Instituto tecnológico de Orizaba

Estructura De Datos

Unidad 3: Recursividad

Carrera: Ingeniería en Sistemas Computacionales

Grupo: 3g2B

Hora: 17:00 – 18:00 hrs

Profesora: Martínez Castillo María Jacinta

Alumno: Moran De la Cruz Aziel

Reporte de Prácticas

Introducción

Las estructuras lineales son un tipo de estructura de datos que se caracterizan por almacenar los datos en una secuencia ordenada, en la que cada elemento está conectado con su predecesor y sucesor. Esta secuencia se puede recorrer en una única dirección, desde el primer elemento hasta el último. Ejemplos comunes de estructuras lineales incluyen listas enlazadas, pilas, colas y arreglos unidimensionales. Las estructuras lineales se utilizan comúnmente en la programación debido a su simplicidad y eficiencia en la manipulación de datos. Además, se pueden utilizar en una variedad de aplicaciones, desde la gestión de datos en bases de datos hasta la creación de algoritmos para la resolución de problemas complejos. Comprender las estructuras lineales y sus características es esencial para cualquier desarrollador de software que busque crear soluciones eficientes y efectivas para una amplia variedad de problemas de programación.

Competencia Específica

Con base al tema que vimos en clase, aprendemos y elaboramos nuestros programas con los diferentes tipos de estructuras de datos que pudimos ver, esto con el fin de almacenar diferentes datos que el usuario puede dar y nosotros como programadores debemos almacenarlos en una base de datos.

Marco Teórico

3.0 Clases genéricas arreglos dinámicos (memoria dinámica)

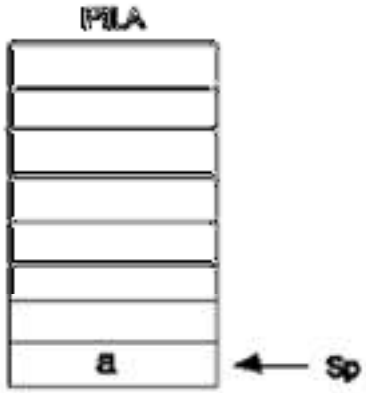
Es una estructura de almacenamiento de datos que crece o mengua dinámicamente conforme los elementos se agregan o se eliminan. Se suministra dentro de las librerías estándar en muchos lenguajes modernos de programación

3.1. Pilas

Es una colección de datos a los cuales se puede acceder mediante un extremo, que se conoce generalmente como tope. Las pilas no son estructuras fundamentales de datos Para su representación requieren el uso de otras estructuras de datos, como arreglos o listas.

3.1.1 Representación en memoria

Nosotros ahora usaremos los arreglos. Por lo tanto, debemos definir el tamaño máximo de la pila, además de un apuntador al último elemento insertado en la pila el cual denominaremos SP. La representación gráfica de una pila es la siguiente:



Como utilizamos arreglos para implementar pilas, tenemos la limitante de espacio de memoria reservada. Una vez establecido un máximo de capacidad para la pila, ya no es posible insertar más elementos.

3.1.2 Operaciones básicas

Agregar datos a la estructura; retirar datos de la estructura; borrar datos de la estructura; actualizar datos en la estructura; recorrer la estructura para ver, leer, modificar, borrar, grabar a disco, o bien realizar algo con los datos de la misma y, posiblemente algunas otras operaciones más o menos complicadas.

3.1.3 Aplicaciones

Nos permiten tener una batería de herramientas para solucionar ciertos tipos de problemas. Además, nos permiten hacer un software más eficiente optimizando recursos, algo muy útil para IoT y para los entornos que trabajan con Big Data.

3.2 Colas

Es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista.

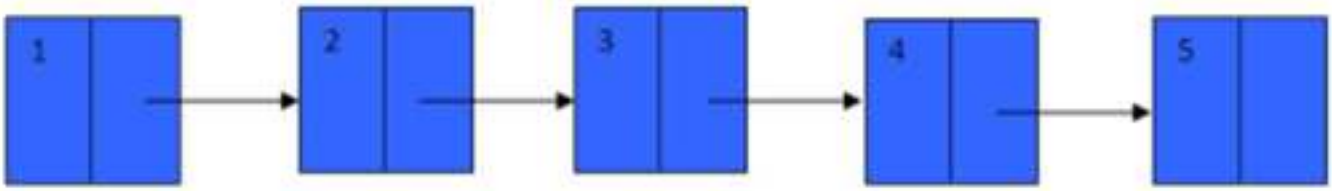
3.2.1 Representación en memoria

Las colas se pueden representar de la misma forma que las pilas.

- Arreglos: Los arreglos son la forma fácil para representar colas, aunque, como se mencionó en el módulo anterior, es el número fijo de elementos que se pueden almacenar.

4	5	9	6

- Listas enlazadas: Permiten almacenar elementos, sin tener problemas de desperdicio de memoria por la inserción y eliminación de elementos.



3.2.2 Operaciones básicas

Agregar datos a la estructura; retirar datos de la estructura; borrar datos de la estructura; actualizar datos en la estructura; recorrer la estructura para ver, leer, modificar, borrar, grabar a disco, o bien realizar algo con los datos de la misma y, posiblemente algunas otras operaciones más o menos complicadas.

3.2.3 Tipos de colas: Simples, Circulares y Bicolás

Cola Lineal o Simple: Las inserciones se llevarán a cabo por el FINAL de la cola, mientras que las eliminaciones se harán por el FRENTE.

Cola circular: Para hacer un uso más eficiente de la memoria disponible se trata a la cola como a una estructura circular. Es decir, el elemento anterior al primero es el último.

DOBLE COLA (BICOLA): Es una generalización de una estructura de cola simple. En una doble cola, los elementos pueden ser insertados o eliminados por cualquiera de los extremos. Es decir, se pueden insertar y eliminar valores tanto por el FRENTE como por el FINAL de la cola

3.2.4 Aplicaciones

Las colas se utilizan en sistemas informáticos, transportes y operaciones de investigación (entre otros), dónde los objetos, personas o eventos son tomados como datos que se almacenan y se guardan mediante colas para su posterior procesamiento.

3.3 Listas

Es una estructura de datos residente en la memoria que se llena con un conjunto de nombres extraídos de una fuente externa, como por ejemplo un archivo sin formato. Una vez creada y llenada con nombres, una lista de datos está disponible para utilizarla en las solicitudes de búsqueda subsiguientes.

3.3.1 Operaciones básicas

Las operaciones más comunes sobre listas son inicializar la lista, destruir la lista, insertar un elemento en la lista, borrar un elemento de la lista, consultar el elemento en una posición, buscar un elemento en la lista, tamaño de la lista o vaciar la lista.

3.3.2 Tipos de listas simplemente enlazadas doblemente enlazadas y circulares

- Listas simplemente enlazadas: cada nodo contiene una sola parte de enlace.
- Listas doblemente enlazadas: cada nodo contiene dos partes de enlace al siguiente nodo y al anterior nodo.
- Listas circulares: cada nodo contiene una parte de enlace al siguiente nodo, pero su diferencia está que el último nodo se enlaza con el primer nodo de la lista.

3.3.3 Aplicación

Las listas tienen las funciones de eliminar e introducir datos nuevos, siempre y cuando estos cumplan con los requisitos del tipo de dato que puede ingresar en las listas.

Material y Equipo

- Computadora.
- Software y versión usados.
- Materiales de apoyo para el desarrollo de la práctica

Desarrollo de las prácticas

Interface Pila TDA

1.- Esta clase es de tipo Interface ya que la usaremos cómo plantilla para próximas clases ya que solo contendrá el nombre de los métodos.

```
public interface PilaTDA <T>{

    //El interface obliga que implementen los métodos que agreguemos

    public boolean isEmptyPila(); //Regresa true si la pila está vacia.
    public void pushPila(T dato); //Inserta el dato en el tope de la pila.
    public T popPila(); //Elimina el elemento que está en el tope de la pila.
    public T peekPila(); //Regresa el elemento que está en el tope, sin quitarlo.

}
```

Pila Estática Pila A

1.- La clase de Pila A lleva dos variables, una privada de T pila[] y otra de byte tope.

```
public class PilaA <T> implements PilaTDA<T> {
    private T pila[];
    private byte tope;
```

2.- El método de PilaA es para determinar el espacio que tendrá la pila, en nuestro caso será el límite de max o máximo de valores permitidos.

```
public PilaA(int max) {
    pila=(T[])(new Object[max]);
    tope=-1;
}
```

3.- El método de isEmptyPila es un método que usaremos para saber si la pila está vacia comparando el tope de dicha pila con el valor de -1.

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

4.- El método de isSpace nos servirá para saber si aún le queda espacio a la pila comparando el tope con la longitud del tamaño de la pila.

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

5.- pushPila será el método que usemos para agregar datos dentro de la pila, primero usaremos un if para saber si hay espacio en la pila, si sí hay entonces mete el valor, si no hay espacio entonces imprimiría un mensaje de error diciendo que la pila está llena.

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

6.- popPila es el método que nos ayudará a eliminar el dato que fue añadido a lo último, en este caso sería el tope de la pila en ese momento ya que saca los valores del último hasta el primero, sin modificar el orden.

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

7.- peekPila será el método que retornará el valor que esté hasta arriba en la pila que igual sería el último agregado.

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

8.- toString es un método de cadena que recopilará los valores agregados, del primero hasta el último, este método se agrega dos veces, el primero para retornar el tope de la pila y el segundo para organizar ese valor de tope para usarlo cómo herramienta en el menú para el método de peek o pop.

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

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

MenuPilaEstatica

Para probar estos métodos realizamos una clase menú en donde usamos case para hacer el llamado a los métodos, en algunos casos cómo el de pop o peek agregamos algunos if y else por si la pila estaba vacía o llena, o para llamar el método toString para imprimir el tope de pila que será eliminado o está en la cima, el case de Free el if lo usamos para saber si la pila está vacía para mandar mensaje que no hay nada en la pila, si no el else vaciaría la pila.

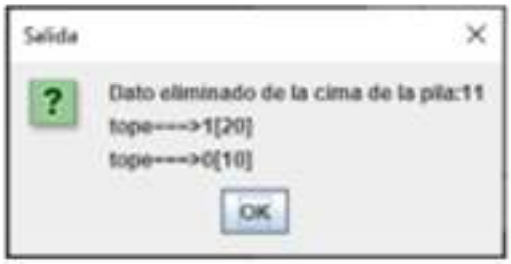
```java
public class MenuPilaEstatica {

    public static void operacionesPilaEstatica() {
        PilaA <Integer> pila = new PilaA((byte) 10);
        String op="";
        do {
            op=Tools.seleccionBotonPilas("PUSH, POP, PEEK, FREE, SALIR");
            switch(op) {
            case "PUSH": pila.pushPila(Tools.leerInt("Dato:"));
            Tools.imprimirMSJE("Datos de la pila:\n"+pila.toString());break;
            case "POP": if (pila.isEmptyPila())Tools.imprimirErrorMSJE("Pila vacía");
            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 vacía");
            else
                Tools.imprimirMSJE("Dato de la cima de la pila: ==>"+pila.peekPila()+"\n"+pila.toString());break;
            case "FREE": if(pila.isEmptyPila())Tools.imprimirErrorMSJE("Pila vacía");
            else {
                pila = new PilaA((byte)10);
            }
            break;
            }
        } while(!op.equals("SALIR"));
    }
}
```

Resultado

Al correr el programa nos sale el menú de opciones que podemos tomar.

1.- "PUSH" esta opción mandará a llamar al método de pushPila que será el encargado de meter nuestros valores en la pila y a la vez nos imprimirá los valores que llevemos guardados después de agregar uno nuevo.

2.- "POP" será la opción que saque los valores de la pila comenzando desde el último valor agregado o el que está siendo el tope de la pila, sale en orden de la posición mayor a la menor.



3.- "PEEK" nos mostrará el valor que esté en el tope de la pila, o sea el último valor agregado.

4.- "FREE" será la opción que limpiará toda la pila y la dejará vacía borrando cada uno de los valores existentes.

5.- "SALIR" cómo lo dice su nombre es la opción para terminar el programa.

Pila Dinámica

Pila C

1.- El método PilaC es para crear el ArrayList con iniciación del tope en el -1.

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

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

2.- Size lo ocupamos para saber el tamaño del espacio de la pila.

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

3.- isEmptyPila lo ocupamos para llamar el método de isEmpty de la clase PilaTDA para verificar que la pila esté vacía.

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

4.- El método vaciar lo ocupamos para limpiar la pila, que quede sin valores dentro.

```
public void vaciar() {
    pila.clear();
}
```

5.- El resto de métodos es similar al de pilas estáticas, solo que cambia las invocaciones de algunos métodos cómo el push usamos el pila.add y aumentamos el tamaño del tope, en pop pila usamos el pila.remove(tope) y en peek usamos pila.ger(tope).

```java
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):"";
}
```

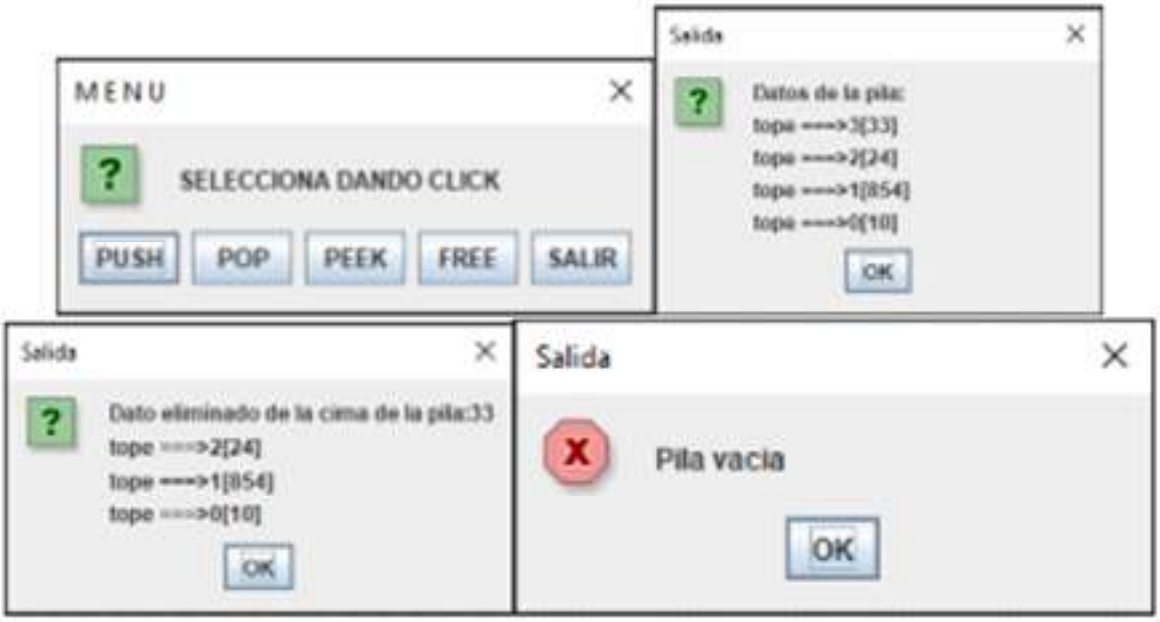
MenuArrayList

Para probar los métodos de la pila dinámica podemos usar el menú de la pila estática y solo hacer las respectivas modificaciones de agregar los métodos de las pilas dinámicas.

```java
public class MenuArrayList {

    public static void operacionesArrayList() {
        PilaC <Integer> pila = new PilaC();
        String op="";
        do {
            op=Tools.seleccionBotonPilas("PUSH, POP, PEEK, FREE, SALIR");
            switch(op) {
            case "PUSH": pila.pushPila(Tools.leerInt("Escribe un valor entero:"));
            Tools.imprimirMSJE("Datos de la 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.vaciar();
            }
            break;
            }
        } while(!op.equals("SALIR"));
    }

}
```

Resultado

Al iniciar el programa nos muestra el mismo menú que pilas estáticas, incluso hace las mismas funciones, aunque la diferencia que encontramos aquí es que no hay limitante de espacio dentro de la pila, ya que se adaptará a lo que necesitemos.



Stack

StackPila

Para esta clase solamente tendremos que importar los métodos del propio Java usando import java.util.Stack y sería casi los mismo métodos que las pilas estáticas, en este caso para optimizar el uso del Stack usamos el mismo menú solamente con algunas modificaciones de los métodos invocados.

```
public class StackPila {

    public static void StackPila() {

        Stack <Integer> pila = new Stack();
        int dato;
        String op="";
        do {
            op=Tools.seleccionBotonPilasD("Insertar Pila, Elimina Pila, Mostrar Cima, Libera, SALIR");
            switch(op) {
            case "Insertar Pila": dato=Tools.leerInt("Dato a insertar");
            pila.push(dato);
            Tools.imprimirMSJE("Dato en la pila: "+pila);break;
            case "Elimina Pila": try {
                Tools.imprimirMSJE("Dato de la cima de la pila antes de eliminar es "+pila.peek());
                pila.pop();
                Tools.imprimirMSJE("La pila ahora tiene "+pila+ " ");
            } catch(EmptyStackException e) {
                Tools.imprimirErrorMSJE("Pila vacia");
            }break;
            case "Mostrar Cima": if(pila.isEmpty())Tools.imprimirErrorMSJE("Pila vacia");
            else
                Tools.imprimirMSJE("Dato de la cima de la pila: ==>"+pila.peek()+"\n"+pila.toString());break;
            case "Libera": if(pila.isEmpty())Tools.imprimirErrorMSJE("Pila vacia");
            else {
                pila=null;
                pila=new Stack();
            }
            break;
            }
        } while(!op.equals("SALIR"));
    }
}
```

Resultados

Los resultados son los mismos de las anteriores pilas ya que la creación de estas 3 clases o métodos son para enseñarnos que hay diferentes formas de crear pila y mostrarnos cómo funciona cada una con sus ventajas y desventajas.

Nodos

Nodo

En esta clase llamada nodo la usaremos para obtener valores y después pasarlos al tope de una pila, y al tomar otro valor el anterior valor lo mueve al tope de una pila y así sucesivamente. Los métodos que tendrá esta clase son los de get y set de las variables info y sig, info tendrá un tipo de variable genérico de T y el de sig tendrá Nodo.

```
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 <T> getSig() {
        return sig;
    }

    public void setSig(Nodo <T >sig) {
        this.sig = sig;
```

PilaD

La clase de pilaD hará lo que se mencionó en la clase nodo, el tomar un dato con el usará el push, después si se agrega otro valor se usa el popPila para sacar el valor anterior y ponerlo en el tope de una pila. Al igual que habrá un método para saber si la pila está vacía y un toString para juntar todos los valores de las pilas para una impresión de los valores que llevemos en los topes.

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

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

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

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

MenuNodo

En esta clase de menú la usaremos para que el usuario pueda interactuar y probar los métodos de las dos clases anteriores.

```
public class MenuNodo {

    public static void operacionesNodo() {
        PilaD <Integer> pila = new PilaD();
        String op="";
        do {
            op=Tools.seleccionBotonPilas("PUSH, POP, PEEK, FREE, SALIR");
            switch(op) {
            case "PUSH": pila.pushPila(Tools.leerInt("Escribe un valor entero:"));
            Tools.imprimirMSJE("Datos de la 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=new PilaD();
            }
            break;
            }
        } while(!op.equals("SALIR"));
    }
}
```

Resultados

Al ejecutar el programa se nos muestra el mismo que los anteriores programas aunque en esta vez algunos botones harán las mismas funciones sólo que con unas pequeñas diferencias.

1.- "PUSH" esta opción pedirá al usuario que ingrese un valor y la insertará, para que después al agregar otro dato el valor anterior para a ser insertado a la pila siendo el nuevo tope y así sucesivamente.

2.- "POP", "PEEK", "FREE" y "SALIR" hacen lo mismo que los anteriores programas.

Colas

ColaTDA

En esta no será una clase, sino que será una interface que usaremos en las siguientes clases para ya tener unos métodos preparados como base.

```
public interface ColaTDA <T>{

    public boolean isEmptyCola(); //Regresa true si la pila está vacia.
    public void pushCola(T dato); //Inserta el dato en el tope de la pila.
    public T popCola(); //Elimina el elemento que está en el tope de la pila.
    public T peekCola(); //Regresa el elemento que está en el tope, sin quitarlo.

}
```

ColaA

Comenzamos creando dos variables privadas una con tipo de variable T que será la cola[] y otro con byte que será u.

```
private T cola[];
private byte u;
```

1.- El primer método será el de ColaA para establecer los valores de las variables siendo cola que tenga un tamaño máximo con [max] y u que será la variable de posición iniciando en -1.

```
public ColaA(byte max) {
    cola = (T[])(new Object[max]);
    u=-1;
}
```

2.- "isEmptyCola" será un método de tipo booleano y hará la función de saber si la cola está vacía comparando u con -1, y si es igual a -1 entonces la cola está vacia.

```
public boolean isEmptyCola() {
    return (u==-1);
}
```

3.- "isSpaceCola" será booleano y hará la función de verificar si hay espacio en la cola comparando la u con la longitud de la cola.

```
public boolean isSpaceCola() {
    return (u<cola.length-1);
}
```

4.- "pushCola" tendrá la función que si la cola tiene espacio entonces agregará un valor introducido por el usuario así aumentando u, sino entonces mandará un mensaje indicando que la cola está llena.

```
public void pushCola(T dato) {
    if(isSpaceCola()) {
        u++;
        cola[u]=dato;
    }
    else {
        Tools.imprimirErrorMSJE("Cola llena");
    }
}
```

5.- "recorrePosición" es un método usando en el tema 1 de búsquedas secuenciales, este método se encargará que al momento de eliminar un valor que en este caso sería el que se encuentre en la posición 0 se recorra el valor que estaba en la posición 1.

```
public void recorrePosicion() {
    for(int j=0; j<u;j++) {
        cola[j]=cola[j+1];
    }
}
```

6.- "popCola" se encargará de sacar el valor que se encuentre en la posición 0 y después manda a llamar el método de "recorrePosición" para acomodar nuevamente el orden de la coal.

```
public T popCola() {
    T dato=cola[0];
        recorrePosicion();
    u--;
    return dato;
}
```

7.- "peekCola" tiene la función de mostrar el frente de la cola, en este caso lo que se encuentre en la posición 0, es por eso que solo retorna cola[0].

```
public T peekCola() {
    return cola[0];
}
```

8.- "freeCola" es el método con el objetivo de borrar todo, cola obtiene el valor de null y la u regresa a tener el valor de -1.

```
public void freeCola() {
    cola=null;
    u=-1;
}
```

9.- "toString" se encarga de juntar los datos de una manera adecuada para su impresión.

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

private String toString(int i) {
    return (i<=u)?"===>"+i+"["+cola[i]+"]\n"+toString(i+1):"";
}
```

MenuColaA

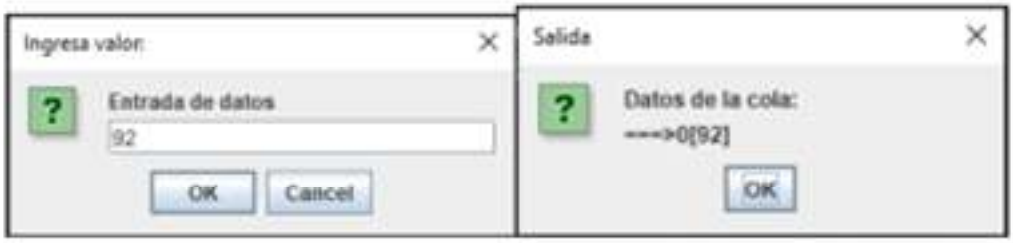
Será la clase para probar los métodos anteriores por medio de un menú de opciones de botones.

```
public class MenuColaA {

    public static void menuColasA() {
        ColaA <String> Cola = new ColaA((byte)10);
        String op="";
        do {
            op=Tools.seleccionBotonPilas("PUSH, POP, PEEK, FREE, SALIR");
            switch(op) {
            case "PUSH": Cola.pushCola(Tools.leerString("Ingresa valor:"));
            Tools.imprimirMSJE("Datos de la cola:\n"+Cola.toString());break;
            case "POP": if (Cola.isEmptyCola())Tools.imprimirErrorMSJE("Cola vacia");
            else
                Tools.imprimirMSJE("Dato eliminado del frente de la cola:" +Cola.popCola()+"\n"+Cola.toString());break;
            case "PEEK": if(Cola.isEmptyCola())Tools.imprimirErrorMSJE("Cola vacia");
            else
                Tools.imprimirMSJE("Dato de la cima de la cola: ==>"+Cola.peekCola()+"\n"+Cola.toString());break;
            case "FREE": if(Cola.isEmptyCola())Tools.imprimirErrorMSJE("Cola vacia");
            else {
                Cola.freeCola();
            }
            break;
            }
        } while(!op.equals("SALIR"));
    }
```

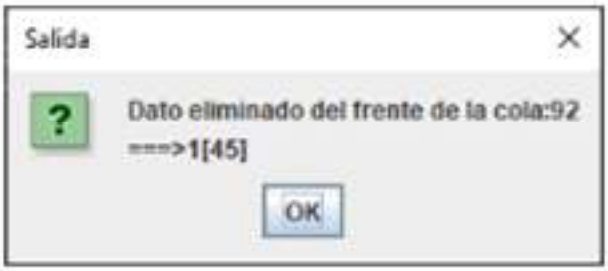
Resultados

Al iniciar el programa se nos mostrará el menú de opciones.

1.- "PUSH" se encargará de registrar el dato dado por el usuario y lo meterá dentro de la cola.

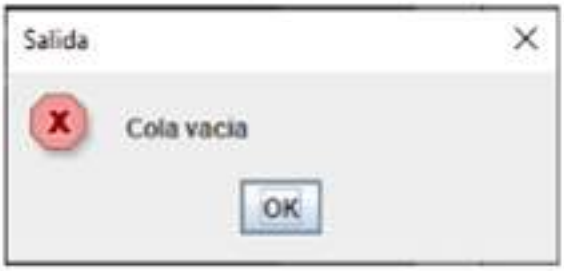


2.- "POP" se encargará de eliminar al valor de la posición 0 y nos mostrará el valor que se recorrerá a su posición.



3.- "PEEK" nos mostrará el valor que se encuentra hasta el frente de la cola.

4.- "FREE" eliminará todo valor que tenga la cola dentro.



ColaB

Esta clase tendrá los mismo métodos que la clase anterior sólo con la diferencia de que nos ahorraremos líneas de código ya que importaremos tres paquetes que están dentro de Java llamados java.util.Iterator, java.util.LinkedList y java.util.Queue. Estos paquetes ya llevan la función de lo que hacen las colas, ya solo invocamos sus funciones.

```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;

public class ColaB <T> implements ColaTDA <T>{ // LinkedList

    private Queue cola;

    public ColaB() {
        cola = new LinkedList();
    }

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

    public boolean isEmptyCola() {
        return (cola.isEmpty());
    }

    public T peekCola() {
        return (T) (cola.element());
    }

    public void vaciar() {
        cola.clear();
    }
```

```java
public void pushCola(T dato) {
    cola.add(dato);
}

public T popCola() {
    T dato;
    dato=(T) cola.element();
    cola.remove();
    return dato;
}

public String toString() {
    String cad="";
    byte j=0;
    for(Iterator i = cola.iterator();i.hasNext();) {
        cad+="["+i.next()+"] "+j;
        j++;
    }
    return cad;
}
```

El método de toString cambia ya que hará que cada valor obtenido se ponga alado del otro, anteriormente en pilas los valores salían en lista vertical, ahora en colas será igual pero de forma horizontal, uno tras de otra, es por eso que ocupamos el i.hasNext() para obtener el valor que sigue y el i.next() para incluirlo a la cadena.

MenuColaB

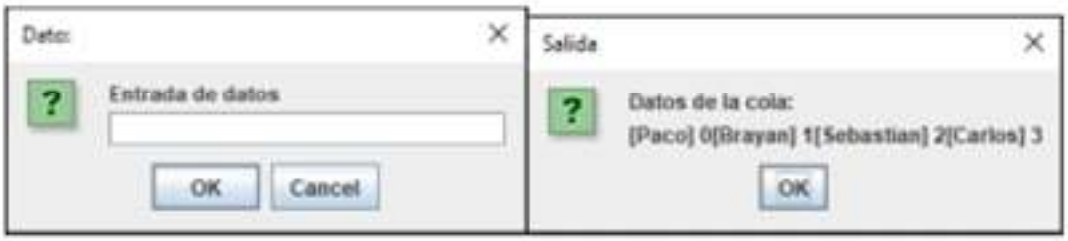
Esta clase es igual a la de "MenuColaA" sólo que cambiaremos el llamado de los métodos por los de la B.

```
public class MenuColaB {

    public static void menuColasB() {
        ColaB <String> Cola = new ColaB();
        String op="";
        do {
            op=Tools.seleccionBotonPilas("PUSH, POP, PEEK, FREE, SALIR");
            switch(op) {
            case "PUSH": Cola.pushCola(Tools.leerString("Dato:"));
            Tools.imprimirMSJE("Datos de la pila:\n"+Cola.toString());break;
            case "POP": if (Cola.isEmptyCola())Tools.imprimirErrorMSJE("Pila vacia");
            else
                Tools.imprimirMSJE("Dato eliminado de la cima de la pila:" +Cola.popCola()+"\n"+Cola.toString());break;
            case "PEEK": if(Cola.isEmptyCola())Tools.imprimirErrorMSJE("Pila vacia");
            else
                Tools.imprimirMSJE("Dato de la cima de la pila: ==>"+Cola.peekCola()+"\n"+Cola.toString());break;
            case "FREE": if(Cola.isEmptyCola())Tools.imprimirErrorMSJE("Pila vacia");
            else {
                Cola.vaciar();
            }
            break;
            }
        } while(!op.equals("SALIR"));
    }
}
```

Resultados

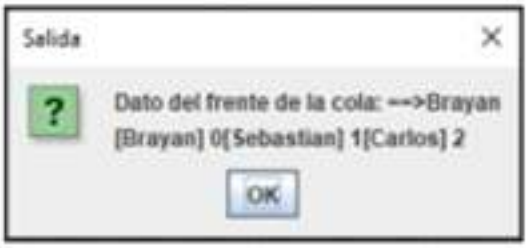
Tiene la misma funcionalidad que la ColaA sólo que a la hora de mostrar los valores podemos verlos de manera ordenada en cola, al usar el pop podemos ver con claridad que el valor en la posición 0 se elimina y el valor que estaba en la posición 1 pasa a estar en la posición 0.

1.- "PUSH"

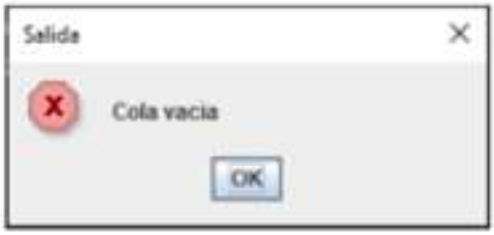


2.- "POP"

3.- "PEEK"



4.- "FREE"



ColaC

La escritura de los métodos de la "ColaC" es muy similar al de la "ColaB" sólo que aquí solo importamos java.util.ArrayList, el primer método que es ColaC contendrá que cola = new ArrayList() para crear una cola, mientras que u=0 para indicar que desde 0 es donde comenzará a aumentar las posiciones de los valores.

```
import java.util.ArrayList;

public class ColaC <T> implements ColaTDA <T>{ // ArrayList

    private ArrayList cola;
    byte u;

    public ColaC() {
        cola = new ArrayList();
        u=0;
    }

    public int Size() {
        return cola.size()-1;
    }

    @Override
    public boolean isEmptyCola() {
        return cola.isEmpty();
    }

    public void vaciar() {
        cola.clear();
    }
```

Los métodos de "pushCola", "popCola", "peekCola" cambian en esta clase ya que en pushCola usaremos cola.add(dato) para meter el valor puesto por el usuario dentro de la cola y aumentar la posición dentro de la misma, en el popCola usaremos un cola.get(0) para tomar el valor de la posición 0 y después removerlo con cola.remove(0) y disminuir la posición de u, peekCola retornara el valor que se encuentre en la posición 0.

```java
public void pushCola(Object dato) {
    cola.add(dato);
    u++;
}

public T popCola() {
    T dato=(T) cola.get(0);
    cola.remove(0);
    u--;
    return dato;
}

public T peekCola() {
    return (T)cola.get(0);
}

public String toString() {
    return toString(0);
}

private String toString(int i) {
    return (i<u)?""+i+"["+cola.get(i)+"] ===>"+toString(i+1):"";
}
```

MenuColaC

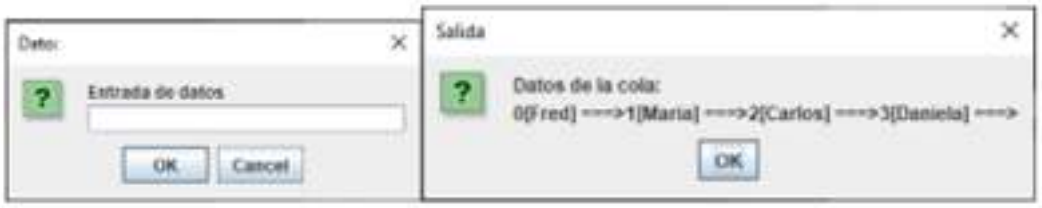
Nuevamente crearemos otra clase en la agregaremos el mismo menú y lo modificaremos con los métodos de la ColaC.

```java
public class MenuColaC {

    public static void menuColasC() {
        ColaC <String> Cola = new ColaC();
        String op="";
        do {
            op=Tools.seleccionBotonPilas("PUSH, POP, PEEK, FREE, SALIR");
            switch(op) {
            case "PUSH": Cola.pushCola(Tools.leerString("Dato:"));
            Tools.imprimirMSJE("Datos de la cola:\n"+Cola.toString());break;
            case "POP": if (Cola.isEmptyCola())Tools.imprimirErrorMSJE("Cola vacia");
            else
                Tools.imprimirMSJE("Dato eliminado del frente de la cola:" +Cola.popCola()+"\n"+Cola.toString());break;
            case "PEEK": if(Cola.isEmptyCola())Tools.imprimirErrorMSJE("Pila vacia");
            else
                Tools.imprimirMSJE("Dato del frente de la cola: ==>"+Cola.peekCola()+"\n"+Cola.toString());break;
            case "FREE": if(Cola.isEmptyCola())Tools.imprimirErrorMSJE("Cola vacia");
            else {
                Cola.vaciar();
            }
            break;
            }
        } while(!op.equals("SALIR"));
    }

}
```

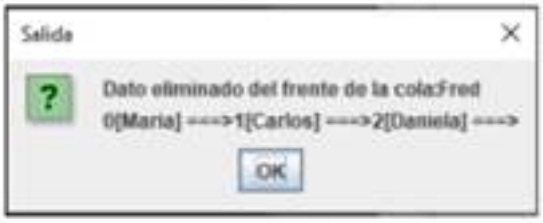
Resultados

Al ejecutarlo se nos muestra el menú con las opciones que podremos elegir.

1. "PUSH" esta opción nos pedirá que ingresemos algún dato, en este ejercicio tomamos valores String, después nos mostrará el cómo va la cola.



2.- "POP" esta opción eliminará el nombre que se encuentre en la posición 0 y será sustituido por el que se encontraba en la posición 1.



3.- "PEEK" esta opción nos mostrará quien se encuentra al frente de la cola, siendo quien esté en la posición 0.

4.- "FREE" libera la cola eliminando todos los valores.

EjercicioPrePosFijo

Este ejercicio se nos encargó antes de salir de vacaciones, pero igual lo realizamos en clase.

El ejercicio consta de que ingresemos una expresión prefija y nos regrese una expresión posfija en una tabla donde se nos muestra el procedimiento que realizó para llegar al resultado final de la expresión posfija, una tabla en donde se mostraba que si el carácter era un "(" se guardaba dentro de la columna de en medio, si era una letra entonces pasaba directo a la columna de expresión posfija, si era un operador se guardaba dentro de la tabla central, si se agregaba un operador de mayor precedencia a nivel jerárquico entonces salía el valor que antes estaba y pasa a la columna posfija y el valor de mayor precedencia se quedaba en la columna central, en caso de que el carácter sea ")" todo lo de la columna central se movía a la posfija y se eliminaban los "()"

1-. Se crea un método que lea la expresión que se ingrese y usar un charAt para que pueda pasar carácter por carácter dentro del comparador, aquí es donde entran los if y lo else, si el carácter es igual a "(" entonces entra dentro de la pila, sino entonces va directo a la columna posfija, sino si el valor es un ")" entonces usamos un while que dice que si no está vacía la pila y la cima de esa pila es "(" saca los valores que están dentro de la pila y se mueven a la columna de posfija, al igual que ocuparemos un while para detectar si es un operador, si sí es un operador entonces manda a llamar al método de jerarquía que se encarga de la precedencia de los operadores.

```
String infij = Tools.leerString("Ingrese una expresión:");
String posfij = "";
Stack<Character> stack = new Stack<Character>();
int longi = infij.length();
```

```
for (int i = 0; i < longi; i++) {
    char caracter = infij.charAt(i);
    String estado = "";
    for (Character c : stack) {
        estado += c;
    }
    if (caracter == '(') {
        stack.push(caracter);
    } else if (Character.isLetterOrDigit(caracter)) {
        posfij += caracter;
    } else if (caracter == ')') {
        while (!stack.empty() && stack.peek() != '(') {
            posfij += stack.pop();
        }
        if (!stack.empty()) {
            stack.pop();
        }
    } else {
        while (!stack.empty() && jerarquia(stack.peek()) >= jerarquia(caracter)) {
            posfij += stack.pop();
        }
        stack.push(caracter);
    }
    modelo.addRow(new Object[] {caracter, estado, posfij});
}
while (!stack.empty()) {
    posfij += stack.pop();
}
```

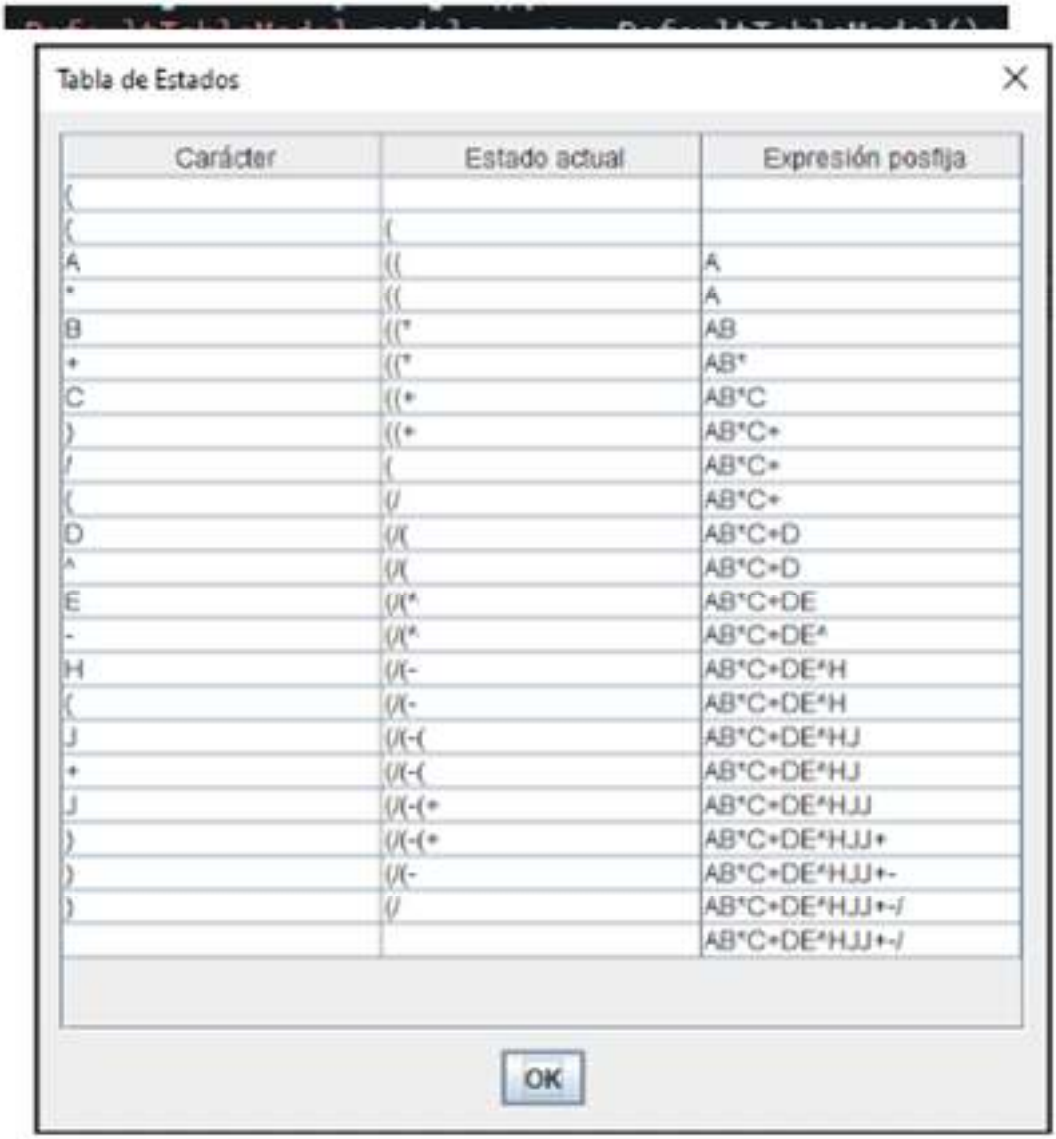
2.- El método de jerarquía como se mencionó antes será el encargado de comparar los operadores para ver cual tiene mayor precedencia que el otro.

```
public static int jerarquia(char c) {
    if (c == '+' || c == '-') {
        return 1;
    } else if (c == '*' || c == '/' || c == '%') {
        return 2;
    } else if (c == '^') {
        return 3;
    } else {
        return -1;
    }
}
```

3.- Para crear la tabla donde se mostrará el procedimiento usamos algunos imports que hay en java, como los son:

```
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
```

4.- Ya después es agregar las columnas:



Carácter	Estado actual	Expresión posfija
(		
(	(	
A	((	A
*	((	A
B	((*	AB
+	((*	AB*
C	((+	AB*C
)	((+	AB*C+
/	(	AB*C+
(	(/	AB*C+
D	(/(	AB*C+D
^	(/(	AB*C+D
E	(/(^	AB*C+DE
-	(/(^	AB*C+DE^
H	(/(-	AB*C+DE^H
(	(/(-	AB*C+DE^H
J	(/(-(	AB*C+DE^HJ
+	(/(-(	AB*C+DE^HJ
J	(/(-(+	AB*C+DE^HJJ
)	(/(-(+	AB*C+DE^HJJ+
)	(/(-	AB*C+DE^HJJ+-
)	(/	AB*C+DE^HJJ+-/
		AB*C+DE^HJJ+-/

Conclusión

En conclusión, hay varias formas de usar las pilas dentro de la programación, ya sea estáticas o dinámicas, al igual que vimos que el propio Java ya tiene sus propios métodos dentro de sus paquetes, estas actividades nos sirvieron para comprender mejor el uso de las pilas y las diferentes maneras que hay de usarlas ya sea para futuros proyectos que necesiten una memoria limitada o alguna que se ajuste a las necesidades del momento. También vimos el tema referente a nodos, aunque al inicio del tema nos costó comprender el tema pero con la práctica se hizo sencilla su comprensión, después en las últimas clases que tuvimos se nos enseñó el tema de colas en el que vimos que es similar al de pilas pero con la diferencia de la elaboración de los métodos que cambian un poquito y en la forma en la que se nos muestran los resultados.