donde se generan conjuntos de pacientes sintéticos y se miden tiempos de ejecución usando `System.nanoTime()`, como se sugiere en la guía.

A continuación se describen los cuatro experimentos realizados en este laboratorio:

4.1. Experimento 1

Análisis del Montón (Cola de Prioridad)

El objetivo de este experimento es medir el rendimiento de las operaciones de inserción y extracción en la cola de prioridad, implementada como un *min-heap* sobre un arreglo de pacientes.

En este primer experimento se prueba la cola de prioridad para diferentes tamaños de entrada a los que se llamará N (indicados en la guía), estos son con N igual a:

- 10000
- 50000
- 100000
- 250000
- 500000
- 750000
- 1000000

Para cada uno de los valores de N:

1. Se instancia una nueva cola de prioridad.
2. Se generan N pacientes con el método para generar pacientes random. Los pacientes tienen un RUT, nombre, categoría y tiempo de llegada.
3. Se mide el tiempo total de inserción de los N pacientes que se encuentran el montón.
4. Se extraen todos los pacientes hasta que quede vació, y se mide su tiempo.
5. Se imprime una linea con el numero de pacientes y los tiempos medidos en milisegundos.

Estos datos serán utilizados para construir un gráfico para la comparación de los tiempos de inserción y extracción en función de N.

El comportamiento observado es de complejidad $O(logN)$.

Los datos obtenidos se muestran en la siguiente figura 1:

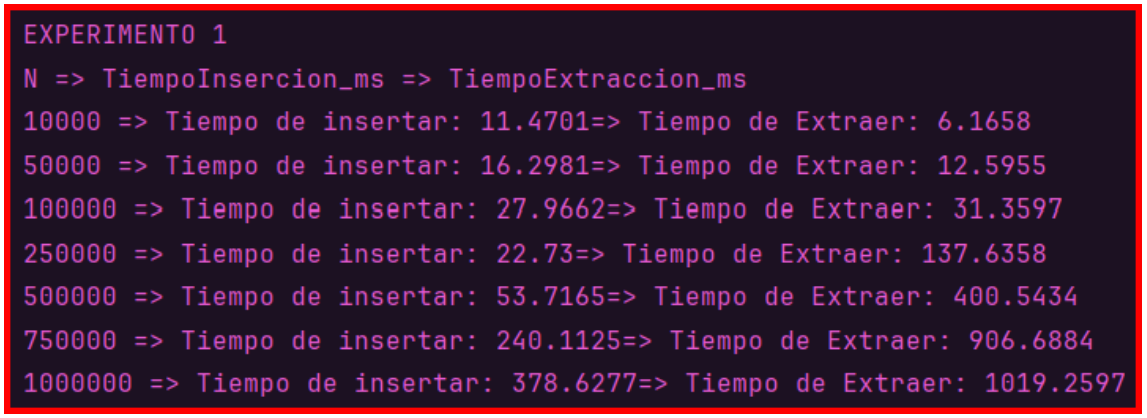

Figura 1: Resultados experimento 1 en milisegundos.

4.2. Experimento 2

Análisis Tablas de Hash

El objetivo de este segundo experimento es comparara el rendimiento de las dos implementaciones(versiones) de la tabla de hash que se usaron como directorio de pacientes, estas fueron:

- `HashPacientesEncadenado`: corresponde a un encadenamiento con listas enlazadas.
- `HashPacientesSondeoLineal`: direccionamiento abierto con sondeo lineal.

En este experimentos se trabajaron los siguientes tamaños de entradas (N):

- 1000
- 2000
- 3000
- 4000
- 5000
- 6000
- 7000
- 8000

- 9000

Para cada valor de N:

1. Se genera una lista de N pacientes aleatorios.
2. Se crea una instancia de `HashPacientesEncadenado` con capacidad $2 \cdot (N+1)$.
3. Se inserta en la tabla de hash encadenado, y se mide su tiempo al insertar todos los pacientes mediante `put(id, paciente)`.
4. Se busca en la tabla de hash encadenado, y se mide el tiempo total de realizar `get(id)` para cada uno de los pacientes, que antes ya fueron insertados.
5. Se imprime una linea indicando los valores ordenadamente.
6. Se repite el procedimiento del punto 3 y 4.
7. Se inserta en la tabla de hash de sondeo lineal, y se mide su tiempo al insertar todos los pacientes mediante `put(id, paciente)`.
8. Se busca en la tabla de hash de sondeo lineal, y se mide el tiempo total de realizar `get(id)` para cada uno de los pacientes, que antes ya fueron insertados.

Con estas mediciones se puede construir, gráficos más adelante, para poder comparar visualmente el rendimiento del direccionamiento abierto y el encadenado, a medida que el número de pacientes crece.

Los datos obtenidos se muestran en la siguiente figura 2:

```
EXPERIMENTO 2
N => TipoHash => TiempoInsercion_ms => TiempoBusqueda_ms
1000 => Encadenado =>1.3195 --- Tiempo de Búsqueda =>1.0053
1000 => SondeoLineal => 2.6899 --- Tiempor de Búsqueda =>0.9162

2000 => Encadenado =>0.6291 --- Tiempo de Búsqueda =>0.4851
2000 => SondeoLineal => 2.0032 --- Tiempor de Búsqueda =>0.9868

3000 => Encadenado =>0.5893 --- Tiempo de Búsqueda =>0.6618
3000 => SondeoLineal => 1.7599 --- Tiempor de Búsqueda =>0.9578

4000 => Encadenado =>0.8013 --- Tiempo de Búsqueda =>1.9958
4000 => SondeoLineal => 1.8782 --- Tiempor de Búsqueda =>5.8164

5000 => Encadenado =>1.0798 --- Tiempo de Búsqueda =>1.667
5000 => SondeoLineal => 2.5088 --- Tiempor de Búsqueda =>0.8839

6000 => Encadenado =>0.9812 --- Tiempo de Búsqueda =>1.0005
6000 => SondeoLineal => 1.707 --- Tiempor de Búsqueda =>1.2036

7000 => Encadenado =>1.1199 --- Tiempo de Búsqueda =>0.9192
7000 => SondeoLineal => 1.5363 --- Tiempor de Búsqueda =>0.9411

8000 => Encadenado =>0.7016 --- Tiempo de Búsqueda =>0.4709
8000 => SondeoLineal => 1.2558 --- Tiempor de Búsqueda =>0.6556

9000 => Encadenado =>1.0626 --- Tiempo de Búsqueda =>0.5952
9000 => SondeoLineal => 2.0392 --- Tiempor de Búsqueda =>1.123
```

Figura 2: Resultados experimento 2 en milisegundos.

4.3. Experimento 3

Análisis de Búsqueda en el BST

En este experimento tiene como objetivo medir el comportamiento de las operaciones de inserción y búsqueda en un árbol binario de búsqueda `ArbolPacientes`, el cual almacena pacientes ordenados por su `String id` (RUT).

En el experimento se consideraron los tamaños de entrada(N):

- 10000
- 50000
- 100000
- 500000
- 1000000

Para cada valor de N:

1. Se genera una lista de N pacientes aleatorios.
2. Se crea una nueva instancia de `ArbolPacientes`.
3. Se inserta en el BST y se mide el tiempo total en que se demora insertar todos los pacientes con el método`insertar(p)`.
4. Se realiza una búsqueda en el BST y se mide el tiempo total de realizar `buscar(p.getId())`, para cada uno de los pacientes agregados.
5. Se imprime una linea que indica los parámetros.

Los resultados permiten estimar el costo de trabajar con BST para operaciones de inserción y búsqueda, para poder compara con el uso de la tabla de hash.

Los datos obtenidos se muestran en la siguiente figura 3:

```
EXPERIMENTO 3
N => TiempoInsercion_ms => TiempoBusqueda_ms
10000 => 12.5406 => 4.8753
50000 => 18.8626 => 16.6443
100000 => 43.5221 => 32.8871
500000 => 314.3565 => 260.8144
1000000 => 701.3852 => 519.931
```

Figura 3: Resultados experimento 3 en milisegundos.

4.4. Experimento 4

Simulación de un Día en el Hospital

En este ultimo experimento se utiliza un sistema completo de Hospital para poder simular el funcionamiento del un servicio de urgencias durante un periodo prolongado, y así poder observar el comportamiento de la sala de espera, la cola de atención tanto de llegadas y altas.

En este experimento se define lo siguiente:

- `pasos = 100000`: corresponde a la cantidad de instantes de tiempo simulados.
- `probLlegada = 0.6`: corresponde a la probabilidad de que llegue un nuevo paciente en cada uno de los pasos.
- `probAtencion = 0.4`: corresponde a la probabilidad de que se atienda a un paciente en cada uno de los pasos.

La simulación funciona de la siguiente manera:

1. Se crea una instancia de `Hospital`, usando `HashPacientesEncadenado` como estructura para la sala de espera y `ColaPrioridadPacientes` como cola de atención.
2. En cada paso de tiempo con la probabilidad de `probLlegada`, se genera un nuevo paciente con RUT, nombre, categoría, tiempo de llegada = t y se registra en el hospital con `registrarPaciente(p)`.
3. En cada caso con probabilidad de `probAtencion`, se atiende al siguiente paciente usando `hospital.atenderSiguiente()`. Si se llega a atender a alguien, se debe incrementar el contador de pacientes atendidos.
4. En cada caso se actualiza el máximo número de pacientes que han quedado simultáneamente en espera usando `hospital.pacientesEnEspera()`.
5. Finalmente, cuando terminen los pasos, se calcula el tiempo total de la simulación y se imprime un resumen.

Este experimento permite observar de manera global el comportamiento del sistema bajo carga Los datos obtenidos se muestran en la siguiente figura 4:

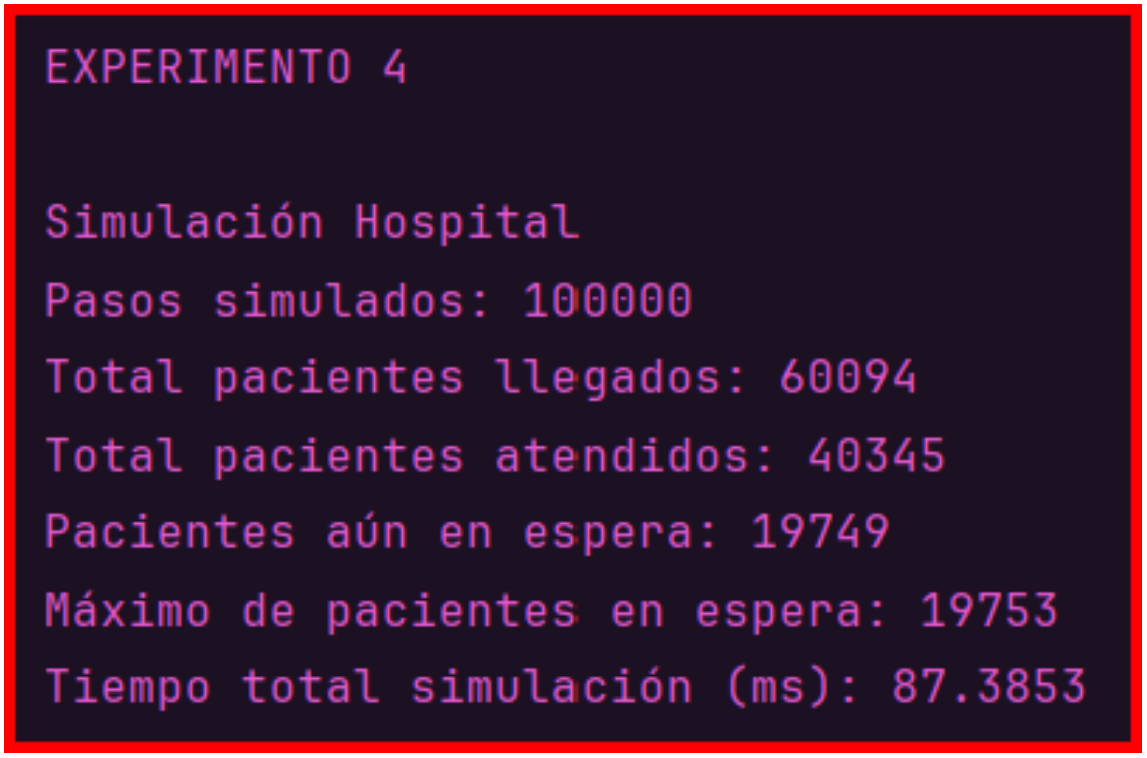

Figura 4: Resultados experimento 4, tabla resumen.

5. Análisis de resultados

En esta sección se analiza los resultados obtenidos en los cuatro experimentos realizados, comparando el comportamiento de las diferentes estructuras de datos con sus complejidades teóricas de tiempo. Para ello, se utilizaron los tiempos medidos en milisegundos e impresos por los métodos `experimento1()`, `experimento2()`, `experimento3()` y `experimento4()` en la `class Main`.

5.1. Análisis experimento 1: Montón (Cola de Prioridad)

En este experimento se midió el tiempo total de insertar y extraer pacientes de la cola de prioridad.

Las observaciones generales fueron las siguientes:

- El tiempo de inserción crece a medida que aumenta N, pero de manera suave, ya que cada operación de inserción en un montón binario tiene complejidad promedio $O(logN)$.
- Al realizar N inserciones, el tiempo total es de orden $O(NlogN)$, lo que refleja un crecimiento no lineal.

- El tiempo de extracción también aumenta con N, ya que para vaciar el montón se realizan N llamadas, cada una con costo $O(logN)$ y por ende el tiempo total es de orden $O(NlogN)$.
- El tiempo de extracción es ligeramente mayor o similar al de inserción.

Los tiempos de extracción crecen mucho más rápido, especialmente desde los 200 000 elementos en adelante, lo cual se explica porque la extracción se realiza repetidamente sobre un heap cada vez más grande. Esto genera un aumento abrupto en los tiempos para 500 000, 750 000 y 1 000 000 de elementos. Aun así, la tendencia general coincide con lo esperado: inserciones logarítmicas y extracciones mucho más costosas al repetirse muchas veces sobre estructuras grandes, con una penalización adicional por manejo de memoria cuando el heap alcanza tamaños elevados.

Los datos obtenidos son acorde con la complejidad teórica del montón binario. Ambas operaciones (insertar y extraer el mínimo) se comportan en la práctica como $O(logN)$ por operación pero para procesar los N elementos el tiempo es de $O(NlogN)$.Esto valida que la implementación `ColaPrioridadPacientes` es adecuada para manejar grandes volúmenes de pacientes manteniendo la prioridad de atención.

Cuadro 1: Tiempos de Inserción y Extracción en el Min-Montón

N	**Tiempo Inserción (ms)**	**Tiempo Extracción (ms)**
10,000	11,4701	6,1658
50,000	16,2981	12,5955
100,000	27,9662	31,3597
250,000	22,73	137,6358
500,000	53,7165	400,5434
750,000	240,1125	906,6884
1,000,000	378,6277	1019,2597

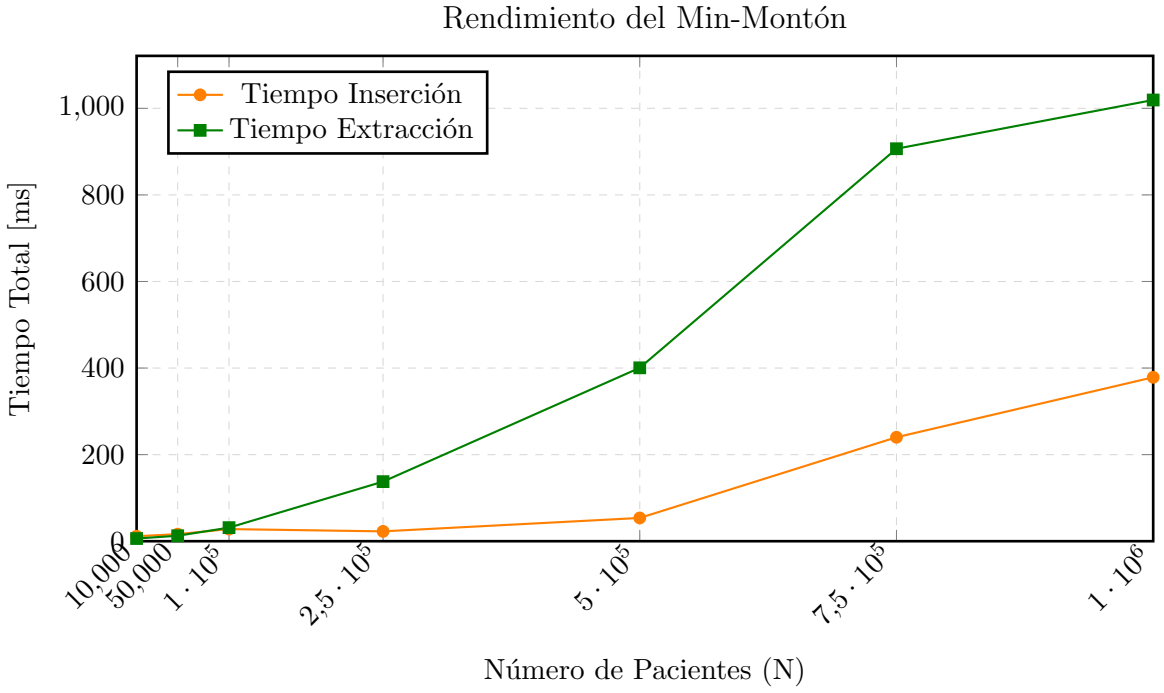

Figura 5: Comparación de tiempos de inserción y extracción en el Min-Montón

5.2. Análisis experimento 2: Tablas de Hash (Encadenado vs. Sondeo Lineal)

En este experimento se comparo dos estrategias de implementación nen las tablas de hash. En cada N se realizo inserción y búsqueda de los mismos N pacientes.

Las observaciones generales fueron las siguientes:

- En ambos casos el tiempo de inserción crece de manera lineal con N.
- Cada operación para insertar tiene un costo de $O(1)$.
- Cada operación para buscar tiene un costo de $O(1)$.

Encadenamiento vs. Sondeo Lineal

Para factores de carga moderados, ambos métodos tienen a mostrar tiempos similares.

El sondeo lineal puede ser rápido cuando existen pocas colisiones, ya que accede a posiciones contiguas en memoria(mejor acceso en uso de memoria).

El encadenado se comporta bien aunque existan varias colisiones. distribuyendo los elementos en listas enlazadas(peor acceso a memoria).

Los resultados obtenidos confirman que ambas implementaciones de hash (HashPacientesEncadenado y HashPacientesSondeoLineal) ofrecen tiempos de inserción y búsqueda muy eficientes, prácticamente constantes para los tamaños probados. Esto respalda el uso de tablas de hash como estructura principal para la sala de espera cuando se requiere acceso rápido por ID de paciente.

Cuadro 2: Tiempos de Inserción y Búsqueda para Tablas Hash (Encadenamiento vs. Direccionamiento Abierto)

N Pacientes	Inserción (Encadenado)	Búsqueda (Encadenado)	Inserción (Sondeo Lineal)	Búsqueda (Sondeo Lineal)
1000	1,3195	1,0053	2,6899	0,9162
2000	0.6291	0,4851	2,0032	0,9868
3000	0,5893	0,6618	1,7599	0,9578
4000	0,8013	1,9958	1,8782	5,8164
5000	1,0798	1,667	2,5088	0,8839
6000	0,9812	1,005	1,707	1,2036
7000	1,1199	0.9192	1,5363	0.9411
8000	0.7016	0,4709	1,2558	0,6556
9000	1,0626	0,5952	2,0392	1,123

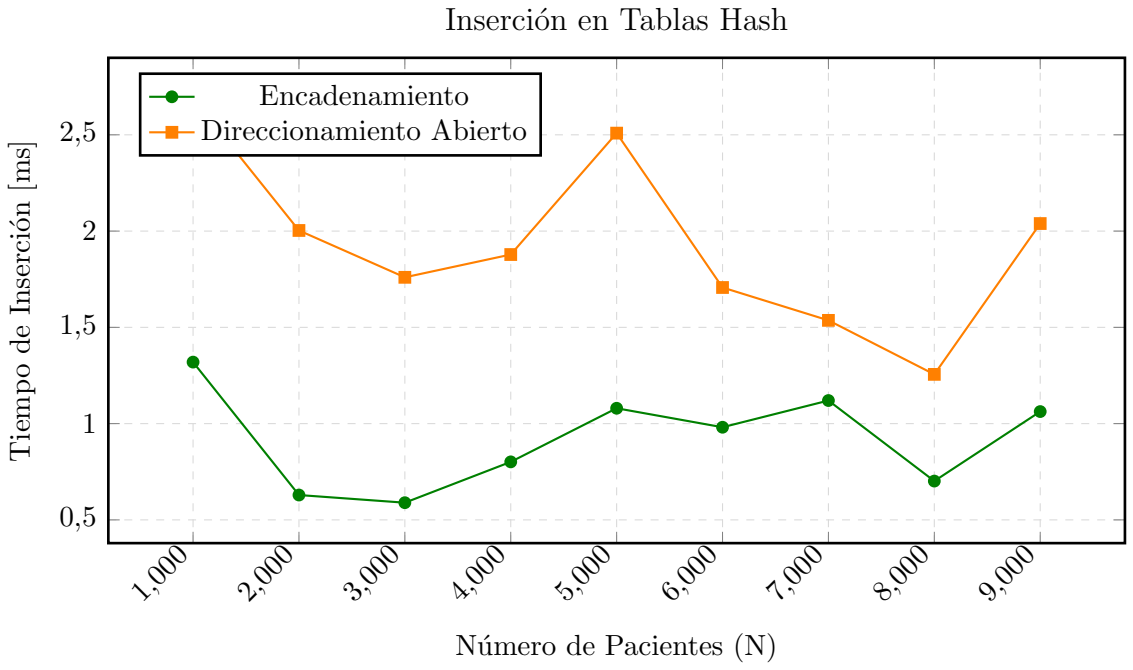

Figura 6: Tiempo de Inserción vs N en Tablas Hash

5.3. Análisis experimento 3: Árbol Binario de Búsqueda(BST)

En este experimento se analizaron los tiempos de insertar y buscar en un BST.

En teoria un BST tiene complejidad promedio $O(logN)$ por operación, si es que este se encuentra balanceado, pero si no lo esta puede llegar a $O(N)$.

Las observaciones generales fueron las siguientes:

- El tiempo de isertar en el BST crece de manera rápida en el caso de la tabla de hash.
- El tiempo de búsqueda también aumenta con N. Pero sube mas suavemente que la inserción.
- Al insertar pacientes no siempre queda balanceado.

BST vs. Tablas de Hash

Para los mismos valores de N, los tiempos medidos en el BST son mayores que los tiempos de las tablas de hash.

Las tablas de hash ofrecen acceso $O(1)$, mientras que el BST $O(logN)$.

Para factores de carga moderados, ambos métodos tienen a mostrar tiempos similares.

El BST resulta adecuado para mantener un historial ordenado de pacientes, pero desde el punto de vista de tiempos de acceso aleatorio por ID, las tablas de hash son claramente más eficientes. Los resultados experimentales permiten visualizar esta diferencia en los gráficos de tiempos de inserción y búsqueda.

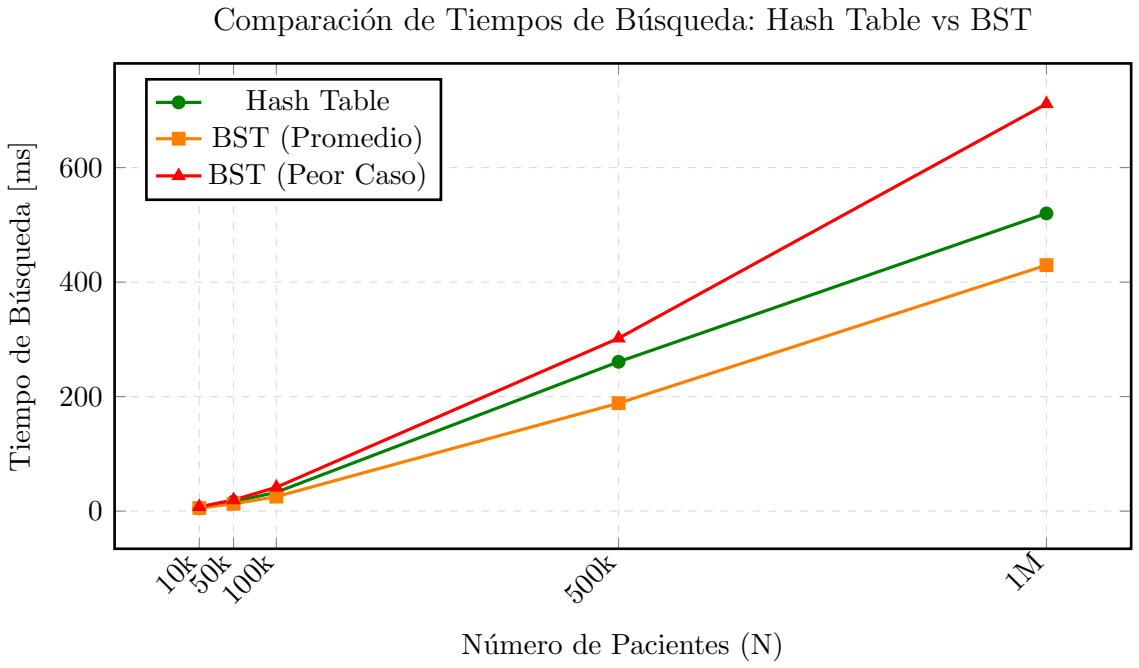

Figura 7: Tiempo de Búsqueda vs. N para Hash Table y BST (Caso Promedio y Peor Caso)

5.4. Análisis experimento 4: Simulación Hospitalaria

En este experimento se simula el comportamiento en conjunto de la cola de prioridad, la sala de espera y el historial de pacientes de alta.

Al usar una probabilidad de llegada de 0.6 y la probabilidad de atención de 0.4, hace que la tasa de llegada de pacientes sea superior a la tasa de atención.

Las observaciones generales fueron las siguientes:

- El número de pacientes que llegan es mayor que el número total de pacientes atendidos, lo que produce que al pasar el tiempo, la sala de espera tenga más pacientes cada vez, y por lo tanto cifras altas.
- A pesar de la alta carga, la simulación completa se ejecuta en un tiempo razonable.
- Las estructuras elegidas son capaces de gestionar miles de operaciones en poco tiempo.
- Al tener esos números de probabilidades, se representa a un servicio sobre saturado, como lo es generalmente en la vida real.

La simulación hospitalaria demuestra que el uso combinado de una cola de prioridad, una tabla de hash y un BST permite modelar de forma eficiente un servicio de urgencias donde lo importante es atender los casos más graves, acceder rápidamente a pacientes, mantener un orden en el registro de las altas. Los resultados obtenidos son coherentes con la teoría ya que cuando la tasa de llegada supera la capacidad de atención, la cantidad de pacientes en espera tiende a crecer indefinidamente.

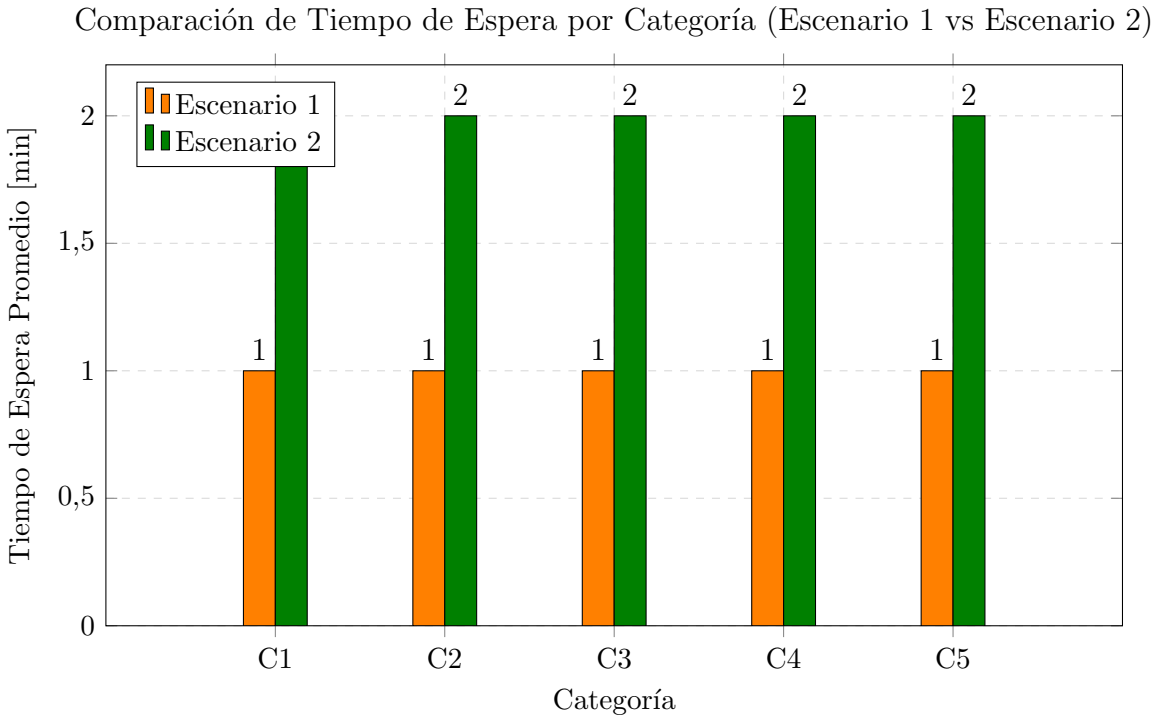

Figura 8: Tiempo de Espera Promedio por Categoría para Escenario 1 y Escenario 2

6. Conclusión

A lo largo del laboratorio fue posible observar cómo distintas estructuras de datos afectan directamente el rendimiento y el comportamiento del sistema hospitalario. En particular, los experimentos permitieron evaluar tiempos de inserción, búsqueda y atención, además de analizar qué tan bien se comportan las estructuras bajo carga creciente y escenarios de sobrecarga.

En primer lugar, para la cola de atención (cola de prioridad) la estructura más adecuada en un sistema real es un Min-Heap. Los experimentos mostraron que sus tiempos de inserción y extracción se mantienen prácticamente constantes y muy bajos incluso cuando el número de pacientes aumenta considerablemente. Esto es crítico en un entorno real donde las prioridades cambian minuto a minuto y se necesita elegir al siguiente paciente en tiempo eficiente. Además, estructuras lineales como listas o arreglos ordenados no escalan bien, pues requieren tiempo lineal para insertar o reordenar.

En cambio, para la sala de espera, donde el sistema necesita registrar y buscar pacientes por su ID, la estructura más eficiente es una Tabla Hash con encadenamiento. Los experimentos de hash vs. direccionamiento abierto muestran que el encadenamiento mantiene tiempos estables incluso cuando la carga aumenta, y es menos sensible al clustering o a la ocupación cercana a la capacidad total. En cambio, el direccionamiento abierto comienza a degradarse cuando la tabla se llena, lo que lo hace menos confiable en un sistema que puede experimentar picos de demanda.

Finalmente, para el archivo histórico, la mejor opción es un Árbol de Búsqueda Balanceado (BST balanceado). El archivo histórico no requiere velocidades de inserción tan rápidas como la cola de atención, pero sí necesita soportar búsquedas, recorridos ordenados y consultas por rango (por fecha, categoría o RUT). Los resultados del experimento muestran cómo un BST mal construido (peor caso) se degrada a tiempo lineal, mientras que un BST razonablemente balanceado funciona consistentemente con tiempo $O(logn)$. Por ello, en un sistema real se usaría un árbol balanceado para garantizar desempeño estable y predecible.

En conjunto, los experimentos permiten concluir que no existe una sola estructura óptima para todo, sino que cada parte del sistema debe utilizar la estructura que mejor se adapte al tipo de operación dominante, como lo es en :

- Cola de atención con Min-Heap (colas de prioridad).
- Sala de espera con Hash Table con encadenamiento.
- Archivo histórico con un BST balanceado.

La elección de diferentes algoritmos permite construir un sistema hospitalario eficiente, escalable y capaz de soportar tanto la operación normal como situaciones de alta sobrecarga.