

Materia:

ESTRUCTURA DE DATOS

Docente:

María Jacinta Martínez Castillo

Integrantes:

Ríos Méndez Cesar

Hernández Acevedo Brayan

Sentíes Robles Pedro

Hora:

17:00 - 18:00

Fecha:

22/04/2023



REPORTE DE PRACTICA.

	Unidad: 3
ENERO-JUNIO 2023	NO. DE PRÁCTICA: 3 NOMBRE DE PRÁCTICA: Estructuras Lineales.
	DOCENTE: María Jacinta Martínez Castillo
NO. DE CONTROL Y NOMBRE: RIOS MENDEZ CESAR 21011029 HERNANDEZ ACEVEDO BRAYAN 21010962 SENTIES ROBLES PEDRO 21011047	
FECHA ENTREGA: 22/04/2023	

INTRODUCCIÓN

Las estructuras lineales son uno de los conceptos fundamentales en la programación. Estas estructuras permiten almacenar y organizar datos de manera secuencial, de tal manera que se puedan acceder a ellos de forma ordenada y eficiente.

En la programación, las estructuras lineales se pueden implementar de diferentes maneras, como por ejemplo utilizando arrays, listas enlazadas, pilas y colas. Cada una de estas estructuras tiene sus propias ventajas y desventajas, dependiendo del tipo de datos y de la operación que se desee realizar.

Las estructuras lineales son muy útiles en diferentes áreas de la programación, como por ejemplo en la manipulación de datos en bases de datos, en la

implementación de algoritmos de ordenamiento y búsqueda, en la gestión de memoria en sistemas operativos, entre otras aplicaciones.

En resumen, las estructuras lineales son esenciales en la programación y su conocimiento es fundamental para poder desarrollar programas eficientes y escalables.

OBJETIVO GENERAL

Desarrollar programas que incorporen las principales estructuras de datos lineales: Pila, Cola y Lista enlazada.

OBJETIVO ESPECÍFICO

El estudio de las estructuras de datos, sin duda es uno de los más importantes dentro de las carreras relacionadas con la computación, ya que el conocimiento eficiente de las estructuras de datos suele ser imprescindible en la formación, debido a la trascendencia que un aprendizaje teórico-práctico de las mismas supondrá para nuestra carrera.

MARCO TEÓRICO

3.0 Clases genéricas, arreglos dinámicos

Las clases genéricas y los arreglos dinámicos son dos conceptos importantes en la programación orientada a objetos y en particular en lenguajes como Java, C# y C++.

Las clases genéricas permiten crear clases que pueden trabajar con diferentes tipos de datos de manera flexible y reutilizable. En lugar de definir una clase para cada tipo de datos que se necesite, se puede crear una clase genérica que acepte parámetros de tipo. Por ejemplo, se puede crear una clase genérica de lista enlazada que pueda almacenar diferentes tipos de datos, como números enteros, cadenas de caracteres o cualquier otro tipo de objeto.

Por otro lado, los arreglos dinámicos son estructuras de datos que permiten almacenar una cantidad variable de elementos en un solo contenedor. A diferencia de los arreglos estáticos, que tienen un tamaño fijo, los arreglos dinámicos pueden aumentar o disminuir su tamaño según sea necesario durante la ejecución del programa. Esto permite una mayor flexibilidad en la manipulación de datos y una gestión más eficiente de la memoria.

3.1 Pilas

En programación, una pila es una estructura de datos lineal que permite almacenar y recuperar elementos de manera LIFO (Last In, First Out), es decir, el último elemento que se inserta en la pila es el primero en ser eliminado.

Las pilas se pueden implementar utilizando arreglos estáticos o dinámicos, o mediante listas enlazadas. La operación principal en una pila es la inserción de un elemento en la cima de la pila, conocida como "push", y la eliminación de un elemento de la cima de la pila, conocida como "pop".

Las pilas son muy útiles en situaciones en las que se necesita un acceso rápido a los últimos elementos insertados, como por ejemplo en la implementación de algoritmos de búsqueda en profundidad, en el procesamiento de expresiones aritméticas, en la gestión de llamadas a funciones en la memoria del programa, entre otros.

3.1.1 Representación en memoria

La representación en memoria de una estructura lineal depende de la implementación específica que se utilice. Sin embargo, en general, se pueden utilizar dos enfoques para representar estructuras lineales en memoria: mediante arreglos o mediante listas enlazadas.

En el caso de los arreglos, la representación en memoria implica reservar un bloque contiguo de memoria para almacenar los elementos de la estructura lineal. Cada elemento se almacena en una posición específica del arreglo y se puede acceder a él utilizando un índice. El índice de un elemento se calcula multiplicando su posición en la estructura lineal por el tamaño de cada elemento y sumándolo a la dirección base del arreglo en memoria.

Por otro lado, en el caso de las listas enlazadas, la representación en memoria implica crear un nodo para cada elemento de la estructura lineal. Cada nodo contiene el elemento y un puntero que apunta al siguiente nodo en la lista. El primer nodo se llama cabeza de la lista y el último se llama cola.

3.1.2 Operaciones básicas

Las operaciones básicas en una pila son las siguientes:

Push: inserta un elemento en la cima de la pila.

Pop: elimina el elemento superior de la pila.

Pop o Peek: devuelve el elemento superior de la pila sin eliminarlo.

isEmpty: comprueba si la pila está vacía o no.

Estas operaciones son fundamentales para el uso de una pila en la programación. El push y el pop permiten agregar y eliminar elementos en la pila respectivamente. La operación top permite consultar el elemento superior de la pila sin eliminarlo, lo que puede ser útil para verificar el contenido de la pila sin modificarlo. Por último, la operación isEmpty comprueba si la pila está vacía o no, lo que puede ser útil para evitar errores al acceder a elementos inexistentes en la pila.

3.1.3 Aplicaciones

1. Implementación de algoritmos de búsqueda en profundidad (DFS): En este algoritmo, se utiliza una pila para almacenar los nodos que se van visitando en el grafo o árbol que se está recorriendo. El último nodo visitado se inserta en la pila y se explora sus vecinos hasta que se agoten. Luego se retira el último nodo de la pila y se explora sus vecinos, y así sucesivamente hasta que se encuentre el nodo buscado o se visiten todos los nodos.
2. Evaluación de expresiones aritméticas: En este caso, se utiliza una pila para almacenar los operandos y operadores de una expresión aritmética en notación posfija (postfix).
3. Gestión de llamadas a funciones: En lenguajes de programación como C++, Java y Python, se utiliza una pila para almacenar los marcos de pila (stack frames) correspondientes a las funciones que se van llamando en un programa. Cada vez que se llama una función, se inserta un nuevo marco de pila en la pila y cuando la función retorna, se elimina el marco de pila correspondiente.
4. Deshacer/Rehacer en editar

3.2 Colas

En programación, una cola es una estructura de datos lineal que se utiliza para almacenar elementos en un orden determinado. La cola funciona siguiendo el principio de "FIFO" (First In, First Out), lo que significa que el primer elemento que se agrega a la cola es el primero que se elimina. En otras palabras, la cola sigue un principio de "el primero en llegar es el primero en salir".

Las operaciones básicas de una cola incluyen:

1. Agregar un elemento al final de la cola.
2. Eliminar el primer elemento de la cola.
3. Obtener el elemento delantero de la cola, sin eliminarlo.
4. isEmpty: Verificar si la cola está vacía o no.

La operación Agregar permite agregar un elemento al final de la cola, mientras que la operación eliminar elimina el primer elemento de la cola. La operación Front

devuelve el primer elemento de la cola sin eliminarlo, lo que puede ser útil para verificar el contenido de la cola sin modificarlo. Por último, la operación `isEmpty` comprueba si la cola está vacía o no, lo que puede ser útil para evitar errores al acceder a elementos inexistentes en la cola.

3.2.1 Representación en memoria

Las colas se pueden representar en programación utilizando diferentes estructuras de datos, dependiendo de las necesidades y restricciones de la aplicación. Un

1. **Arreglos:** Una forma común de representar una cola es mediante un arreglo unidimensional. En este enfoque, los elementos de la cola se almacenan en un arreglo y se utilizan dos índices para rastrear el primer y último elemento de la cola. La operación agregar se realiza agregando un elemento al final del arreglo y el índice que rastrea el último elemento se incrementa. La operación eliminar se realiza eliminando el primer elemento del arreglo y el índice que rastrea el primer elemento se incrementa. Sin embargo, esta representación
2. **Listas enlazadas:** Otra forma común de representar una cola es mediante una lista enlazada. En este enfoque, los elementos de la cola se almacenan en nodos, donde cada nodo contiene un elemento y un puntero al siguiente nodo. El puntero al primer nodo de la lista representa el frente de la cola, mientras que el puntero al último nodo de la lista representa el final de la cola. La operación agregar se realiza agregando un nodo al final de la lista y la operación eliminar se realiza eliminando el primer nodo de la lista.
3. **Listas doblemente enlazadas:** Las colas también se pueden representar mediante listas doblemente enlazadas, donde cada nodo tiene un puntero al siguiente y al nodo anterior. En este enfoque, la cabeza de la lista representa el frente de la cola y la cola representa el final de la cola. Esta representa

3.2.2 Operaciones básicas

Las operaciones básicas de las colas son las siguientes:

1. **Agregar:** también conocida como push o insertar, agrega un elemento al final de la cola.
2. **Pop:** también conocida como pop o eliminar, elimina el primer elemento de la cola.
3. **Peek:** devuelve el primer elemento de la cola sin eliminarlo.
4. **isEmpty:** verifica si la cola está vacía.
5. **isFull:** verifica si la cola está llena. Sin embargo, en la mayoría de las implementaciones dinámicas de las colas, la cola puede crecer

automáticamente en tamaño según sea necesario, por lo que este método no suele ser relevante.

Estas operaciones son esenciales para trabajar con colas en programación y se pueden implementar utilizando diferentes estructuras de datos, como arreglos, listas enlazadas, listas doblemente enlazadas y heaps. La elección de la estructura de datos adecuada para implementar una cola depende de las necesidades específicas de la aplicación y de los recursos disponibles.

MATERIAL

- *Equipo de computo*
- *Eclipse (u otro software de programación).*

METODOLOGIA

En este tema se estudia la primera gran familia de TADs, todos ellos derivados del concepto de secuencia. Primero se definen las secuencias como conjuntos de elementos entre los que se establece una relación de predecesor y sucesor. Los diferentes TADs basados en este concepto se diferenciarán por las operaciones de acceso a los elementos y manipulación de la estructura. Desde el punto de vista de la informática, existen tres estructuras lineales especialmente importantes: las pilas, las colas y las listas. Su importancia radica en que son muy frecuentes en los esquemas algorítmicos.

Las operaciones básicas para dichas estructuras son:

- **crear** la secuencia vacía
- **añadir** un elemento a la secuencia
- **borrar** un elemento a la secuencia
- **consultar** un elemento de la secuencia
- **comprobar** si la secuencia está vacía

La diferencia entre las tres estructuras que se estudiarán vendrá dada por la posición del elemento a añadir, borrar y consultar:

- **Pilas:** las tres operaciones actúan sobre el final de la secuencia
- **Colas:** se añade por el final y se borra y consulta por el principio
- **Listas:** las tres operaciones se realizan sobre una posición privilegiada de la secuencia, la cual puede desplazarse

Se presenta el TAD de las pilas de elementos arbitrarios.

Para los ejercicios que vamos a realizar estaremos utilizando la estructura de las pilas, pero para esto necesitamos saber que es una pila y saber los tipos de pilas a utilizar.

Una pila es una lista ordinal o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO que permite almacenar y recuperar datos. Esta estructura se aplica en multitud de ocasiones en el área de informática debido a su simplicidad y ordenación implícita de la propia estructura.

Para el manejo de los datos se cuenta con dos operaciones básicas: apilar (push), que coloca un objeto en la pila, y su operación inversa, retirar (o desapilar, pop), que retira el último elemento apilado.

En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado. La operación retirar permite la obtención de este elemento, que es retirado de la pila permitiendo el acceso al siguiente, que pasa a ser el nuevo TOS.

Por analogía con objetos cotidianos, una operación apilar equivaldría a colocar un plato sobre una pila de platos, y una operación retirar a retirarlo.

¿Qué es una pila dinámica?

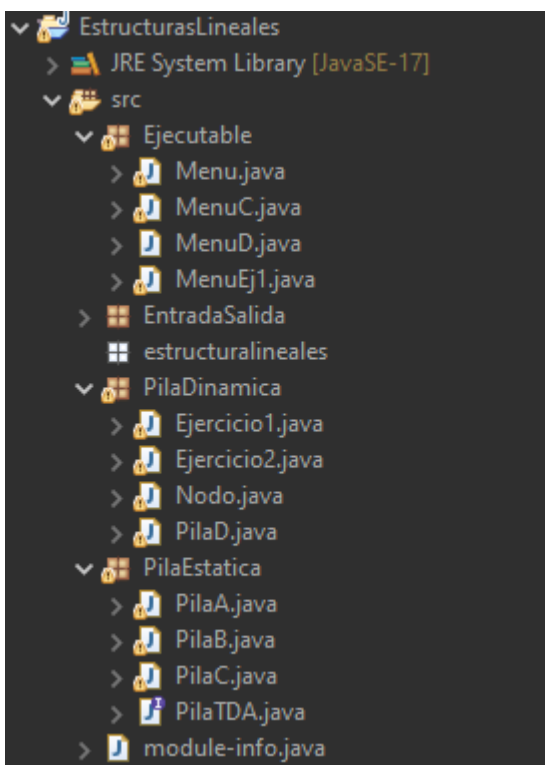
Una pila es una estructura dinámica que “apila” elementos de forma que para llegar al primero, hay que quitar todos los nodos que se hayan añadido después. Utiliza LIFO (Last Input First Output) que significa que el último que entra es el primero que saldrá.

¿Qué son las pilas estáticas?

Estructuras de datos Estáticas Son aquellas que se implementan usando vectores por lo que tienen un número de elementos limitado. Una pila es un contenedor de datos cuyo comportamiento está regido por el principio LIFO (Last Input First Output).

DESARROLLO

Primero antes que todo dudemos crear otra carpeta con el nombre de “Estructura Lineales” dentro de esa carpeta crearemos 4 paquetes los cuales contendrán las clases, los nombres de los paquetes serán los siguientes:



Anterior mente vimos los nombres de los paquetes, importante el crear la clase pilaTDA ya que será una parte importante para poder correr las demás clases con los métodos dentro.

Contenido de la clase pilaTDA:

```
public interface PilaTDA <T>{
    public boolean isEmptyPila();// Regresa true si la pila esta vacia.
    public void pushPila(T dato);// Inserta el dato en el tope de la pila.
    public T popPila();// elimina el elemento que esta en el tope de la pila.
    public T peekPila();//Regresa el elemento que esta en el tope, sin quitarlo
}
```

Gracias a esta clase y sus métodos podemos usar las demás clases y que funcionen sus métodos como, por ejemplo:

Ejercicio 1 Pila Estática: PilaA

A continuación, enseñaremos la estructura de la clase PilaA y hablaremos de sus métodos y sus funciones.

Para comenzar podemos ver que en la clase PilaA se tiene que implementar el TDA por lo que lo primero en hacer será implementar la clase PilaTDA y posterior a eso declaramos las variables *pila* y *tope*, y posterior a eso la inicializamos en un constructor.

```
public class PilaA <T> implements PilaTDA<T> {
    private T pila[];
    private byte tope;
    public PilaA(int max) {
        pila=(T[]) (new Object[max]);
        tope=-1;
    }
}
```

El método comprueba si una cadena está vacía o no. Este método devuelve verdadero si la cadena dada está vacía; de lo contrario, devuelve falso. En otras palabras, puede decir que este método devuelve verdadero si la longitud de la cadena es 0.

```
public boolean isEmptyPila()
{
    return (tope == -1);
}
```

El método isSpace(char ch) devuelve un valor booleano, es decir, verdadero si el carácter dado (o especificado) es un espacio en blanco. De lo contrario, el método devuelve falso.

```
public boolean isSpace()
{
    return (tope<pila.length-1);
}
```

El método pushPila() nos ayudara a insertar datos a la pila siempre se insertara el dato en la cima de la pila y lo que se ve en el método es que: primero busca si hay un espacio en blanco con el anterior método visto isSpace si es verdad, podremos meter un dato, si por lo contrario no hay espacios en blanco nos dirá que la pila esta llena.

```
public void pushPila(T dato)
{
    if (isSpace()) {
        tope++;
        pila[tope]=dato;
    }
    else
        Tools.imprimirErrorMSJE("PILA LLENA...!!");
}
```

El método popPila() nos ayudara a eliminar el dato que se encuentre en ese momento en la cima de la pila.

```
public T popPila() {
    T dato= pila[tope];
    tope--;
    return dato;
}
```

El método `peekPila()` nos dará el valor del dato que se encuentre en ese instante en la cima de la pila, importante mencionar que el dato solo será mostrado mas no eliminado.

```
public T peekPila() {
    return pila[tope];
}
```

Y por último los métodos que nos ayudaran a imprimir los datos, esto lo vamos a conseguir gracias a la recursividad y al concatenar los datos. Ya lo que esta haciendo el método es ir recorriendo y imprimiendo los datos.

```
public String toString() {
    return toString(tope);
}

private String toString(int i) {
    return (i>=0)?"tope==>" + i + "[" + pila[i] + "]" + "\n" + toString(i-1) : "";
}
}
```

Y como podemos ver el funcionamiento este método, pues gracias al menú: El menú contendrá los métodos de agregar datos, eliminar dato de la cima, ver el dato de la cima, y liberar la pila.

```
public class Menu {
    public static void main(String[] args) {
        operacionesPilaEstatica();
    }

    public static void operacionesPilaEstatica() {
        PilaA <Integer> pila = new PilaA((byte)10);
        String op="";
        do {
            op=Tools.boton("PUSH,POP,PEK,FREE,SALIR");
            switch(op) {
                case "PUSH":
                    pila.pushPila(Tools.LeerInt("Escrube un dato entero"));

```

```
Tools.imprimirMSJE("Datos de pila:\n"+pila.toString());
break;
case "POP":
if (pila.isEmptyPila())Tools.imprimirErrorMSJE("Pila vacia");
else
Tools.imprimirMSJE("Dato eliminado de la cima de la
pila:==>" +pila.popPila()+"\n"+pila.toString());
break;
case "PEEK":
if(pila.isEmptyPila()) Tools.imprimirErrorMSJE("Pila vacia");
else
Tools.imprimirMSJE("Dato de la cima de la pila:
==>" +pila.peekPila()+"\n"+pila.toString());
break;
case "FREE":
if (pila.isEmptyPila()) Tools.imprimirErrorMSJE("Pila vacia");
else {
pila=null;
pila= new PilaA<Integer>((byte)10);
}
break;
}
}while(!op.equals("SALIR"));
}
}
```

Ventanas emergentes:

MENU

×

?

SELECCIONA DANDO CLICK

PUSH

POP

PEEK

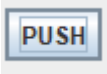
FREE


SALIR

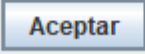
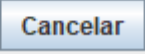


Agregar datos:


Entrada de datos



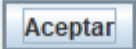
 Escriba un dato entero

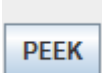
Salida

 Datos de pila:


tope==>4[5]
tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]



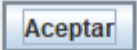
Ver dato de la sima:




Salida

 Dato de la cima de la pila: ==>5


tope==>4[5]
tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]



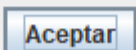
Borrar dato de la sima:



Salida

 Dato eliminado de la cima de la pila:==>5

tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]



Liberar pila:



Salida

 Pila vacia



Ejercicio 2 Pila Estática: PilaB

Lo que tiene de especial esta clase es que no depende de otra clase para utilizar sus métodos ya que solo haciendo el menú y utilizando el Stack podremos utilizar los métodos que ya se encuentran en el "Stack" por eso solo es un menú con las opciones antes vistas: agregar datos, eliminar dato de la cima, ver el dato de la cima, y liberar la pila.

```
public class PilaB {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Stack <Integer> pila = new Stack();  
        int dato;  
        String op="";  
        do {  
            op=Tools.boton2("Push,Pop,Peek,Free,Salir");  
            switch (op) {  
                case "Push": //agregar  
                    dato=(Tools.LeerInt("Escribe un entero"));  
                    pila.push(dato);  
                    Tools.imprimirMSJE("Datos de la pila: \n"+pila);  
                    break;  
                case "Pop":  
                    if(pila.isEmpty()) Tools.imprimirErrorMSJE("Pila vacia");  
                    else  
                        Tools.imprimirMSJE("Dato eliminado de la cima de la pila:  
                        "+pila.pop()+"\n"+pila.toString());  
                    break;  
                case "Peek":  
                    if (pila.isEmpty()) Tools.imprimirErrorMSJE("Pila vacia");  
                    else  
                        Tools.imprimirMSJE("Dato de la cima de la pila:  
                        ==>"+pila.peek()+" "+pila.toString());  
                    break;  
                case "Free":
```

```
if(pila.isEmpty()) Tools.imprimirErrorMSJE("Pila vacia");
else {
pila=null;
pila=new Stack();
}
break;
}
} while(!op.equals("Salir"));
```

Este metodo hace lo mismo que el anterior visto en la PilaA solo que este saca los métodos de el "Stack" como vamos a ver a continuación es la misma función:

Ventanas emergentes:



Ejercicio 3 Pila Estática: PilaC

En esta clase lo distinto a usar será que envés de usar el Stack o una pila, estaremos usando una lista por eso vamos a declarar la variable pila como una lista y posterior a eso en los demás métodos la diferencia es que vamos a estar invocando o llamando a la variable lista en todos los métodos, esas son las diferencias de una clase a otra porque en si tienen el mismo funcionamiento, A continuación vamos a enseñar la clase PilaC:

```
import java.util.ArrayList;

public class PilaC <T> implements PilaTDA <T> {
private ArrayList pila;
```



```
int tope;

public PilaC() {
    pila = new ArrayList();
    tope = -1;
}

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

public boolean isEmptyPila(){
    return pila.isEmpty();
}

public void vacia() {
    pila.clear();
}

public void pushPila(Object dato) {
    pila.add(dato);
    tope++;
}

public T popPila() {
    T dato = (T) pila.get(tope);
    pila.remove(tope);
    tope--;
    return dato;
}

public T peekPila() {
    return (T) pila.get(tope);
}

public String toString() {
    return toString(tope);
}

private String toString(int i) {
    return (i>=0)?"tope==>" + i + "["+pila.get(i)+"]\n"+toString(i-1):"";
}
```

```
}
```

Posterior mente se mostrará lo que es el menú para arrancar la clase:

```
public class MenuC {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        PilaC <Integer> pila = new PilaC();
        String op="";
        do {
            op=Tools.boton("PUSH,POP,PEK,FREE,SALIR");
            switch(op) {
                case "PUSH":
                    pila.pushPila(Tools.LeerInt("Escribe un dato entero"));
                    Tools.imprimirMSJE("Datos de pila:\n"+pila.toString());
                    break;
                case "POP":
                    if (pila.isEmptyPila())Tools.imprimirErrorMSJE("Pila vacia");
                    else
                        Tools.imprimirMSJE("Dato eliminado de la cima de la
                        pila:==>" +pila.popPila()+"\n"+pila.toString());
                    break;
                case "PEEK":
                    if(pila.isEmptyPila()) Tools.imprimirErrorMSJE("Pila vacia");
                    else
                        Tools.imprimirMSJE("Dato de la cima de la pila:
                        ==>" +pila.peekPila()+"\n"+pila.toString());
                    break;
                case "FREE":
                    if (pila.isEmptyPila()) Tools.imprimirErrorMSJE("Pila vacia");
                    else {
                        pila=null;
                        pila= new PilaC<Integer>();
                    }
                    break;
            }
        }
```

```
}while(!op.equals("SALIR"));  
}
```

Ventana emergente:

MENU X

? SELECCIONA DANDO CLICK

PUSH POP PEEK FREE SALIR

Agregar datos:

Entrada de datos X

PUSH

i Escriba un dato entero

1

Aceptar Cancelar

Salida X

? Datos de pila:

tope==>4[5]
tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]

Aceptar

Ver dato de la cima:

PEEK

Salida X

? Dato de la cima de la pila: ==>5

tope==>4[5]
tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]

Aceptar

Borrar dato de la cima:

POP

Salida X

? Dato eliminado de la cima de la pila:==>5

tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]

Aceptar

Liberar pila:

Salida X

X Pila vacia

Aceptar

Ejercicio 4 Pila Dinámica: Pila D.

Antes de utilizar la clase PilaD es importante crear una clase llamada Nodo ya que esta será la siguiente forma de utilizar una pila y para esto creamos la clase Nodo que contendrá a dentro los siguiente:

```
public class Nodo<T> {
    public T info;
    public Nodo<T> sig;
    public Nodo (T info) {
        this.info=info;
        this.sig=null;
    }
    public T getInfo() {
        return info;
    }
    public void setInfo(T info) {
        this.info = info;
    }
    public Nodo getSig() {
        return sig;
    }
    public void setSig(Nodo<T> sig) {
        this.sig = sig;
    }
}
```

Ya gracias a esta clase podremos utilizar correcta mente nuestra clase PilaD, el contenido de la clase PilaD será el siguiente mostrado.

```
public class PilaD <T> implements PilaTDA<T> {
    private Nodo <T> pila;
    public PilaD() {
        pila=null;
    }
}
```

```

}

public boolean isEmptyPila() {
    return (pila==null);
}

public void pushPila(T dato) {
    Nodo<T> tope = new Nodo<T>(dato);
    if (isEmptyPila()) pila=tope;
    else{
        tope.sig = pila;
        pila = tope;
    }
}

public T peekPila() {
    return (T) pila.getInfo();
}

public T popPila() {
    Nodo <T> tope = pila;
    T dato=(T) pila.getInfo();
    pila=pila.getSig();
    tope=null;
    return dato;
}

public String toString() {
    Nodo <T> tope=pila;
    return toString(tope);
}

private String toString(Nodo i) {
    return (i!=null)?"tope ==>"+ "["+i.getInfo()+"]\n"+toString(i.getSig()):"";
}

```


Esta clase es prácticamente igual a la de la lista solo que en este caso estaremos utilizando un nodo, como se puede ver a continuación su funcionamiento es el mismo que los demás, también cuenta con un menú y las mismas funciones solo que con otra clase.

```
public class MenuD {
```

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    PilaD<Integer> pila = new PilaD<Integer>();  
    String op="";  
    do {  
        op=Tools.boton("PUSH,POP,PEK,FREE,SALIR");  
        switch(op) {  
            case "PUSH":  
                pila.pushPila(Tools.LeerInt("Escrube un dato entero"));  
                Tools.imprimirMSJE("Datos de pila:\n"+pila.toString());  
                break;  
            case "POP":  
                if (pila.isEmptyPila())Tools.imprimirErrorMSJE("Pila vacia");  
                else  
                    Tools.imprimirMSJE("Dato eliminado de la cima de la  
pila:==>" +pila.popPila()+"\n"+pila.toString());  
                break;  
            case "PEEK":  
                if(pila.isEmptyPila()) Tools.imprimirErrorMSJE("Pila vacia");  
                else  
                    Tools.imprimirMSJE("Dato de la cima de la pila:  
==>" +pila.peekPila()+"\n"+pila.toString());  
                break;  
            case "FREE":  
                if (pila.isEmptyPila()) Tools.imprimirErrorMSJE("Pila vacia");  
                else {  
                    pila=null;  
                    pila= new PilaD<Integer>();  
                }  
                break;  
        }  
    }while(!op.equals("SALIR"));  
}
```

Ventanas emergentes:


MENU ×


 SELECCIONA DANDO CLICK

PUSH **POP** **PEEK** **FREE** **SALIR**

Agregar datos:

Entrada de datos ×


 **PUSH**

 **Escriba un dato entero**

1

Aceptar **Cancelar**

Salida ×


 **Datos de pila:**

tope==>4[5]
tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]

Aceptar

Ver dato de la cima:

Salida ×

 **Dato de la cima de la pila: ==>5**


tope==>4[5]
tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]

Aceptar

PEEK

Borrar dato de la cima:

Salida ×

 **Dato eliminado de la cima de la pila:==>5**

tope==>3[4]
tope==>2[3]
tope==>1[2]
tope==>0[1]

Aceptar

POP

Liberar pila:

FREE

Salida

×



Pila vacia

Aceptar



EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA



TECNOLÓGICO
NACIONAL DE MÉXICO



Ejercicios propuestos por la maestra:

Ejercicio 1: Teníamos que hacer en clase nuestra propia pila haciendo una clase y eligiendo uno de las clases enseñadas por la maestra podría ser con: arraylist, Nodo, Stack, Normal.

En mi caso elegí hacer el ejercicio con la clase arraylist, a continuación, presentare mi código echo en clase:

```
public class Ejercicio1<T> implements PilaTDA <T> {
private ArrayList pila;
int tope;
public Ejercicio1() {
pila = new ArrayList();
tope = -1;
}
public int Size() {
return pila.size();
}
public boolean isEmptyPila(){
return pila.isEmpty();
}
public void vacia() {
pila.clear();
}
public void pushPila(Object dato) {
pila.add(dato);
tope++;
}
```

```

}

public T popPila() {
    T dato = (T) pila.get(tope);
    pila.remove(tope);
    tope--;
    return dato;
}

public T peekPila() {
    return (T) pila.get(tope);
}

public String toString() {
    return toString(tope);
}

private String toString(int i) {
    return (i >= 0) ? "tope==>" + i + "[" + pila.get(i) + "]\n" + toString(i-1) : "";
}
}
}

```

Ejercicio 2: Balanceo de una ecuación.

```

public class Ejercicio2 {
    public static void main(String[] args) {
        String expresion = JOptionPane.showInputDialog(null, "Ingrese expresión:");
        boolean balanceado = esbalanceado(expresion);
        if (balanceado) {
            JOptionPane.showMessageDialog(null, "Paréntesis balanceados.");
        } else {
            JOptionPane.showMessageDialog(null, "No tiene paréntesis balanceados.");
        }
    }

    public static boolean esbalanceado(String cade) {
        Ejercicio1<Character> pila = new Ejercicio1();
        String op = "";
        for (int i = 0; i < cade.length(); i++) {
            char character = cade.charAt(i);

```

```
if(caracter == '(') {
    pila.pushPila(caracter);
}else if(caracter == ')') {
    if(!pila.isEmptyPila() && pila.peekPila() == '(') {
        pila.popPila();
    }else {
        return false;
    }
}
return pila.isEmptyPila();
}
```

Ventanas emergentes:

Es balanceado:

Entrada	×	Mensaje	×
<div style="border: 1px solid #ccc; padding: 10px; background-color: #f9f9f9;"> <div style="display: flex; align-items: center;"> <div style="background-color: #4caf50; color: white; border-radius: 50%; width: 30px; height: 30px; display: flex; align-items: center; justify-content: center; margin-right: 10px;">?</div> <div> Ingrese expresión: <input style="width: 150px;" type="text" value="(a-b)/(a+b)"/> </div> </div> <div style="display: flex; justify-content: flex-end; gap: 10px; margin-top: 10px;"> <div style="border: 1px solid #ccc; padding: 5px 15px; background-color: #4caf50; color: white;">Aceptar</div> <div style="border: 1px solid #ccc; padding: 5px 15px; background-color: #ccc;">Cancelar</div> </div> </div>		<div style="border: 1px solid #ccc; padding: 10px; background-color: #f9f9f9;"> <div style="display: flex; align-items: center;"> <div style="background-color: #2196f3; color: white; border-radius: 50%; width: 30px; height: 30px; display: flex; align-items: center; justify-content: center; margin-right: 10px;">i</div> <div> Paréntesis balanceados. </div> </div> <div style="text-align: right; margin-top: 10px;"> <div style="border: 1px solid #ccc; padding: 5px 15px; background-color: #2196f3; color: white;">Aceptar</div> </div> </div>	

No es balanceado:

Entrada	×	Mensaje	×
<div style="border: 1px solid #ccc; padding: 10px; background-color: #f9f9f9;"> <div style="display: flex; align-items: center;"> <div style="background-color: #4caf50; color: white; border-radius: 50%; width: 30px; height: 30px; display: flex; align-items: center; justify-content: center; margin-right: 10px;">?</div> <div> Ingrese expresión: <input style="width: 150px;" type="text" value="((a+b/c))"/> </div> </div> <div style="display: flex; justify-content: flex-end; gap: 10px; margin-top: 10px;"> <div style="border: 1px solid #ccc; padding: 5px 15px; background-color: #4caf50; color: white;">Aceptar</div> <div style="border: 1px solid #ccc; padding: 5px 15px; background-color: #ccc;">Cancelar</div> </div> </div>		<div style="border: 1px solid #ccc; padding: 10px; background-color: #f9f9f9;"> <div style="display: flex; align-items: center;"> <div style="background-color: #2196f3; color: white; border-radius: 50%; width: 30px; height: 30px; display: flex; align-items: center; justify-content: center; margin-right: 10px;">i</div> <div> No tiene paréntesis balanceados. </div> </div> <div style="text-align: right; margin-top: 10px;"> <div style="border: 1px solid #ccc; padding: 5px 15px; background-color: #2196f3; color: white;">Aceptar</div> </div> </div>	

EJERCICIO DE TABLA CONVERSIÓN A POSTFIJA

```

1 package Ejemplos;
2
3 import java.util.Stack;
4 import javax.swing.JOptionPane;
5 import javax.swing.JScrollPane;
6 import javax.swing.JTable;
7 import javax.swing.table.DefaultTableModel;
8
9 public class ConversionInfijaPosfija {
10     public static void main(String[] args) {
11         String infija = JOptionPane.showInputDialog(null, "Ingrese una expresión infija:");
12         String posfija = "";
13         Stack<Character> pila = new Stack<Character>();
14         int longitud = infija.length();
15         DefaultTableModel modelo = new DefaultTableModel();
16         modelo.addColumn("Carácter");
17         modelo.addColumn("Estado actual de la pila");
18         modelo.addColumn("Expresión posfija");
19         for (int i = 0; i < longitud; i++) {
20             char caracter = infija.charAt(i);
21             String estadoPila = "";
22             for (Character c : pila) {
23                 estadoPila += c;
24             }
25             if (caracter == '(') {
26                 pila.push(caracter);
27             } else if (Character.isLetterOrDigit(caracter)) {
28                 posfija += caracter;
29             } else if (caracter == ')') {
30                 while (!pila.empty() && pila.peek() != '(') {
31                     posfija += pila.pop();
32                 }
33                 if (!pila.empty()) {
34                     pila.pop();
35                 }
36             } else {
37                 while (!pila.empty() && obtenerPrecedencia(pila.peek()) >= obtenerPrecedencia(caracter)) {
38                     posfija += pila.pop();
39                 }
40                 pila.push(caracter);
41             }
42             modelo.addRow(new Object[] {caracter, estadoPila, posfija});
43         }
44         while (!pila.empty()) {
45             posfija += pila.pop();
46         }
47         modelo.addRow(new Object[] {"", "", posfija});
48         JTable tabla = new JTable(modelo);
49         JScrollPane scroll = new JScrollPane(tabla);
50         JOptionPane.showMessageDialog(null, scroll, "Estados de la pila", JOptionPane.PLAIN_MESSAGE);
51     }
52
53     public static int obtenerPrecedencia(char operador) {
54         switch (operador) {
55             case '+':
56             case '-':
57                 return 1;
58             case '*':
59             case '/':
60             case '%':
61                 return 2;
62             case '^':
63                 return 3;
64             default:
65                 return -1;
66         }
67     }
68 }

```

```

36         } else {
37             while (!pila.empty() && obtenerPrecedencia(pila.peek()) >= obtenerPrecedencia(caracter)) {
38                 posfija += pila.pop();
39             }
40             pila.push(caracter);
41         }
42         modelo.addRow(new Object[] {caracter, estadoPila, posfija});
43     }
44     while (!pila.empty()) {
45         posfija += pila.pop();
46     }
47     modelo.addRow(new Object[] {"", "", posfija});
48     JTable tabla = new JTable(modelo);
49     JScrollPane scroll = new JScrollPane(tabla);
50     JOptionPane.showMessageDialog(null, scroll, "Estados de la pila", JOptionPane.PLAIN_MESSAGE);
51 }
52
53 public static int obtenerPrecedencia(char operador) {
54     switch (operador) {
55         case '+':
56         case '-':
57             return 1;
58         case '*':
59         case '/':
60         case '%':
61             return 2;
62         case '^':
63             return 3;
64         default:
65             return -1;
66     }
67 }
68 }
69

```

Este programa convierte una expresión aritmética en notación infija a notación posfija utilizando una pila. La notación infija es la forma común de escribir una expresión

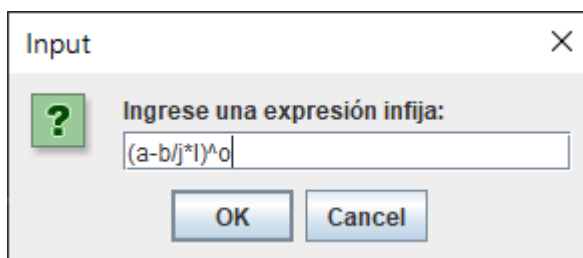
aritmética, como por ejemplo $(5+3)*2$, mientras que la notación posfija, también conocida como notación polaca inversa, coloca los operadores después de los operandos, como $5\ 3\ +\ 2\ *$.

El programa comienza pidiendo al usuario que ingrese una expresión infija utilizando un cuadro de diálogo de entrada de texto. A continuación, el programa crea una pila de caracteres y un modelo de tabla con tres columnas: "Carácter", "Estado actual de la pila" y "Expresión posfija". El programa luego itera a través de cada carácter de la expresión infija y realiza una serie de operaciones.

Si el carácter es un paréntesis izquierdo, se coloca en la pila. Si el carácter es una letra o un número, se agrega directamente a la expresión posfija. Si el carácter es un paréntesis derecho, se extraen los operadores de la pila y se agregan a la expresión posfija hasta que se encuentra el paréntesis izquierdo correspondiente, que se descarta. Si el carácter es un operador, se extraen los operadores de la pila y se agregan a la expresión posfija hasta que se encuentra un operador con una precedencia menor o igual que la del operador actual, y luego se agrega el operador actual a la pila.

Después de cada iteración, se agrega una fila a la tabla con información sobre el carácter actual, el estado actual de la pila y la expresión posfija. Cuando se han procesado todos los caracteres de la expresión infija, se extraen todos los operadores restantes de la pila y se agregan a la expresión posfija. Finalmente, se agrega una última fila a la tabla con la expresión posfija completa.

La tabla se muestra en un cuadro de diálogo de mensaje utilizando un JScrollPane para permitir desplazamiento si es necesario. La tabla muestra el estado de la pila después de cada carácter procesado y la expresión posfija en construcción, lo que puede ayudar a entender cómo funciona el algoritmo.



Estructuras de Datos Lineales.
(s. f.). http://webdiis.unizar.es/~elvira/eda/addeda/teruel/Estructuras_Lineales.html

Henry, R. (2022). ¿Qué es una estructura de datos en programación? | Henry. Henry.
<https://blog.soyhenry.com/que-es-una-estructura-de-datos-en-programacion/#:~:text=Las%20estructuras%20de%20datos%20lineales,otro%20relacionados%20en%20forma%20lineal>

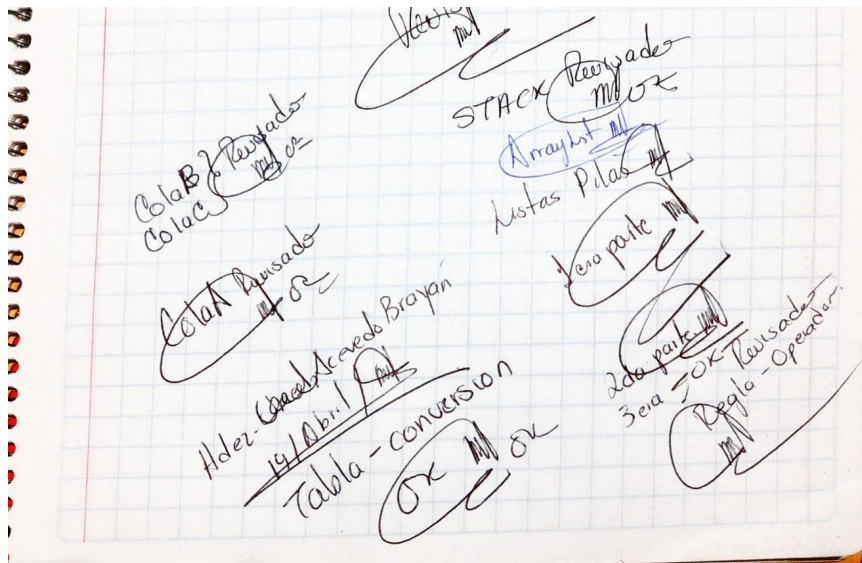
Flores, G. (s. f.). Que son las pilas estaticas y dinamicas? – La-Respuesta.com.
<https://la-respuesta.com/articulos-populares/que-son-las-pilas-estaticas-y-dinamicas/>

School, T. (2022). Estructura de datos en Java. Tokio School.
<https://www.tokioschool.com/noticias/estructura-datos-java/#:~:text=%2C%20ac%C3%ADlico%2C%20etc.,Estructuras%20de%20datos%20lineales,entre%20s%C3%AD%20en%20forma%20lineal>.

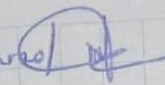
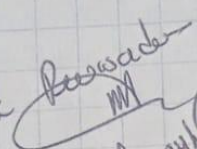
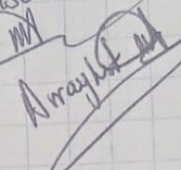
Ramón Toala Dueñas. (2020, 9 julio). Estructuras Lineales: Pilas y Colas en java [Vídeo]. YouTube. <https://www.youtube.com/watch?v=2Mr11Zr77nw>

Prof. Edgar Tista. (2020, 23 octubre). Definición de estructuras lineales [Vídeo]. YouTube. <https://www.youtube.com/watch?v=61tQkR8A6sk>

Firmas:



Brayan:

Estructura
Pila 22/Klaro/ 
Tema 3
STACx 
Amaylt 
24/03/2023

19/Abr/2023
Pilas / falta concatenar

ColaA 20/Abr/2023

Pedro:

La ultima no pude ir a su clase porque el clima se puso feo (le mande mensaje).

Cesar: No las mando ☹