

Tecnológico Nacional de México

Campus Orizaba

ESTRUCTURA DE DATOS

Carrera:

Ingeniería en sistemas computacionales

Tema 3: Estructuras Lineales

Alumnos:

Tezoco Cruz Pedro-20011328

Flores Domínguez Ángel Gabriel-21010951

Grupo:3g2B

Fecha 22/04/23

Introducción

En la programación, una estructura de datos es una forma de organizar y almacenar información en una computadora para que pueda ser utilizada de manera eficiente. Las estructuras de datos lineales son un tipo de estructura que se caracteriza por tener un orden secuencial en sus elementos, es decir, cada elemento tiene un sucesor y un predecesor en la estructura.

En esta práctica, exploraremos las estructuras de datos lineales en el lenguaje de programación Java. Específicamente, nos enfocaremos en tres tipos de estructuras lineales: listas enlazadas, pilas y colas. Aprenderemos cómo implementar cada una de estas estructuras en Java, cómo realizar operaciones básicas como agregar y eliminar elementos, y cómo utilizarlas en la resolución de problemas.

Es importante destacar que cada estructura lineal tiene sus propias características y ventajas, lo que las hace adecuadas para diferentes situaciones y problemas. Por lo tanto, es fundamental entender los conceptos y principios detrás de cada estructura para poder elegir la adecuada para cada caso.

En resumen, esta práctica tiene como objetivo familiarizarnos con las estructuras de datos lineales en Java, aprender cómo implementarlas y utilizarlas para resolver problemas prácticos.

Competencia específica:

Comprende y aplica estructuras de datos lineales para la solución de problemas.

Marco Teórico:

3.0. Clases genéricas, arreglos dinámicos (memoria dinámica)

Las clases genéricas son una característica de la programación orientada a objetos que permite definir una clase que puede trabajar con diferentes tipos de datos, sin tener que definir una clase diferente para cada tipo. Las clases genéricas se definen mediante la especificación de un tipo de variable genérico, que se sustituye por el tipo real en el momento de la creación del objeto.

La programación genérica es útil porque permite la reutilización de código y aumenta la seguridad del tipo en tiempo de compilación. Los errores de tipo de tiempo de ejecución se reducen significativamente cuando se utilizan clases genéricas. Además, la programación genérica mejora la legibilidad del código, ya que los tipos de datos están explícitamente indicados.

Java ofrece una implementación de clases genéricas con los parámetros de tipo, que permiten a los programadores escribir código genérico. Los parámetros de tipo se utilizan en la definición de una clase o método, y se sustituyen por un tipo real en el momento de la creación de una instancia de la clase o al llamar al método.

Arreglos dinámicos (memoria dinámica):

Los arreglos dinámicos, también conocidos como memoria dinámica, son una forma de asignar memoria en tiempo de ejecución para un arreglo. A diferencia de los arreglos estáticos, que tienen un tamaño fijo, los arreglos dinámicos permiten cambiar su tamaño durante la ejecución del programa.

En Java, los arreglos dinámicos se implementan mediante la clase `ArrayList`. La clase `ArrayList` es una lista dinámica que se expande automáticamente cuando se agrega un nuevo elemento y se encoge cuando se elimina un elemento.

Los arreglos dinámicos son útiles en situaciones en las que no se sabe de antemano el tamaño de un arreglo o cuando se necesita agregar o eliminar elementos de forma dinámica. Los arreglos dinámicos también proporcionan una forma eficiente de acceder a los elementos del arreglo mediante un índice.

Sin embargo, los arreglos dinámicos también tienen algunas desventajas. En comparación con los arreglos estáticos, los arreglos dinámicos son menos eficientes en términos de uso de memoria y de rendimiento. Además, los arreglos dinámicos pueden requerir más tiempo de procesamiento para agregar o eliminar elementos, ya que se necesita cambiar el tamaño del arreglo.

3.1. Pilas

Una pila es una estructura de datos lineal en la que los elementos se agregan y eliminan en la misma posición, conocida como la cima de la pila. Es una estructura de datos LIFO (Last-In-First-Out), lo que significa que el último elemento que se agregó a la pila es el primero en salir.

En Java, una pila se puede implementar utilizando la clase `Stack`, que proporciona métodos para agregar elementos a la cima de la pila, eliminar elementos de la cima de la pila y buscar elementos en la pila. La clase `Stack` es una subclase de la clase `Vector`, por lo que tiene todas las capacidades de un vector dinámico.

3.1.1. Representación en la memoria

En la memoria, una pila se representa como una región contigua de memoria, donde cada elemento se almacena en una dirección de memoria consecutiva y la cima de la pila se representa como un puntero a la dirección de memoria del último elemento agregado.

En una implementación típica de pila, se utiliza un array para almacenar los elementos de la pila y se utiliza un puntero para indicar la cima de la pila. El puntero apunta a la última posición ocupada en el array, es decir, la posición del último elemento que se agregó a la pila. Cuando se agrega un nuevo elemento a la pila, se incrementa el puntero de la cima de la pila para apuntar a la siguiente posición libre en el array y se almacena el nuevo elemento en esa posición. Cuando se elimina un elemento de la pila, se decrementa el puntero de la cima de la pila para apuntar al elemento anterior en el array y se devuelve el elemento eliminado.

3.1.2. Operaciones básicas

Las operaciones básicas que se pueden realizar con una pila son las siguientes:

`push()`: Agrega un elemento a la cima de la pila.

`pop()`: Elimina el elemento que se encuentra en la cima de la pila y devuelve su valor.

`peek()`: Devuelve el valor del elemento que se encuentra en la cima de la pila sin eliminarlo.

`isEmpty()`: Devuelve verdadero si la pila está vacía, de lo contrario, devuelve falso.

`size()`: Devuelve la cantidad de elementos que hay en la pila.

3.1.3. Aplicaciones

Las pilas tienen varias aplicaciones en la programación y se utilizan en muchos algoritmos y estructuras de datos. A continuación, se presentan algunas de las aplicaciones más comunes de las pilas:

Evaluación de expresiones aritméticas: Las pilas se utilizan para evaluar expresiones aritméticas como las que se utilizan en las calculadoras. Los operandos se colocan en una pila y los operadores se aplican a ellos en orden de precedencia.

Recursión: Las pilas se utilizan en la implementación de algoritmos recursivos. Cada vez que se llama a una función recursiva, se almacena una copia de los parámetros y variables locales en una pila.

Manejo de llamadas a funciones: Las pilas se utilizan para almacenar la información necesaria para el manejo de las llamadas a funciones. Cuando se llama a una función, se almacena la dirección de retorno y los parámetros en la pila.

Historial de navegación: Las pilas se utilizan en los navegadores web para almacenar el historial de navegación. Cada vez que se visita una página web, se agrega una entrada a la pila. Si se presiona el botón de retroceso, se elimina la entrada más reciente de la pila.

Implementación de algoritmos de búsqueda y recorrido de árboles: Las pilas se utilizan para implementar algoritmos de búsqueda y recorrido de árboles como la búsqueda en profundidad o el recorrido en profundidad primero.

Validación de paréntesis: Las pilas se utilizan para validar la sintaxis de expresiones que contienen paréntesis. Cada vez que se encuentra un paréntesis de apertura, se agrega a la pila. Cada vez que se encuentra un paréntesis de cierre, se elimina el paréntesis de apertura correspondiente de la pila. Si la pila está vacía al final del proceso, la sintaxis es correcta.

3.2. Colas

Las colas son una estructura de datos en la que los elementos se agregan al final y se eliminan del frente, lo que sigue el principio de "primero en entrar, primero en salir" (FIFO, por sus siglas en inglés). En Java, las colas se implementan como una interfaz llamada Queue en la biblioteca estándar de Java (Java Standard Library).

Las colas son una estructura de datos útil y versátil en la programación que se utiliza en una variedad de aplicaciones. En Java, las colas se implementan como una interfaz y existen varias implementaciones disponibles en la biblioteca estándar de Java.

3.2.1. Representación en memoria

En Java, una cola se puede representar en memoria utilizando una estructura de datos llamada "array circular" o una "lista enlazada".

En una estructura de datos de array circular, los elementos de la cola se almacenan en un array. En lugar de asignar espacio para cada elemento de la cola, se utiliza un puntero "front" y un puntero "rear" que indican el inicio y el final de la cola, respectivamente. Al agregar un elemento a la cola, se agrega al final del array (posición indicada por "rear") y "rear" se mueve al siguiente espacio disponible en el array. Cuando se elimina un elemento de la cola, se elimina el elemento en la posición "front" y "front" se mueve al siguiente elemento en la cola.

En una estructura de datos de lista enlazada, cada elemento de la cola se representa como un nodo que contiene el valor del elemento y un puntero al siguiente nodo en la cola. El puntero "front" apunta al primer nodo en la cola y el puntero "rear" apunta al último nodo en la cola. Al agregar un elemento a la cola, se crea un nuevo nodo con el valor del elemento y se establece el puntero "next" del último nodo en la cola al nuevo nodo. Luego, "rear" se actualiza para apuntar al nuevo nodo. Cuando se elimina un elemento de la cola, se elimina el primer nodo en la cola y "front" se actualiza para apuntar al siguiente nodo en la cola.

3.2.2. Operaciones básicas:

Las operaciones básicas con las colas son:

Enqueue: agrega un elemento al final de la cola.

Dequeue: elimina y devuelve el primer elemento de la cola.

Peek: devuelve el primer elemento de la cola sin eliminarlo.

Size: devuelve el número de elementos en la cola.

IsEmpty: comprueba si la cola está vacía.

3.2.3. Tipos de colas

Existen varios tipos de colas que se pueden utilizar en diferentes aplicaciones. Los principales tipos de colas son:

Colas simples: también conocidas como colas FIFO (First In, First Out), son la forma más común de cola. En una cola simple, los elementos se agregan al final de la cola y se eliminan del frente de la cola.

Colas circulares: en una cola circular, el último elemento de la cola está conectado con el primer elemento para formar un bucle. De esta manera, cuando se agrega un nuevo elemento al final de la cola, el puntero de "rear" se mueve al principio de la cola.

Colas dobles: también conocidas como dequeues, permiten agregar y eliminar elementos tanto al frente como al final de la cola.

Colas con prioridad: en una cola con prioridad, los elementos se organizan según su prioridad, y el elemento con mayor prioridad se elimina primero. Esto se puede implementar utilizando una estructura de datos como un montículo binario.

3.2.4. Aplicaciones

Las colas son una estructura de datos útil que se utiliza en muchas aplicaciones diferentes, algunas de las cuales incluyen:

Sistemas operativos: los sistemas operativos a menudo utilizan colas para programar procesos y administrar recursos compartidos. Por ejemplo, cuando varios procesos solicitan acceso a un recurso compartido, el sistema operativo puede utilizar una cola para asignar el recurso a los procesos en orden de llegada.

Algoritmos de búsqueda: los algoritmos de búsqueda como BFS (Breadth-First Search) utilizan colas para almacenar los nodos que aún no se han visitado. De esta manera, los nodos se visitan en orden de su profundidad en el árbol de búsqueda.

Colas de impresión: las colas se utilizan comúnmente en los sistemas de impresión para manejar múltiples trabajos de impresión en orden de llegada. Los trabajos de impresión se agregan a la cola y se imprimen en orden de llegada.

Procesamiento de datos: en muchas aplicaciones de procesamiento de datos, como procesamiento de imágenes o procesamiento de señales, las colas se utilizan para almacenar los datos que aún no se han procesado.

Simulaciones: las colas se utilizan comúnmente en simulaciones de eventos discretos para modelar eventos que ocurren en orden de tiempo. Por ejemplo, en una simulación de tráfico, los vehículos se agregan a una cola y se mueven por una carretera en orden de llegada.

3.3. Listas

Las listas en Java son una estructura de datos dinámica y flexible que permite almacenar y manipular un conjunto de elementos en un orden determinado. A diferencia de los arreglos estáticos, las listas pueden crecer y disminuir de tamaño según sea necesario, lo que las hace ideales para aplicaciones que requieren una cantidad variable de elementos.

En Java, las listas se pueden implementar utilizando la interfaz List. La interfaz List define un conjunto de operaciones básicas que se pueden realizar en una lista, como agregar elementos, eliminar elementos, buscar elementos y ordenar elementos. Hay varias clases que implementan la interfaz List en Java, como ArrayList, LinkedList y Vector.

3.3.1 Operaciones básicas

Las operaciones básicas que se pueden realizar con las listas en Java incluyen:

Agregar elementos: se puede agregar elementos al final de la lista utilizando el método `add()`. También es posible agregar elementos en una posición específica de la lista utilizando el método `add(index, element)`.

Eliminar elementos: se puede eliminar elementos de la lista utilizando el método `remove()`. Este método puede eliminar un elemento específico o el elemento en una posición específica.

Obtener elementos: se puede obtener un elemento específico de la lista utilizando el método `get()`. Este método devuelve el elemento en una posición específica.

Reemplazar elementos: se puede reemplazar un elemento específico de la lista utilizando el método `set()`. Este método cambia el valor del elemento en una posición específica.

Tamaño de la lista: se puede obtener el tamaño de la lista utilizando el método `size()`. Este método devuelve el número de elementos en la lista.

Verificar si la lista está vacía: se puede verificar si la lista está vacía utilizando el método `isEmpty()`. Este método devuelve `true` si la lista no tiene elementos.

Búsqueda de elementos: se puede buscar un elemento específico en la lista utilizando el método `indexOf()`. Este método devuelve la posición del elemento en la lista. Si el elemento no está en la lista, devuelve `-1`.

Ordenar elementos: se puede ordenar los elementos de la lista utilizando el método `sort()`. Este método ordena los elementos de la lista en orden ascendente.

3.3.2. Tipos de listas

En Java, hay tres tipos principales de listas: listas simplemente enlazadas, listas doblemente enlazadas y listas circulares. Cada tipo de lista tiene sus propias ventajas y se puede elegir según las necesidades específicas de la aplicación.

Las listas simplemente enlazadas son la forma más básica de lista en Java. Cada elemento de la lista tiene un enlace que apunta al siguiente elemento en la lista. La lista comienza con un elemento especial llamado "cabeza", que apunta al primer elemento de la lista.

Las listas doblemente enlazadas, por otro lado, tienen dos enlaces para cada elemento: uno que apunta al elemento anterior y otro que apunta al elemento siguiente. Esto permite una navegación más fácil en ambas direcciones.

Las listas circulares, como su nombre lo indica, tienen un enlace que apunta al primer elemento de la lista al final de la lista. Esto crea un bucle, lo que significa que la lista se puede recorrer continuamente en un bucle.

3.3.3 Aplicaciones

Las listas son una estructura de datos muy versátil y se utilizan en una amplia variedad de aplicaciones en Java. Algunas de las aplicaciones más comunes incluyen:

Manejo de colecciones de datos: Las listas son una forma conveniente de almacenar y manipular colecciones de objetos en Java. Las listas se pueden utilizar para almacenar una gran cantidad de datos en una estructura fácilmente accesible.

Implementación de estructuras de datos complejas: Las listas se pueden utilizar para implementar estructuras de datos más complejas como árboles, grafos y mapas.

Implementación de algoritmos de búsqueda y ordenación: Las listas son una estructura de datos ideal para implementar algoritmos de búsqueda y ordenación. Los algoritmos de búsqueda y ordenación se utilizan en una amplia variedad de aplicaciones, desde motores de búsqueda hasta sistemas de gestión de bases de datos.

Implementación de pilas y colas: Las listas también se pueden utilizar para implementar pilas y colas. En el caso de las pilas, se pueden agregar y eliminar elementos solo en la parte superior de la pila. En el caso de las colas, se pueden agregar elementos al final y eliminar elementos solo del frente de la cola.

3.2 colas

Las colas en Java son una estructura de datos fundamental que se utiliza para almacenar y administrar elementos en un orden específico. Una cola es una colección de elementos que se mantienen en orden secuencial y se procesan en el mismo orden en que se agregaron a la cola.

Las colas se utilizan comúnmente en situaciones en las que se necesita procesar elementos en un orden específico, como en sistemas de gestión de tareas, procesamiento de mensajes y programación concurrente. En Java, las colas se

implementan utilizando la interfaz Queue y sus subinterfaces, como Deque y PriorityQueue. Estas interfaces proporcionan una serie de métodos para agregar, eliminar y consultar elementos en la cola. Además, las colas en Java también son compatibles con el enfoque de programación orientado a objetos, lo que significa que se pueden crear objetos de cola y tratarlos como entidades independientes en el código.

En resumen, las colas en Java son una estructura de datos fundamental que proporciona un enfoque ordenado y secuencial para almacenar y procesar elementos. Son esenciales para muchas aplicaciones de programación y se utilizan comúnmente en situaciones en las que se requiere procesamiento de datos en un orden específico.

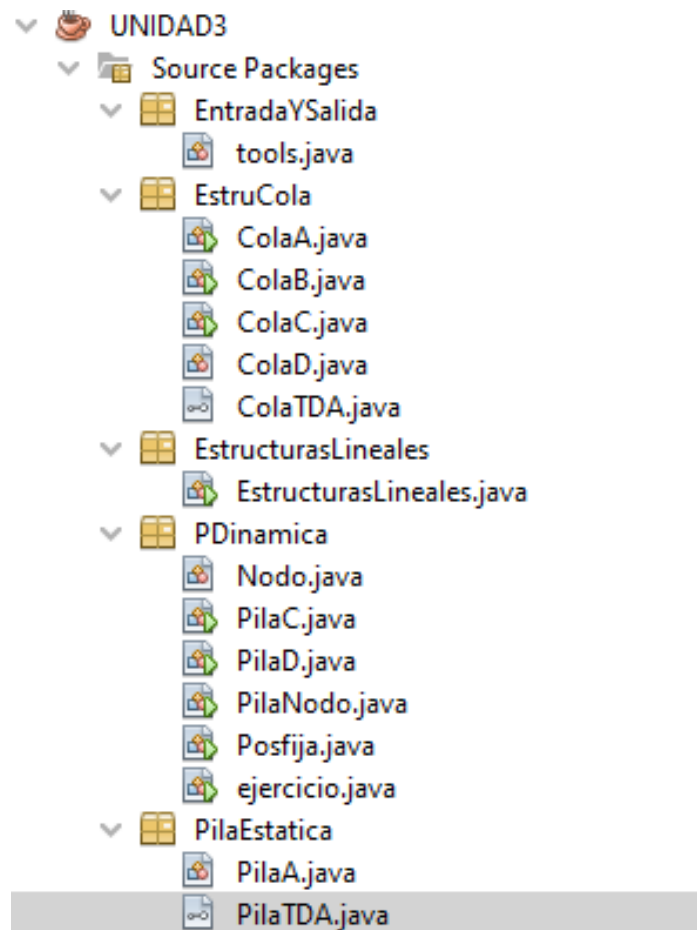
Material y equipo.

- Computadora
- Netbeans
- Word
- Chrome

Desarrollo de la practica.

Primero iniciamos con pilas, empezamos creando un proyecto en el cual agregamos unos paquetes, uno llamado pila dinámica y la otra llamada pila estática, primero diremos que es la pila Las pilas en Java son como una pila de platos en la cocina: cada vez que añades un plato, se apila encima del anterior. En programación, las pilas funcionan de la misma manera: los elementos se apilan uno encima del otro y sólo se puede acceder al elemento en la parte superior de la pila. Puedes agregar elementos nuevos a la pila y quitar elementos existentes de la misma forma que agregar o quitar platos de la pila de la cocina. ¡Es una estructura de datos muy útil en programación!, después empezamos a ver acerca de las colas, creamos un paquete llamado estrucola, en el cual creamos diferentes clases dentro de ellas

Resultados.



```
Source History
1 package PilaEstatica;
2
3 public interface PilaTDA <T> {
4     public boolean isemptyPila();//registra true si la pila esta vacia.
5     public void pushPila(T dato);// inserta el dato en el tope de la pila.
6     public T popPILA();//elimina el elemento que esta en el tope de la pila.
7     public T peekpila ();//regresa el elemento que esta en el tope, sin quitarlo.
8
9
```

Este código es una interfaz en Java llamada "PilaTDA" que define cuatro métodos para operar en una pila genérica.

El primer método "isemptyPila" se utiliza para verificar si la pila está vacía o no, y devuelve true si la pila está vacía. El segundo método "pushPila" se utiliza para insertar un elemento en la parte superior de la pila.

El tercer método "popPILA" se utiliza para eliminar el elemento que está en la parte superior de la pila.

El cuarto método "peekpila" se utiliza para obtener el elemento que está en la parte superior de la pila, pero sin eliminarlo.

Al utilizar esta interfaz para implementar una pila en Java, se pueden utilizar estos métodos para insertar, eliminar y consultar elementos en la pila.

```
1 package PilaEstatica;
2
3 import EntradaYSalida.tools;
4
5 public class PilaA <T> implements PilaTDA <T> {
6     private T pila [];
7     private byte tope;
8
9     public PilaA (int max) {
10         pila =(T[]) (new Object[max]);
11         tope=-1;
12     }
13
14     @Override
15     public boolean isemptyPila() {
16         return (tope==-1);
17     }
18
19     public boolean isSpace() {
20         return (tope<pila.length-1);
21     }
22
23     @Override
24     public void pushPila(T dato) {
25         if (isSpace()) {
26             tope++;
27             pila[tope]=dato;
28         }
29         else
```

```

30         tools.errorMsj("PILA LLENA!!!!");
31     }
32
33     @Override
34     public T popPILA () {
35         T dato = pila [tope];
36         tope--;
37         return dato;
38     }
39
40     @Override
41     public T peekpila() {
42         return pila [tope];
43     }
44
45     @Override
46     public String toString() {
47         return toString(tope);
48     }
49
50     private String toString(int i) {
51         return (i >= 0) ? "tope==>" + i + "[" + pila [i] + "]\n" + toString (i-1): "";
52     }
53 }

```

```

1 package EstructurasLineales;
2 import EntradaYSalida.tools;
3 import PilaEstatica.PilaA;
4 import PilaEstatica.PilaTDA;
5 import javax.swing.JOptionPane;
6 public class EstructurasLineales {
7
8     public static void Menu() {
9
10         PilaA <Integer> pila = new PilaA ((byte)10);
11         String op="";
12         do{
13             op=tools.boton("PUSH,POP,PEEK,FREE,SALIR");
14             switch(op){
15                 case "PUSH":
16                     pila.pushPila(tools.leeInt("escribe un dato entero: "));
17                     tools.imprime("datos de la pila: \n" + pila.toString());
18                     break;
19                 case "POP":
20                     if (pila.isemptyPila())tools.errorMsj("pila Vacía");
21                     else
22                         tools.imprime("dato eliminado de la cima de la pila: " + pila.popPILA()+ "\n"+ pila.toString());
23                     break;
24                 case "PEEK":
25                     if (pila.isemptyPila())tools.errorMsj("pila vacía");
26                     else
27                         tools.imprime("dato de la cima de la pila: ==>" + pila.peekpila()+ "\n"+ pila.toString());
28                     break;
29                 case "FREE":

```

Este código implementa una pila en Java utilizando un arreglo estático. La clase PilaA<T> implementa la interfaz PilaTDA<T>, que define los métodos que se pueden utilizar para manipular la pila. La clase tiene un arreglo pila que almacena los elementos de la pila y un índice tope que indica la posición del último elemento insertado en la pila.

El método isEmptyPila() verifica si la pila está vacía y devuelve un valor booleano true en caso de que así sea. El método pushPila(T dato) inserta el dato en la pila si hay espacio disponible y actualiza el valor de tope. El método popPILA() elimina y devuelve el elemento que está en el tope de la pila. El método peekpila() devuelve el elemento que está en el tope de la pila sin eliminarlo.

El método toString() devuelve una cadena con todos los elementos de la pila. En el método Menu() se muestra un menú con opciones para realizar operaciones con la pila, como insertar (PUSH), eliminar (POP), obtener el elemento en el tope (PEEK) y limpiar la pila (FREE). El método main() llama al método Menu() para iniciar la interacción con el usuario.

```
8      public class PilaC<T> implements PilaTDA<T> {  
9          private ArrayList pila;  
10         int tope;  
11  
12         public PilaC() {  
13             pila= new ArrayList();  
14             tope=-1;  
15         }  
16  
17         public int Size() {  
18             return pila.size();  
19         }  
20     }  
21 }
```



```
18         return pila.size();
19     }
20
21     @Override
22     public boolean isemptyPila() {
23         return pila.isEmpty();
24     }
25
26     public void vaciar () {
27         pila.clear();
28     }
29
30     @Override
31     public void pushPila (Object dato) {
32         pila.add(dato);
33         tope++;
34     }
35
36     @Override
37     public int Size() {
38         return pila.size();
39     }
40
41     @Override
42     public boolean isemptyPila() {
43         return pila.isEmpty();
44     }
45
46     public void vaciar () {
47         pila.clear();
48     }
49
50     @Override
51     public void pushPila (Object dato) {
52         pila.add(dato);
53         tope++;
54     }
55
56     @Override
57     public T popPILA() {
58         T dato= (T) pila.get(tope);
59         pila.remove(tope);
60         tope--;
61         return dato;
62     }
```

```

42     }
43
44     @Override
45     public T peekpila() {
46         return (T)pila.get(tope);
47     }
48
49     @Override
50     public String toString() {
51         return toString(tope);
52     }
53     private String toString(int i) {
54         return (i>=0)?"tope==>" + i + "[" + pila.get(i)+"\n" + toString(i-1):"";
55     }
56 }
57

```

```

60 public static void Menu() {
61     PilaC pila= new PilaC ();
62     String dato,op;
63     do{
64         op=tools.boton("PUSH,POP,PEEK,FREE,SALIR");
65         switch(op){
66             case "PUSH":
67                 pila.pushPila(tools.leeInt("escribe un dato entero: "));
68                 tools.imprime("datos de la pila: \n" + pila.toString());
69                 break;
70             case "POP":
71                 if (pila.isEmptyPila())tools.errorMsj("pila Vacia");
72                 else
73                     tools.imprime("dato eliminado de la cima de la pila: " + pila.popPILA()+ "\n"+ pila.toString());
74                 break;
75             case "PEEK":
76                 if (pila.isEmptyPila())tools.errorMsj("pila vacia");
77                 else
78                     tools.imprime("dato de la cima de la pila: ==>" + pila.peekpila()+ "\n"+ pila.toString());
79                 break;
80             case "FREE":
81                 if (pila.isEmptyPila())tools.errorMsj("pila vacia");
82                 else

```

```

83         {
84             pila.vaciar();
85             pila = new PilaC();
86         }
87         break;
88     }
89 }
90 while(!op.equals("SALIR"));
91 }
92 public static void main(String[] args) {
93     Menu();
94 }
95 }

```

Este código define una implementación de una Pila (stack) en Java, utilizando una ArrayList para almacenar los elementos. La clase se llama **PilaC** y implementa la interfaz **PilaTDA**, que define los métodos básicos que debe tener una Pila.

El atributo **pila** es un ArrayList que almacena los elementos de la pila, y el atributo **tope** es un entero que representa el índice del elemento más recientemente agregado.

En el constructor **PilaC**, se inicializa la lista **pila** y se asigna **-1** a **tope**, lo que indica que la pila está vacía.

Luego se definen los métodos básicos para manipular la pila:

- **Size()** devuelve el tamaño de la pila, es decir, la cantidad de elementos que contiene.
- **isemptyPila()** devuelve **true** si la pila está vacía, y **false** en caso contrario.
- **vaciar()** vacía la pila, eliminando todos los elementos.
- **pushPila(dato)** agrega el elemento **dato** a la pila, incrementando el valor de **tope**.
- **popPILA()** elimina y devuelve el elemento en el tope de la pila, decrementando el valor de **tope**.
- **peekpila()** devuelve el elemento en el tope de la pila, sin eliminarlo.
- **toString()** devuelve una representación en forma de cadena de la pila, mostrando todos los elementos en orden, desde el tope hasta el fondo.

Finalmente, se define un método **Menu()** que utiliza la clase **tools** (que no se muestra en el código) para crear un menú interactivo que permite al usuario agregar, eliminar y ver elementos de la pila. El método **main()** simplemente llama a este método para iniciar el programa.

```
1 package PDinamica;
2 import EntradaYSalida.tools;
3 import javax.swing.*;
4 import java.util.Stack;
5 import java.util.EmptyStackException;
6 public class PilaD {
7
8     public static void Menu() {
9         Stack pila= new Stack();
10        String dato,op;
11        do{
12            op=tools.boton("PUSH,POP,PEEK,FREE,SALIR");
13            switch(op){
14                case "PUSH":
15                    pila.push(tools.leeInt("escribe un dato entero: "));
16                    tools.imprime("datos de la pila: \n" + pila.toString());
17                    break;
18                case "POP":
19                    if (pila.empty())tools.errorMsj("pila Vacia");
20                    else
21                        tools.imprime("dato eliminado de la cima de la pila: " + pila.pop()+ "\n"+ pila.toString());
22                    break;
23                case "PEEK":
24                    if (pila.empty())tools.errorMsj("pila vacia");
25                    else
26                        tools.imprime("dato de la cima de la pila: ==>" + pila.peek()+ "\n"+ pila.toString());
27                    break;
28                case "FREE":
29                    if (pila.empty())tools.errorMsj("pila vacia");
30
31                else
32                {
33                    pila=null;
34                    pila = new Stack();
35                }
36                break;
37            }
38        }
39        while(!op.equals("SALIR"));
40    }
41
42    public static void main(String[] args) {
43        Menu();
44    }
45 }
```

Este código implementa una pila dinámica utilizando la clase **Stack** de Java. La pila se utiliza a través de un menú de opciones, que permite al usuario realizar las siguientes operaciones:

- PUSH: insertar un elemento en la pila.
- POP: eliminar el elemento de la cima de la pila.
- PEEK: consultar el elemento de la cima de la pila.

- **FREE**: vaciar la pila completamente.

El código utiliza la clase **tools** para realizar la entrada y salida de datos, a través de ventanas de diálogo en el entorno de Java Swing.

La implementación de la pila se realiza mediante un objeto de la clase **Stack**. En la opción **PUSH**, se utiliza el método **push()** de la clase **Stack** para insertar un elemento en la pila. En la opción **POP**, se utiliza el método **pop()** de la clase **Stack** para eliminar el elemento de la cima de la pila. En la opción **PEEK**, se utiliza el método **peek()** de la clase **Stack** para consultar el elemento de la cima de la pila.

En la opción **FREE**, se vacía la pila utilizando el método **clear()** de la clase **Stack**, y se crea un nuevo objeto de la clase **Stack**.

Cada vez que se realiza una operación en la pila, se muestra su contenido actual utilizando el método **toString()** de la clase **Stack**.

En resumen, este código implementa una pila dinámica mediante la clase **Stack** de Java, y permite al usuario interactuar con ella a través de un menú de opciones.



```
1 package PDinamica;
2
3
4 public class Nodo <T> {
5     public T info;
6     public Nodo sig;
7
8     public Nodo (T info){
9         this.info=info;
10        this.sig=null;
11    }
12
13    public T getInfo() {
14        return info;
15    }
16
17    public void setInfo(T info) {
18        this.info = info;
19    }
20
21    public Nodo getSig() {
22        return sig;
23    }
24
25    public void setSig(Nodo sig) {
26        this.sig = sig;
27    }
28 }
```



```
1 package PDinamica;
2
3 import EntradaYSalida.tools;
4 import static PDinamica.PilaD.Menu;
5 import PilaEstatica.PilaTDA;
6 import java.util.Stack;
7
8 public class PilaNodo <T> implements PilaTDA<T>{
9     private Nodo pila;
10    public PilaNodo() {
11        pila=null;
12    }
13
14    @Override
15    public void pushPila(T dato) {
16        Nodo tope = new Nodo(dato);
17        if (isemptyPila()) pila=tope;
18        else{
19            tope.sig=pila;
20            pila=tope;
21        }
22    }
23    @Override
24    public boolean isemptyPila() {
25        return (pila== null);
26    }
27    public void vaciar () {
28        pila = null;
29    }
```

```

30  @Override
31  public T peekpila() {
32      return (T) (pila.getInfo());
33  }
34  @Override
35  public T popPILA() {
36      Nodo tope= pila;
37      T dato= (T) pila.getInfo();
38      pila=pila.getSig();
39      tope=null;
40      return dato;
41  }
42  @Override
43  public String toString() {
44      Nodo tope=pila;
45      return toString(tope);
46  }
47  private String toString(Nodo i) {
48      return (i!=null)?"TOPE ==>" + "[" + i.getInfo()+"] \n"+ toString(i.getSig()):"";
49  }
50
51
52
53  public static void Menu() {
54      PilaNodo pila= new PilaNodo();
55      String dato,op;
56      do{
57          op=tools.boton("PUSH, POP, PEEK, FREE, SALIR");
58          switch (op) {
59              case "PUSH":
60                  pila.pushPila(tools.leeInt("escribe un dato entero: "));
61                  tools.imprime("datos de la pila: \n" + pila.toString());
62                  break;
63              case "POP":
64                  if (pila.isEmptyPila())tools.errorMsj("pila vacia");
65                  else
66                      tools.imprime("dato eliminado de la cima de la pila: " + pila.popPILA()+ "\n"+ pila.toString());
67                  break;
68              case "PEEK":
69                  if (pila.isEmptyPila())tools.errorMsj("pila vacia");
70                  else
71                      tools.imprime("dato de la cima de la pila: ==>" + pila.peekpila()+ "\n"+ pila.toString());
72                  break;
73              case "FREE":
74                  if (pila.isEmptyPila())tools.errorMsj("pila vacia");
75                  else
76                      {
77                          pila.vaciar();
78                          pila = new PilaNodo();
79                      }
80                  break;
81          }
82      }
83      while (!op.equals("SALIR"));
84  }
85
86  public static void main(String[] args) {
87      Menu();

```


El código define una implementación de una Pila Dinámica utilizando una estructura de datos basada en nodos enlazados. En particular, la clase **PilaNodo** implementa la interfaz **PilaTDA** que define los métodos básicos de una pila (push, pop, peek, isempty) y también define un método **toString** para poder imprimir la pila en un formato legible.

El método **pushPila** agrega un nuevo elemento a la pila, creando un nuevo nodo con el dato pasado como argumento y estableciéndolo como el nuevo tope de la pila. El método **isemptyPila** verifica si la pila está vacía o no. El método **peekpila** devuelve el elemento en el tope de la pila sin sacarlo de la pila. El método **popPILA** devuelve el elemento en el tope de la pila y lo saca de la pila.

El método **toString** recorre la pila desde el tope hasta el fondo y devuelve una cadena que representa los elementos en la pila en orden inverso (ya que la pila es una estructura LIFO - Last-In-First-Out).

El método **Menu** implementa un menú de opciones para interactuar con la pila, permitiendo al usuario agregar elementos a la pila (push), sacar elementos de la pila (pop), ver el elemento en el tope de la pila sin sacarlo (peek), vaciar la pila (free) y salir del programa (salir). El programa utiliza la clase **tools** para manejar la entrada y salida de datos, y al llamar al método **Menu** se crea una nueva instancia de la clase **PilaNodo** para utilizar como pila.

```

1 package PDinamica;
2
3 import javax.swing.*;
4 import javax.swing.table.DefaultTableModel;
5
6 public class Posfija<T> extends JFrame {
7
8     private JTextField txtEntrefija;
9     private JTable tblDatos;
10    private DefaultTableModel model;
11    private PilaNodo<Character> pila = new PilaNodo<Character>();
12
13    public Posfija() {
14        setTitle("Convertidor de entrefija a posfija");
15        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16        setSize(500, 300);
17        setLocationRelativeTo(null);
18        getContentPane().setLayout(null);
19
20        JLabel lblEntrefija = new JLabel("Expresión entrefija:");
21        lblEntrefija.setBounds(20, 20, 120, 20);
22        getContentPane().add(lblEntrefija);
23
24        txtEntrefija = new JTextField();
25        txtEntrefija.setBounds(150, 20, 200, 20);
26        getContentPane().add(txtEntrefija);
27
28        JButton btnConvertir = new JButton("Convertir");
29        btnConvertir.setBounds(370, 20, 100, 20);

```

```
30         getContentPane().add(btnConvertir);
31
32         tblDatos = new JTable();
33         model = new DefaultTableModel();
34         model.addColumn("Entrefija");
35         model.addColumn("Pila");
36         model.addColumn("Posfija");
37         tblDatos.setModel(model);
38         JScrollPane scrollPane = new JScrollPane(tblDatos);
39         scrollPane.setBounds(20, 60, 450, 180);
40         getContentPane().add(scrollPane);
41
42         btnConvertir.addActionListener(event -> convertir());
43
44         setVisible(true);
45     }
46
47     private void convertir() {
48         String entrefija = txtEntrefija.getText();
49         String posfija = "";
50
51         model.setRowCount(0);
52         for (int i = 0; i < entrefija.length(); i++) {
53             char c = entrefija.charAt(i);
54
55             if (c == '(' || c == '+' || c == '-' || c == '*' || c == '/') {
56                 pila.pushPila(c);
57             } else if (Character.isLetter(c)) {
58                 posfija += c;
59
60             } else if (c == ')') {
61                 while (!pila.isEmptyPila() && pila.peekPila() != '(') {
62                     posfija += pila.popPILA();
63                 }
64                 if (!pila.isEmptyPila() && pila.peekPila() == '(') {
65                     pila.popPILA();
66                 }
67             }
68
69             model.addRow(new Object[]{entrefija.substring(0, i + 1), pila.toString(), posfija});
70         }
71
72         while (!pila.isEmptyPila()) {
73             posfija += pila.popPILA();
74             model.addRow(new Object[]{entrefija, pila.toString(), posfija});
75         }
76
77         model.addRow(new Object[]{entrefija, "", posfija});
78     }
79
80     public static void main(String[] args) {
81         new Posfija();
82     }
```

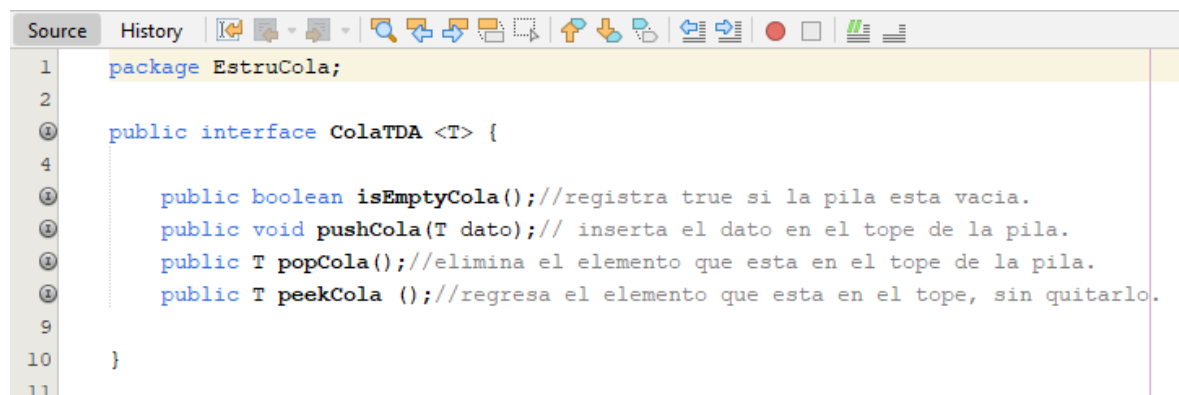
Este código es una implementación de una aplicación de escritorio en Java que convierte una expresión infija a una expresión posfija utilizando una pila.

La clase Posfija extiende de JFrame para crear una ventana de aplicación, que contiene una etiqueta JLabel, un campo de texto JTextField, un botón JButton y una tabla JTable. En el constructor de la clase se inicializan los componentes y se define el comportamiento del botón al hacer clic en él, lo que invoca al método convertir().

El método convertir() obtiene el texto del campo de texto txtEntrefija, y luego itera a través de cada carácter de la cadena de entrada. Si el carácter es un operador o un paréntesis, se agrega a la pila. Si el carácter es una letra, se agrega a la cadena de salida posfija. Si el carácter es un paréntesis derecho, se eliminan todos los operadores de la pila y se agregan a la cadena de salida posfija hasta encontrar un paréntesis izquierdo.

En cada iteración se actualiza la tabla con la expresión entrefija, el estado actual de la pila y la expresión posfija. Después de la iteración, cualquier operador restante en la pila se agrega a la cadena de salida posfija y se actualiza la tabla con la expresión completa entrefija, la pila vacía y la expresión posfija final.

El método main crea una instancia de la clase Posfija y la muestra en la pantalla. En resumen, esta aplicación de escritorio convierte una expresión infija a una expresión posfija y la muestra en una tabla.



```
1 package EstruCola;
2
3
4 public interface ColaTDA <T> {
5
6     public boolean isEmptyCola(); //registra true si la pila esta vacia.
7     public void pushCola(T dato); // inserta el dato en el tope de la pila.
8     public T popCola(); //elimina el elemento que esta en el tope de la pila.
9     public T peekCola (); //regresa el elemento que esta en el tope, sin quitarlo.
10
11 }
```

```
1 package EstruCola;
2
3 import EntradaYSalida.tools;
4
5 public class ColaA <T> implements ColaTDA <T> {
6     private T Cola[];
7     private byte U;
8
9     public ColaA (int max) {
10         Cola =(T[]) (new Object[max]);
11         U=-1;
12     }
13
14     @Override
15     public boolean isEmptyCola() {
16         return (U== -1);
17     }
18
19     public boolean isSpace() {
20         return (U<Cola.length-1);
21     }
22
23     @Override
24     public void pushCola(T dato) {
25         if (isSpace()) {
26             U++;
27             Cola[U]=dato;
28         }
29         else
30             tools.errorMsj("PILA LLENA!!!!");
```

```

30     }
31
32     @Override
33     public T popCola () {
34         T dato = Cola [0];
35         recorrePos();
36         U--; return dato;
37     }
38
39     @Override
40     public T peekCola() {
41         return Cola [0];
42     }
43
44     public void VaciarCola() {
45         Cola=null;
46         U=-1;
47     }
48
49     public void recorrePos() {
50
51         for (int j = 0; j < U; j++) {
52             Cola[j] = Cola[j + 1];
53         }
54     }
55

```

```

57     @Override
58     public String toString(){
59         return toString(U);
60     }
61     private String toString(int i){
62         return(i<=U?"tope==>" + i + "[" + Cola [i] + "]" + toString (i+1):"";
63         //return(i>=0?"tope==>" + i + "[" + Cola [i] + "]" + toString (i-1):"";
64     }
65
66     public static void Menu() {
67         ColaA <String> cola = new ColaA ((byte)10);
68         String op="";
69         do{
70             op=tools.boton("PUSH,POP,PEEK,FREE,SALIR");
71             switch(op){
72                 case "PUSH":
73                     cola.pushCola(tools.leeString("escribe un dato entero: "));
74                     tools.imprime("datos de la pila: \n" + cola.toString());
75                     break;
76                 case "POP":
77                     if (cola.isEmptyCola())tools.errorMsj("pila Vacía");
78                     else
79                         tools.imprime("dato eliminado de la cima de la pila: " + cola.popCola()+ "\n" + cola.toString());
80                     break;
81                 case "PEEK":
82                     if (cola.isEmptyCola())tools.errorMsj("pila vacía");
83                     else

```

```
84         tools.imprime("dato de la cima de la pila: ==>" + cola.peekCola() + "\n" + cola.toString());
85         break;
86     case "FREE":
87         if (cola.isEmptyCola()) tools.errorMsj("pila vacia");
88         else
89         {
90             cola=null;
91             cola = new ColaA ((byte)10);
92         }
93         break;
94     }
95
96     }
97     while(!op.equals("SALIR"));
98 }
99
100 public static void main(String[] args) {
101
102     Menu();
103
104 }
105 }
```

El código es una implementación en Java de una cola abstracta (cola genérica) utilizando un arreglo de tamaño fijo para almacenar los elementos.

La clase **ColaA** implementa la interfaz **ColaTDA** y define los siguientes métodos:

- **ColaA(int max):** constructor que inicializa el arreglo con tamaño **max** y establece el índice **U** en -1 (que representa la posición del último elemento de la cola).
- **isEmptyCola():** devuelve **true** si la cola está vacía (es decir, **U** es -1).
- **isSpace():** devuelve **true** si todavía hay espacio disponible en el arreglo para agregar elementos.
- **pushCola(T dato):** agrega un elemento **dato** al final de la cola si todavía hay espacio disponible en el arreglo. Si la cola está llena, se lanza un mensaje de error usando la clase **tools**.
- **popCola():** elimina y devuelve el elemento del frente de la cola (es decir, el primer elemento). El método **recorrePos()** se utiliza para reorganizar los elementos restantes en la cola después de eliminar el primer elemento.
- **peekCola():** devuelve el elemento del frente de la cola sin eliminarlo.
- **VaciarCola():** elimina todos los elementos de la cola estableciendo el arreglo en **null** y establece **U** en -1.

- **recorrePos()**: método auxiliar utilizado por **popCola()** para mover los elementos restantes en el arreglo una posición hacia el frente después de eliminar el primer elemento.
- **toString()**: devuelve una cadena que representa todos los elementos en la cola. Este método utiliza el método privado **toString(int i)** para construir la cadena de elementos en orden.
- **Menu()**: método principal que define un bucle que permite al usuario elegir entre cuatro opciones: agregar un elemento (**PUSH**), eliminar el primer elemento (**POP**), obtener el primer elemento sin eliminarlo (**PEEK**) o vaciar completamente la cola (**FREE**). Cada opción utiliza los métodos definidos en la clase **ColaA** para realizar las operaciones correspondientes.
- **main()**: método principal que llama a **Menu()** para ejecutar el programa.

```
Source History [Icons]
1 package EstruCola;
2
3 import EntradaYSalida.tools;
4 import java.util.Iterator;
5 import java.util.LinkedList;
6 import java.util.Queue;
7
8 public class ColaB <T> implements ColaTDA<T> {
9     private Queue cola;
10    public ColaB(){
11        cola=new LinkedList();
12    }
13    public int Size(){
14        return cola.size();
15    }
16
17    @Override
18    public boolean isEmptyCola(){
19        return (cola.isEmpty());
20    }
21
22    @Override
23    public T peekCola(){
24        return (T) (cola.element());
25    }
26    public void vaciar(){
27        cola.clear();
28    }
29 }
```



```

30      @Override
31      public void pushCola(T dato) {
32          cola.add(dato);
33      }
34
35      @Override
36      public T popCola() {
37          T dato;
38          dato=(T) cola.element();
39          cola.remove();
40          return dato;
41      }
42
43      @Override
44      public String toString() {
45          String cad="";
46          byte j=0;
47          for (Iterator i = cola.iterator(); i.hasNext();) {
48              cad+="["+ j+ " ] ==>> ["+ i.next()+ "]+ "\n";
49              j++;
50          }
51          return cad;
52      }
53
54      public static void Menu() {
55          ColaB cola= new ColaB();
56          String dato,op;
57          do{
58
59              op=tools.boton("PUSH,POP,PEEK,FREE,SALIR");
60              switch(op){
61                  case "PUSH":
62                      cola.pushCola(tools.leeString("escribe el dato que deseas agregar: "));
63                      tools.imprime("datos de la pila: \n" + cola.toString());
64                      break;
65                  case "POP":
66                      if (cola.isEmptyCola())tools.errorMsj("pila Vacía");
67                      else
68                          tools.imprime("dato eliminado de la cima de la pila: " + cola.popCola()+ "\n"+ cola.toString());
69                      break;
70                  case "PEEK":
71                      if (cola.isEmptyCola())tools.errorMsj("pila vacía");
72                      else
73                          tools.imprime("dato de la cima de la pila: ==>" + cola.peekCola()+ "\n"+ cola.toString());
74                      break;
75                  case "FREE":
76                      if (cola.isEmptyCola())tools.errorMsj("pila vacía");
77                      else
78                      {
79                          cola=null;
80                          cola = new ColaB();
81                      }
82                      break;
83              }
84          }
85          while(!op.equals("SALIR"));

```

```
85     while(!op.equals("SALIR"));
86   }
87   public static void main(String[] args) {
88       Menu();
89   }
90 }
```

Este código implementa una cola genérica en Java utilizando una lista enlazada (clase `LinkedList`) como la estructura de datos subyacente. La cola se define como una clase que implementa la interfaz `ColaTDA<T>`, que define los métodos que se esperan en una cola.

La clase `ColaB<T>` tiene los siguientes métodos:

- Constructor: Inicializa una nueva instancia de la cola creando una instancia de `LinkedList`.
- `Size()`: Devuelve el número de elementos en la cola.
- `isEmptyCola()`: Devuelve true si la cola está vacía, de lo contrario, devuelve false.
- `peekCola()`: Devuelve el elemento en el frente de la cola sin eliminarlo.
- `vaciar()`: Elimina todos los elementos de la cola.
- `pushCola(T dato)`: Agrega el elemento especificado al final de la cola.
- `popCola()`: Elimina y devuelve el elemento en el frente de la cola.
- `toString()`: Devuelve una cadena que representa todos los elementos en la cola.

Además, la clase tiene un método estático `"Menu()"` que proporciona una interfaz de usuario simple para interactuar con la cola utilizando la consola. El usuario puede elegir entre varias opciones: agregar un elemento a la cola (`"PUSH"`), eliminar el elemento en el frente de la cola (`"POP"`), obtener el elemento en el frente de la cola sin eliminarlo (`"PEEK"`), vaciar la cola (`"FREE"`), o salir del programa (`"SALIR"`).



```
1 package EstruCola;
2
3 import EntradaYSalida.tools;
4 import java.util.ArrayList;
5
6 public class ColaC<T> implements ColaTDA<T> {
7     private ArrayList cola;
8     byte u;
9
10    public ColaC() {
11        cola= new ArrayList();
12        u=0;
13    }
14
15    public int Size() {
16        return cola.size();
17    }
18
19    @Override
20    public boolean isEmptyCola() {
21        return cola.isEmpty();
22    }
23
24    public void vaciar () {
25        cola.clear();
26    }
27
28    @Override
29    public void pushCola(Object dato) {
```

```

30     cola.add(dato);
31     u++;
32 }
33
34 @Override
35 public T popCola() {
36     T dato =(T) cola.get(0);
37     cola.remove(0);
38     u--;
39     return dato;
40 }
41
42 @Override
43 public T peekCola() {
44     return (T) cola.get(0);
45 }
46
47 @Override
48 public String toString(){
49     return toString(0);
50 }
51
52 private String toString(int i){
53     return(i<u?"["+ i + " ] ==> ["+ cola.get(i) + "]\n" + toString (i+1):"";
54 }
55
56 public static void Menu() {
57     ColaC cola= new ColaC();

```

```

59 String dato,op;
60 do{
61     op=tools.boton("PUSH,POP,PEEK,FREE,SALIR");
62     switch(op){
63         case "PUSH":
64             cola.pushCola(tools.leeString("escribe el dato que deseas agregar: "));
65             tools.imprime("datos de la pila: \n" + cola.toString());
66             break;
67         case "POP":
68             if (cola.isEmptyCola())tools.errorMsj("pila Vacía");
69             else
70                 tools.imprime("dato eliminado de la cima de la pila: " + cola.popCola()+ "\n"+ cola.toString());
71             break;
72         case "PEEK":
73             if (cola.isEmptyCola())tools.errorMsj("pila vacía");
74             else
75                 tools.imprime("dato de la cima de la pila: ==> " + cola.peekCola()+ "\n"+ cola.toString());
76             break;
77         case "FREE":
78             if (cola.isEmptyCola())tools.errorMsj("pila vacía");
79             else
80             {
81                 cola=null;
82                 cola = new ColaC();
83             }
84             break;
85     }
86 }

```

```
86     }  
87     while(!op.equals("SALIR"));  
88 }  
89 public static void main(String[] args) {  
90     Menu();  
91 }  
92 }  
93 }
```

El código define una clase llamada "ColaC" que implementa la interfaz "ColaTDA". La clase utiliza un ArrayList para implementar la cola.

La clase tiene varios métodos para realizar operaciones básicas de una cola, como "pushCola" para agregar elementos, "popCola" para eliminar el primer elemento, "peekCola" para obtener el primer elemento sin eliminarlo y "isEmptyCola" para comprobar si la cola está vacía.

También hay un método "toString" que devuelve una representación en cadena de la cola y un método "vaciar" que borra todos los elementos de la cola.

La clase incluye un método "Menu" que se encarga de la interacción con el usuario a través de la consola. El usuario puede agregar elementos a la cola, eliminar elementos de la cola, obtener el primer elemento de la cola sin eliminarlo y vaciar la cola. El programa termina cuando el usuario selecciona la opción "SALIR".

Conclusión

En conclusión, en este reporte se exploraron las estructuras de datos lineales más comunes en programación: colas, pilas y listas. Se discutieron las características únicas de cada estructura, cómo se representan en memoria, las operaciones básicas que se pueden realizar y las aplicaciones prácticas de cada una.

Además, se implementaron métodos para cada una de estas estructuras de datos lineales en Java. Los métodos incluyen operaciones básicas como agregar y eliminar elementos, así como operaciones más avanzadas como recorrer los elementos de una lista vinculada. Se demostró que las estructuras de datos lineales son esenciales en la programación y que cada una tiene sus fortalezas y debilidades en función de la tarea que se deba realizar. Por ejemplo, las pilas son útiles para

almacenar y procesar información en el orden de "último en entrar, primero en salir", mientras que las colas son útiles para almacenar y procesar información en el orden de "primero en entrar, primero en salir".

En general, comprender las estructuras de datos lineales y cómo implementarlas en Java es fundamental para desarrollar aplicaciones efectivas y eficientes. Con el conocimiento de estas estructuras y sus métodos, se puede abordar una amplia gama de problemas de programación y desarrollar soluciones elegantes y eficientes.

Bibliografía

Ciberaula. (s/f). Pilas en Java. Ciberaula.com. Recuperado el 22 de abril de 2023, de https://www.ciberaula.com/cursos/java/pilas_java.php

Pilas en Java. (s/f). CÓDIGO Libre. Recuperado el 22 de abril de 2023, de <http://codigolibre.weebly.com/blog/pilas-en-java>

Ciberaula. (s/f-a). Colas en Java. Ciberaula.com. Recuperado el 22 de abril de 2023, de https://www.ciberaula.com/cursos/java/colas_java.php

Gonzalez, M. V. (2021, enero 24). Listas en Java. Codmind. <https://blog.codmind.com/listas-en-java/>

Tokio School. (2021, julio 8). Estructura de datos en Java. Tokio School. <https://www.tokioschool.com/noticias/estructura-datos-java/>