INSTITUTO TECNOLÓGICO DE MEXICO

CAMPUS ORIZABA

NOMBRE DE LA ASIGNATURA:

ESTRUCTURA DE DATOS

TEMA:

REPORTE DE PRACTICA TEMA 3 ESTRUCTURAS LINEALES

NOMBRE DE LOS INTEGRANTES:

HERNÁNDEZ DÍAZ ARIEL - 21010195

PALACIOS MENDEZ XIMENA MONSERRAT - 21010209

CARRERA:

INGENIERIA INFORMATICA

GRUPO:

3a3A

FECHA DE ENTREGA:

22 DE ABRIL DE 2023

INTRODUCCIÓN

La pila es una secuencia de elementos del mismo tipo en la que el acceso a la misma se realiza por un único lugar denominado cima:

Vemos como el acceso a los elementos de la pila se realiza siempre sobre un único extremo. Las operaciones que caracterizan la pila son las de introducir un nuevo elemento sobre la cima (push) y la de extraer el elemento situado en la cima (pop). Una forma de ver esta estructura de datos es como una pila de libros en la que sólo se puede coger el libro que está en la cima o apilar más libros sobre la misma, pero los libros que sostienen la pila no son accesibles pues de otro modo todo se desmoronaba.

De aquí que decimos que una lista se comporta como una pila, ya que las inserciones y las extracciones sólo pueden hacerse desde la parte superior de la pila. Esto se conoce también como LIFO (Last In First Out).

Las pilas son muy útiles en varios escenarios de programación. Dos de los más comunes son:

- Pilas que contienen direcciones de retorno:
- Cuando el código llama a un método, la dirección de la primera instrucción que sigue a la llamada se inserta en la parte superior de la pila de llamadas de métodos del thread actual. Cuando el método llamado ejecuta la instrucción return, se saca la dirección de la parte superior de la pila y la ejecución continúa en esa dirección. Si un método llama a otro método, el comportamiento LIFO de la pila asegura que la instrucción return del segundo método transfiere la ejecución al primer método, y la del primer método transfiere la ejecución al código que sigue al código que llamó al primer método. Como resultado una pila "recuerda" las direcciones de retorno de los métodos llamados.
- Pilas que contienen todos los parámetros del método llamado y las variables locales:

- Cuando se llama a un método, la JVM reserva memoria cerca de la dirección de retorno y almacena todos los parámetros del método llamado y las variables locales de ese método. Si el método es un método de ejemplar, uno de los parámetros que almacena en la pila es la referencia this del objeto actual

Los métodos básicos para manipular una pila son push (empujar) y pop (sacar). El método push agrega un nuevo nodo a la parte superior de la pila. El método pop elimina un nodo de la parte superior de la pila y devuelve los datos del nodo que se quitó.

Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, cuando un programa llama a un método, el método llamado debe saber cómo regresar a su invocador, por lo que la dirección de retorno del método que hizo la llamada se mete en la pila de ejecución del programa. Si ocurre una serie de llamadas a métodos, las direcciones de retorno sucesivas se meten en la pila en el orden “último en entrar, primero en salir”, para que cada método pueda regresar a su invocador. Las pilas soportan llamadas recursivas a métodos de la misma manera que para las llamadas no recursivas convencionales.

Otra estructura de datos que se utiliza comúnmente es la cola. Una cola es similar a la fila para pagar en un supermercado: el cajero atiende primero a la persona que se encuentra hasta adelante. Los demás clientes entran a la fila sólo por su parte final y esperan a que se les atienda. Los nodos de una cola se eliminan sólo desde el principio (cabeza) de la misma y se insertan sólo al final (cola) de ésta. Por esta razón, a una cola se le conoce como estructura de datos PEPS (primero en entrar, primero en salir). Las operaciones para insertar y eliminar se conocen como enqueue (agregar a la cola) y dequeue (retirar de la cola).

MARCO TEÓRICO

Una pila es una estructura lineal en la que los elementos pueden ser añadidos o eliminados sólo por el final y una **cola** es una lista lineal en la que los elementos solo pueden ser añadidos por un extremo y eliminados por el otro.

Las pilas y las colas son dos conceptos que en programación se enseñan al mismo tiempo. Quizás por el parecido de las estructuras a nivel de implementación, pues varían solo en la forma de recuperar o eliminar sus elementos En Java, las pilas y colas ya vienen definidas como una estructura de datos, que simplemente se instala, a diferencia de otros lenguajes de programación donde se deben implementar o crear.

Primero, comencemos con algunas definiciones. En el uso común de la palabra, una **cola es FIFO** (primero en entrar, primero en salir). Al igual que la primera persona en la fila en la oficina de correos se sirve primero. Una **pila es LIFO** (último en entrar primero en salir). Piense en una pila de libros: el último libro que coloca en la pila es el primer libro que saca. Stack y Queue son dos de las estructuras de datos más importantes en el mundo de la programación y tienen una variedad de usos. A diferencia de la matriz y la lista enlazada, que se consideran una estructura de datos primaria, son una estructura de datos secundaria que se puede construir utilizando una matriz o una lista enlazada. Puede usar la pila para resolver problemas recursivos y la cola puede usarse para el procesamiento ordenado.

La diferencia entre la estructura de datos de la pila y la cola es también una de las preguntas comunes, no solo en las entrevistas de Java, sino también en C, C ++ y otras entrevistas de trabajo de programación. Bueno, la principal diferencia es la forma en que se utilizan estas estructuras de datos, StackEs la estructura de datos LIFO (último en entrar, primero en salir), lo que significa que el elemento que se inserta por última vez se recupera primero, similar a una pila de platos en una cena, donde cada invitado recoge el plato de la parte superior de la pila. Por otro lado, la estructura de datos de la cola representa literalmente una cola, que es una estructura de datos FIFO (primero en entrar, primero en salir), es decir, el objeto

que se inserta primero, primero se consume, porque la inserción y el consumo ocurren en el extremo opuesto de la cola.

Dentro de las pilas tenemos las pilas estáticas y las pilas dinámicas.

Pila Dinámica: Una pila representa una estructura lineal de datos en que se puede agregar o quitar elementos únicamente por uno de los dos extremos. Una pila representa una estructura lineal de datos en que se puede agregar o quitar elementos únicamente por uno de los dos extremos.

Pila Estática: Una PILA es una estructura de datos dinámica cuyos elementos se manipulan siguiendo una política LIFO: last-in, first-out, es decir el último dato almacenado es el primero en ser sacado y procesado. De esta forma, los elementos de una pila son almacenados y eliminados de la misma por un extremo común o 'tope'

En una Cola los elementos se añaden desde la parte de atrás o la parte final de la cola, sin embargo la información se extrae desde el frente, es decir, los elementos que se añadieron primero serán los primeros en salir, esto se conoce como estructura FIFO (First In First Out).

Los elementos de la cola se añaden y se eliminan de tal manera que el primero en entrar es el primero en salir. La adición de elementos se realiza a través de una operación llamada encolar (enqueue), mientras que la eliminación se denomina desencolar (dequeue). La operación de encolar inserta elementos por un extremo de la cola, mientras que la de desencolar los elimina por el otro.

Dentro de las colas tenemos que se encuentran las estáticas y las dinámicas

Cola Estática: Es una estructura de datos en donde los elementos se insertan sólo por un extremo de la fila y la extracción de los datos, por el otro. Es una estructura de datos en donde los elementos se insertan sólo por un extremo de la fila y la extracción de los datos, por el otro.

Cola dinámica: Agrupa elementos como si fuera una cola, por ejemplo una fila de personas. Haciendo que cada vez que se mete un elemento este se añada a la última posición. Utiliza FIFO (First Input First Output) que significa que el primero que entra es el primero que saldrá.

TDA Pila (Stack)

Pila: Colección lineal de objetos actualizada en un extremo llamado tope usando una política LIFO (last-in first-out, el primero en entrar es el último en salir).

Operaciones:

• **push(e):** Inserta el elemento e en el tope de la pila

• **pop():** Elimina el elemento del tope de la pila y lo entrega como resultado. Si se aplica a una pila vacía, produce una situación de error.

• **isEmpty():** Retorna verdadero si la pila no contiene elementos y falso en caso contrario.

• **top():** Retorna el elemento del tope de la pila. Si se aplica a una pila vacía, produce una situación de error.

• **size():** Retorna un entero natural que indica cuántos elementos hay en la pila.

Implementación de Pila

Definición de una interfaz Pila:

- Se abstrae de la ED con la que se implementará
- Se documenta el significado de cada método en lenguaje natural
- Se usa un parámetro formal de tipo representando el tipo de los elementos de la pila
- Se definen excepciones para las condiciones de error

Interfaz Stack

En UML las operaciones se dan con una sintaxis Pascallike y las excepciones se dan como comentarios:

<<interface>>

Stack<E>

+push(item : E)

+pop( ) : E

+top() : E

+isEmpty() : boolean

+size() : int

Implementaciones de pilas

1) Con un arreglo
2) Con una estructura de nodos enlazados
3) En términos de una lista (lo dejamos pendiente hasta dar el TDA Lista)

TDA Cola

Cola: Colección lineal de objetos actualizada en sus extremos llamados frente y rabo siguiendo una política FIFO (first-in firstout, el primero en entrar es el primero en salir) (También se llama FCFS = First-Come First-Served).

Operaciones:

- enqueue(e): Inserta el elemento e en el rabo de la cola
- dequeue(): Elimina el elemento del frente de la cola y lo retorna. Si la cola está vacía se produce un error.
- front(): Retorna el elemento del frente de la cola. Si la cola está vacía se produce un error.
- isEmpty(): Retorna verdadero si la cola no tiene elementos y falso en caso contrario
- size(): Retorna la cantidad de elementos de la cola.

Implementación de Cola

Definición de una interfaz Cola:

- Se abstrae de la ED con la que se implementará
- Se documenta el significado de cada método en lenguaje natural
- Se usa un parámetro formal de tipo representando el tipo de los elementos de la cola
- Se definen excepciones para las condiciones de error

Implementaciones de colas

1) Con un arreglo circular
2) Con una estructura enlazada
3) En términos de una lista (lo dejamos pendiente hasta dar el TDA Lista)

Esta estructura implica cuatro conceptos: clase auto-referenciada, nodo, campo de enlace y enlace.

Clase auto-referenciada: una clase con al menos un campo cuyo tipo de referencia es el nombre de la clase:

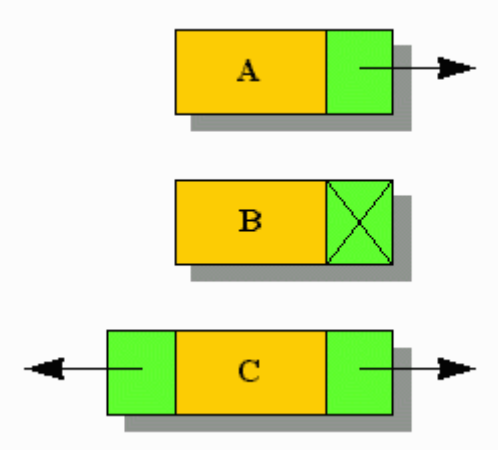
```
1. clase Empleado {
2.    privado int empno;
3.    cadena privada nombre;
4.    doble salario privado;
5.
6.    Empleado público siguiente;
7.
8.    // Otros miembros
9.
10. }
```

Employee es una clase auto-referenciada porque su campo next tiene el tipo Employee .

- **Nodo:** un objeto creado desde una clase auto-referenciada.

- **Campo de enlace:** un campo cuyo tipo de referencia es el nombre de la clase. En el fragmento de código anterior, next es un campo de enlace. Por el contrario, empno , nombre , y salario son campos no de enlace.
- **Enlace:** la referencia a un campo de enlace. En el fragmento de código anterior, la referencia siguiente a un nodo Employee es un enlace.

  Los cuatro conceptos de arriba nos llevan a la siguiente definición: *una **lista enlazada** es una secuencia de nodos que se interconectan mediante sus campos de enlace* . En ciencia de la computación se utiliza una notación especial para ilustrar las listas enlazadas. En la siguiente imagen aparece una variante de esta notación que utilizaráé a lo largo de esta sección:

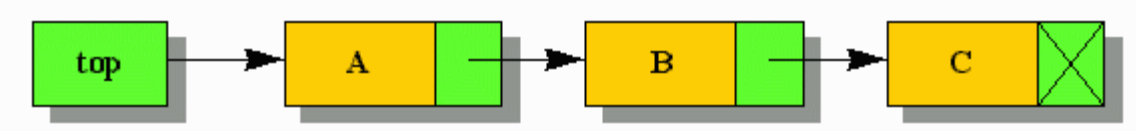


La figura anterior presenta tres nodos: A, B y C. Cada nodo se divide en áreas de contenido (en naranja) y una o más áreas de enlace (en verde). Las áreas de contenido representan todos los campos que no son enlaces, y cada área de enlace representa un campo de enlace. Las áreas de enlace de A y C tienen unas flechas para indicar que referencian a otro nodo del mismo tipo (o subtipo). El único área de enlace de B incorpora una X para indicar una referencia nula. En otras palabras, B no está conectado a ningún otro nodo.

Aunque se pueden crear muchos tipos de listas enlazadas, las tres variantes más populares son la lista de enlace simple, la lista doblemente enlazada y la lista enlazada circular. Exploremos esas variantes, comenzando con la lista enlazada.

Lista de enlaces simples

Una **lista de enlace simple** es una lista enlazada de nodos, donde cada nodo tiene un único campo de enlace. Una variable de referencia contiene una referencia al primer nodo, cada nodo (excepto el último) enlaza con el nodo siguiente, y el enlace del último nodo contiene nulo para indicar el final de la lista. Aunque normalmente a la variable de referencia se la suele llamar arriba , usted puede elegir el nombre que quiera. La siguiente figura presenta una lista de enlace simple de tres nodos, donde top referencia al nodo A, A conecta con B y B conecta con C y C es el nodo final:



Un algoritmo común de las listas de enlace simple es la inserción de nodos. Este algoritmo está implicado de alguna forma porue tiene mucho que ver con cuatro casos: cuando el nodo se debe insertar antes del primer nodo; cuando el nodo se debe insertar después del último nodo; cuando el nodo se debe insertar entre dos nodos; y cuando la lista de enlace simple no existe. Antes de estudiar cada caso consideramos el siguiente pseudocódigo:

Nodo DECLARE CLASS
  DECLARAR STRING nombre
  Nodo DECLARE siguiente
DECLARACIÓN FIN

DECLARE Nodo superior = NULL

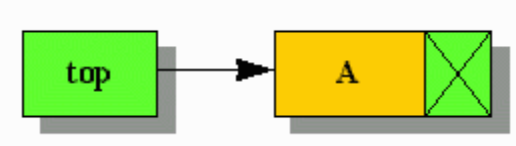
Este pseudocódigo declara una clase auto-referenciada llamada Node con un campo no de enlace llamado name y un campo de enlace llamado next . También declara una variable de referencia top (del tipo Node) que contiene una referencia al primer Node de una lista de enlace simple. Como la lista todavía no existe, el valor inicial de top es NULL . Cada uno de los siguientes cuatro casos asume las declaraciones de Node y top :

- **La lista de enlace simple no existe::**
  Este es el caso más simple. Se crea un Node , se asigna su referencia a top ,

se inicializa su campo no de enlace, y se asigna NULL a su campo de enlace. El siguiente pseudocódigo realiza estas tareas:

- superior = NUEVO nodo
-
- arriba.nombre = "A"
- arriba.siguiente = NULL

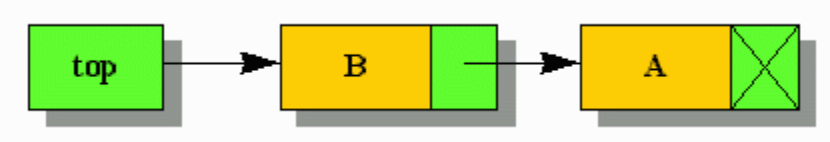
En la siguiente imagen se puede ver la lista de enlace simple que emerge del pseudocódigo anterior:



- **El nodo debe insertarse antes del primer nodo:** . Se crea un Node , se inicialia su campo no de enlace, se asigna la referencia de top al campo de enlace next , y se asigna la referencia del Node recien creado a top . El siguiente pseudocódigo (que asume que se ha ejecutado el pseudocódigo anterior) realiza estas tareas:

- DECLARAR temperatura del nodo
-
- temp = NUEVO nodo
- temp.nombre = "B"
- temp.siguiente = arriba
- arriba = temperatura

El resultado del listado anterior aparece en la siguiente imagen:



- **El nodo debe insertarse detrás del último nodo:** Se crea un Node , se inicializa su campo no de enlace, se asigna NULL al campo de enlace, se atraviesa la lista de enlace simple hasta el último Node , y se asigna la referencia del Node recien creado al campo next del último nodo. El siguiente pseudocódigo realiza estas tareas:

1. temp = NUEVO nodo
1. temp.nombre = "C"
2. temp.siguiente = NULL

3. DECLARAR Nodo temp2

4. temp2 = superior

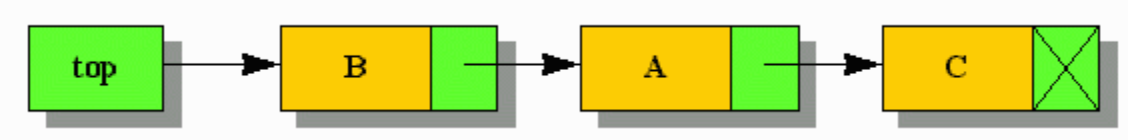
5. // Asumimos que top (y temp2) no son NULL
6. // por el pseudocódigo anterior

7. MIENTRAS temp2.next NO ES NULO
8. temp2 = temp2.siguiente
9. TERMINAR MIENTRAS

10. // temp2 ahora hace referencia al último nodo

11. temp2.siguiente = temp

    La siguiente imagen revela la lista después de la inserción del nodo C después del nodo A .



- **El nodo se debe insertar entre dos nodos:** Este es el caso más complejo. Se crea un Node , se inicializa su campo no de enlace, se atraviesa la lista hasta encontrar el Node que aparece antes del nuevo Node , se asigna el campo de enlace del Node anterior al campo de enlace del Node recien creado, y se asigna la referencia del Node recien creado al campo del enlace del Node anterior. El siguiente pseudocódigo realiza estas tareas:

- temp = NUEVO nodo
- temp.nombre = "D"
-
- temp2 = superior
-
- // Suponemos que el Nodo recién creado se inserta después del Nodo
- // A y que el Nodo A existe. En el mundo real, no hay
- // garantiza que existe cualquier nodo, por lo que tendríamos que verificar
- // para temp2 que contiene NULL en el encabezado del ciclo WHILE
- // y después de que se complete el bucle WHILE.
-
- MIENTRAS temp2.name NO ES "A"
- temp2 = temp2.siguiente
- TERMINAR MIENTRAS
-
- // temp2 ahora hace referencia al Nodo A.
-
- temp.siguiente = temp2.siguiente
- temp2.siguiente = temp

La siguiente imagen muestra la inserción del nodo D entre los nodos A y C .

El siguiente listado presenta el equivalente Java de los ejemplos de pseudocódigo de inserción anteriores:

```
// SLLInsDemo.java

clase SLLInsDemo {
  Nodo de clase estática {
    Nombre de cadena;
    Nodo siguiente;
  }
```

```
public static void principal (String [] args) {
  Nodo superior = nulo;

  // 1. La lista de enlaces simples no existe

  arriba = nuevo Nodo ();
  arriba.nombre = "A";
  superior.siguiente = nulo;

  dump ("Caso 1", arriba);

  // 2. La lista enlazada individualmente existe y se debe insertar el nodo
  // antes del primer nodo

  temperatura del nodo;

  temp = nuevo nodo ();
  temp.nombre = "B";
  temp.siguiente = arriba;
  arriba = temperatura;

  dump ("Caso 2", arriba);

  // 3. La lista enlazada individualmente existe y se debe insertar el nodo
  // después del último nodo

  temp = nuevo nodo ();
  temp.nombre = "C";
  temp.siguiente = nulo;
```

```
    nodo temp2;

    temp2 = arriba;

    while (temp2.next != null)
      temp2 = temp2.siguiente;

    temp2.siguiente = temp;

    dump ("Caso 3", arriba);

    // 4. La lista enlazada individualmente existe y el nodo debe insertarse
    // entre dos nodos

    temp = nuevo nodo ();
    temp.nombre = "D";

    temp2 = arriba;

    while (temp2.name.equals ("A") == falso)
      temp2 = temp2.siguiente;

    temp.siguiente = temp2.siguiente;
    temp2.siguiente = temp;

    dump ("Caso 4", arriba);
  }

  volcado de vacío estático (String msg, Node topNode) {
    System.out.print (mensaje + " ");
```

```
      while (topNode != null) {
        System.out.print (topNode.name + " ");
        nodosuperior = nodosuperior.siguiente;
      }
      Sistema.out.println ();
    }
}
```

El método static void dump(String msg, Node topNode) itera sobre la lista e imprime su contenido. Cuando se ejecuta SLLInsDemo , las repetidas llamadas a este método dan como resultado la siguiente salida, lo que coincide con las imágenes anteriores:

Caso 1A

Caso 2BA

Caso 3 CAB

Caso 4 BADC

No un:

SLLInsDemo y los ejemplos de pseudocódigo anteriores emplean un algoritmo de búsqueda lineal orientado a listas enlazadas para encontrar un Nodo específico. Indudablemente usted ajusta este otro algoritmo en sus propios programas:

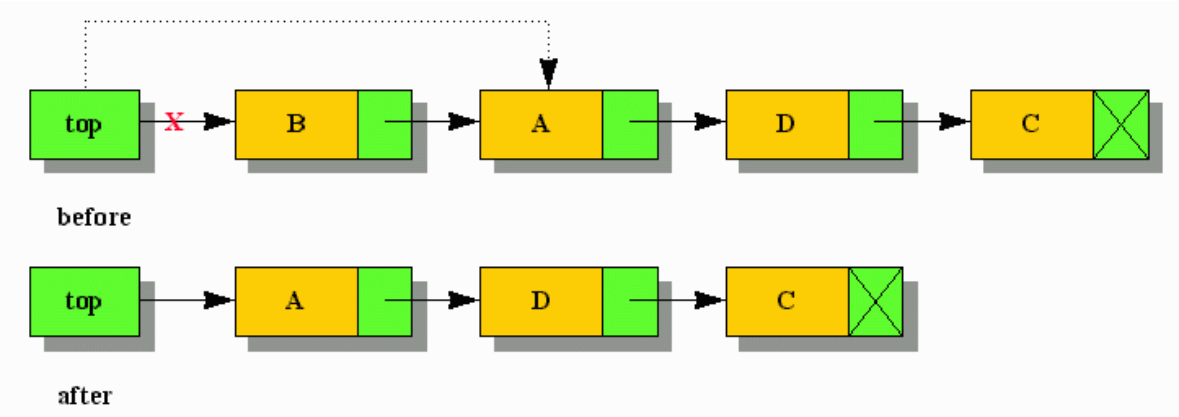
- **Búsqueda del último nodo :**
- // Supongamos que top hace referencia a una lista enlazada individualmente de al menos un Nodo.
-
- Node temp = top // Usamos temp y no top. Si se usara top,
- // no se pudo acceder a la lista enlazada individualmente después
- // la búsqueda terminó porque top se referiría
- // al nodo final.
- MIENTRAS temp.next NO ES NULO
- temp = temp.siguiente
- TERMINAR MIENTRAS

-
- // temp ahora hace referencia al último nodo.
- **Búsqueda de un Nodo específico:**
- // Supongamos que top hace referencia a una lista enlazada individualmente de al menos un Nodo.
-
- Temperatura del nodo = superior
-
- MIENTRAS que temp NO ES NULO Y temp.name NO ES "A" // Busque "A".
- temp = temp.siguiente
- TERMINAR MIENTRAS
-
- // temp hace referencia al Nodo A o contiene NULL si no se encuentra el Nodo A.

  Otro algoritmo común de las listas de enlace simples es el borrado de nodos. Al contrario que la inserción de nudos, sólo hay dos casos a considerar:
- **Borrar el Primer nodo:**

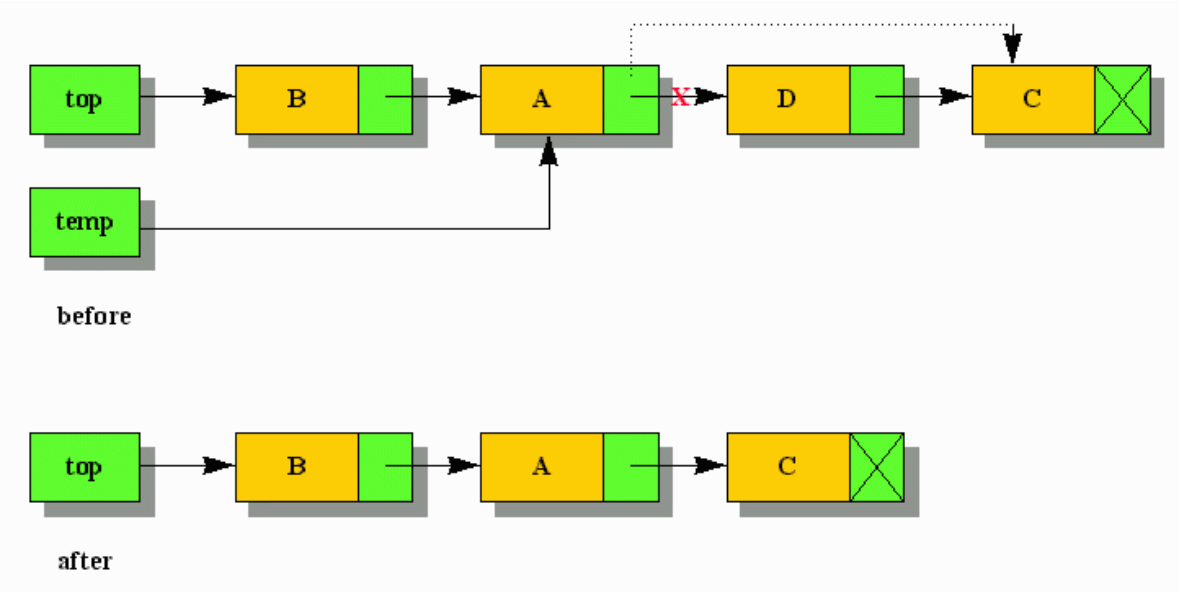
  Asigna el enlace del campo next del nodo referenciado por top a top :
- arriba = arriba.siguiente; // Hace referencia al segundo Nodo (o NULL si solo hay un Nodo)

  La siguiente imagen presenta las vistas anterior y posterior de una lista donde se ha borrado el primer nodo. en esta figura, el nodo B desaparece y el nodo A se convierte en el primer nodo.



- **Borrar cualquier nodo que no sea el primero:** Localiza el nodo que precede al nodo a borrar y le asigna el enlace que hay en el campo next del nodo a borrar al campo next del nodo que le precede. El siguiente pseudocódigo borra el nodo D :
- temperatura = superior
- MIENTRAS que temp.name NO ES "A"
- temp = temp.siguiente
- TERMINAR MIENTRAS
-
- // Asumimos que la temperatura hace referencia al Nodo A
-
- temp.siguiente = temp.siguiente.siguiente
-
- // El nodo D ya no existe

  La siguiente figura presenta las vistas anterior y posterior de una lista donde se ha borrado un nodo intermedio. En esa figura el nodo D desaparece.



El siguiente listado representa el equivalente Java a los pseudocódigos de borrado anteriores:

```
// SLLDelDemo.java

clase SLLDelDemo {
  Nodo de clase estática {
    Nombre de cadena;
    Nodo siguiente;
  }

  public static void principal (String [] args) {
    // Construir la lista de enlaces simples de la Figura 6 (es decir, BADC)

    Nodo superior = nuevo Nodo ();
    arriba.nombre = "C";
    superior.siguiente = nulo;

    Nodo temporal = nuevo Nodo ();
    temp.nombre = "D";
    temp.siguiente = arriba;
```

```
arriba = temperatura;

temp = nuevo nodo ();
temp.nombre = "A";
temp.siguiente = arriba;
arriba = temperatura;

temp = nuevo nodo ();
temp.nombre = "B";
temp.siguiente = arriba;
arriba = temperatura;

dump ("Lista inicial de enlaces simples", arriba);

// 1. Eliminar el primer nodo

arriba = arriba.siguiente;

dump ("Después de la eliminación del primer nodo", arriba);

// Devolver B

temp = nuevo nodo ();
temp.nombre = "B";
temp.siguiente = arriba;
arriba = temperatura;

// 2. Eliminar cualquier nodo menos el primero

temperatura = superior;
```

```
      while (temp.name.equals ("A") == falso)
        temp = temp.siguiente;

      temp.siguiente = temp.siguiente.siguiente;

      dump ("Después de la eliminación del nodo D", arriba);
    }

    volcado de vacío estático (String msg, Node topNode) {
      System.out.print (mensaje + " ");

      while (topNode != null) {
        System.out.print (topNode.name + " ");
        nodosuperior = nodosuperior.siguiente;
      }
      Sistema.out.println ();
    }
}
```

Cuando ejecute SLLDelDemo , observará la siguiente salida:

```
Lista inicial de enlaces simples BADC
Después de la eliminación del primer nodo ADC
Después de la eliminación del nodo D BAC
```

Cuidado:

Como java inicializa los campos de referencias de un objeto a null durante la construcción del objeto, no es necesario asignar cleanmente null a un campo de enlace. No olvide estas asignaciones de null en su código fuente; su ausencia reduce la claridad del código.

Después de estudiar SLLDelDemo , podría preguntarse qué sucede si asigna null al nodo referenciado por top : ¿el recolector de basura recogerá toda la lista? Para responder a esta cuestión, compile y ejecute el código del siguiente listado:

```
// GCDemo.java

clase GCDemo {
  Nodo de clase estática {
    Nombre de cadena;
    Nodo siguiente;

    Vacío protegido Finalizar () lanza Throwable {
      System.out.println ("Finalizando " + nombre);
      super.finalizar ();
    }
  }

  public static void principal (String [] args) {
    // Construir la lista de enlaces simples de la Figura 6 (es decir, BADC)

    Nodo superior = nuevo Nodo ();
    arriba.nombre = "C";
    superior.siguiente = nulo;

    Nodo temporal = nuevo Nodo ();
    temp.nombre = "D";
    temp.siguiente = arriba;
    arriba = temperatura;

    temp = nuevo nodo ();
    temp.nombre = "A";
    temp.siguiente = arriba;
    arriba = temperatura;

    temp = nuevo nodo ();
```

```
    temp.nombre = "B";
    temp.siguiente = arriba;
    arriba = temperatura;

    dump ("Lista inicial de enlaces simples", arriba);

    superior = nulo;
    temperatura = nulo;

    para (int i = 0; i < 100; i++)
        Sistema.gc ();
  }

  volcado de vacío estático (String msg, Node topNode) {
    System.out.print (mensaje + " ");

    while (topNode != null){
      System.out.print (topNode.name + " ");
      nodosuperior = nodosuperior.siguiente;
    }
    Sistema.out.println ();
  }
}
```

GCDemo crea la misma lista de cuatro nodos que SLLDelDemo . Después de volcar los nodos a la salida estándar, GCDemo asigna null a top ya temp . Luego, GCDemo ejecuta Sys tem.gc(); hasta 100 veces. ¿Qué sucede después? Mire la salida (que observó en mi plataforma Windows):

```
Lista inicial de enlaces simples BADC
Finalizando C
Finalizando D
```

Finalizando A

Finalizando B

La salida revela que todos los nodos de la lista de enlace simple han sido finalizados (y recolectados). Como resultado, no tiene que preocuparse de poner a null todos los enlaces de una lista de enlace simple cuando se quiera deshacer de ella. (Podría necesitar tener que incrementar el número de ejecuciones de System.gc (); si su salida no incluye los mensajes de finalización.)

COLAS

Definición:

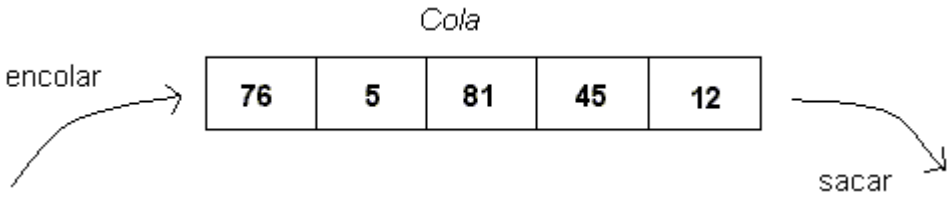
Una cola es un tipo especial de lista abierta en la que sólo se pueden insertar nodos en uno de los extremos de la lista y sólo se pueden eliminar nodos en el otro. Además, como sucede con las pilas, las escrituras de datos siempre son inserciones de nodos, y las lecturas siempre eliminan el nodo leído.

Este tipo de lista es conocido como lista FIFO (First In First Out), el primero en entrar es el primero en salir. Existen tres tipos de colas:

- cola lineal (sencilla)
- cola circular
- cola doble (bicola)

Cola sencilla

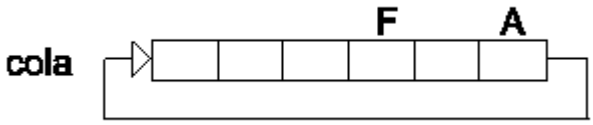
La cola lineal es un tipo de almacenamiento creado por el usuario que trabaja bajo la técnica FIFO (primero en entrar primero en salir). Las colas lineales se representan gráficamente de la siguiente manera:



Las colas lineales tienen un grave problema, como las extracciones sólo pueden realizarse por un extremo, puede llegar un momento en que el apuntador sea igual al máximo número de elementos en la cola, siendo que al frente de la misma existan lugares vacíos, y al insertar un nuevo elemento nos mandará un error de overflow (cola llena).

Cola circular

En este tipo de cola existe un apuntador desde el último elemento al primero de la cola. La representación gráfica de esta estructura es la siguiente:



La condición de vacío en este tipo de cola es que el apuntador F sea igual a cero.
Las condiciones que debemos tener presentes al trabajar con este tipo de estructura son las siguientes:

• Over flow, cuando se realice una inserción.
• Under flow, cuando se requiera de una extracción en la cola.
• Vació

Cola doble o bicola

Esta estructura es una cola bidimensional en que las inserciones y eliminaciones se pueden realizar en cualquiera de los dos extremos de la bicola. Gráficamente representamos una bicola de la siguiente manera:

Existen dos variantes de la doble cola:

• Doble cola de entrada restringida.

• Doble cola de salida restringida.

Uso de Clases Genéricas en Java

El tema de las clases genéricas es bastante amplio, y algunas de ellas están lo suficientemente avanzadas. Sin embargo, un conocimiento básico de los genéricos es necesario para todos los programadores de Java. Aunque la sintaxis de los genéricos puede parecer un poco intimidante las clases genéricas son sorprendentemente fáciles de usar.

Las clases genéricas nos evitan duplicar clases que administran tipos de datos distintos, pero implementan algoritmos o una lógica similar. En pocas palabras evitan la redundancia de código.

Por lo que, una ventaja principal del código genérico es que trabajará automáticamente con el tipo de dato pasado como parámetro de tipo.

Para mostrar cómo funciona una clase genérica vamos a construir una clase llamada Bolsa que nos permitirá almacenar objetos de varios tipos.

El principio de una clase genérica es que esta clase en este caso una bolsa, puede ser un objeto donde almacenar Sodas o Chocolates, pero no de las dos a la vez*.

```
import java.util.ArrayList;

public class Bolsa < T > {
 private ArrayList < T > lista = new ArrayList < T >();

 public void add(T objeto){
```

```
    lista.add(objeto);
  }

  public ArrayList<T> getProducts(){
    return lista;
  }
}
```

El parámetro de tipo T será el que la bolsa manipulará. Este parámetro se utiliza en la declaración de la clase Bolsa para representar el tipo del elemento.

Entonces para utilizar esta clase genérica, al momento de instanciar se deberá de mandar como argumento el tipo de dato que vamos a querer almacenar en nuestra bolsa.

```
public static void main(String[] args) {

Bolsa<Soda> bolsaDeSodas = new Bolsa<Soda>();
 Bolsa<Chocolate> bolsaDeChocolates = new Bolsa<Chocolate>();

 bolsaDeSodas.add(new Soda(“Soda_1”, “Limon”));
 bolsaDeSodas.add(new Soda(“Soda_2”, “Fresa”));

 bolsaDeChocolates.add(new Chocolate(“Chocolate_1”, “Negro”));
 bolsaDeChocolates.add(new Chocolate(“Chocolate_2”, “Blanco”));

}
```

Lo que se hizo fue que dinámicamente la Bolsa cambio el tipo de dato que se está manipulando por lo que internamente la Bolsa<Soda> se vería así:

```
import java.util.ArrayList;
```

```
public class Bolsa {
 private ArrayList < Soda > lista = new ArrayList < Soda >();

 public void add(Soda objeto){
    lista.add(objeto);
 }

 public ArrayList<Soda> getProducts(){
    return lista;
 }

}
```

Por lo cual una clase genérica se va a adaptar al tipo de dato que sea mandado como argumento y evitamos tener una clase por cada tipo de dato. Por ejemplo, de tener las clases **BolsaChocolate** y **BolsaSoda**, mejor, tenemos una genérica Bolsa. De las clases genéricas más utilizadas de java se encuentran las colecciones como **ArrayList**, **HashSet**, **HashMap**, **LinkedList**.

*Si no indicamos un tipo de dato para la clase genérica, tomará **Object** como parámetro y entonces nuestra Bolsa podrá almacenar cualquier tipo de objeto.

LISTAS ENLAZADAS

La estructura de datos tipo lista dinámica consiste en un conjunto de datos del mismo tipo de forma que cada elemento de la lista (nodo) consta de dos campos. Un campo contiene la información que almacena la lista y el otro indica cual es el siguiente nodo de la lista. A la información normalmente contenida en cada nodo de la lista se le añade otra, para enlazar unos nodos con otros. Esta información adicional apunta a otro u otros nodos de la estructura. En general hay dos tipos de listas: las estáticas, en las que esta información adicional son indices a los elementos de un vector (o matriz), cuyos elementos son nodos de la lista; y las

dinámicas, en las que los nodos son referenciados directamente mediante su dirección de memoria, con una variable de tipo puntero, de las soportadas por los lenguajes de programación. El manejo de estructuras de tipo lista dinámica es mucho más sencillo que el de las estáticas, pues el sistema operativo del ordenador se encarga de gestionar la memoria ocupada por la estructura en cada momento, en vez de tener que hacerlo el usuario.

El hecho de que pueda haber más de un apuntador en cada nodo permite que los nodos de la estructura se encadenen según criterios distintos. Se dice que una lista es ordenada si sus nodos están encadenados según un criterio de ordenación (ascendente, descendente, numérico, etcétera.) de su campo de información.

Definición.

La estructura de datos tipo lista dinámica se diferencia de la estática, en que se utilizan punteros para enlazar los nodos que forman la lista. En la figura 1 se muestra gráficamente la estructura de datos lista dinámica.

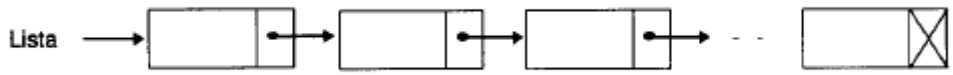


Figura 1: estructura de datos tipo lista dinámica.

Las operaciones definidas sobre la estructura de datos de tipo lista ordenada y enlazada mediante punteros (lista dinámica) son las siguientes:

- InicializarLista: crear una lista dinamica vacia.
- ListaVacia: averiguar si una lista dinamica esta vacia.
- InsertarLista: insertar un nodo en una lista dinamica.
- SuprimirLista: suprimir un nodo de una lista dinamica.

InicializarLista (Lista)

Función: Inicializar lista a estado vacío.

Entrada: Lista a inicializar.

Salida: Lista inicializada.

Precondiciones: Ninguna.

Postcondiciones: Lista vacia.

ListaVacia (Lista)

Funcion:Averiguar si la lista esta vacia.

Entrada: Lista a comprobar.

Salida: Valor booleano que indica si la pila esta vacia.

Precondiciones: Lista inicializada.

Postcondiciones: Indica si la lista esta vacia.

InsertarLista (Lista, NuevoElemento)

Función: Insertar NuevoElemento a la lista.

Entrada: Lista, elemento a insertar (Lista, NuevoElemento).

Salida: Lista con NuevoElemento insertado (Lista).

Precondiciones: Lista inicializada.

Postcondiciones: Lista convenientemente actualizada.

SuprimirLista (Lista, QuitarElemento)

Función: Suprimir QuitarElemento de la lista.

Entrada: Lista sobre la que suprimir un nodo (Lista).

Salida: Lista y elemento suprimido (Lista, QuitarElemento).

Precondiciones: Lista no vacia.

Postcondiciones: Lista convenientemente actualizada.

Es posible definir otras estructuras de tipo lista dinamica, con características diferentes de la lista que vamos a desarrollar:

- Lista dinamica circular, es decir, aquella en la que el último elemento de la lista apunta al primero (Figura 2).
- Lista dinamica doblemente enlazada, es decir, aquella con dos campos de tipo TipoPuntero: el siguiente que apunta al nodo siguiente al actual y el anterior que apunta al nodo anterior al actual (Figura 3).

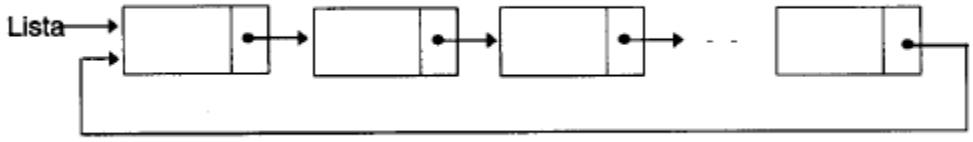


Figura 2: lista dinamica circular.

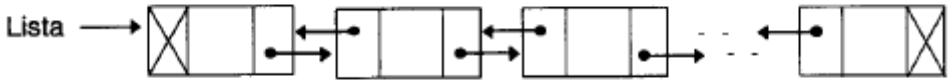


Figura 3: lista dinamica doblemente enlazada.

Lista de enlace simple

La lista de enlace simple es una estructura dinámica donde el número de nodos puede variar rápidamente dependiendo de los requerimientos del proceso: los nodos aumentan por inserciones a la lista o disminuyen por eliminación.

La lista de enlace simple se caracteriza por tener únicamente un enlace al siguiente nodo. Esta lista cuenta con un nodo cabeza y un nodo al final de la lista.

Se accede a la lista mediante el primer nodo de esta llamado “cabeza” o “cabecera” y el último nodo llamado “cola”, cada enlace del nodo apuntará al siguiente, el último

nodo apuntará a nulo. Se debe contar con un apuntador que se encarga de referenciar al primer nodo de la lista y otro apuntador al nodo final de la lista.

Lista de enlace doble

Es una estructura dinámica, donde el número de nodos puede variar dependiendo de las necesidades del proceso: agregando nodos por inserciones o disminuyendo nodos por eliminación.

La lista de enlace doble está caracterizada por tener únicamente dos enlaces: uno al siguiente nodo y otro al anterior nodo. Cuenta con un nodo cabeza y un nodo al final de la lista. Se accede a la lista mediante el primer nodo de la lista llamado “cabeza” o “cabecera” y el último nodo llamado “cola”, cada enlace del nodo apuntará al siguiente y al anterior nodo.

Se debe contar con un apuntador que se encargar de referenciar al primer nodo de la lista y otro apuntador al nodo final de la lista.

Lista circular de enlace simple

La lista circular de enlace simple es una estructura dinámica donde el número de nodos puede variar rápidamente dependiendo de los requerimientos del proceso: aumentando los nodos por inserciones a la lista o disminuyendo nodos por eliminación.

La lista circular de enlace simple se caracteriza por tener únicamente un enlace al siguiente nodo, pero que el enlace del último nodo apunta al primer nodo de la lista. Cuenta con un nodo cabeza y un nodo al final de la lista.

Se accede a la lista mediante el primer nodo llamado “cabeza” o “cabecera” y el último nodo llamado “cola”. Cada enlace del nodo apuntará al siguiente y el último nodo apuntará al primer nodo.

Se debe contar con un apuntador que se encargar de referenciar al primer nodo de la lista y otro apuntador al nodo final de la lista. Esta lista cuenta con unos métodos y operaciones propios.

Lista circular de enlace doble

Es una estructura dinámica donde el número de nodos puede variar rápidamente dependiendo de los requerimientos del proceso: aumentando los nodos por inserciones a la lista o disminuyendo nodos por eliminación.

La lista circular de enlace doble se caracteriza por tener dos enlaces al siguiente nodo o predecesor y otro al anterior nodo de la lista o antecesor, pero que el enlace del último nodo apunta al primer nodo de la lista y el primer nodo (cabeza), apunta al último nodo de la lista (cola). Cuenta con un nodo cabeza y un nodo al final de la lista.

Se accede a la lista mediante el primer nodo de la lista llamado “cabeza” o “cabecera” y el último nodo llamado “cola”, cada enlace del nodo apuntará al siguiente, el último nodo apuntará al primer nodo y este a su vez, apuntará al último nodo.

Se debe contar con un apuntador que se encarga de referenciar al primer nodo de la lista y otro apuntador al nodo final de la lista. Al igual que las anteriores listas, la circular de enlace doble tiene los mismos métodos comunes de inserción de un nodo, y las mismas operaciones.

COMPETENCIA(S) ESPECÍFICA(S):

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

RECURSOS, MATERIALES Y EQUIPO

- Computadora

- Java
- Lectura de los materiales de apoyo del tema 3 (PADLET)
- Notas de clase (problemas resueltos en clase y materiales de trabajo de la profesora)

CLASE PILA TDA

```
public interface PilaTDA <T>{
 public boolean isEmpty();
public boolean isSpace();
 public T pop();
 public void push(T dato);
 public T peek();
public void freePila();
```

En la clase anterior se crearon unos métodos los cuales vamos a ocupar para agregar, eliminar, mostrar, elementos en la pila. Para esto aquí se explicará para que sirve cada uno de estos métodos:

- **public boolean isEmpty();//Regresa true si la pila no tiene elementos**
- **public boolean isSpace();**
- **public T pop();//debe quitar el elemento que está en el tope y regresarlo**
- **public void push(T dato);//inserta el dato en el tope de la pila**
- **public T peek();//regresa el elemento que está en el tope, sin quitarlo**
- **public void freePila();//limpia pila**

Estos los vamos a ocupar más adelante para las siguientes clases.

CLASE PILA A

En este método lo que se va a hacer es que vamos a crear una PilaA estática en la cual por medio de la interfaz **PILATDA** vamos a meter, eliminar, mostrar, ver si la pila está llena o no y liberar elementos en la pila. No sin antes vamos a declarar un

array llamado pila y otra que lleve el tope que indica la posición del último elemento insertado, seguido del **CONSTRUCTOR DE LA CLASE EL CUAL ES EL ENCARGADO DE CREAR MI PILA:**

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

El tope-1 es necesario porque como sabemos cuándo recorremos una, la primer posición es la 0 y la última posición es el tamaño -1, para darnos cuenta cuando está vacía o cuando ingresemos

Para esto tenemos el siguiente **MÉTODO PUSH,** el cual nos sirve para meter elementos a la pila:

```
public void push(T dato) {
    // TODO Auto-generated method stub
    if(isSpace()) {
        tope++;
        pila[tope]=dato;
    }
    else Tools.imprime("PILA LLENA...");
}
```

En ese método vamos a preguntar si la pila está llena, si lo hay vamos a incrementar el tope para que nos inserte en la posición 0, después vamos a insertar un objeto a pila y seguido del dato que ingresamos. Sino vamos a mandar un mensaje que la pila está llena y ya no se puede ingresar más elementos.

Después el **MÉTODO POP** nos sirve para eliminar un elemento de la pila:

```
public T pop() {
    // TODO Auto-generated method stub
    T dato= pila[tope];
    tope--;
    return dato;
}
```

En T dato va a almacenar el número que va a sacar para retornarlo, entonces en pila[tope] le decimos sácame lo que había en tope, obviamente a tope lo decrementamos para que lo saque. Y retornara el dato que se va a eliminar. Por otro lado, **en el Método isEmpty** checamos si la pila está vacía o no:

```
public boolean isEmpty() {
    // TODO Auto-generated method stub
    return (tope==-1);
}
```

En pocas palabras vamos a retornar cuando el tope sea igual a -1 que quiere decir que está vacía

MÉTODO ISSPACE

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

Cuando la medida del tope es mayor a la medida es el tamaño del array. Entonces va a retornar un true de que aún no está llena o false de que está llena.

METODO FREEPILA

```
public void freePila() {
    // TODO Auto-generated method stub

}
```

Este método pregunta que si la pila está vacía nos mandara un mensaje de que está vacía, sino lo estuviera vamos a seguir metiendo elementos a la pila.

METODO PEEK

```
public T peek() {
    // TODO Auto-generated method stub

    return pila[tope];
}
```

En este método va a retornar lo que hay en pila, en este caso el tope(último dato que se encuentra en la cima) de la pila.

METODOS TOSTRING

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

Por consiguiente en este método nos va a retornar lo que hay en toString en este caso tope, posteriormente pregunta que si “i” es mayor o igual a 0, entonces nos va a concatenar la posición, los elementos en cada posición y finalmente mostrarlos en pantalla. Sino lo fuera no va a mostrar nada.

CLASE MENU PILA A

En este menú vamos a crear un objeto de Pila llamado pila en el cual vamos a decir que hay que ingresar 10 elementos en la pila, para que este funcione hay que crear un switch en cual tendrá los 5 métodos de “push,pop,peek,free y salir” y fuera del switch tendrá un do-while que va a terminar hasta que seleccionemos la opción “salir”. Y este es el código de dicho método:

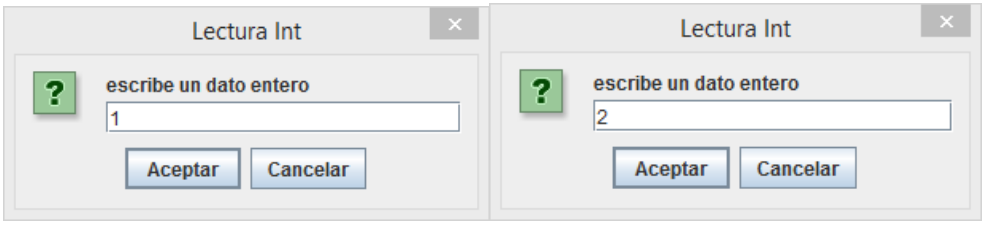
```
public class Menu {
    public static void menu3(String menu){
        String sel="";
        PilaA <Integer> pila = new PilaA((byte)10);
```

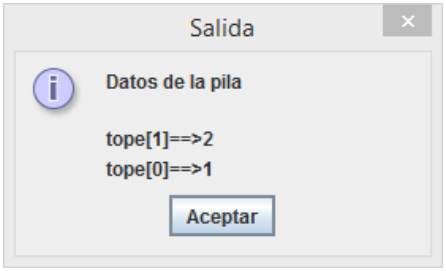
Este es la parte anterior explicada.

En el caso de PUSH ingresamos elementos en la pila por medio del método push, para después llamar al toString y mostrar los elementos antes ingresados.

```
case "push":
    pila.push(Tools.leerInt("escribe un dato entero"));
    Tools.imprime("Datos de la pila \n"+pila.toSring());
    break;
```

EJECUTABLES DEL METODO PUSH

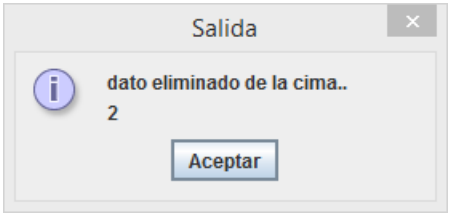




En el caso POP aquí nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino muestra un mensaje que se eliminó el dato que está en la cima, por medio del método pop.

```
case "pop":
    if(pila.isEmpty())Tools.imprimeErrorMsje("pila vacia...!!");
    else Tools.imprime("dato eliminado de la cima..\n"+pila.pop());
    break;
```

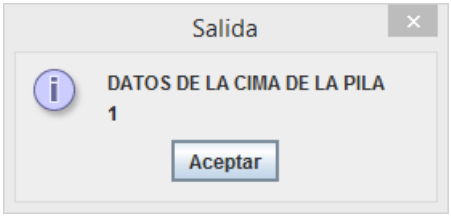
EJECUTABLE DEL METODO POP



En el caso PEEK aquí nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino está vacía nos mostrara el elemento de la cima y llamamos al método peek.

```
case "peek":
    if(pila.isEmpty())Tools.imprimeErrorMsje("pila vacia!!");
    else {
        Tools.imprime("DATOS DE LA CIMA DE LA PILA \n"+pila.peek());
    }
    break;
```

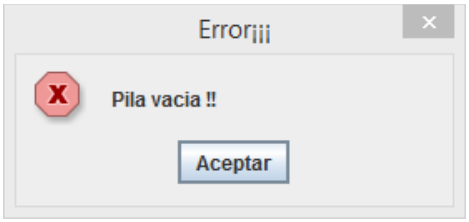
EJECUTABLE DEL METODO PEEK



En el caso de FREE nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino nos pide que ingresemos elementos en la pila para que se llene hasta el punto de que quisimos.

```
case "Free":
    if (pila.isEmpty())Tools.imprimeErrorMsje("Pila vacia !!");
    else {
        pila= new PilaA((byte)10);
    }
    break;
```

EJECUTABLE DEL METODO FREE

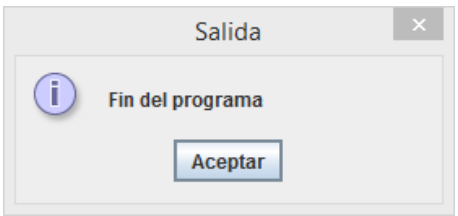


Si está llena la pila va a limpiar la pila y cuando queramos volver a darle free nos dirá que la pila está vacía.

El último caso SALIR solo se encarga de salir del menú y ya el programa termina, no sin antes mostrar un mensaje de Fin del programa.

```
case"Salir": Tools.imprime("Fin del programa");
    break;
```

EJECUTABLE DEL METODO SALIR



MAIN PARA LLAMAR AL METODO MENU

```
public static void main(String []args) {
    String menu2="Push,Pop,Peek,Free,Salir";
    menu3(menu2);
}
```

CLASE PILA B

Dentro de esta pila encontraremos diferentes métodos usando la interfaz de PILATDA <T>, la cual se implementa dentro de la clase

```
public class PilaB<T>implements PilaTDA<T> {
```

Para crear una variable privada de tipo Stack llamado pila . Un objeto de la clase Stack es una pila. Permite almacenar objetos y luego recuperarlos en el orden inverso en el cual se insertaron, es decir, siempre se recupera el último elemento insertado. Para insertar una la pila se invoca el método push.

```
private  Stack<T> pila;
```

Crear un constructor llamado igual que la clase PILAB. Aquí la pila va a apuntar al Stack.

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

SIZE

El primer método Size nos ayuda a obtener la cantidad de elementos de una pila

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

ISEMPTY

El método de isEmpty el cual nos valida si la pila está vacía o no ,el cual nos va a retornar pila más el mismo método de manera diferente

```
public boolean isEmpty() {
    // TODO Auto-generated method stub
    return (pila.isEmpty());
}
```

POP

Este método tiene saca un elemento de la cima de la pila , donde se crea una variable de tipo T llamada dato, y dato es igual al elemento que está en la cima de la pila y saca el elemento para retornar el contenido de la lista sin el elemento que se eliminó

```
public T pop() {
    // TODO Auto-generated method stub
    T dato;
    dato = (T)pila.peek();
    pila.pop();
    return dato;
}
```

PUSH

El método de push nos sirve para meter elementos dentro de la pila, en el cual solamente se manda a llamar en dato ,en pila con el método de push

```java
public void push(T dato) {
    // TODO Auto-generated method stub

    pila.push(dato);

}
```

PEEK

Este método muestra el elemento que se encuentra hasta la cima de la pila ,y retorna lo que hay en pila hasta tope extrayendo el número de la cima

```java
public T peek() {
    // TODO Auto-generated method stub
    return (T)(pila.peek());
}
```

VACIAR

El siguiente método es vaciar el cual como su nombre nos dice lo que hará es limpiar la pila al momento de salir

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

TOSTRING

Para este método nos sirve para mostrar los elementos dentro de la pila en pila size -1 , con ayuda de la variable i haciendo uso de la recursividad retornara i mayor o igual a 0 y lo que hay dentro de pila en i , devolviendo el mismo método toString en i-1

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

CLASE MENU PILA B

En este menú vamos a crear un objeto de PilaB llamado pila en el cual vamos a decir que hay que ingresar elementos en la pila, para que este funcione hay que crear un switch en cual tendrá los 5 métodos de “push,pop,peek,free y salir” y fuera del switch tendrá un do-while que va a terminar hasta que seleccionemos la opción “salir”. Y este es el código de dicho método:

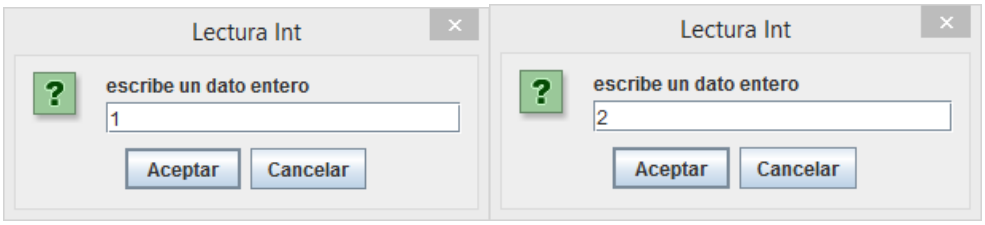
```
public class MenuPilaB {
    public static void menu3(String menu){
        String sel="";
        PilaB <Integer> pila = new PilaB<Integer>();
```

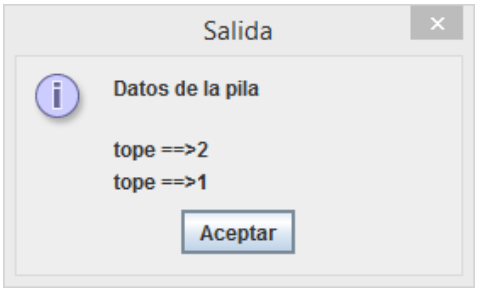
Este es la parte anterior explicada.

En el caso de PUSH ingresamos elementos en la pila por medio del método push, para después llamar al toString y mostrar los elementos antes ingresados.

```
case "push":
    pila.push(Tools.leerInt("escribe un dato entero"));
    Tools.imprime("Datos de la pila \n"+pila.toSring());
    break;
```

EJECUTABLES DEL METODO PUSH

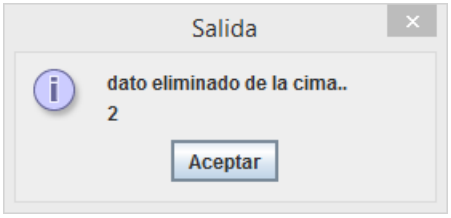




En el caso POP aquí nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino muestra un mensaje que se eliminó el dato que está en la cima, por medio del método pop.

```
case "pop":
    if(pila.isEmpty())Tools.imprimeErrorMsje("pila vacia...!!");
    else Tools.imprime("dato eliminado de la cima..\n"+pila.pop());
    break;
```

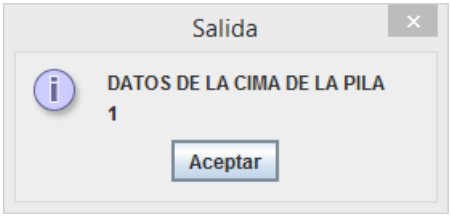
EJECUTABLE DEL METODO POP



En el caso PEEK aquí nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino está vacía nos mostrara el elemento de la cima y llamamos al método peek.

```
case "peek":
    if(pila.isEmpty())Tools.imprimeErrorMsje("pila vacia!!");
    else {
        Tools.imprime("DATOS DE LA CIMA DE LA PILA \n"+pila.peek());
    }
    break;
```

EJECUTABLE DEL METODO PEEK

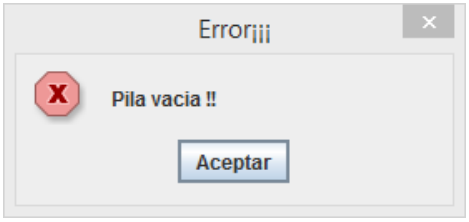


En el caso de FREE nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino llamamos a vaciar para que vacié la pila.

```
case "Free":
    if (pila.isEmpty())Tools.imprimeErrorMsje("Pila vacia !!");
    else {
    pila.vaciar();

    }
    break;
```

EJECUTABLE DEL METODO FREE

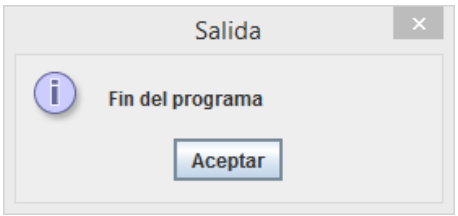


Si está llena la pila va a limpiar la pila y cuando queramos volver a darle free nos dirá que la pila está vacía.

El último caso SALIR solo se encarga de salir del menú y ya el programa termina, no sin antes mostrar un mensaje de Fin del programa.

```
case"Salir": Tools.imprime("Fin del programa");
    break;
```

EJECUTABLE DEL METODO SALIR



MAIN PARA LLAMAR AL METODO MENU

```
public static void main(String []args) {
    String menu2="Push,Pop,Peek,Free,Salir";
    menu3(menu2);
}
```

CLASE PILA C

Dentro de esta pila encontraremos diferentes métodos usando la interfaz de PILATDA <T>, la cual se implementa dentro de la clase

```
public class PilaC<T> implements PilaTDA<T>{
```

Para crear luego un arreglo de lista llamado pila y una variable de tipo entero llamado tope.

```
private ArrayList pila;
int tope;
```

Para luego crear PilaC como método

inicializando pila con un nuevo ArrayList el cual almacena elementos en la memoria ,y también inicializamos tope con -1 ,lo cual esto nos indicará que la pila está vacía

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

}
```

SIZE

El primer método Size nos ayuda a obtener la de elementos de una pila

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

ISEMPTY

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

El método de isEmpty el cual nos valida si la pila está vacía o no ,el cual nos va a retornar pila más el método .

VACIAR

El siguiente método es vaciar el cual como su nombre nos dice lo que hará es limpiar la pila al momento de salir

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

PUSH

El método de push nos sirve para meter elementos dentro de la pila en el cual pide un dato de tipo de T, donde se añade el dato a la pila y tope incrementa

```
public void push(T dato) {
    pila.add(dato);
    tope++;
}
```

POP

Este método tiene saca un elemento de la cima de la pila , donde se crea una variable de tipo T para mostrar el elemento que hay dentro de pila hasta tope pues ahí se encuentra lo que está hasta arriba de la cima y con pila remove elimina este elemento de la pila y tope decrementa retornando dato

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

PEEK

Este método muestra el elemento que se encuentra hasta la cima de la pila ,y retorna lo que hay en pila hasta tope extrayendo el número de la cima

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

TOSTRING

El método toString() es un método integrado del objeto Number de JavaScript que le permite convertir cualquier valor de tipo numérico en su representación de tipo cadena. Para este método nos sirve para mostrar los elementos dentro de la pila con ayuda de la variable i haciendo uso de la recursividad retornara i mayor igual a 0 y lo que hay dentro de pila en i ,devolviendo el mismo método toString en i-1

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

IS SPACE

Por último este método que es la que nos ayuda a validar si la pila está vacía o no el cual retorna comparando si tope es menor a lo que hay dentro de pila con el método de size -1

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

FREE PILA

En este método lo que hará es liberar la pila, hasta que ya no haya ningún elemento y salga un cuadro que diga pila vacía.

```
public void freePila() {
    // TODO Auto-generated method stub

}
```

CLASE MENU PILA C

En este menú vamos a crear un objeto de PilaC llamado pila en el cual vamos a decir que hay que ingresar elementos en la pila, para que este funcione hay que crear un switch en cual tendrá los 5 métodos de “push,pop,peek,free y salir” y fuera del switch tendrá un do-while que va a terminar hasta que seleccionemos la opción “salir”. Y este es el código de dicho método:

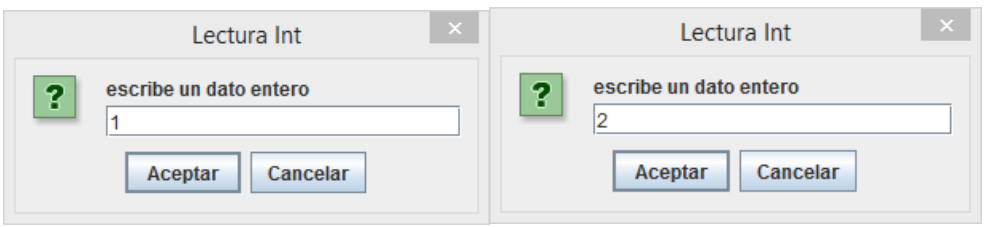
```
public class MenuPilaC {
    public static void menu3(String menu){
        String sel="";
        PilaC <Integer> pila = new PilaC<Integer>();
```

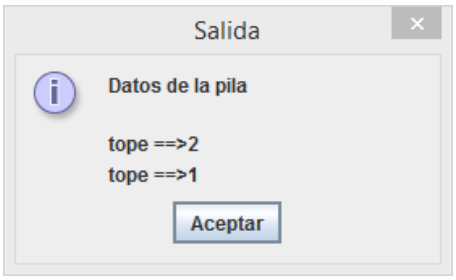
Este es la parte anterior explicada.

En el caso de PUSH ingresamos elementos en la pila por medio del método push, para después llamar al toString y mostrar los elementos antes ingresados.

```
case "push":
    pila.push(Tools.leerInt("escribe un dato entero"));
    Tools.imprime("Datos de la pila \n"+pila.toSring());
    break;
```

EJECUTABLES DEL METODO PUSH

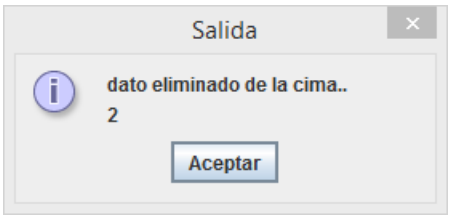




En el caso POP aquí nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino muestra un mensaje que se eliminó el dato que está en la cima, por medio del método pop.

```
case "pop":
    if(pila.isEmpty())Tools.imprimeErrorMsje("pila vacia...!!");
    else Tools.imprime("dato eliminado de la cima..\n"+pila.pop());
    break;
```

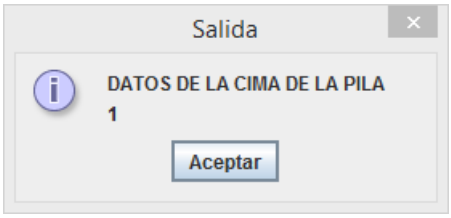
EJECUTABLE DEL METODO POP



En el caso PEEK aquí nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino está vacía nos mostrara el elemento de la cima y llamamos al método peek.

```
case "peek":
    if(pila.isEmpty())Tools.imprimeErrorMsje("pila vacia!!");
    else {
        Tools.imprime("DATOS DE LA CIMA DE LA PILA \n"+pila.peek());
    }
    break;
```

EJECUTABLE DEL METODO PEEK

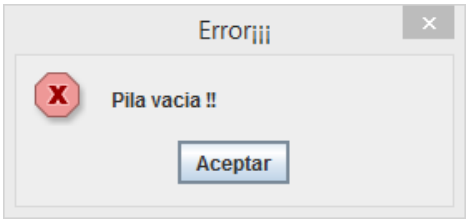


En el caso de FREE nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino llamamos a vaciar para que vacié la pila.

```
case "Free":
    if (pila.isEmpty())Tools.imprimeErrorMsje("Pila vacia !!");
    else {
    pila.vaciar();

    }
    break;
```

EJECUTABLE DEL METODO FREE

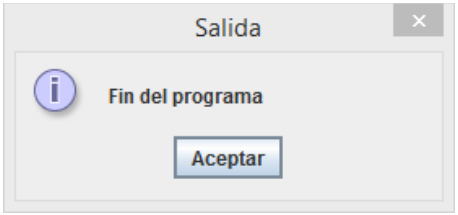


Si está llena la pila va a limpiar la pila y cuando queramos volver a darle free nos dirá que la pila está vacía.

El último caso SALIR solo se encarga de salir del menú y ya el programa termina, no sin antes mostrar un mensaje de Fin del programa.

```
case"Salir": Tools.imprime("Fin del programa");
    break;
```

EJECUTABLE DEL METODO SALIR



MAIN PARA LLAMAR AL METODO MENU

```
public static void main(String []args) {
    String menu2="Push,Pop,Peek,Free,Salir";
    menu3(menu2);
}
```

CLASE NODO

Tendremos un tipo de dato en este caso “T” que va seguido de la variable dato, después un nodo llamado siguiente.

```
public class Nodo<T> {
public T info;
public Nodo sig;
```

Posteriormente creamos un constructor de la misma clase, en el cual siguiente llevara la dirección del elemento, e info el elemento.

```
public Nodo(T info) {
    super();
    this.info = info;
    this.sig =null;
}
```

Finalmente tenemos sus correspondientes get y sets los cuales sabemos que el método set es un método público, el cual se encarga de darle un valor a una propiedad o atributo de un objeto, y el get se encarga de mostrar un valor a una propiedad o atributo de un objeto.

```
public T getInfo() {
    return info;
}


public void setInfo(T info) {
    this.info = info;
}


public Nodo getSig() {
    return sig;
}
```

CLASE PILA D

En esta clase vamos a hacer un **implements** con la clase de PILATDA para que con los métodos que hay ahí los podamos usar en esta clase PILAD. Después hay que encapsular un puntero llamado **nodo** el cual se va a llamar pila, después vamos a crear un constructor llamado igual que la clase PILAD. Aquí la pila va a apuntar a nulo por defecto

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

METODO ISEMPTY

```
  public boolean isEmpty() {
      // TODO Auto-generated method stub
      return (pila==null);
  }
```

Este Método nos va a regresar un true o un false si la pila está vacía, con que quiero decir esto pues si la pila es igual a nulo pues no hay nada entonces nos va a retornar un true sino un false si es caso contrario.

METODO PUSH

```
public void push(T dato) {
    // TODO Auto-generated method stub
    Nodo tope = new Nodo(dato);
    if (isEmpty()) pila=tope;
    else
            { tope.sig=pila;
              pila=tope;
            }
}
```

Este método va a recibir un dato de tipo T. después dentro de esto vamos a crear un nuevo puntero de Nodo llamado tope y después del new tomaremos como parámetro el dato que reciba, ya con eso pregunta que si está vacía , si está vacía dirá que lo que hay en tope se lo asigne a pila. En caso contrario, vamos a decir que tope de su siguiente lo vamos a apuntar a la pila, y después de eso vamos a decirle que la pila sea igual a tope para que la pila (cima) sea el que acaba de ser insertado.

METODO POP

```
public T pop() {
    // TODO Auto-generated method stub
    T dato = (T) pila.getInfo();
    pila=pila.getSig();
    return dato;
}
```

Este método no toma nada de parámetro, pero si retornara un dato que es el la cima que será eliminado, creamos una variable llamada dato y le vamos a decir que lo que tenemos en la pila de dato que nos traiga por medio de getInfo el valor del atributo info. Además hay que pila (cima) ya no será igual al que estaba, sino que ahora le pondremos de pila pero del siguiente (llamamos a getSig para que muestre el valor que estaba después de la cima ósea debajo de la cima). Y finalmente retornamos el tipo de dato llamado dato.

METODO PEEK

```
public T peek() {
    // TODO Auto-generated method stub
    return (T)(pila.getInfo());
}
```

En este método solo retornamos lo que se encuentra en pila, ósea el elemento de la cima.

METODO VACIAR

```
public void vaciar() {
    pila=null;
}
```

Este se encarga de apuntar pila a nulo cuando este vacía la pila. Este lo vamos a explicar más adelante.

METODO FREE

```
public void freePila() {
    // TODO Auto-generated method stub

}
```

Este método en pocas palabras nos limpia la pila, no hay nada ahí pero para eso sirve. Mas adelante hay una parte donde tiene que ver este método.

METODOS TOSTRING

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

En el primer método le asignamos lo que hay en pila a tope de tipo Nodo, después retornamos lo que hay en tope. Ya en segundo método en el parámetro declaramos la variable “i” de tipo nodo, finalmente retornamos (con el operador ternario) y preguntamos que si “i” es diferente o igual a nulo, si es verdadero entonces va a concatenar lo que hay en getInfo (el elemento ingresado) y lo que hay en getSig (la dirección del anterior). Sino fuera verdadero entonces concatena pero sin nada.

CLASE MENU PILA D

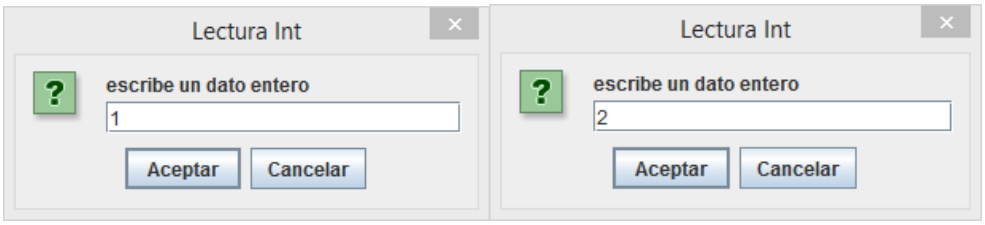
En este menú vamos a crear un objeto de PilaD llamado pila en el cual vamos a ingresar los elementos que sean en la pila, para que este funcione hay que crear un switch en cual tendrá los 5 métodos de “push,pop,peek,free y salir” y fuera del switch tendrá un do-while que va a terminar hasta que seleccionemos la opción “salir”, aquí el código del menú PILAD:

```
public class MenuPilaD {
    public static void menu3(String menu){
        String sel="";
        PilaD<Integer> pila = new PilaD<Integer>();
```

Caso PUSH ingresamos elementos en la pila por medio del método push, para después llamar al toString y mostrar los elementos antes ingresados.

```
case "push":
    pila.push(Tools.leerInt("escribe un dato entero"));
    Tools.imprime("Datos de la pila \n"+pila.toSring());
    break;
```

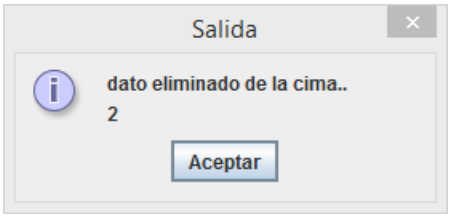
EJECUTABLES DEL METODO PUSH



Caso POP aquí nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino muestra un mensaje que se eliminó el dato que está en la cima, por medio del método pop.

```
case "pop":
    if(pila.isEmpty())Tools.imprimeErrorMsje("pila vacia...!!");
    else Tools.imprime("dato eliminado de la cima..\n"+pila.pop());
    break;
```

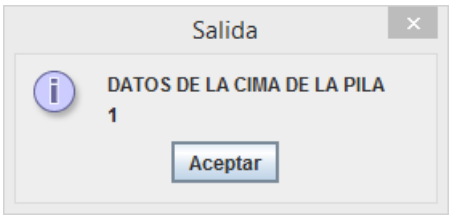
EJECUTABLE DEL METODO POP



Caso PEEK aquí nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino está vacía nos mostrara el elemento de la cima y llamamos al método peek.

```
case "peek":
    if(pila.isEmpty())Tools.imprimeErrorMsje("pila vacia!!");
    else {
        Tools.imprime("DATOS DE LA CIMA DE LA PILA \n"+pila.peek());
    }
    break;
```

EJECUTABLE DEL METODO PEEK

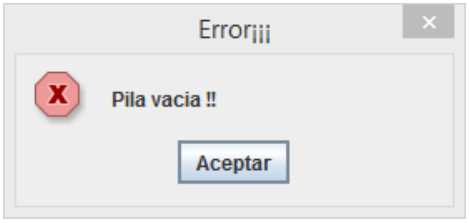


En el caso de FREE nos pregunta que si la pila está vacía, si lo está nos manda un mensaje que está vacía, sino entonces llamamos a vaciar por medio del objeto llamado pila para que la vacié.

```
case "Free":
    if (pila.isEmpty())Tools.imprimeErrorMsje("Pila vacia !!");
    else {
    pila.vaciar();

    }
    break;
```

EJECUTABLE DEL METODO FREE

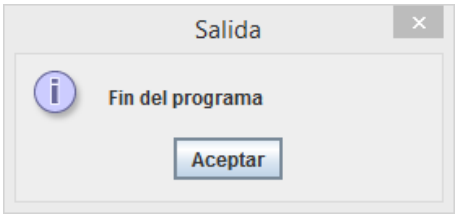


Si está llena la pila va a limpiar la pila y cuando queramos volver a darle free nos dirá que la pila está vacía.

Caso SALIR solo se encarga de salir del menú y ya el programa termina, no sin antes mostrar un mensaje de Fin del programa.

```
case"Salir": Tools.imprime("Fin del programa");
    break;
```

EJECUTABLE DEL METODO SALIR



MAIN PARA LLAMAR AL METODO MENU

```
public static void main(String []args) {
    String menu2="Push,Pop,Peek,Free,Salir";
    menu3(menu2);
}
```

CLASE COLA TDA

```
public interface ColaTDA<T>{
    public boolean isEmptyCola();
    public void pushCola(T dato);
    public T popCola();
    public T peekCola();
    public void freeCola();
}
```

En la clase anterior se crearon unos métodos los cuales vamos a ocupar para agregar, eliminar, mostrar, elementos en la pila. Para esto aquí se explicará para que sirve cada uno de estos métodos:

- **public boolean isEmptyCola();//Regresa true si la cola no tiene elementos**
- **public boolean isSpace();**
- **public T popCola();//debe quitar el primer elemento y regresarlo**
- **public void pushCola (T dato);//inserta el dato en el tope de la cola**
- **public T peekCola ();//regresa el elemento que está en el tope (el primer elemento ingresado), sin quitarlo**

- **public void freeCola();//limpia cola**

Estos los vamos a ocupar más adelante para las siguientes clases.

CLASE COLA A

```
public class ColaA<T> implements ColaTDA<T>{
private T cola[];
private byte u;
public ColaA(int max) {
cola = (T[]) (new Object[max]);
u=-1;
}
```

En este método lo que se va a hacer es que vamos a crear una Cola estática en la cual por medio de la interfaz **COLATDA** vamos a meter, eliminar, mostrar, ver si la cola está llena o no y liberar elementos de esta. No sin antes vamos a declarar un array llamado cola y otra que lleve "u" ósea el ultimo que indica la posición del primer elemento insertado, seguido del **CONSTRUCTOR DE LA CLASE EL CUAL ES EL ENCARGADO DE CREAR LA COLA:**

El u-1 es necesario porque como sabemos cuándo recorremos un, la primer posición es la 0 y la última posición es el tamaño -1, para darnos cuenta cuando está vacía o cuando ingresemos

Para esto tenemos el siguiente **MÉTODO PUSH COLA,** el cual nos sirve para meter elementos a la COLA:

```
public void pushCola(T Dato) {
if(isSpace()) {
u++;
cola[u]=Dato;
}else {
Tools.imprimeErrorMsje("Cola llena...");
}
}
```

En ese método vamos a preguntar si la cola está llena, si lo hay vamos a incrementar "u" para que nos inserte en la posición 0, después vamos a insertar un objeto a cola

y seguido del dato que ingresamos. Sino vamos a mandar un mensaje que la cola está llena y ya no se puede ingresar más elementos.

Después el **MÉTODO POP COLA** nos sirve para eliminar un elemento de la COLA:

```
@Override
public T popCola() {
T Dato = cola[0];
for(int k=0; k<=u; k++) {
cola[k]=cola[k+1];
}
u--;
return Dato;
}
```

En T dato va a almacenar el número que va a sacar para retornarlo, entonces en cola[0] le decimos sácame lo que había en u, obviamente a u lo decrementamos para que lo saque. En pocas palabras saca el primer elemento que se ingresó a la cola, es como decir “el primero que entra es el primero que sale”. Finalmente retornamos el dato para eliminarlo. Por otro lado, **en el Método isEmptyCola** checamos si la cola está vacía o no:

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

En pocas palabras vamos a retornar cuando el “U” (último elemento) sea igual a -1 que quiere decir que está vacía

MÉTODO ISSPACECOLA

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

Cuando la medida del último (u) es mayor a la medida es el tamaño del array-1. Entonces va a retornar un true de que aún no está llena o false de que está llena.

METODO FREECOLA

```
@Override
public void freeCola() {
u=-1;
}
```

Este método nos va a liberar la cola, en pocas palabras va a borrar los elementos que hay en ella.

METODO PEEKCOLA

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

En este método va a retornar lo que hay en cola, en este caso el 0(primer dato que se encuentra en el principio) de la cola.

METODOS TOSTRING

```
public String toString() {
return toString(0);
}
public String toString(int i) {
return (i<=u)?" "+i+"=>["+""+cola[i]+"] "+toString(i+1):"";
}
```

Por consiguiente en este método nos va a retornar lo que hay en toString en este caso el 0, posteriormente pregunta que si “i” es menor o igual a “u”, entonces nos va a concatenar la posición, los elementos en cada posición y finalmente mostrarlos en pantalla. Sino lo fuera no va a mostrar nada.

CLASE MENU COLA A

En este menú vamos a crear un objeto de **ColaA** llamado cola en el cual vamos a decir que hay que ingresar 10 elementos en la cola, para que este funcione hay que crear un switch en cual tendrá los 5 métodos de “push,pop,peek,free y salir” y fuera del switch tendrá un do-while que va a terminar hasta que seleccionemos la opción “salir”. Y este es el código de dicho método:

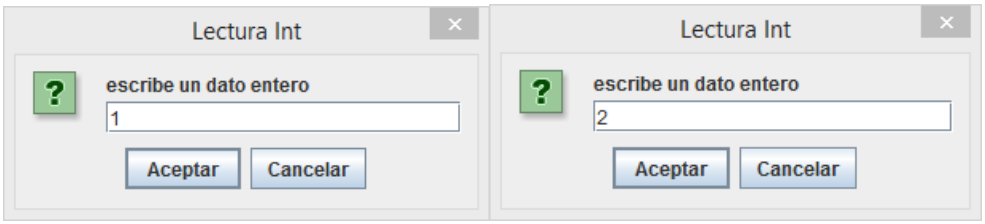
```
public class MenuColaA {
    public static void menu3(String menu){
        String sel="";
        ColaA <Integer> cola = new ColaA<Integer>((byte)10);
```

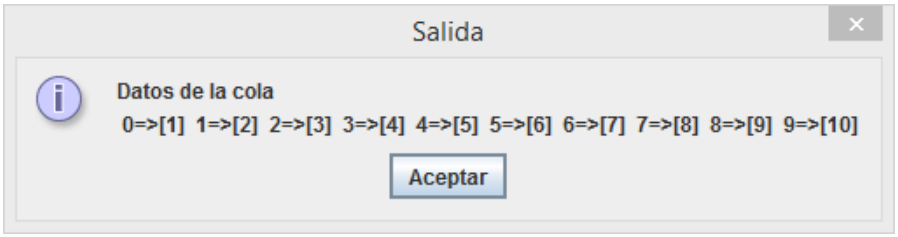
Este es la parte anterior explicada.

En el caso de PUSH ingresamos elementos en la cola por medio del método push, para después llamar al toString y mostrar los elementos antes ingresados.

```
case "Push":
    cola.pushCola(Tools.leerInt("Escribe un dato entero"));
    Tools.imprime("Datos de la cola \n"+cola.toString());
    break;
```

EJECUTABLES DEL METODO PUSH

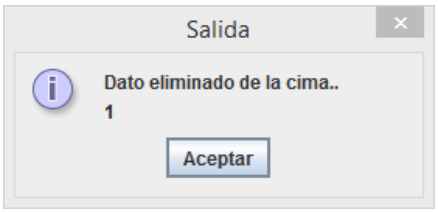




En el caso POP aquí nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino muestra un mensaje que se eliminó el dato que está en la cima (el primer elemento que ingreso), por medio del método pop.

```
case "Pop":
    if(cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia...!!");
    else Tools.imprime("Dato eliminado de la cima..\n"+cola.popCola());
    break;
```

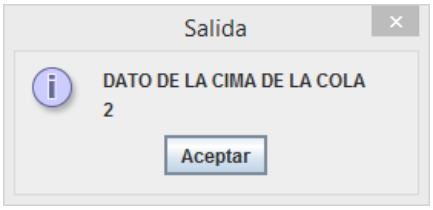
EJECUTABLE DEL METODO POP



En el caso PEEK aquí nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino está vacía nos mostrara el elemento de la cima (el primer elemento) y llamamos al método peek.

```
case "Peek":
    if(cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia!!");
    else {
        Tools.imprime("DATO DE LA CIMA DE LA COLA \n"+cola.peekCola());
    }
    break;
```

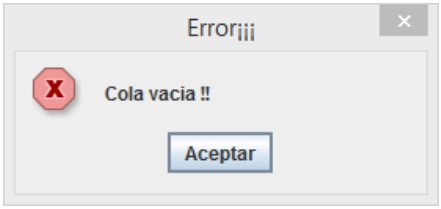
EJECUTABLE DEL METODO PEEK



En el caso de FREE nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino nos pide que ingresemos elementos en la cola para que se llene hasta el punto de que pedimos.

```
case "Free":
    if (cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia !!");
    else {
        cola= new ColaA<Integer>((byte)10);
    }
    break;
```

EJECUTABLE DEL METODO FREE



Si está llena la cola va a limpiar la cola y cuando queramos volver a darle free nos dirá que la cola está vacía.

El último caso SALIR solo se encarga de salir del menú y ya el programa termina, no sin antes mostrar un mensaje de Fin del programa.

```
case"Salir": Tools.imprime("Fin del programa");
    break;
```

EJECUTABLE DEL METODO SALIR

CLASE COLA B

Otro ejemplo de colas dinámicas es **ColaB** empezando por crear una clase llamada ColaB donde implementamos también los métodos de ColaTDA ,,y creamos una variable privada global Queue llamada cola , y un constructor llamado como la clase ”ColaB”, ahí vamos a crear un objeto LinkedList el cual nos permite almacenar valores de objetos de un mismo tipo, y empezar con los métodos los cuales no son muy distintos a los que hemos llevado hasta ahora

```
public class ColaB<T>implements ColaTDA<T> {
    private Queue cola;

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

SIZE nos ayuda a obtener el número de elementos que hay dentro de una cola y disminuye en 1 en el proceso

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

ISEMPTY nos valida si la cola está vacía o no ,el cual nos va a retornar cola llamando al método .

```
public boolean isEmptyCola() {
	// TODO Auto-generated method stub
	return (cola.isEmpty());
}
```

EL MÉTODO PUSH nos sirve para meter elementos dentro de la cola en el cual por medio de la acción “add” va a añadirnos datos.

```
public void pushCola(T dato) {
	// TODO Auto-generated method stub

	cola.add(dato);
}
```

POP COLA nos ayuda a eliminar el elemento que se encuentra al inicio de cola dentro de una variable de tipo T, llamado dato , utilizando el implemento de **element** en el cual los elementos de la cola se añaden y se eliminan de tal manera que el primero en entrar es el primero en salir y cola **remove** elimina este elemento

```
public T popCola() {
	// TODO Auto-generated method stub
	T dato;
	dato = (T)cola.element();
	cola.remove();
	return dato;
}
```

PEEKCOLA nos ayudará a ver cuál es el elemento que se encuentra primero en la cola con ayuda de cola **element**

```
public T peekCola() {
	// TODO Auto-generated method stub
	return (T)  (cola.element());
}
```

FREE COLA lo único que nos hace es limpiar los elementos de la cola eliminándolos de esta

```
public void freeCola() {
    // TODO Auto-generated method stub
}
```

En este caso el **TOSTRING** es muy diferente puesto que ocupamos el iterador Un iterador es un objeto que nos permite recorrer una lista y presentar por pantalla todos sus elementos. De tal manera que mostramos fácilmente los elementos de la cola, y para esto usamos un variable llamada **cad** de tipo **String** para concatenar los elementos de la cola, y posteriormente retornar la cadena para mostrarla.

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

CLASE MENU COLA B

En este menú vamos a crear un objeto de **ColaB** llamado cola en el cual vamos a decir que hay que ingresar elementos en la cola, para que este funcione hay que crear un switch en cual tendrá los 5 métodos de “push,pop,peek,free y salir” y fuera del switch tendrá un do-while que va a terminar hasta que seleccionemos la opción “salir”. Y este es el código de dicho método:

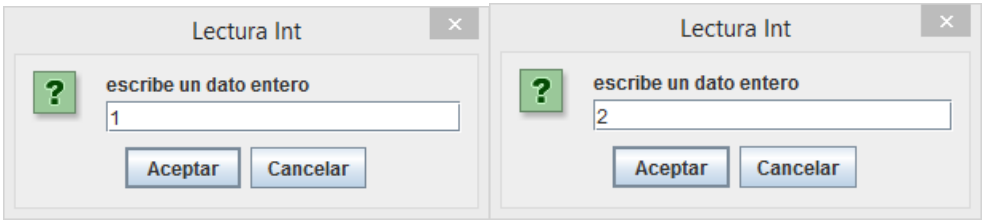
```
public class MenuColaB {
    public static void menu3(String menu){
        String sel="";
        ColaB<Integer> cola = new ColaB<Integer>();
```

Este es la parte anterior explicada.

En el caso de PUSH ingresamos elementos en la cola por medio del método push, para después llamar al toString y mostrar los elementos antes ingresados.

```
case "Push":
    cola.pushCola(Tools.leerInt("Escribe un dato entero"));
    Tools.imprime("Datos de la cola \n"+cola.toString());
    break;
```

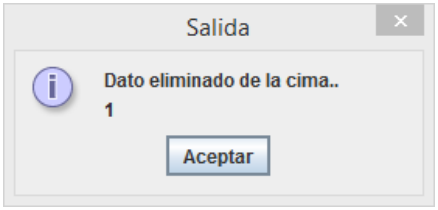
EJECUTABLES DEL METODO PUSH



En el caso POP aquí nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino muestra un mensaje que se eliminó el dato que está en la cima (el primer elemento que ingreso), por medio del método pop.

```
case "Pop":
    if(cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia...!!");
    else Tools.imprime("Dato eliminado de la cima..\n"+cola.popCola());
    break;
```

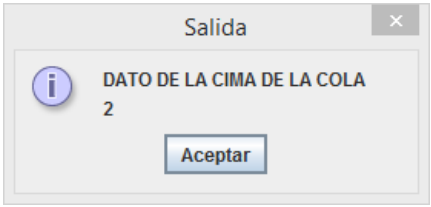
EJECUTABLE DEL METODO POP



En el caso PEEK aquí nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino está vacía nos mostrara el elemento de la cima (el primer elemento) y llamamos al método peek.

```
case "Peek":
    if(cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia!!");
    else {
        Tools.imprime("DATO DE LA CIMA DE LA COLA \n"+cola.peekCola());
    }
    break;
```

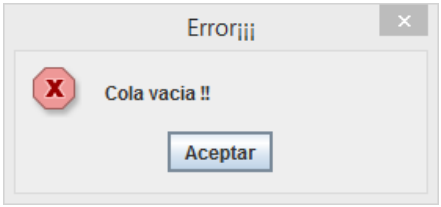
EJECUTABLE DEL METODO PEEK



En el caso de FREE nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino llamamos a “vaciar” para que vacié la cola.

```
case "Free":
    if (cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia !!");
    else {
        cola.vaciar();
    }
    break;
```

EJECUTABLE DEL METODO FREE

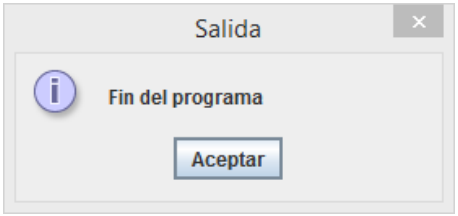


Si está llena la cola va a limpiar la cola y cuando queramos volver a darle free nos dirá que la cola está vacía.

El último caso SALIR solo se encarga de salir del menú y ya el programa termina, no sin antes mostrar un mensaje de Fin del programa.

```
case"Salir": Tools.imprime("Fin del programa");
    break;
```

EJECUTABLE DEL METODO SALIR



CLASE COLA C

Esta cola se caracteriza por usar ArrayList; La clase ArrayList en Java, es una clase que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los Arrays, por esto es por lo que esta cola es dinámica

Empezando por crear la clase llamada cola implementando los métodos de **ColaTDA** que son los métodos que nos ayudaran, y creando una variables de tipo ArrayList llamado cola para crear el objeto cola de tipo ArrayList ,una variable local "u" la cual estará inicializada con 0

```
public class ColaC <T>implements ColaTDA<T>{
private ArrayList cola;
byte u;
public ColaC() {
    cola = new ArrayList();
    u=0;
}
```

Para luego agregar **EL MÉTODO SIZE** que nos ayuda a obtener el número de elementos que hay dentro de una cola y disminuye en 1 en el proceso

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

El siguiente método lo implementaremos gracias a la interfaz de ColaTDA , llamado **ISEMPTYCOLA,** el cual nos valida si la cola está vacía o no ,el cual nos va a retornar cola llamando al método.

```
public boolean isEmptyCola() {
    // TODO Auto-generated method stub
    return cola.isEmpty();
}
```

EL MÉTODO PUSH nos sirve para meter elementos dentro de la COLA en el cual pide por medio de la acción “add” añadimos “dato” ósea datos en la cola, por eso también se esta incrementando “u”.

```
public void pushCola(Object dato) {
    // TODO Auto-generated method stub
    cola.add(dato);
    u++;
}
```

VACIAR es el método el cual lo que hace es limpiar y como su nombre lo dice vaciar la cola

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

EL MÉTODO POPCOLA saca un elemento de principio de la cola, donde se crea una variable de tipo T para mostrar el elemento que hay dentro de cola hasta tope pues ahí se encuentra lo que está hasta arriba de la cima y con cola remove elimina este elemento del inicio de la cola y nos retorna los datos dentro de la cola . por eso en el get se pone cero porque es la primera posición que eliminara.

```
public T popCola() {
    // TODO Auto-generated method stub
    T dato = (T)cola.get(0);
    cola.remove(0);
    u--;
    return dato;
}
```

PEEKCOLA es un método que nos ayudará a ver cuál es el elemento que se encuentra primero en la cola, por eso en el get se pone cero porque es la primera posición que mostrara.

```
public T peekCola() {
    // TODO Auto-generated method stub
    return (T)cola.get(0);
}
```

EL MÉTODO TOSTRING , nos sirve para mostrar los elementos dentro de la cola con ayuda de la variable i haciendo uso de la recursividad retornara si “i” menor o igual a “u” , lo que hay dentro de **cola.get** ,devolviendo el mismo método **toString** en i+1

```
public String toString() {
    return toString(0);
}
private String toString(int i) {
    return (i<u)?"" + i + " [" +cola.get(i) + "] ----> " + toString(i+1):"";
}
```

CLASE MENU COLA C

En este menú vamos a crear un objeto de ColaC llamado cola en el cual vamos a decir que hay que ingresar elementos en la cola, para que este funcione hay que crear un switch en cual tendrá los 5 métodos de “push,pop,peek,free y salir” y fuera del switch tendrá un do-while que va a terminar hasta que seleccionemos la opción “salir”. Y este es el código de dicho método:

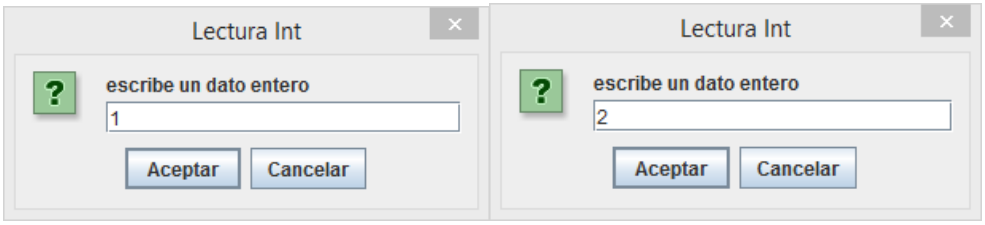
```
public class MenuColaC {
    public static void menu3(String menu){
        String sel="";
        ColaC <Integer> cola = new ColaC<Integer>();
```

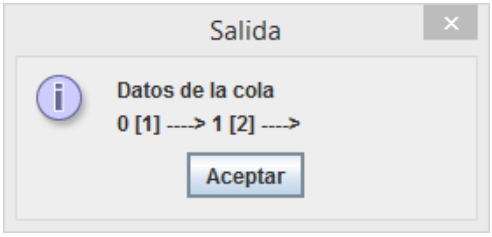
Este es la parte anterior explicada.

En el caso de PUSH ingresamos elementos en la cola por medio del método push, para después llamar al toString y mostrar los elementos antes ingresados.

```
case "Push":
    cola.pushCola(Tools.leerInt("Escribe un dato entero"));
    Tools.imprime("Datos de la cola \n"+cola.toString());
    break;
```

EJECUTABLES DEL METODO PUSH

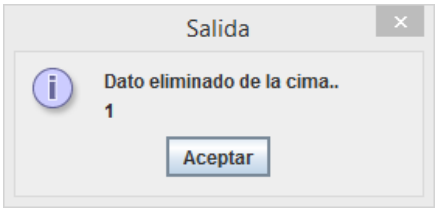




En el caso POP aquí nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino muestra un mensaje que se eliminó el dato que está en la cima (el primer elemento que ingreso), por medio del método pop.

```
case "Pop":
    if(cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia...!!");
    else Tools.imprime("Dato eliminado de la cima..\n"+cola.popCola());
    break;
```

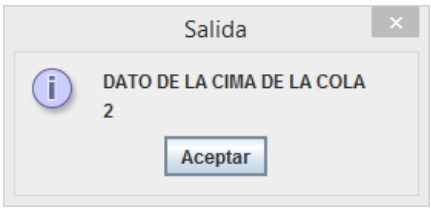
EJECUTABLE DEL METODO POP



En el caso PEEK aquí nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino está vacía nos mostrara el elemento de la cima (el primer elemento) y llamamos al método peek.

```
case "Peek":
    if(cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia!!");
    else {
        Tools.imprime("DATO DE LA CIMA DE LA COLA \n"+cola.peekCola());
    }
    break;
```

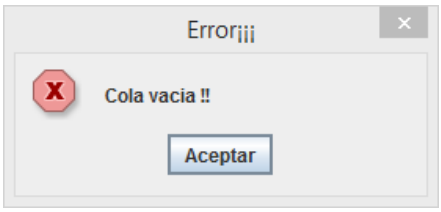
EJECUTABLE DEL METODO PEEK



En el caso de FREE nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino **ENTONCES POR MEDIO DEL METODO VACIAR, VA A VACIAR LA COLA (SI ES QUE HABIA ELEMENTOS).**

```
case "Free":
    if (cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia !!");
    else {
        cola.vaciar();
    }
    break;
```

EJECUTABLE DEL METODO FREE

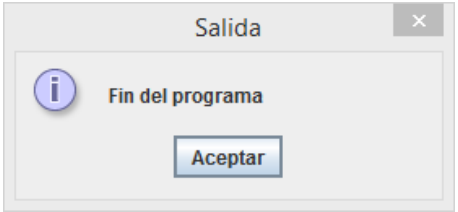


Si está llena la cola va a limpiar la cola y cuando queramos volver a darle free nos dirá que la cola está vacía.

El último caso SALIR solo se encarga de salir del menú y ya el programa termina, no sin antes mostrar un mensaje de Fin del programa.

```
case"Salir": Tools.imprime("Fin del programa");
    break;
```

EJECUTABLE DEL METODO SALIR



CLASE COLA NODO

Tendremos un tipo de dato en este caso “T” que va seguido de la variable dato, después un nodo llamado siguiente.

```
public class Nodo<T> {
public T info;
public Nodo sig;
```

Posteriormente creamos un constructor de la misma clase, en el cual siguiente llevara la dirección del elemento, e info el elemento.

```
public Nodo(T info) {
    super();
    this.info = info;
    this.sig =null;
}
```

Finalmente tenemos sus correspondientes get y sets los cuales sabemos que el método set es un método público, el cual se encarga de darle un valor a una propiedad o atributo de un objeto, y el get se encarga de mostrar un valor a una propiedad o atributo de un objeto.

```
public T getInfo() {
    return info;
}


public void setInfo(T info) {
    this.info = info;
}


public Nodo getSig() {
    return sig;
}
```

CLASE COLA D

```
public class ColaD<T>implements ColaTDA<T> {
private Nodo cola;
private Nodo f;
```

En este método lo que se va a hacer es que vamos a crear una Cola Nodo en la cual por medio de la interfaz **COLATDA** vamos a meter, eliminar, mostrar, ver si la cola está llena o no y liberar elementos de esta. No sin antes vamos a declarar a Nodo llamado cola y otro Nodo que lleve "f" ósea el fondo al final de la lista, seguido del **CONSTRUCTOR DE LA CLASE EL CUAL ES EL ENCARGADO DE CREAR LA COLA:**

```
public ColaD() {
    cola=null;
}
```

El cola igual a nulo es necesario porque cuando este nulo, habrá que ingresar elementos obviamente.

Para esto tenemos el siguiente **MÉTODO PUSH COLA,** el cual nos sirve para meter elementos a la COLA:

```
public void pushCola(T dato) {
    Nodo u=new Nodo (dato);
    if (isEmptyCola())
            cola=u;
    else
    {
    f.sig=u;
    }
    f=u;
}
```

En ese método vamos a crear un objeto Nodo llamado u, en el cual vamos a tomar el dato que metamos en la cola, y posteriormente vamos a preguntar si la cola está llena, si lo hay diremos que lo que hay en **"u"** se lo asignemos a cola, después en caso contrario Debemos enlazar el puntero sig del último nodo con el nodo recién

creado **“f.sig=u”.** Y por último el puntero externo **“F”** debe apuntar al nodo apuntado por **“u”:**

Después el **MÉTODO POP COLA** nos sirve para eliminar un elemento de la COLA:

```
public T popCola() {
    Nodo u = cola;
    T dato = (T) cola.getInfo();
    cola=cola.getSig();
    u=null;
    return dato;
}
```

Aquí vamos a volver a llamar a Nodo con un puntero llamado “u” y en esta parte cola será asignado a el Nodo, después en T dato va a almacenar el número por medio de **getInfo** (en este caso el primer elemento introducido), entonces en cola traeremos lo que hay en siguiente, obviamente a u le asignaremos nulo, que es cuando ya en la cola se eliminaron todos los elementos.. En pocas palabras saca el primer elemento que se ingresó a la cola, es como decir “el primero que entra es el primero que sale”. Finalmente retornamos el dato para eliminarlo. Por otro lado, **en el Método isEmptyCola** checamos si la cola está vacía o no:

```
public boolean isEmptyCola() {
    return (cola==null);
}
```

En pocas palabras vamos a retornar cuando el “COLA” (último elemento) sea igual a nulo que quiere decir que está vacía, sino lo está nos seguirá pidiendo que introduzcamos más elementos.

MÉTODO VACIAR

```
public void vaciar() {
    cola=null;
}
```

En este método va a vaciar la cola hasta que dé a nulo.

METODO FREECOLA

```
public void freeCola() {
    cola = null;
    f=null;
}
```

Este método se le asignara a cola nulo y “f” también ya que con nulo nos dice que no hay nada en la cola.

METODO PEEKCOLA

```
public T peekCola() {
    return (T) (cola.getInfo());
}
```

En este método va a retornar lo que hay en **cola.getInfo**, en este caso el 0 que es el primer dato que se encuentra en el principio de la cola.

METODOS TOSTRING

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

Por último en este método lo que hay en cola se lo asignarlo a Nodo y a su puntero llamado “u”, por lo cual nos va a retornar lo que hay en **toString** en este caso el “u”, posteriormente pregunta que si “i” es diferente e igual a nulo, entonces nos va a concatenar la posición, los elementos en cada posición y finalmente mostrarlos en pantalla. Sino lo fuera no va a mostrar nada.

CLASE MENU COLA D

En este menú vamos a crear un objeto de ColaD llamado cola en el cual vamos a decir que hay que ingresar elementos en la cola, para que este funcione hay que

crear un switch en cual tendrá los 5 métodos de “push,pop,peek,free y salir” y fuera del switch tendrá un do-while que va a terminar hasta que seleccionemos la opción “salir”. Y este es el código de dicho método:

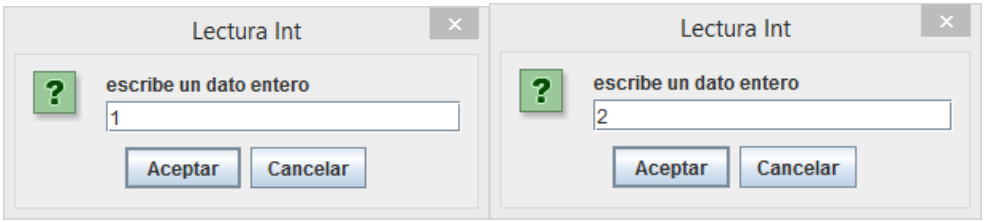
```
public class MenuColaD {
    public static void menu3(String menu){
        String sel="";
        ColaD<Integer> cola = new ColaD<Integer>();
```

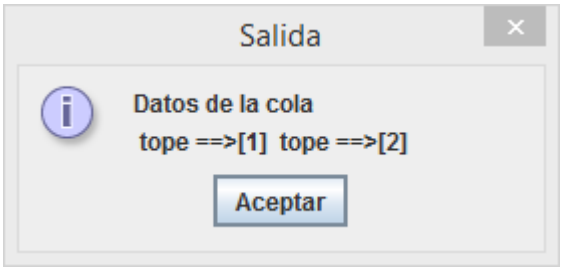
Este es la parte anterior explicada.

En el caso de PUSH ingresamos elementos en la cola por medio del método push, para después llamar al toString y mostrar los elementos antes ingresados.

```
case "Push":
    cola.pushCola(Tools.leerInt("Escribe un dato entero"));
    Tools.imprime("Datos de la cola \n"+cola.toString());
    break;
```

EJECUTABLES DEL METODO PUSH

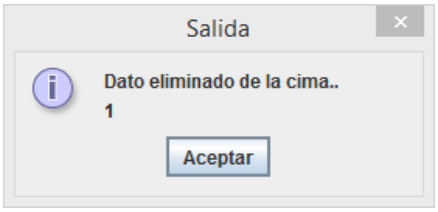




En el caso POP aquí nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino muestra un mensaje que se eliminó el dato que está en la cima (el primer elemento que ingreso), por medio del método pop.

```
case "Pop":
    if(cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia...!!");
    else Tools.imprime("Dato eliminado de la cima..\n"+cola.popCola());
    break;
```

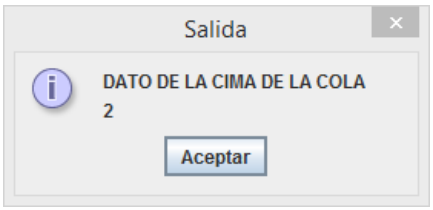
EJECUTABLE DEL METODO POP



En el caso PEEK aquí nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino está vacía nos mostrara el elemento de la cima (el primer elemento) y llamamos al método peek.

```
case "Peek":
    if(cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia!!");
    else {
        Tools.imprime("DATO DE LA CIMA DE LA COLA \n"+cola.peekCola());
    }
    break;
```

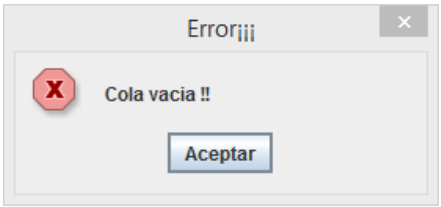
EJECUTABLE DEL METODO PEEK



En el caso de FREE nos pregunta que si la cola está vacía, si lo está nos manda un mensaje que está vacía, sino llamamos a “vaciar” para que vacié la cola.

```
case "Free":
    if (cola.isEmptyCola())Tools.imprimeErrorMsje("Cola vacia !!");
    else {
        cola.vaciar();
    }
    break;
```

EJECUTABLE DEL METODO FREE



Si está llena la cola va a limpiar la cola y cuando queramos volver a darle free nos dirá que la cola está vacía.

El último caso SALIR solo se encarga de salir del menú y ya el programa termina, no sin antes mostrar un mensaje de Fin del programa.

```
case"Salir": Tools.imprime("Fin del programa");
    break;
```

EJECUTABLE DEL METODO SALIR

CONCLUSIÓN

Con lo anterior expuesto en la práctica, podemos decir que las estructuras de datos lineales son aquellas en las que los elementos ocupan lugares sucesivos en la estructura y cada uno de ellos tiene un único sucesor y predecesor, es decir, sus elementos están ubicados uno al lado del otro relacionados en forma lineal.

De las cuales vimos las siguientes estructuras de datos lineales:

- Pilas
- Colas

Aparte de que sabemos que para que sirve cada una:

Obviamente la pila es un tipo especial de lista lineal dentro de las estructuras de datos dinámicas que permite almacenar y recuperar datos, siendo el modo de acceso a sus elementos de tipo LIFO (del inglés Last In, First Out, es decir, último en entrar, primero en salir). Y por otro lado, la cola es un tipo especial de lista abierta en la que sólo se pueden insertar nodos en uno de los extremos de la lista y sólo se pueden eliminar nodos en el otro. Además, como sucede con las pilas, las escrituras de datos siempre son inserciones de nodos, y las lecturas siempre eliminan el nodo

leído. Solo que en este caso este tipo de lista es conocido como lista FIFO (First In First Out), el primero en entrar es el primero en salir.

BIBLIOGRAFÍA

- Computación I. (s/f). Uchile.cl. Recuperado el 10 de abril de 2023, de https://users.dcc.uchile.cl/~lmateu/CC10A/cc10a-lib/stack.html
- de Roer, DD (2017, 28 de junio). Pila Dinámica en Java. Discoteca Duro de Roer - . https://www.discoduroderoer.es/pila-dinamica-en-java /
- (S/f). Edu.ar. Recuperado el 10 de abril de 2023, de http://www.frlp.utn.edu.ar/materias/algoritmos/TP10_2012.pdf
- Androide, Y. (2018, julio 10). Pilas y colas en Java - Teoría, definiciones y ejemplos. YoAndroide. https://yoandroide.xyz/pilas-y-colas-en-java-teoria-definiciones-y-ejemplos/
- Marcos, M., León, A., Noralys, F. M., Maldonado, M. M., & Juan, A. (s/f). PILAS Y COLAS Y SU IMPLEMENTACIÓN EN OBJECT PASCAL. PILES AND QUEENS AND THEIR IMPLEMENTATION IN OBJECT PASCAL. Informatica-juridica.com. Recuperado el 11 de abril de 2023, de https://www.informatica-juridica.com/wp-content/uploads/2019/02/Art%C3%ADculo-sobre-Pilas-y-Colas-converted.pdfPilas en Java. (s/f). CÓDIGO Libre. Recuperado el 11 de abril de 2023, de http://codigolibre.weebly.com/blog/pilas-en-java
- Pilas y Colas en Java. (s/f). Blogspot.com. Recuperado el 11 de abril de 2023, de http://estructuradedatosjp.blogspot.com/2015/11/pilas-y-colas-en-java.html
- 3.3 Listas enlazadas. (2010, April 17). Estructura de datos. https://estruturadedatos.wordpress.com/unidad-3-estructuras-lineales-estaticas-y-dinamicas/3-3-listas-enlazadas/
- Cerinza, N. G. (n.d.). FAEDIS. Edu.co. Retrieved April 22, 2023, from http://virtual.umng.edu.co/distancia/ecosistema/odin/odin_desktop.php?path

=Li4vb3Zhcy9pbmdlbmllcmlhX2luZm9ybWF0aWNhL2VzdHJ1Y3R1cmFfZGVfZGF0b3MvdW5pZGFkXzMv

- Morales, O. (2020, December 20). Uso de Clases genéricas en java. Medium. https://omorales71.medium.com/uso-de-clases-gen%C3%A9ricas-en-java-3695a3d15397