ESTRUCTURA DE DATOS

INSTITÚTO TECNOLÓGICO DE MEXICO

CAMPUS ORIZABA

NOMBRE DE LA ASIGNATURA:
ESTRUCTURA DE DATOS

TEMA:
REPORTE DE UNIDAD 3

NOMBRE DE LOS INTEGRANTES:
ESPINDOLA OLIVERA PALOMA - 21010186
VELAZQUEZ SARMIENTO CELESTE- 210223

CARRERA:
INGENIERIA INFORMATICA

GRUPO:
3a3A

INTRODUCCION

En esta unidad 3 explicaremos los códigos que se ocuparon para entender mejor el tema y propósito de estos, así como también su estructura y los métodos que los componen. Las pilas y colas son estructuras de datos fundamentales en programación que se utilizan para almacenar y organizar elementos. Estas estructuras siguen principios de organización específicos que determinan cómo se agregan y eliminan los elementos.

Las pilas y colas son estructuras de datos esenciales en programación que permiten organizar elementos de manera eficiente. En Java, se pueden implementar utilizando las clases Stack y LinkedList para pilas, y la interfaz Queue con su implementación LinkedList para colas. Estas estructuras son ampliamente utilizadas en diversos problemas y algoritmos, y su conocimiento es fundamental para desarrollar aplicaciones robustas y eficientes.

El tema de pilas y colas en Java es fundamental para comprender y aplicar eficientemente estructuras de datos en tus programas. Las pilas y colas son dos estructuras de datos lineales que siguen principios de organización específicos y son ampliamente utilizadas en programación. Comprender el tema de pilas y colas en Java implica familiarizarse con los conceptos y principios fundamentales, así como conocer las clases e interfaces relacionadas en la biblioteca estándar de Java. Esto te permitirá utilizar las operaciones y métodos proporcionados por estas clases para agregar, eliminar y acceder a los elementos en una pila o cola.

Además, es esencial comprender cómo aplicar pilas y colas en diferentes escenarios de programación, como la gestión de historiales, el procesamiento de tareas en orden, la evaluación de expresiones matemáticas, entre otros. A través de la práctica y la resolución de problemas, podrás adquirir habilidades prácticas para aplicar pilas y colas en situaciones reales.

Al comprender y aplicar eficientemente las pilas y colas en Java, podrás mejorar la organización y la manipulación de datos en tus programas, así como optimizar la eficiencia y la legibilidad del código. Esto te ayudará a desarrollar aplicaciones más robustas y eficientes, y a enfrentar desafíos de programación de manera más efectiva.

COMPETENCIAS ESPECÍFICAS

- Conocimiento básico de estructuras de datos: Es importante comprender los conceptos básicos de estructuras de datos, como el concepto de pila y cola, así como los principios de LIFO (Last In, First Out) y FIFO (First In, First Out).
- Familiaridad con el lenguaje Java: Es necesario tener un conocimiento básico del lenguaje de programación Java, incluyendo la sintaxis, la declaración de variables, los tipos de datos y los conceptos de programación orientada a objetos.
- Comprensión de las clases e interfaces relacionadas: Debes familiarizarte con las clases y interfaces proporcionadas por Java para implementar pilas y colas, como **Stack**, **LinkedList** y **Queue**. Comprender los métodos y operaciones disponibles en estas clases e interfaces es fundamental.
- Conocimiento de los principales métodos y operaciones: Es importante comprender los métodos y operaciones clave para trabajar con pilas y colas en Java, como **push()**, **pop()**, **peek()** para pilas, y **add()**, **remove()**, **peek()** para colas.
- Aplicación de pilas y colas en problemas reales: Es útil practicar la aplicación de pilas y colas en situaciones reales de programación. Esto implica comprender cómo utilizar estas estructuras de datos en diferentes problemas, como el procesamiento de tareas, la evaluación de expresiones matemáticas o el recorrido de estructuras de datos.
- Resolución de problemas y ejercicios prácticos: Resolver problemas y ejercicios prácticos relacionados con pilas y colas en Java te ayudará a fortalecer tus habilidades y comprensión del tema. Puedes buscar problemas en línea o crear tus propios desafíos para aplicar los conceptos aprendidos.
- Investigación y lectura adicional: La lectura de recursos adicionales, como libros de estructuras de datos y algoritmos en Java, tutoriales en línea y documentación oficial de Java, te brindará una comprensión más profunda y te ayudará a estar al tanto de las mejores prácticas y las últimas actualizaciones en el tema.
- Al desarrollar estas competencias, podrás comprender y utilizar eficazmente las pilas y colas en Java, y aplicarlas en una variedad de problemas y situaciones de programación.

MARCO TEORICO

Las pilas y colas son estructuras de datos fundamentales en programación que se utilizan para almacenar y organizar elementos. En Java, se pueden implementar pilas y colas utilizando las clases proporcionadas en la biblioteca estándar, como Stack y LinkedList. A continuación se presenta un marco teórico básico para comprender estas estructuras de datos.
Pila (Stack): Una pila es una estructura de datos lineal que sigue el principio de "último en entrar, primero en salir" (LIFO, por sus siglas en inglés). Esto significa que el último elemento agregado a la pila es el primero en ser eliminado.

En Java, se puede implementar una pila utilizando la clase Stack o la clase LinkedList. La clase Stack extiende la clase Vector y proporciona métodos como push() para agregar elementos a la pila, pop() para eliminar y retornar el elemento superior de la pila, peek() para obtener el elemento superior sin eliminarlo, y empty() para verificar si la pila está vacía.

PILAS Y COLAS EJERCICIOS

```
1   package Colas;
2
3   import Tools.ToolsPanel;
4
5   public class ColaA<T> implements ColaTDA<T>{
6
7           private T cola[];
8           private byte u;
9
10          public ColaA(int max) {
11                  cola = (T[]) (new Object[max]);
12                  u=-1;
13          }
```

Creamos una clase llamada ColaA, esta clase va a llevar el control de la cola, creamos nuestro constructor.

- ⇨ Se le implementa la interfaz ColaTDA
- ⇨ Creamos dos variables de tipo private
- ⇨ Se crea la cola que será de tipo genérica
- ⇨ Se crea una variable U que es de tipo byte
- ⇨ El constructor recibe un parámetro, es de tipo entero, se llama max, inicializamos la cola de tipo genérica, con el tamaño que nos mandó anteriormente el usuario, en este caso sería la variable max, u la inicializamos con -1.

```
18
19          @Override
20          public boolean isEmptyCola() {
21                  return (u==-1);
22          }
```

- ⇨ Este método regresa un booleano, es el encargado de decirnos si la cola esta vacía o llena.

```
23
24          public boolean isSpace() {
25                  return (u<cola.length-1);
26          }
27
```

- ⇨ Este método retorna un dato booleano también, es el encargado de decirnos si aún hay espacio para almacenar más datos.

```
28          @Override
29          public void pushCola(T Dato) {
30                  if(isSpace()) {
31                          u++;
32                          cola[u]=Dato;
33                  }else {
34                          ToolsPanel.imprimeError("Cola llena...");
35                  }
36          }
37
```

⇨ Este método recibe un dato genérico como parámetro que se llama dato, primero verifica si hay espacio en el arreglo, si lo hay aumenta la u en 1, que es nuestro apuntador y guarda el dato en la posición donde se encuentra el apuntador, en caso contrario que no haya espacio imprime “Cola llena”.

```
38          @Override
39          public T popCola() {
40                  T Dato = cola[0];
41                  for(int k=0; k<=u; k++) {
42                          cola[k]=cola[k+1];
43                  }
44                  u--;
45                  return Dato;
46          }
47
```

⇨ Este método elimina el último dato, decrementa u en 1 para poder ser eliminado.

```
48          @Override
49          public T peekCola() {
50                  return (T)cola[0];
51          }
52
```

⇨ Este método solo muestra el primer dato de la cola

```
52
53          public String toString() {
54                  return toString(0);
55          }
56
57          public String toString(int i) {
58                  return (i<=u)?" "+i+"=>["+""+cola[i]+"] "+toString(i+1):"";
59          }
60
```

⇨ Tenemos 2 toString, su función es para imprimir la cola

```
61              @Override
62              public void freeCola() {
63                      u=-1;
64              }
65
66      }
```

⇨ Este método funciona para eliminar toda la cola

```
1   package Colas;
2
3   public interface ColaTDA <T>{
4
5           public boolean isEmptyCola();
6
7           public void pushCola(T Dato);
8
9           public T popCola();
10
11          public T peekCola();
12
13          public void freeCola();
14
15  }
```

⇨ ColaTDA es una interfaz genérica, contiene los métodos que vamos a utilizar, las cuales podemos utilizar en diferentes clases

```java
package Memoria_dinamica;

import java.util.ArrayList;

import Pila_Estatica.Pila_TDA;

public class PilaC<T> implements Pila_TDA<T>{

        private ArrayList pila;

        int tope=-1;

        public PilaC() {
                pila = new ArrayList();
                int tope;
        }
```

- Creamos una nueva clase llamada PilaC, que es de tipo genérica
- Le implementamos una interfaz genérica que es PilaTDA, implementamos sus métodos en esta clase
- Creamos una variable de tipo private llamada pila, es de tipo ArrayList, con esa inicamos nuestra pila
- Creamos un tope de tipo entero, se inicializa con -1 para indicar que en el momento que se inicie no exista ningún dato.

```java

        public int Size() {
                return pila.size();
        }
```

- Este método regresa el tamaño de la pila

```java
        @Override
        public boolean isEmpty() {
                return pila.isEmpty();
        }

```

- Este método es por si la pila esta vacia

```
27          public void vaciar() {
28                  pila.clear();
29          }
30
```

⇨ Este método vacía la pila por completo

```
31          @Override
            public void push(T dato) {
33                  pila.add(dato);
34                  tope++;
35          }
```

⇨ Este metodo recibe un parámetro de tipo genérico llamado dato, el cual lo agrega a la pila y tope aumenta en 1 para indicar que se agregó un nuevo dato.

```
37          @Override
            public T pop() {
39                  T dato=(T)pila.get(tope);
40                  pila.remove(tope);
41                  tope--;
42                  return dato;
43          }
```

⇨ Este método saca el ultimo digito de la pila, tope decrementa en 1 para eliminarlo

```
44
45          @Override
            public T peek() {
47                  return (T)pila.get(tope);
48          }
49
```

⇨ Este método regresa el último dato de la pila, pero sin eliminarlo

```
49
50          public String toString() {
51                  return toString(tope);
52          }
53
54          public String toString(int i) {
55                  return (i>=0)?"tope ===>"+i+"["+pila.get(i)+"]\n"+toString(i-1):"";
56          }
57
```

⇨ Los toString imprimen la pila.

```
1   package Pila_Estatica;
2
3   import Tools.ToolsPanel;
4
5   public class PilaA <T> implements Pila_TDA<T>{
6
7           private T pila[];
8           private byte tope;
9
10          public PilaA(int max) {
11                  pila=(T[])(new Object[max]);
12                  tope=-1;
13          }
14
15      public PilaA(byte b) {
16          throw new UnsupportedOperationException("Not supported yet.");
17      }
18
19          public boolean isEmpty(){
20                  return (tope==-1);
21          }
22
23          public boolean isSpace() {
24                  return(tope<pila.length-1);
25          }
26
27          public void push(T dato) {
28                  if(isSpace()) {
29                          tope++;
30                          pila[tope]=dato;
31                  }else {
32                          ToolsPanel.imprimeError("Pila llena...");
33                  }
34          }
35
36          public T pop() {
37                  T dato = pila[tope];
38                  tope--;
39                  return dato;
40          }
41
42          public T peek() {
43                  return pila[tope];
44          }
45
46          public String toString() {
47                  return toString(tope);
48          }
```

⇨ Creamos una nueva clase llamada PilaA, que es de tipo genérica

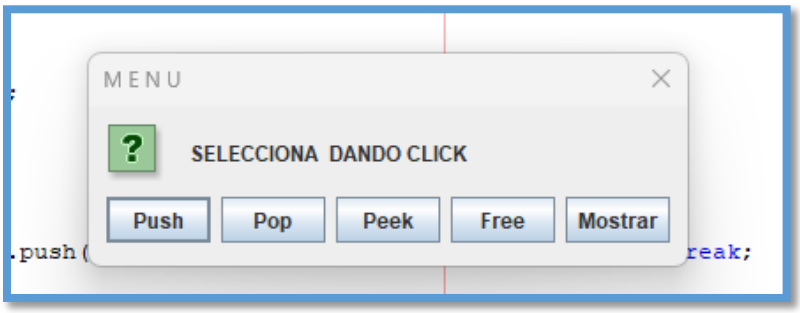
⇨ Le implementamos una interfaz genérica que es PilaTDA, implementamos sus métodos en esta clase

```java
package Pila_Estatica;

import java.util.Stack;

public class PilaB<T> implements Pila_TDA<T>{

        private Stack<T> pila;

        public PilaB() {
                pila=new Stack<T>();
        }

        @Override
        public boolean isEmpty() {
                return (pila.empty());
        }

        @Override
        public T pop() {
                T dato;
                dato=(T)pila.peek();
                pila.pop();
                return dato;
        }

        @Override
        public void push(T dato) {
                pila.push(dato);
        }

        @Override
        public T peek() {
                return (T)(pila.peek());
        }

        @Override
        public void freePila() {
                pila.clear();
        }

        public int Size() {
                return pila.size();
        }

        public String toString() {
                return toString(Size()-1);
        }
```

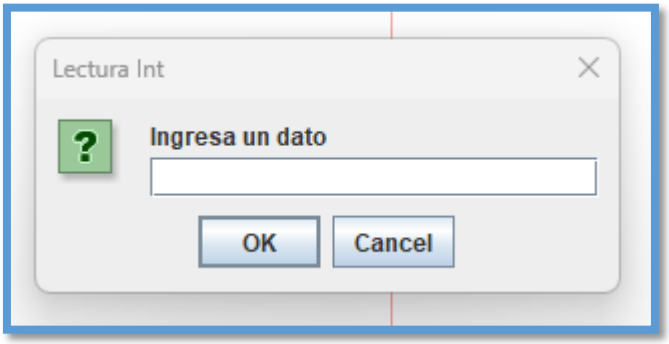
- ⇨ Creamos una variable llamada pila de tipo stack, el constructor creara el espacio en la memoria para la pila
- ⇨ Implementamos la interfaz genérica que es PilaTDA y sus métodos

```java
package Pila_Estatica;

public interface Pila_TDA<T>{

        public boolean isEmpty();

        public T pop();

        public void push(T dato);

        public T peek();

        public void freePila();

}
```

- PilaTDA es una interfaz genérica que contiene los métodos los cuales ocupamos

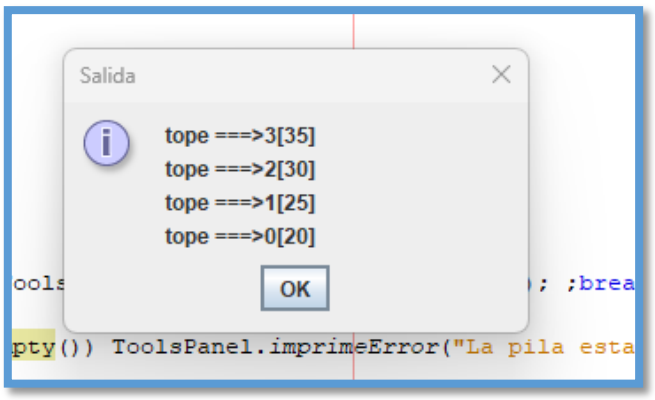


- Ejecutamos y como pueden observar muestra las siguientes opciones del menú:
- Push
- Pop
- Peek
- Free
- Mostrar

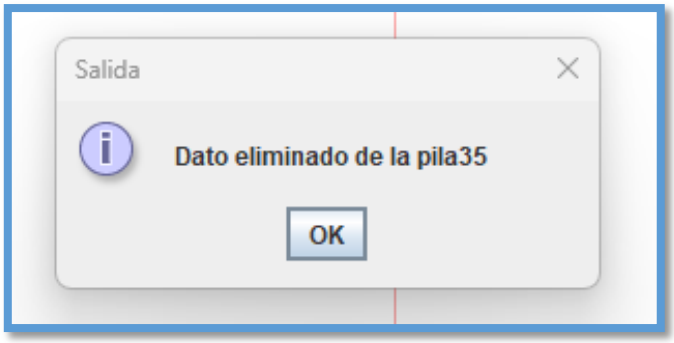


- "PUSH"

Insertamos los números a ocupar

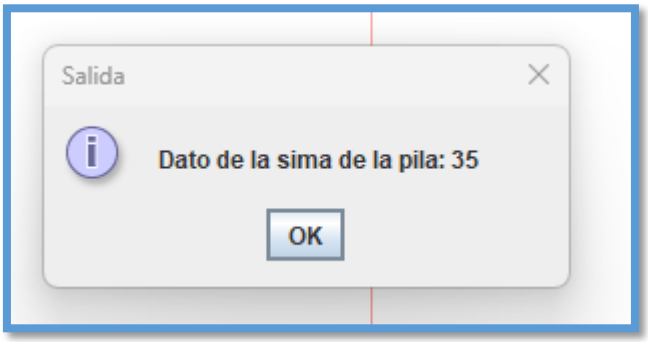


⇨ "MOSTRAR"
Muestra los datos que insertamos anteriormente



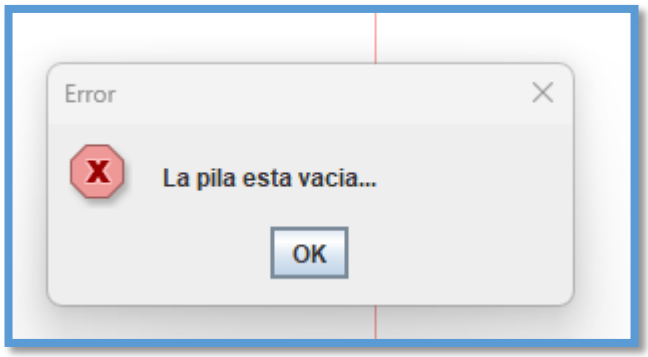
⇨ "POP"

Elimina el último dato de la



⇨ "PEEK"

Muestra el primer dato de la cola



⇨ "FREE"
Elimina toda la cola

CONCLUSIONES

En conclusión, comprender y aplicar pilas y colas en Java es esencial para mejorar la organización y manipulación de datos en tus programas. Al utilizar estas estructuras de datos, puedes resolver problemas de manera eficiente, optimizar la eficiencia y la legibilidad del código, y desarrollar aplicaciones más robustas y eficientes. Las pilas y colas en Java son estructuras de datos esenciales que ofrecen diferentes principios de organización (LIFO y FIFO) y se utilizan en una amplia variedad de aplicaciones y problemas de programación. Comprender y utilizar eficientemente las pilas y colas en Java tiene varias ventajas:

- ⇨ Organización de datos: Las pilas y colas permiten organizar y manipular datos de manera estructurada y eficiente, siguiendo principios específicos de ordenamiento.
- ⇨ Implementación sencilla: Java proporciona clases e interfaces en su biblioteca estándar que facilitan la implementación y el uso de pilas y colas, como Stack, LinkedList y Queue. Estas estructuras están disponibles de forma nativa, lo que simplifica su implementación en tus programas.
- ⇨ Solución de problemas específicos: Las pilas y colas son útiles en una variedad de situaciones, como la evaluación de expresiones matemáticas, el procesamiento de tareas en orden, la gestión de historiales de navegación y la búsqueda en amplitud de estructuras de datos. Comprender cómo utilizar pilas y colas te permite abordar y resolver eficientemente estos problemas.

REFERENCIAS

http://estructuradedatos10111248.blogspot.com/2015/07/estructuras-lineales-y-no-lineales.html