



UNIVERSIDAD DIEGO PORTALES

Laboratorio 2: Listas, Colas y Pilas

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 del sistema de mensajería Slok, un sistema que utiliza 3 tipo de estructuras de datos como listas enlazadas, pilas (stacks) y colas (queues) para la gestión eficiente de mensajes y canales. El sistema permite la creación de mensajes asociados a canales específicos, manteniendo un historial de mensajes mediante el uso de una pila que facilita el acceso a los últimos mensajes creados. Asimismo, se ha implementado una lista enlazada para gestionar los mensajes reportados, que actúa como una cola para el almacenamiento y procesamiento de estos. Además, se ha utilizado una estructura de lista enlazada para gestionar los canales de manera eficiente. A lo largo del desarrollo, se han calculado los requerimientos de memoria para almacenar los mensajes según distintas restricciones de tamaño, permitiendo así optimizar el uso de recursos. También se ha propuesto una extensión al sistema para integrar la gestión de usuarios, asociando cada mensaje a un usuario específico, lo que implica cambios en las estructuras ya existentes.

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 Clase base: Message.

Para dar inicio a este programa se crea la clase `Message` esta clase es la base para el sistema la cual tiene como atributos: `id`, `content`, `reported`, `channelName` y `next`. También se crea su constructor de tipo `Message` donde `reported` y `next` se inicializan en `null`.

```
1  class Message{
2  private int id; // todo mensaje tiene id único.
3  private String content; //contenido del mensaje
4  private boolean reported; //true = reportado, false = no reportado.
5  public String channelName; // Canal donde se envió el mensaje.
6  public Message next; //next para lista enlazada tipo Message.
7
8  public Message(String content, String channelName, int id){
9      this.content = content;
10     this.channelName = channelName;
11     this.id = id;
12     this.reported = false;
13     this.next = null;
14 }
```

Listing 1: Clase y constructor `Message`.

Así también se crea un constructor de copiado el cual nos servirá para poder acceder a los atributos directamente de una instancia de tipo `Message` en las siguientes clases.

```
1  // Constructor de copiado.
2  public Message(Message m) {
3      this.id = m.id;
4      this.content = m.content;
5      this.channelName = m.channelName;
6      this.reported = m.reported;
7      this.next = null;
8  }
```

Listing 2: Constructor de copiado `Message`.

Luego se crean los métodos de `Message` donde:

`report()`: reporta el mensaje.

`IsReported()`: retorna si está reportado el mensaje.

`getContent()`: retorna el contenido del mensaje.

`setNext(Message m)`: recibe un mensaje y lo modifica.

`getId()`: retorna el ID del mensaje.

```
1 public void report(){this.reported = true;}
2
3 public boolean IsReported(){return this.reported;}
4
5 public String getContent(){return this.content;}
6
7 public void setNext(Message m){this.next = m;}
8
9 public int getId(){return this.id;}
```

Listing 3: Métodos de `Message`.

2.2. Listas Enlazadas: Clase Channel.

Ahora se crea la clase `Channel` es por donde se enviarán los mensajes tiene como atributos: `name`, `headMessage`, `next` y `tailMessage`.

También se crea su constructor dónde `headMessage`, `next` y `tailMessage` se inicializan en `null`.

```
1 class Channel{
2     public String name; //Nombre del canal.
3     private Message headMessage; //head de la lista enlazada.
4     public Channel next; // next de lista enlazada tipo channel.
5     private Message tailMessage; // tail = ultimo mensaje.
6
7
8     Channel(String name){
9         this.name = name;
10        this.headMessage = null;
11        this.next = null;
12        this.tailMessage = null;
13    }
```

Listing 4: Clase y constructor de `Channel`.

Una vez se crea la base de la clase, se implementa los métodos el primero de ellos es `appendMessage(Message m)` el cual recibe un mensaje y lo añade al final de la lista enlazada, en este caso lo implementé con head y un tail así mejorando la eficiencia en términos de bigO puesto que al poder manejar dos nodos distintos no es necesario hacerlo con un ciclo.

```
1 public void appendMessage(Message m){
2     Message newMessage = new Message(m);
3     if (headMessage == null){
4         headMessage = newMessage;
5         tailMessage = newMessage;
6     } else {
7         tailMessage.next = newMessage;
8         tailMessage = newMessage;
9     }
10 }
```

Listing 5: Método `appendMessage` de `Channel`.

Luego se implementa el método `search(int id)` este recibe el ID de un mensaje y busca el mensaje que le corresponde si lo encuentra retorna el mensaje sino retorna null. Para listas enlazadas se necesita crear un `auxiliar` para así recorrer la lista enlazada sin modificar la original.

```
1 public Message search(int id){
2     Message aux = this.headMessage;
3     while (aux != null){
4         if (aux.getId() == id){
5             return aux;
6         }
7         aux = aux.next;
8     }
9     return null;
10 }
```

Listing 6: Método `search` de `Channel`.

Para finalizar, se implementan los métodos:

`empty()`: retorna si la lista de mensajes está vacía o no.

`setNext()`: recibe un canal y lo modifica.

`getHead()`: retorna el primer mensaje de la lista.

`getTail()`: retorna el último mensaje de la lista.




```
1 public boolean empty(){return this.headMessage == null;}
2
3     public void setNext(Channel c){this.next = c;}
4
5     public Message getHead(){return this.headMessage;}
6
7     public Message getTail(){return this.tailMessage;}
```

Listing 7: Métodos `empty`, `setNext`, `getHead` y `getTail` de `Channel`.

2.3. Stack: Clase History.

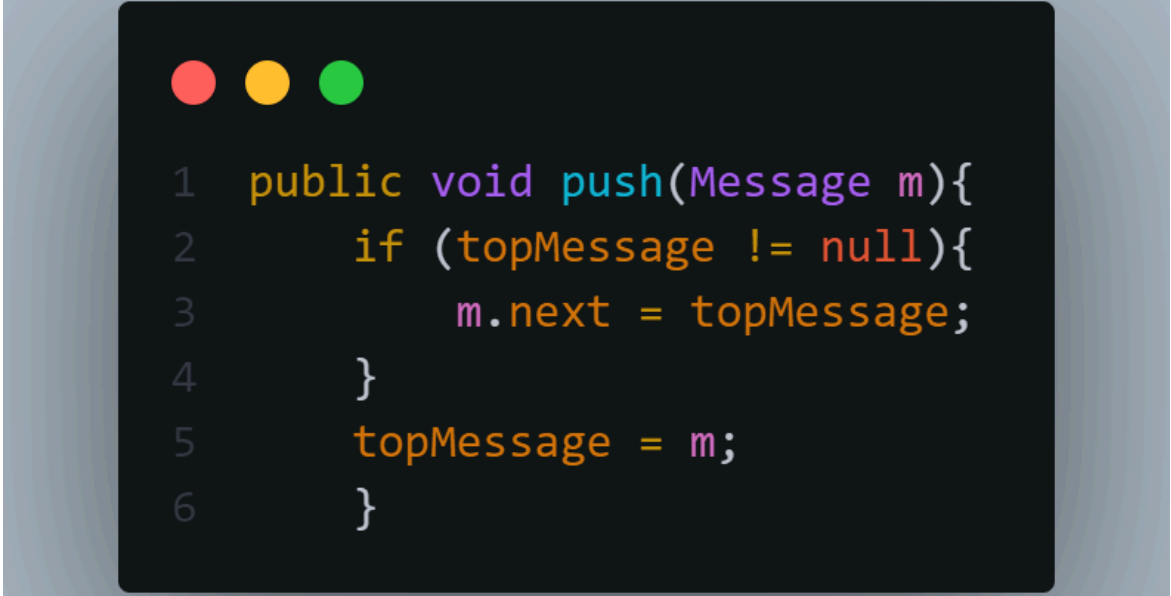
Para crear la clase `History` solo se necesita un atributo de tipo `Message`: `topMessage`, el cual se inicializa en el constructor en `null`.



```
1 class History {
2     private Message topMessage;
3     History (){
4         this.topMessage = null;
5     }
}
```

Listing 8: Clase y constructor `History`.

Luego se tienen que implementar los métodos correspondientes para una estructura de tipo LIFO (Last in : First Out), el primero de ellos es `push(Message m)` que recibe un mensaje verifica si no está vacío, luego referencia al siguiente nodo y lo agrega al historial.



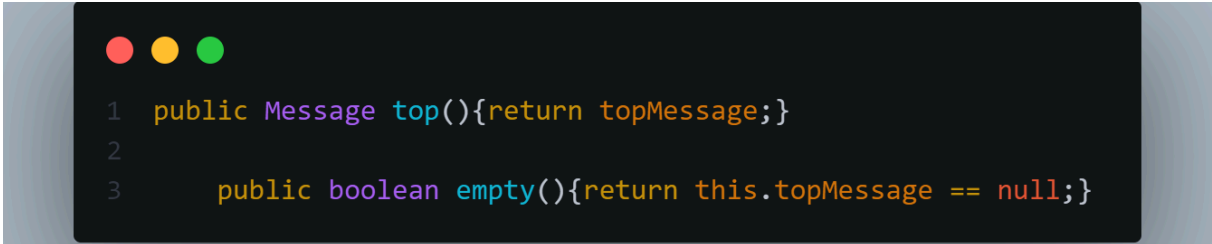
```
1 public void push(Message m){
2     if (topMessage != null){
3         m.next = topMessage;
4     }
5     topMessage = m;
6 }
```

Listing 9: Método `push` de `History`.

En segundo lugar se agregan los métodos:

`top()` = retorna el último mensaje agregado del historial.

`empty()` = retorna si el historial está vacío o no.



```
1 public Message top(){return topMessage;}
2
3 public boolean empty(){return this.topMessage == null;}
```

Listing 10: Método `top` y `empty` de `History`.

Finalmente el último de los métodos es `lastKMessages(int k)`, que recibe el número de últimos mensajes `k` del historial. Primero, se crea un auxiliar para recorrer la pila y contar el número total de mensajes. Si la pila está vacía (`topMessage` es `null`), se retorna un array vacío. Si `k` es mayor que el número total de mensajes, se ajusta `k` para que sea igual al tamaño del historial. Luego, se crea un array de tamaño `k` y se llena con los últimos `k` mensajes del historial, recorriendo la pila desde `topMessage` y agregando los mensajes al array. Finalmente, se retorna el array con los últimos `k` mensajes.

```
1 public Message[] lastKMessages(int k) {
2     if (topMessage == null) { // retorna el array vacío.
3         return new Message[0];
4     }
5     Message aux = this.topMessage; // crea un auxiliar de topMessage.
6     int counter = 0;
7
8     while (aux != null) {
9         counter++;
10        aux = aux.next;
11    }
12    if (k > counter) { // Ajusta k segun el tamaño del stack.
13        k = counter;
14    }
15    Message[] arrhist = new Message[k];
16    aux = this.topMessage;
17    for (int i = 0; i < k; i++) { // Agrega los últimos mensajes al array.
18        arrhist[i] = aux;
19        aux = aux.next;
20    }
21    return arrhist;
22 }
```

Listing 11: Método `lastKMessages` de `History`.

2.4. Queue: Clase ReportList.

Para implementar la clase `ReportList` se debe crear 2 atributos de tipo `Message`: `firstMessage`, `lastMessage`. Estos métodos se inicializan en el constructor en `null`.

```
1 class ReportList{
2     private Message firstMessage;
3     private Message lastMessage;
4
5     ReportList(){
6         this.firstMessage = null;
7         this.lastMessage = null;
8     }
```

Listing 12: Clase y constructor `ReportList`.

Luego de crear los atributos y el constructor, se pueden implementar los métodos. El primero de ellos es `enqueue(Message m)`, que debe verificar si el mensaje ha sido reportado. Si el mensaje no está reportado, el método no hace nada y retorna. Si el mensaje está reportado, se crea una nueva instancia del mensaje y se añade a la cola. Antes de agregar el nuevo mensaje, se verifica si la cola está vacía; si es así, el nuevo mensaje se convierte en el primer y último mensaje de la cola. Si la cola ya contiene mensajes, el nuevo mensaje se añade al final de la cola, actualizando la referencia del último mensaje para que apunte al nuevo mensaje.

```
1 public void enqueue(Message m){
2     if (m.IsReported() == false){
3         return;
4     }
5     Message newMessage = new Message(m);
6     if (firstMessage == null){
7         firstMessage = lastMessage = newMessage;
8     } else {
9         lastMessage.next = newMessage;
10        lastMessage = newMessage;
11    }
12 }
```

Listing 13: Método `enqueue` de `ReportList`.

El segundo de los métodos es `firstKReportedMessages(int k)`, que es similar al método del Listing 11, pero con diferencias importantes. Este método trabaja con una estructura FIFO (First in : First Out), recorriendo desde el primer mensaje ingresado. A diferencia del otro método, este verifica si cada mensaje ha sido reportado y ajusta `k` basado en el número de mensajes reportados encontrados.

```
1  public Message[] firstKReportedMessages(int k){
2      if (firstMessage == null) {
3          return new Message[0];
4      }
5      Message aux = firstMessage;
6      int counter = 0;
7
8      while (aux != null && counter < k) {
9          if (aux.IsReported()) {
10             counter++;
11         }
12         aux = aux.next;
13     }
14     if (k > counter) {
15         k = counter;
16     }
17
18     Message[] reportedMessages = new Message[k];
19     aux = firstMessage;
20     int index = 0;
21
22     while (aux != null && index < k) {
23         if (aux.IsReported()) {
24             reportedMessages[index] = aux;
25             index++;
26         }
27         aux = aux.next;
28     }
29     return reportedMessages;
30 }
```

Listing 14: Método `firstKReportedMessages` de `ReportList`.

Por último los métodos finales que se implementaron fueron:
`first()`: retorna el primer mensaje ingresado en la cola.
`empty()`: retorna si la cola de reportes está vacía.

```
1 public Message first(){return this.firstMessage;}
2
3 public boolean empty(){return this.firstMessage == null;}
```

Listing 15: Método `firstKReportedMessages` de `ReportList`.

2.5. Main: Clase `Slok`.

Para terminar se crea la clase `Slok`, esta es la principal del programa ya que en este será donde se manejan los métodos anteriormente creados, esta clase tiene atributos de tipo `Channel`, `History`, `ReportList`: `headChannel`, `history`, `reportList` y `idCounter`, respectivamente. Donde en su constructor `headChannel` se inicializa en `null`, se instancia: `historial` y lista de reportes, y se inicializa el contadorID en 1.

```
1 public class Slok {
2     private Channel headChannel;
3     private History history;
4     private ReportList reportList;
5     private int idCounter;
6
7     // Constructor de Slok
8     public Slok() {
9         this.headChannel = null;
10        this.history = new History();
11        this.reportList = new ReportList();
12        this.idCounter = 1;
13    }
```

Listing 16: Clase y constructor `Slok`.

Luego de crear los atributos y su constructor, se puede empezar a crear los métodos el primero de ellos es `reportMessage(int id, String channelName)` recibe el ID y el canal del mensaje a reportar, Si el mensaje se encuentra, se marca como reportado. Luego, se crea una copia del mensaje y se agrega a la lista de reportes. Si el mensaje no se encuentra en el canal especificado o en la lista de canales, se imprime un mensaje indicando que el mensaje no fue encontrado.

```
1 public void reportMessage(int id, String channelName) {
2     Channel channel = headChannel;
3     while (channel != null) {
4         if (channel.name == channelName) {
5             Message message = channel.search(id);
6             if (message != null) {
7                 message.report();
8                 reportList.enqueue(message);
9                 return;
10            }
11        }
12        channel = channel.next;
13    }
14    System.out.println("Mensaje no encontrado.");
15 }
```

Listing 17: Método `reportMessage` de `Slok`.

El segundo método es `createMessage(String content, String Channel)` este crea un nuevo mensaje con el contenido y el nombre del canal proporcionados. Primero, busca el canal que coincide. Si encuentra el canal, crea un nuevo mensaje con el contenido, el nombre del canal y un ID único. Luego, añade el mensaje al canal utilizando `appendMessage()`, lo guarda en el historial y actualiza el contador de IDs. Si el canal no se encuentra, imprime un mensaje indicando que el canal no fue encontrado.

```
1 public void createMessage(String content, String channelName) {
2     Channel channel = headChannel;
3     while (channel != null) {
4         if (channel.name.equals(channelName)) {
5             Message newMessage = new Message(content, channelName, idCounter);
6             channel.appendMessage(newMessage);
7             history.push(newMessage);
8             incrementIDCounter();
9             return;
10        }
11        channel = channel.next;
12    }
13    System.out.println("Canal no encontrado.");
14 }
```

Listing 18: Método `createMessage` de `Slok`.

El tercer método es `createChannel(String name)` crea un nuevo canal con el nombre proporcionado. Si no hay canales en la lista, el nuevo canal se establece como el primer canal en la lista. Si ya existen canales, el método recorre la lista hasta encontrar el último canal y añade el nuevo canal al final de la lista.

```
1 public void createChannel(String name) {
2     Channel newChannel = new Channel(name);
3     if (headChannel == null) {
4         headChannel = newChannel;
5     } else {
6         Channel aux = headChannel;
7         while (aux.next != null) {
8             aux = aux.next;
9         }
10        aux.next = newChannel;
11    }
12 }
```

Listing 19: Método `createChannel` de `Slok`.

Al final se implementan los métodos:

`incrementIDCounter()` : cada vez que se utiliza incrementa en 1 el contador.

`getIDCounter()`: retorna el valor del contador.

```
1 public void incrementIDCounter() { this.idCounter++; }
2     public int getIDCounter() { return this.idCounter; }
```

Listing 20: Método `incrementIDCounter` y `getIDCounter` de `Slok`.

Así mismo después se pueden llamar todos estos métodos en el main, creando mensajes genéricos e ir probando diferentes situaciones para comprobar el correcto funcionamiento del programa de mensajería.

```
1 public static void main(String[] args) {
2
3     Slok slok = new Slok();
4     slok.createChannel("General");
5     slok.createChannel("General_2");
6     slok.createMessage("Hola que tal, todo bien?", "General");
7     slok.createMessage("Todo bien y tu crack?", "General");
8     slok.createMessage("Muy bien", "General");
9     slok.createMessage("HOLA", "General");
10    slok.createMessage("alo", "General_2");
11    slok.createMessage("si", "General_2");
12    slok.createMessage("no", "General_2");
13    slok.createMessage("HOLA", "General_2");
14    slok.reportMessage(4, "General");
15    slok.reportMessage(8, "General_2");
16    slok.reportMessage(5, "General_2");
17
18    Message[] firstReportedMessages = slok.reportList.firstKReportedMessages(5);
19    System.out.println("Primeros mensajes reportados:");
20    for (Message message : firstReportedMessages) {
21        System.out.println("Mensaje: " + message.getContent() + " ID: " + message.getId());
22    }
23
24    int k = 6;
25    Message[] lastMessages = slok.history.lastKMessages(k);
26    System.out.println("Últimos " + k + " mensajes:");
27    for (Message message : lastMessages) {
28        System.out.println(message.getContent());
29    }
30 }
```

Listing 21: Main.

3. Análisis:

1. Cree una tabla analizando la complejidad de tiempo teórica, con la notación $O(f(n))$, de cada método implementado.

Clase	Método	Complejidad Temporal
Message	report	$O(1)$
Message	IsReported	$O(1)$
Message	getContent	$O(1)$
Message	setNext	$O(1)$
Message	getId	$O(1)$
Channel	appendMessage	$O(1)$
Channel	search	$O(n)$
Channel	empty	$O(1)$
Channel	setNext	$O(1)$
Channel	getHead	$O(1)$
Channel	getTail	$O(1)$
History	push	$O(n)$
History	top	$O(1)$
History	empty	$O(1)$
History	lastKMessages	$O(n)$
ReportList	enqueue	$O(1)$
ReportList	firstKReportedMessages	$O(n)$
ReportList	first	$O(1)$
ReportList	empty	$O(1)$
Slok	reportMessage	$O(n)$
Slok	createMessage	$O(n)$
Slok	createChannel	$O(n)$
Slok	incrementIDCounter	$O(1)$
Slok	getIDCounter	$O(1)$

Tabla 1: Complejidad Big Oh.

2. Puesto que el ID de cada mensaje es asignado en base al atributo `idCounter`, ¿cuál es la cantidad máxima de mensajes que pueden ser creados en *Slok* (sin generar problemas)?

Para calcular la cantidad máxima de mensajes que pueden ser creados en *Slok*, se debe saber el rango máximo de un entero en Java (en bytes), este se puede saber con la clase `Integer.MAX_VALUE` y el rango es: 2,147,483,647 por lo tanto es la cantidad máxima de mensajes que se podrían crear sin generar problemas. Entonces para cada tamaño de mensaje en bytes, se debe multiplicar el tamaño del mensaje por la cantidad máxima de mensajes para obtener la memoria total en bytes.

Bytes	Memoria Necesaria (Bytes)	MB
128	$2.147.483.647 * 128 =$ $274.877.906.816 \div 1.048.576 =$	262.144
256	$2.147.483.647 * 256 =$ $549.755.813.888 \div 1.048.576 =$	524.288
512	$2.147.483.647 * 512 =$ $1.099.511.627.776 \div 1.048.576 =$	1.048.576
1024	$2.147.483.647 * 1024 =$ $2.199.023.255.552 \div 1.048.576 =$	2.097.152
2024	$2.147.483.647 * 2024 =$ $4.341.869.783.488 \div 1.048.576 =$	4.145.344

Tabla 2: Cantidad Máxima de mensajes (MB).

3. Proponga cómo se podría modificar el sistema para agregar gestión de usuarios a *Slok*. Los usuarios deben estar asociados a los mensajes que creen.

Para agregar gestión de usuarios a *Slok*, se podría crear una clase llamada `User` que represente a cada usuario del sistema. Esta clase tendría atributos como `username`, `userID`, y un historial de mensajes creados por el usuario. Además, cada mensaje debería estar vinculado al usuario que lo creó, lo cual se lograría con el `userID` en la clase `Message`, permitiendo identificar qué usuario envió cada mensaje. La clase `User` podría tener métodos para retornar el `username` y el `userID`, además de gestionar los mensajes creados por el usuario. También se podría implementar una lista enlazada que mantenga un registro de todos los usuarios en el sistema. Los usuarios podrían reportar mensajes, y, si es necesario, podrían tener permisos para crear nuevos canales.

4. Ventajas y desventajas de la implementación propuesta:

Ventaja 1: Una de las grandes ventajas de esta implementación es la organización y claridad que tiene cada uno de los métodos, por lo tanto es fácil de entender y se puede notar fácilmente como fue desarrollado.

Ventaja 2: Tiene una gran eficiencia, puesto que el máximo complejidad teórica (BIG Oh) fue de $O(n)$ lo que apunta a que la implementación fue correcta.

Desventaja 1: La implementación usa listas enlazadas para manejar mensajes y canales, lo que puede complicar la gestión a gran escala y limitar la eficiencia en la organización a medida que aumenta el número de elementos.

Desventaja 2: La información se mantiene únicamente en memoria RAM, por lo que se perderá al cerrar o reiniciar el programa, y no hay mecanismos para asegurar la recuperación de datos en caso de fallos del sistema.

4. Conclusión:

En conclusión, el sistema Slok ha demostrado ser una solución eficiente para la gestión de mensajes y canales, gracias a la implementación de estructuras de datos como listas enlazadas, pilas y colas. Estas estructuras permiten manejar de forma óptima la creación, almacenamiento y reporte de mensajes, asegurando un acceso rápido a los últimos mensajes mediante el historial y un procesamiento eficiente de los mensajes reportados. El análisis de la memoria requerida para almacenar mensajes según distintas restricciones de tamaño también fue clave para comprender las limitaciones del sistema y optimizar su rendimiento. Además, la propuesta de agregar gestión de usuarios expande las funcionalidades del sistema, ofreciendo la posibilidad de asociar mensajes a usuarios específicos y mejorar la trazabilidad y control en la plataforma. En general, el uso adecuado de las estructuras de datos de diferentes tipos como listas enlazadas, pilas y colas. Garantizan que Slok sea más grande y adaptable a futuras mejoras.