

Estructuras de Datos

Clase 4 – Pilas y colas



Dr. Sergio A. Gómez
<http://cs.uns.edu.ar/~sag>



Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Tipo de dato abstracto

- Un *tipo de dato abstracto* (TDA) (en inglés, ADT por Abstract Data Type) es un tipo definido solamente en términos de sus operaciones y de las restricciones que valen entre las operaciones.
- Las restricciones las daremos en términos de comentarios.
- Cada TDA se representa en esta materia con una (o varias) interfaces.
- Una o más implementaciones del TDA se brindan en términos de clases concretas.
- Hoy daremos los TDAs Pila (Stack) y Cola (Queue) (la clase que viene daremos sus aplicaciones).

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.

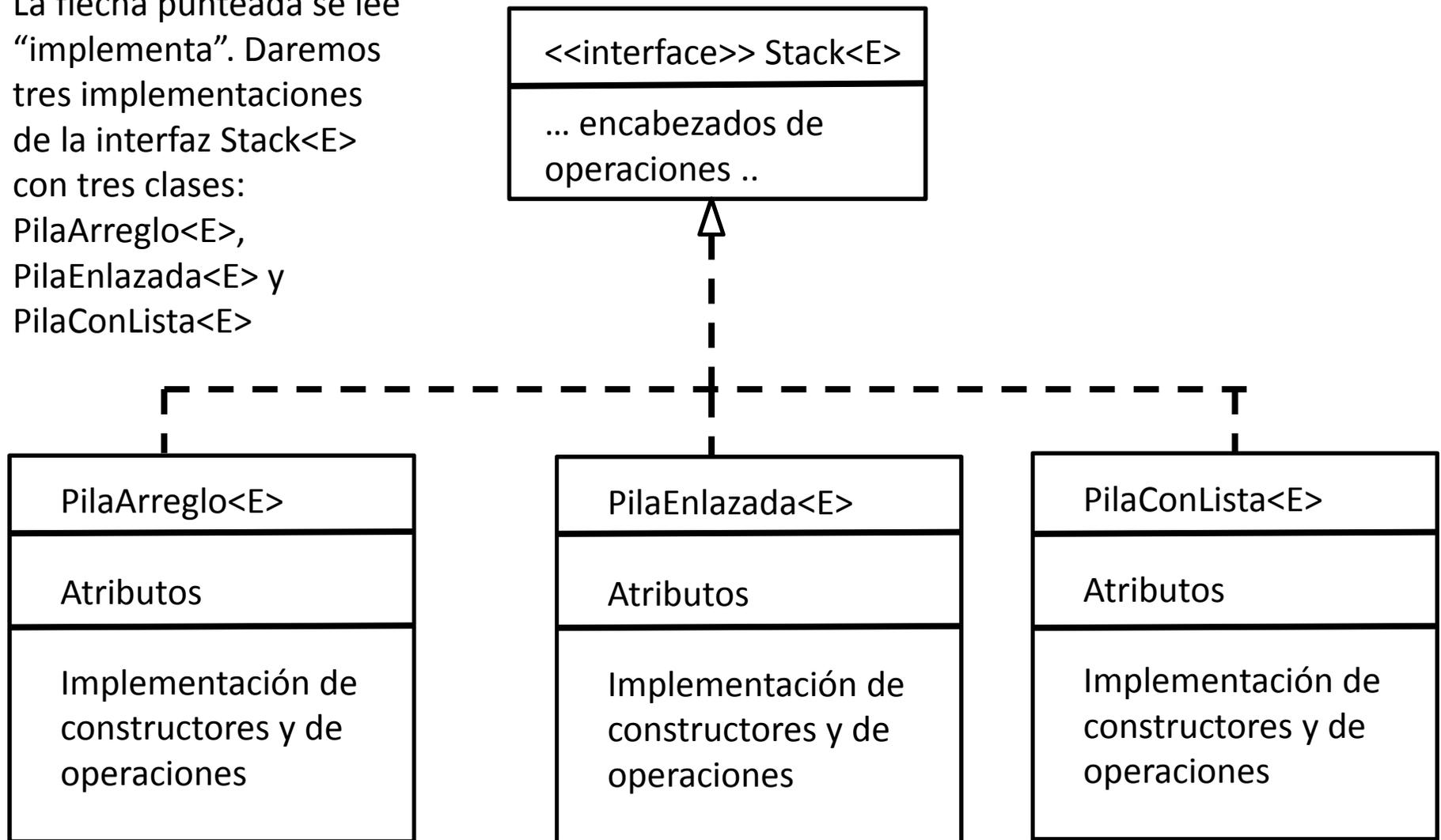
Implementación de Pila

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

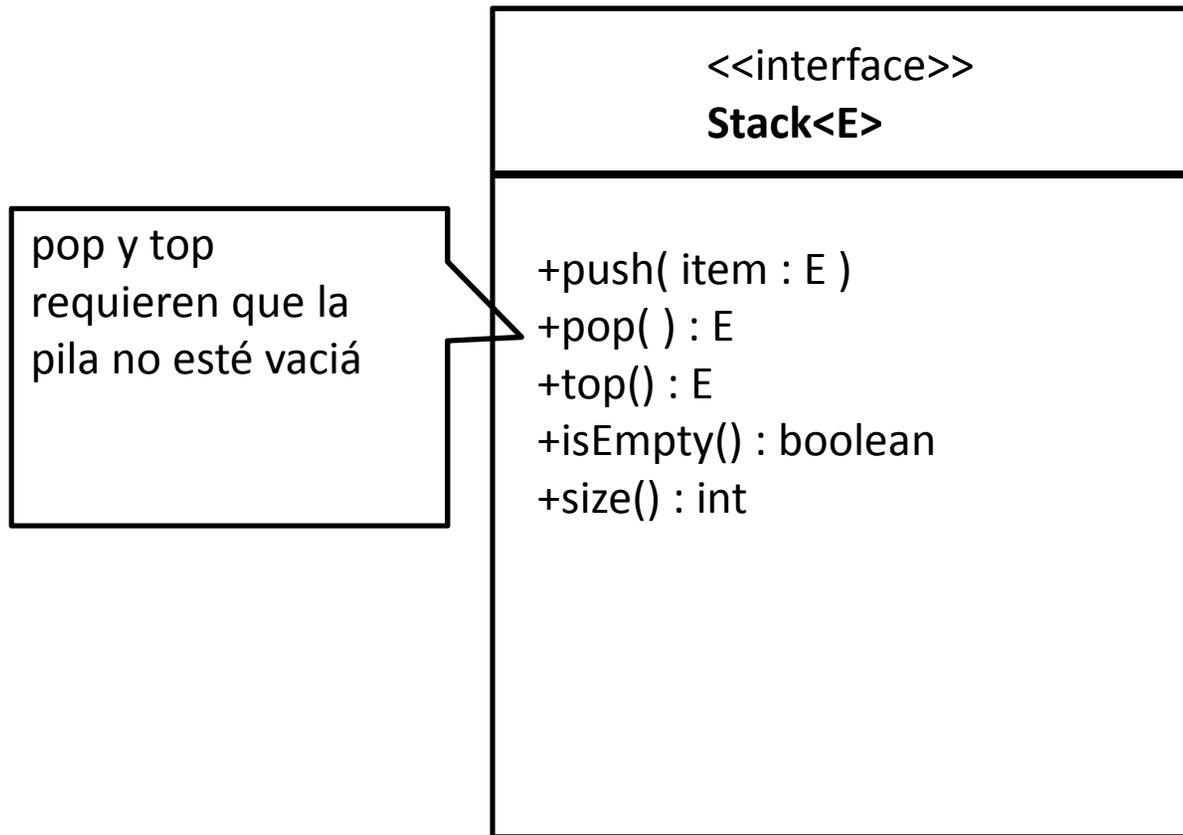
Diagrama UML del diseño

La flecha punteada se lee “implementa”. Daremos tres implementaciones de la interfaz Stack<E> con tres clases: PilaArreglo<E>, PilaEnlazada<E> y PilaConLista<E>



Interfaz Stack

En UML las operaciones se dan con una sintaxis Pascal-like y las excepciones se dan como comentarios:



Código Java en archivo *Stack.java*:

```
public interface Stack<E> {  
    // Inserta item en el tope de la pila.  
    public void push( E item );  
  
    // Retorna true si la pila está vacía y falso en caso contrario.  
    public boolean isEmpty();  
  
    // Elimina el elemento del tope de la pila y lo retorna.  
    // Produce un error si la pila está vacía.  
    public E pop() throws EmptyStackException;  
  
    // Retorna el elemento del tope de la pila y lo retorna.  
    // Produce un error si la pila está vacía.  
    public E top() throws EmptyStackException;  
  
    // Retorna la cantidad de elementos de la pila.  
    public int size();  
}
```

Nota: Los comentarios a la hora de programar los haremos como comentarios Javadoc :

```
/**
 * Inserta un elemento en el tope de la pila.
 * @param element Elemento a insertar.
 */
public void push(E element);

/**
 * Remueve el elemento que se encuentra en el tope de la pila.
 * @return Elemento removido.
 * @throws EmptyStackException si la pila está vacía.
 */
public E pop() throws EmptyStackException;
```

Nota: Luego se compilan con la herramienta Javadoc.exe para generar un sitio web que forma la documentación del código fuente y se ve así:

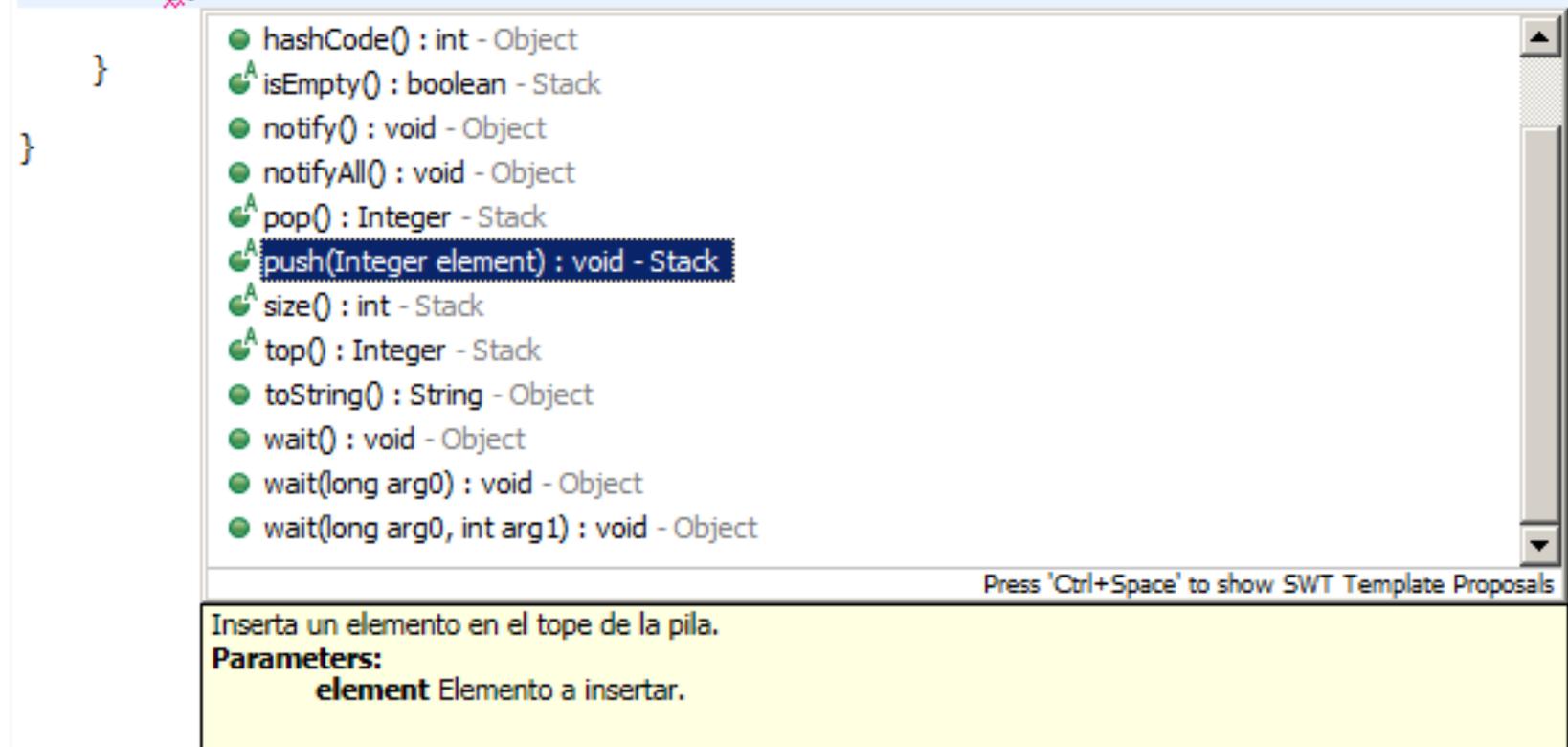
Method Summary

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
boolean	<code>isEmpty()</code>	Consulta si la pila está vacía.
E	<code>pop()</code>	Remueve el elemento que se encuentra en el tope de la pila.
void	<code>push(E element)</code>	Inserta un elemento en el tope de la pila.
int	<code>size()</code>	Consulta la cantidad de elementos de la pila.
E	<code>top()</code>	Examina el elemento que se encuentra en el tope de la pila.

push
<code>void push(E element)</code>
Inserta un elemento en el tope de la pila.
Parameters: element - Elemento a insertar.
pop
<code>E pop()</code> <code>throws EmptyStackException</code>
Remueve el elemento que se encuentra en el tope de la pila.
Returns: Elemento removido.
Throws: <code>EmptyStackException</code> - si la pila está vacía.

Nota: Es importante escribir los comentarios Javadoc junto con el código y no al final porque los mismos se usan en el Intellisense (i.e. los globitos amarillos de ayuda del autocompletado de código en el editor de código de Eclipse).

```
public class App {  
    public static void main(String[] args) {  
        Stack<Integer> s;  
        .  
    }  
}
```



- hashCode() : int - Object
- isEmpty() : boolean - Stack
- notify() : void - Object
- notifyAll() : void - Object
- pop() : Integer - Stack
- push(Integer element) : void - Stack**
- size() : int - Stack
- top() : Integer - Stack
- toString() : String - Object
- wait() : void - Object
- wait(long arg0) : void - Object
- wait(long arg0, int arg1) : void - Object

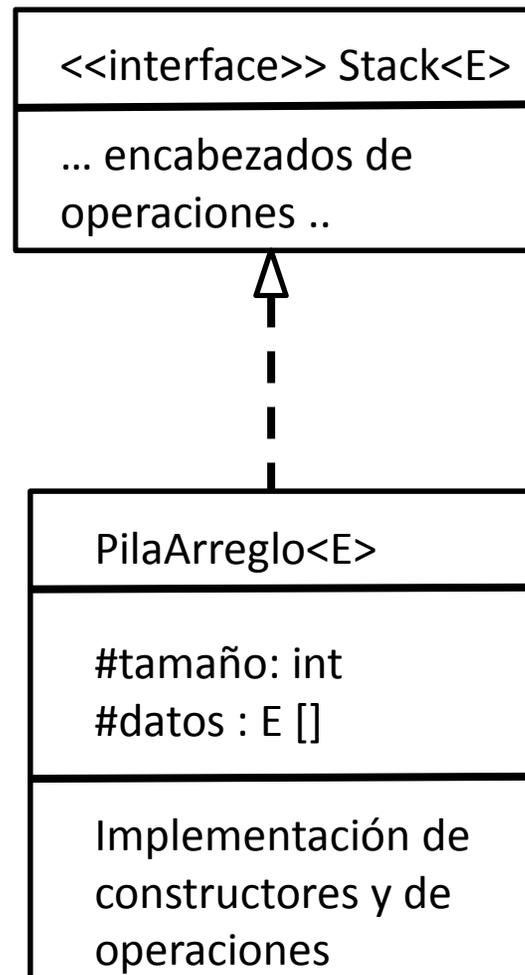
Press 'Ctrl+Space' to show SWT Template Proposals

Inserta un elemento en el tope de la pila.
Parameters:
element Elemento a insertar.

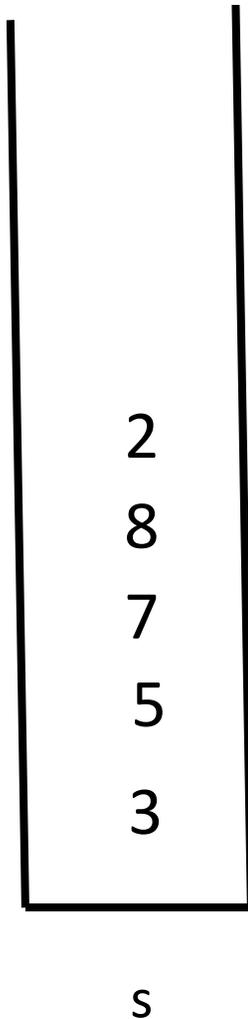
Implementaciones de pilas

- 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)

Implementación de pila con arreglo



Mostramos la clase *PilaArreglo* (para completar) y otra clase *Aplicación...* que la utiliza para armar la pila s:

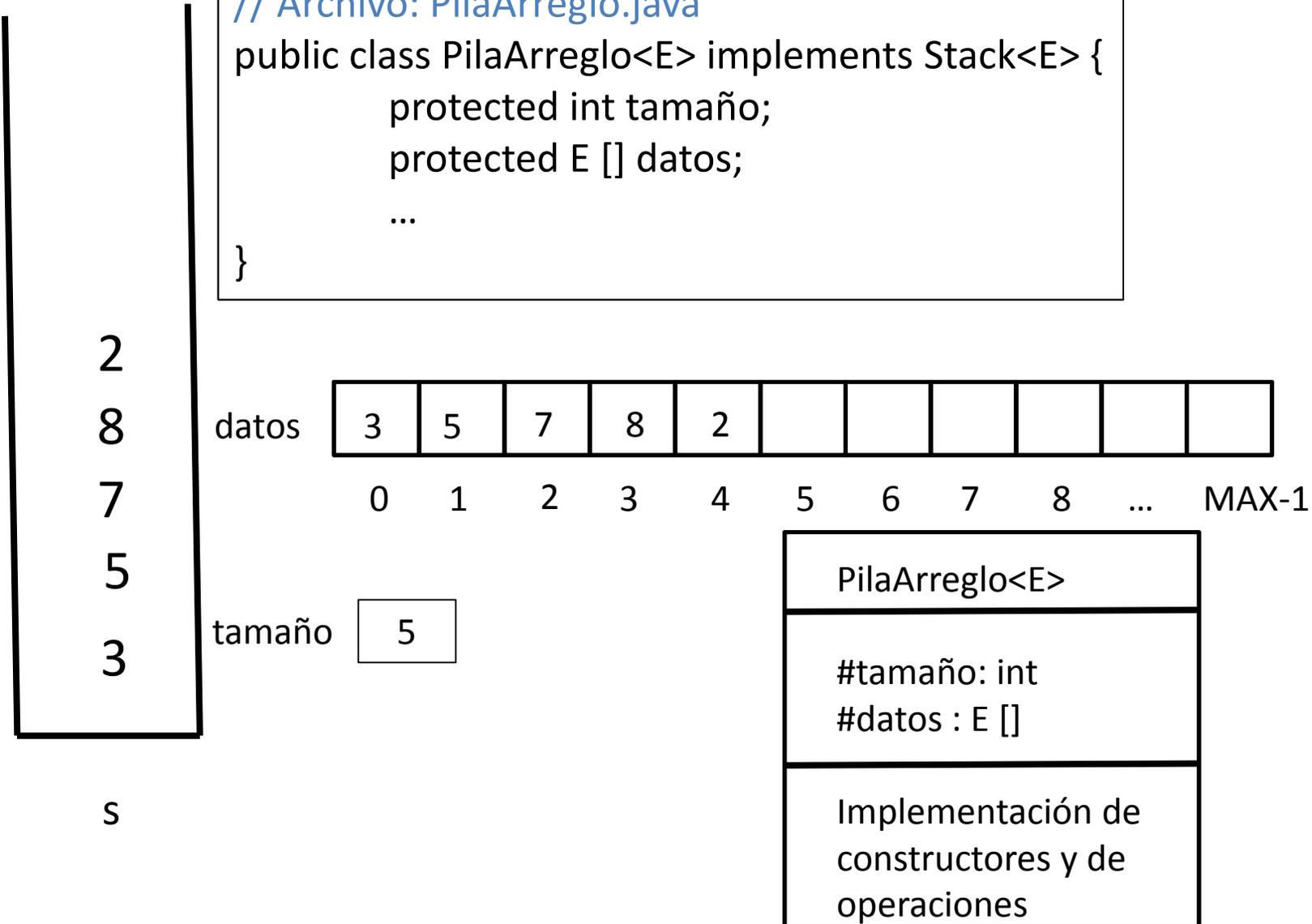


```
// Archivo: PilaArreglo.java  
public class PilaArreglo<E> implements Stack<E> {  
    ...  
}
```

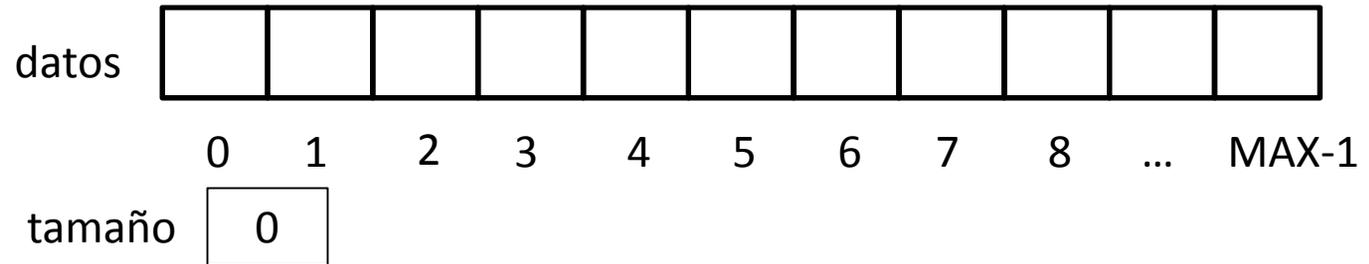
```
// Archivo: AplicacionUsaPilaArreglo.java  
public class AplicacionUsaPilaArreglo {  
    public static void main( String [] args ) {  
        Stack<Integer> s = new PilaArreglo<Integer>();  
        s.push( 3 );  
        s.push( 5);  
        s.push( 7 );  
        s.push( 8 );  
        s.push( 2 );  
    }  
}
```

Pila: Atributos de la implementación con arreglo junto con su representación

```
// Archivo: PilaArreglo.java
public class PilaArreglo<E> implements Stack<E> {
    protected int tamaño;
    protected E [] datos;
    ...
}
```



Constructor: Crea una pila vacía



Constructor 1:

```
PilaArreglo( MAX : int ) {  
    tamaño = 0;  
    datos = (E []) new Object[MAX];  
}
```

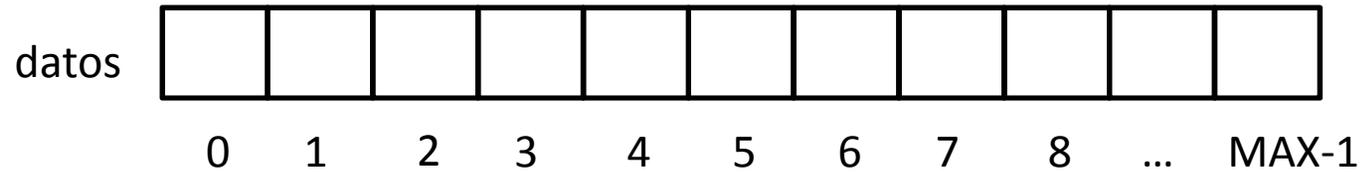
Constructor 2:

```
PilaArreglo() { this(20); }
```

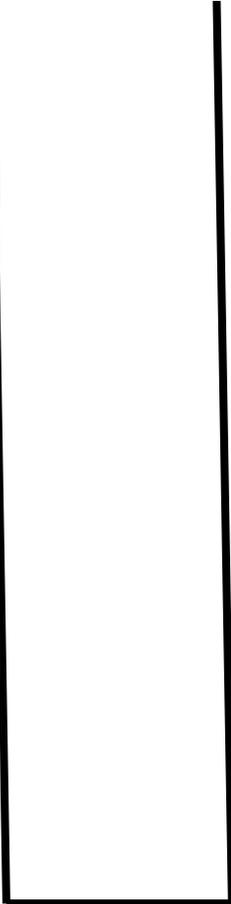
El constructor2 crea una pila que tiene 20 elementos y delega en el constructor1.

Nota: Si MAX fuera negativo podríamos lanzar una excepción

Implementación de isEmpty



tamaño



S

Pila Vacía:

isEmpty() : boolean

{

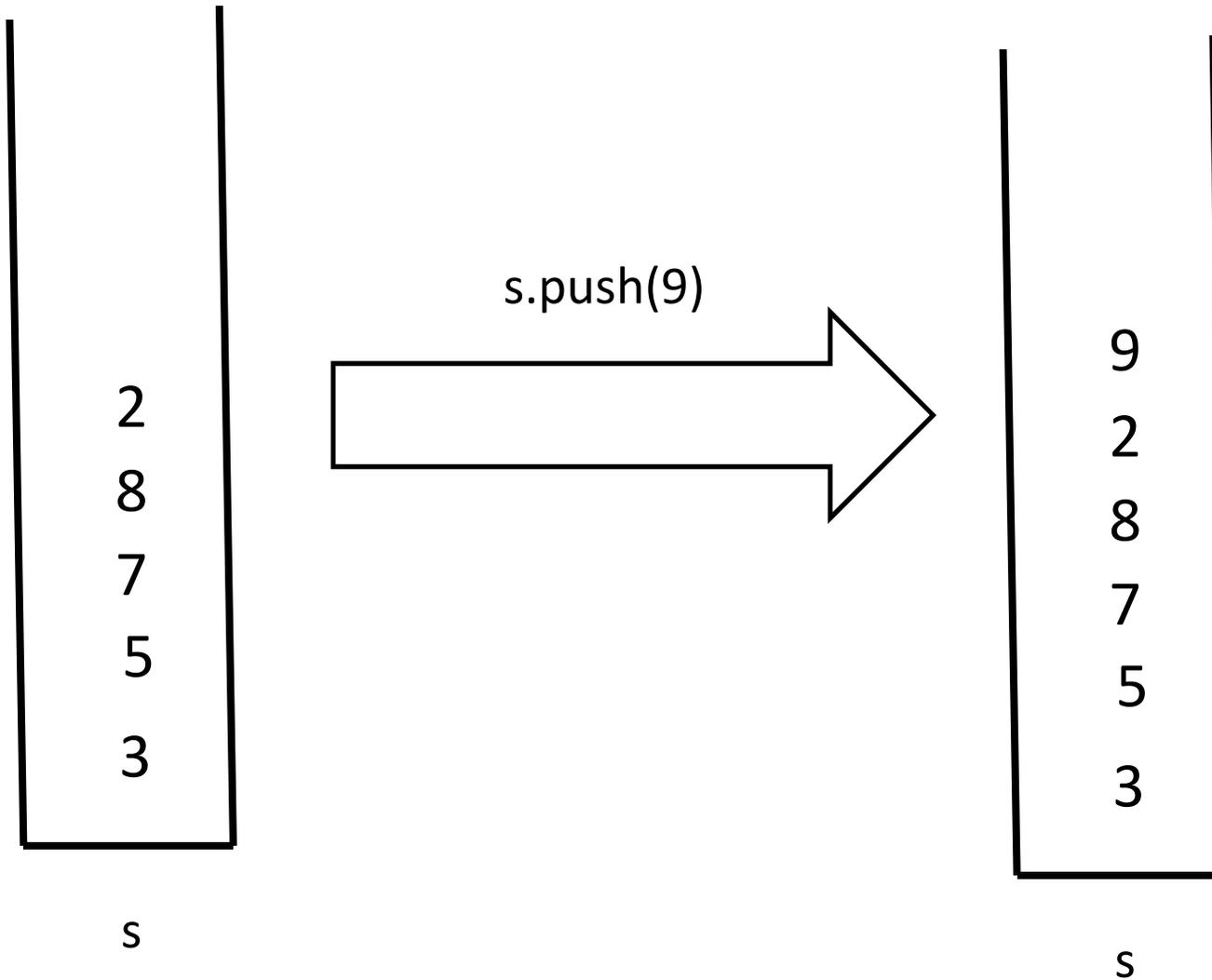
 Retornar tamaño == 0

}

Sea n = la cantidad de elementos de la pila this:

$T_{isEmpty}(n) = O(1)$

Implementación de Apilar: Dada la pila de la izquierda luego de apilar un 9 obtenemos la pila de la derecha



Implementación de Apilar

Opción1: Si hay lugar, insertar el nuevo elemento al final del arreglo. Si no hay lugar, generar una situación de error lanzando una excepción *FullStackException*.

```
Apilar: Push(item : E) {  
    Si tamaño == TAMAÑO DEL ARREGLO entonces  
        error (“Pila llena”)  
  
    Sino  
        datos[tamaño] = item  
        tamaño = tamaño + 1  
}
```

Nota: $T_{\text{push}}(n) = O(1)$

Problema: La interfaz Stack no declara una excepción para push.

Soluciones:

- (1) Cambiar la signatura de la operación push declarando la excepción *FullStackException*. Elegante pero requiere recompilar otras implementaciones existentes de la pila (A veces, si la interfaz es importada no la puedo cambiar).
- (2) Utilizar una *RuntimeException*, que es unchecked y no requiere ser declarada. Poco elegante (las *RuntimeExceptions* no deben ser usadas para programar) pero no requiere recompilar otras implementaciones existentes.

Implementación de apilar

Opción2: Cuando el arreglo esté lleno, incrementar el tamaño del arreglo (por ejemplo agregarle 10 elementos), luego copiar los elementos al nuevo arreglo y después insertar el nuevo elemento.

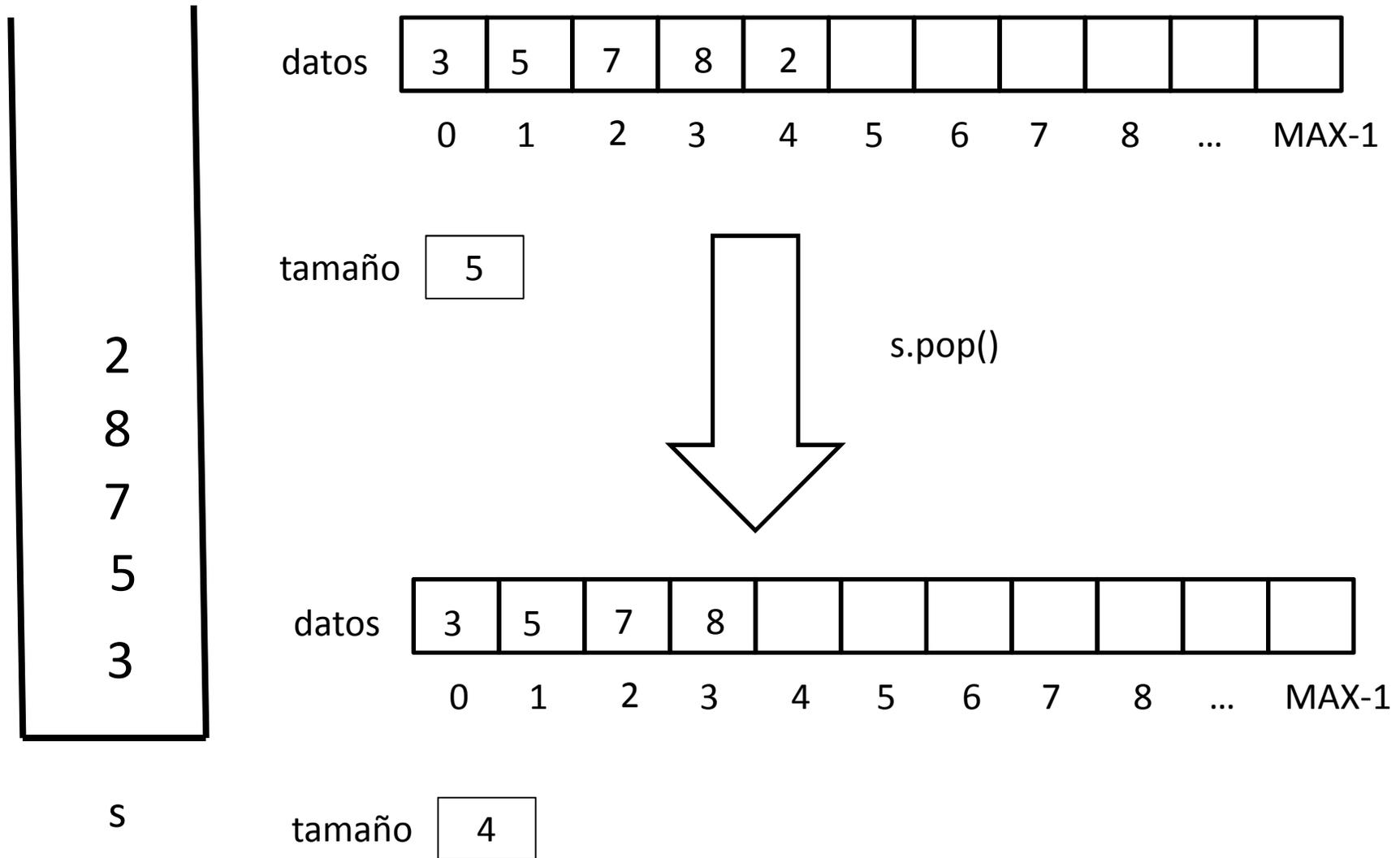
Ventaja: La pila puede ahora crecer en forma no acotada.

Desventaja: Ahora $T_{push}(n) = O(n)$.

Nota: Le podemos dar libertad al usuario con un nuevo constructor que reciba además del tamaño inicial, el incremento a utilizar cada vez que se llena el arreglo

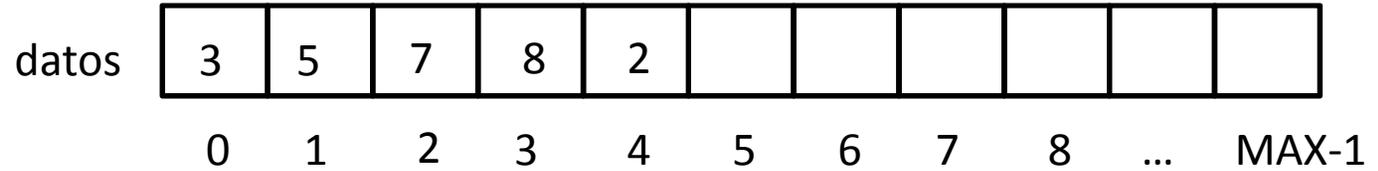
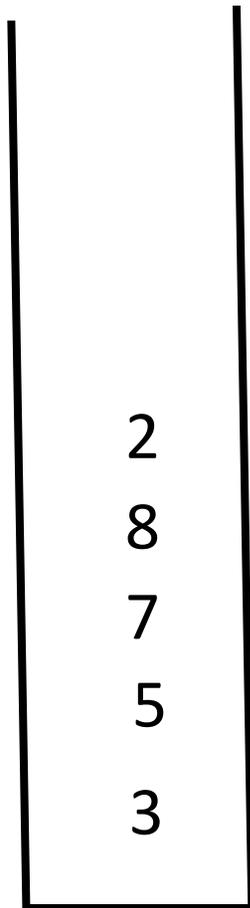
PilaConArreglo(MAX: int, INCREMENTO_TAMAÑO: int) { ... }

Implementación de desapilar



Pop() elimina el 2 del tope de la pila

Implementación de desapilar



tamaño

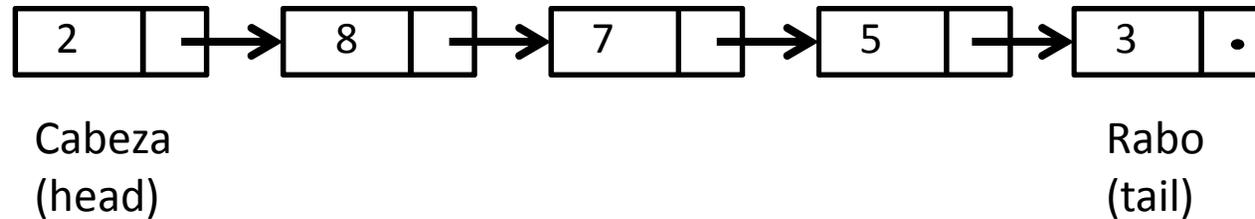
5

```
pop() : E
{
  Si tamaño == 0 entonces
    error ("Pila vacía")
  Sino
    aux = datos[tamaño-1]
    datos[tamaño-1] = null
    tamaño = tamaño - 1
    retornar aux
}
```

Notas: El error se modela con excepción `EmptyStackException` y $T_{\text{pop}}(n) = O(1)$

Implementación con nodos enlazados

Utilizaremos una estructura de nodos enlazados. La pila mostrada se representará con nodos (celdas) enlazados:



Notas:

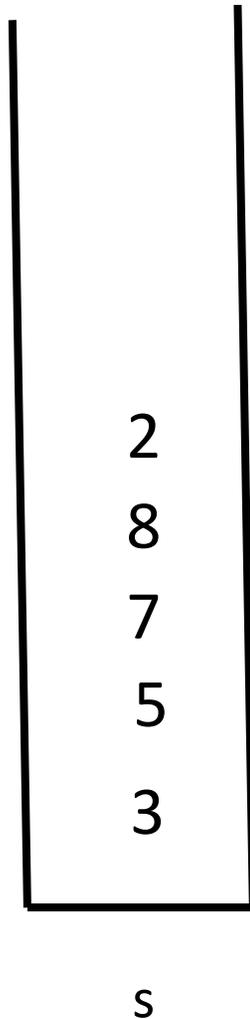
La pila conoce sólo el nodo *cabeza*, que corresponde al tope o último dato apilado.

Todas las operaciones tienen $O(1)$ pues la estructura es no acotada y no hay casos especiales porque nunca se llena.

Es más difícil de codificar, depurar y mantener pero se gana en flexibilidad.

Se puede almacenar el tamaño como un atributo más para evitar calcular `size()` en $O(n)$.

Pila: Implementación con nodos enlazados

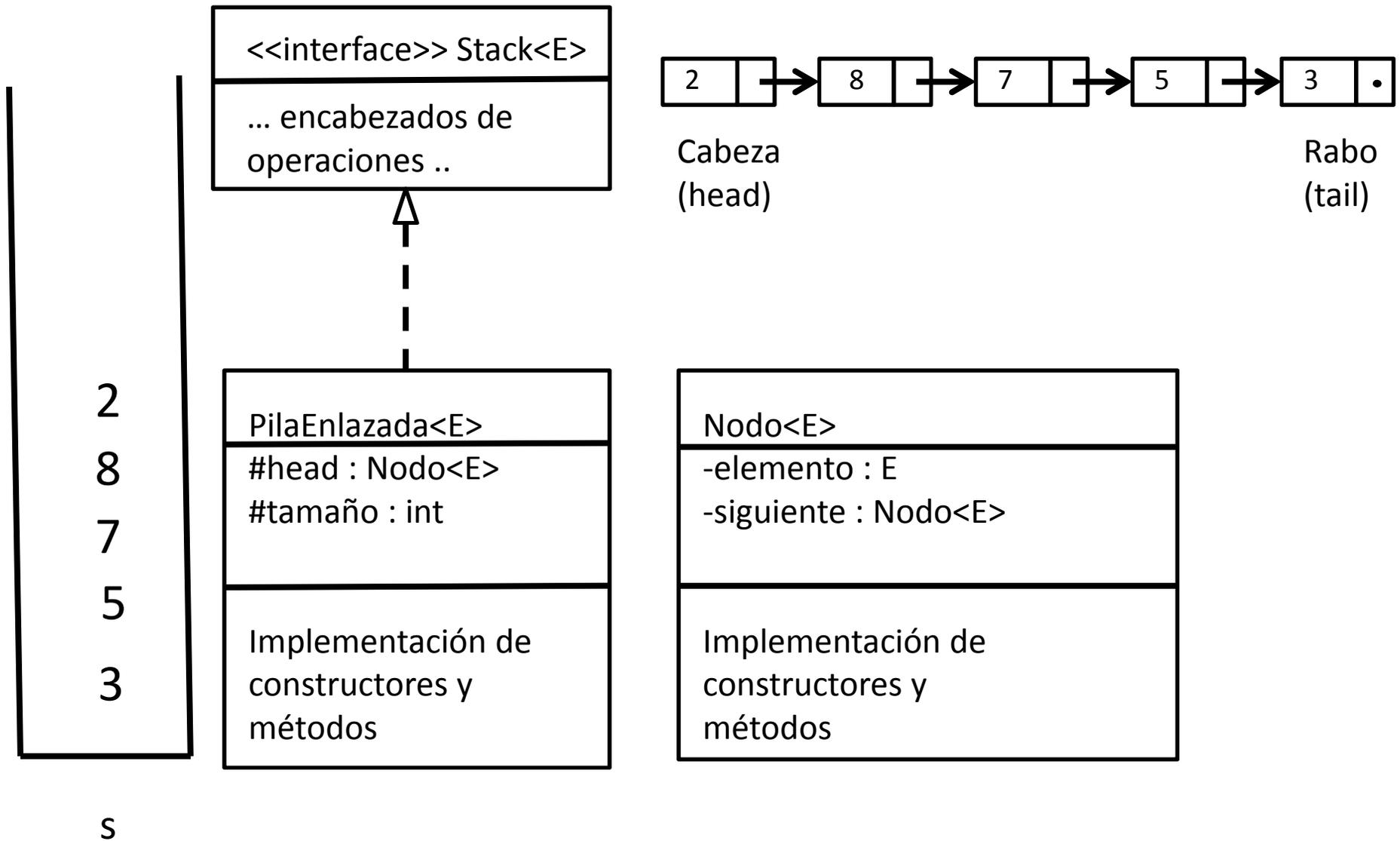


```
// Archivo: PilaEnlazada.java
public class PilaEnlazada<E> implements Stack<E> {
...
}
```

```
// Archivo: AplicacionUsaPilaEnlazada.java
public class AplicacionUsaPilaEnlazada {
    public static void main( String [] args ) {
        Stack<Integer> s = new PilaEnlazada<Integer>();
        s.push( 3 );
        s.push( 5);
        s.push( 7 );
        s.push( 8 );
        s.push( 2 );
    }
}
```

Ahora la aplicación debe invocar el constructor de la nueva clase pero como la variable s es de tipo Stack el resto de la implementación de main no cambia con respecto al ejemplo de diapositiva 12.

Implementación de pilas con nodos enlazados



Pila: Implementación con nodos enlazados

```
public class Nodo<E> {
    private E elemento;
    private Nodo<E> siguiente;

    // Constructores:
    public Nodo( E item, Nodo<E> sig )
    { elemento=item; siguiente=sig; }
    public Nodo( E item ) {this(item,null); }

    // Setters:
    public void setElemento( E elemento )
    { this.elemento=elemento;}
    public void setSiguiente( Nodo<E> siguiente ){
        this.siguiente = siguiente;
    }

    // Getters:
    public E getElemento() { return elemento;}
    public Nodo<E> getSiguiente()
    { return siguiente;}
}
```

```
Public class PilaEnlazada<E>
    implements Stack<E> {
    protected Nodo<E> head;
    protected int tamaño;

    ....
}
```

Nota: La clase `Nodo` es *recursiva*, ya que los nodos están definidos en términos de nodos pues el campo `siguiente` hace que nodo conozca a otro nodo.

Pila con nodos enlazados: Constructor

El constructor crea una pila vacía.

Recordemos que:

```
Public class PilaEnlazada<E>
    implements Stack<E> {
    protected Nodo<E> head;
    protected int tamaño;

    ....
}
```

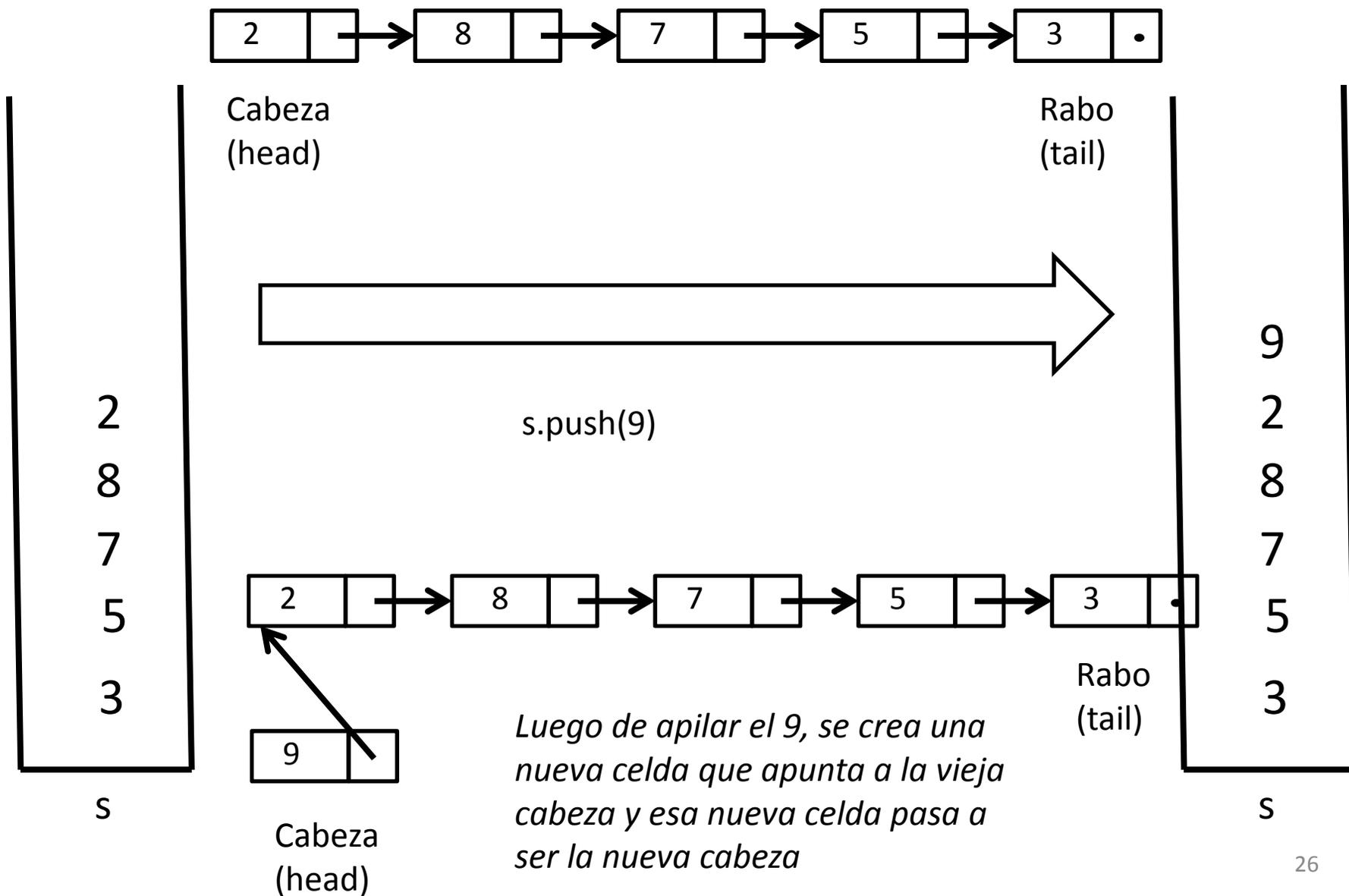
Constructor:

```
PilaEnlazada() {
    Head = null
    Tamaño = 0
}
```

Un caso de prueba para validar el constructor e isEmpty puede ser testear: `(new PilaEnlazada<E>()).isEmpty() == true`

S

Apilar



Apilar

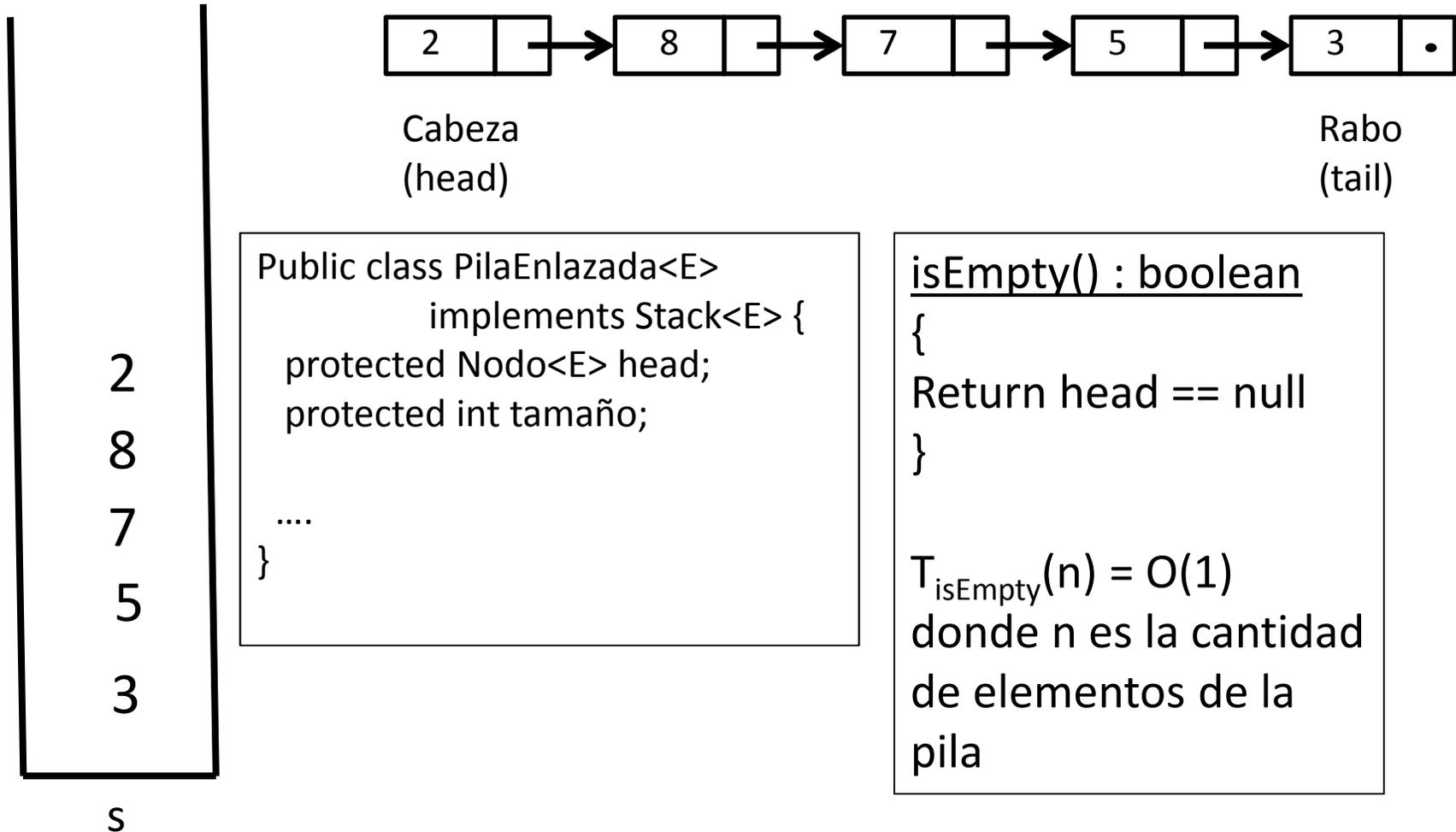
```
Public class PilaEnlazada<E>
    implements Stack<E> {
    protected Nodo<E> head;
    protected int tamaño;

    ....
}
```

```
push(item : E)
{
    Nodo<E> aux = new Nodo<E>( )
    aux.setElemento( item )
    aux.setSiguiete( head )
    head = aux
    tamaño = tamaño + 1
}
```

$T_{\text{push}}(n) = O(1)$

Pila: Implementación con nodos enlazados



Desapilar



Cabeza
(head)

Rabo
(tail)

Desapilar requiere eliminar la celda que contiene el 2 y hacer que la celda que contiene el 8 sea la nueva cabeza.

Nota: Como la celda del 2 no está referenciada por nadie, eventualmente será reclamada por el recolector de basura aunque la celda del 2 siga apuntando a la celda del 8.

Nota: Cuando la pila tiene un único elemento, el código tiene como efecto que head termine con valor null.

```
pop(): E {
```

```
Si isEmpty() entonces  
    error( "Pila vacia" )
```

```
Sino
```

```
    aux = head.getElemento()  
    head = head.getSiguiente()  
    tamaño = tamaño - 1  
    retornar aux
```

```
}
```

```
 $T_{pop}(n) = O(1)$ 
```

2

8

7

5

3

s

Comentarios



Cabeza
(head)

rabo
(tail)

Pregunta: ¿Por qué conviene insertar en la cabeza de la lista y no en el rabo?

Si insertáramos en el rabo, la lista de nodos representaría la pila de la izquierda.

Si la pila tiene n elementos, al insertar en el rabo tenemos $T_{push}(n) = O(n)$

Entonces: insertar en el rabo sería ineficiente y NO LO VAMOS A HACER.

3

5

7

8

2

s

TDA Cola

Cola: Colección lineal de objetos actualizada en sus extremos llamados *frente* y *rabo* siguiendo una política FIFO (first-in first-out, 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:

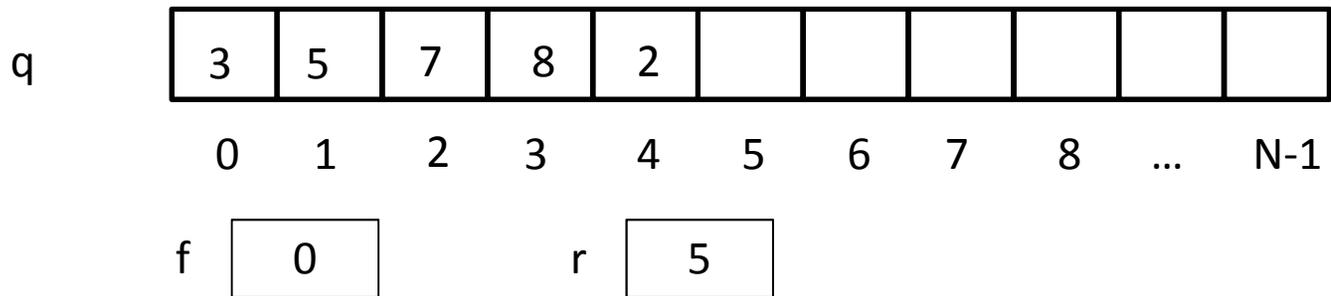
- Se abstrae de la ED con la que se implementará
- Se documenta el significado de cada método en lenguaje natural
- Se usa un parámetro formal de tipo representando el tipo de los elementos de la cola
- Se definen excepciones para las condiciones de error

```
public interface Queue<E> {  
    // Inserta el elemento e al final de la cola  
    public void enqueue(E e);  
  
    // Elimina el elemento del frente de la cola y lo retorna.  
    // Si la cola está vacía se produce un error.  
    public E dequeue() throws EmptyQueueException;  
  
    // Retorna el elemento del frente de la cola.  
    // Si la cola está vacía se produce un error.  
    public E front() throws EmptyQueueException;  
  
    // Retorna verdadero si la cola no tiene elementos  
    // y falso en caso contrario  
    public boolean isEmpty();  
  
    // Retorna la cantidad de elementos de la cola.  
    public int size();  
}
```

Implementaciones de colas

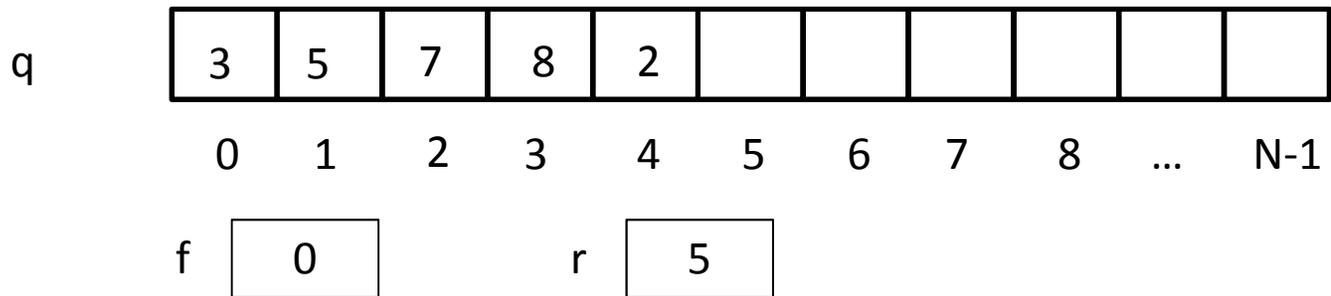
- 1) Con un arreglo circular
- 2) Con una estructura enlazada
- 3) En términos de una lista (lo dejamos pendiente hasta dar el TDA Lista)

Implementación de cola con un arreglo circular



- q es un arreglo de N componentes y mantiene los elementos de la cola
- El tamaño máximo de q es $N-1$ (lo que permite diferenciar la cola vacía de la cola llena)
- f es la posición en q del próximo elemento a eliminar en un dequeue
- r es la posición en la cual se va a insertar el siguiente elemento con un enqueue.
- Implementaremos operaciones en $O(1)$

Implementación de cola con un arreglo circular

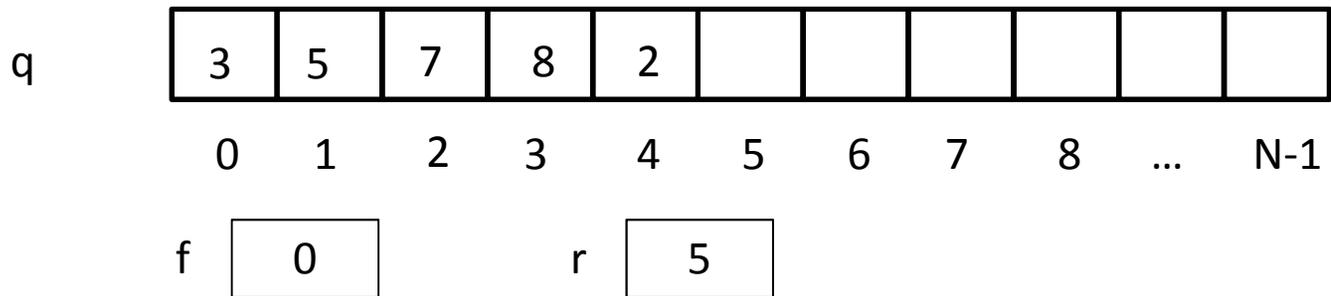


Constructor:

f = 0

r = 0

Implementación de cola con un arreglo circular



Constructor:

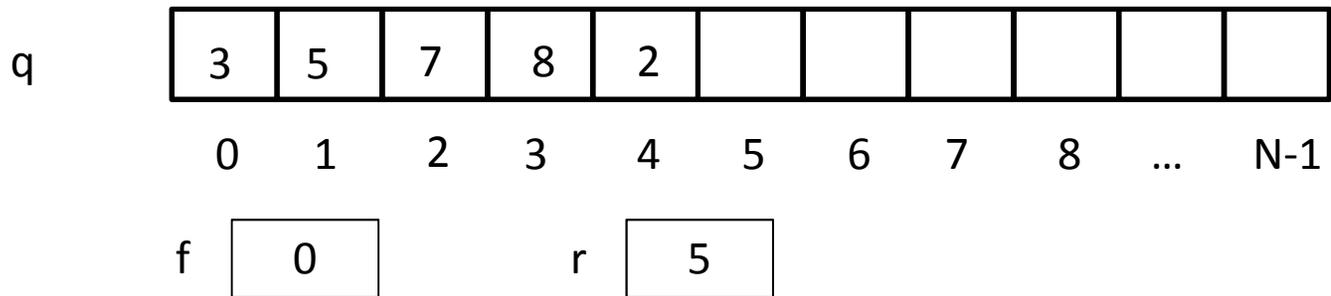
f = 0

r = 0

isEmpty():

Retornar f == r

Implementación de cola con un arreglo circular



Constructor:

f = 0

r = 0

isEmpty():

Retornar f == r

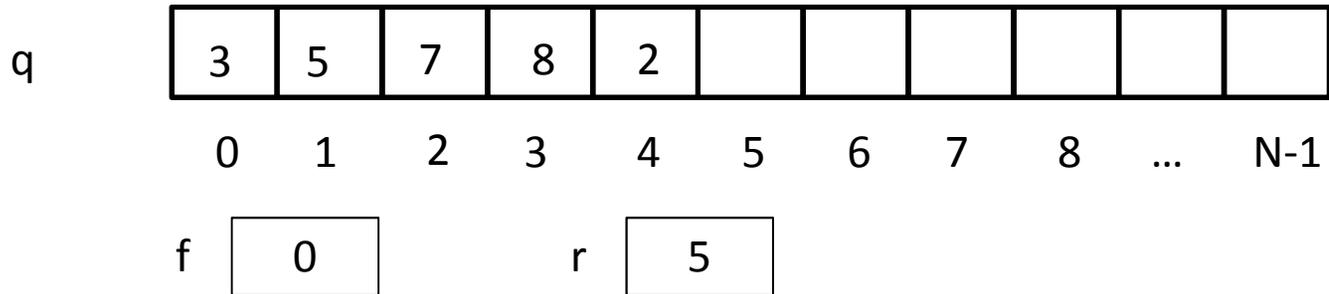
front():

Si isEmpty() entonces
error("cola vacía")

Sino

retornar q[f]

Implementación de cola con un arreglo circular



Constructor:

f = 0

r = 0

isEmpty():

Retornar f == r

front():

Si isEmpty() entonces
error("cola vacía")

Sino
retornar q[f]

dequeue():

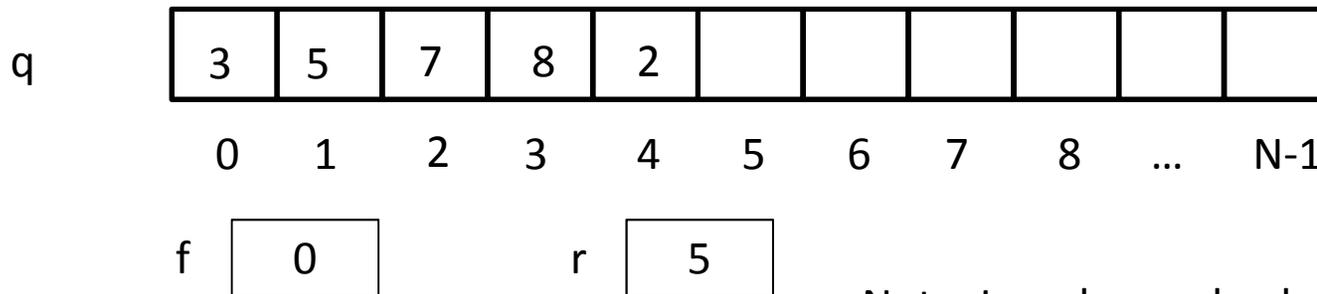
Si isEmpty() entonces error("cola vacía")

Sino

temp = q[f]; q[f] = null;

f = (f+1) mod N; retornar temp

Implementación de cola con un arreglo circular



Size():

Retornar $(N-f+r) \bmod N$

Nota: La cola puede almacenar N-1 elementos a lo sumo

Nota: Otra opción más sencilla es usar un atributo más para representar size.

enqueue(e):

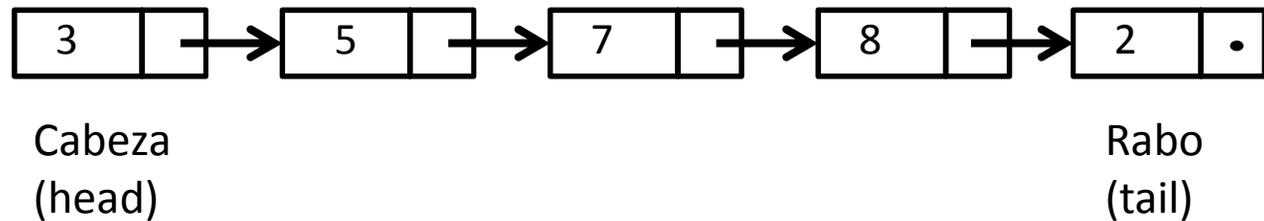
Si $size() == N-1$ entonces
error(“cola llena”)

Sino

$q[r] = e$

$r = (r+1) \bmod N$

Cola: Implementación con nodos enlazados



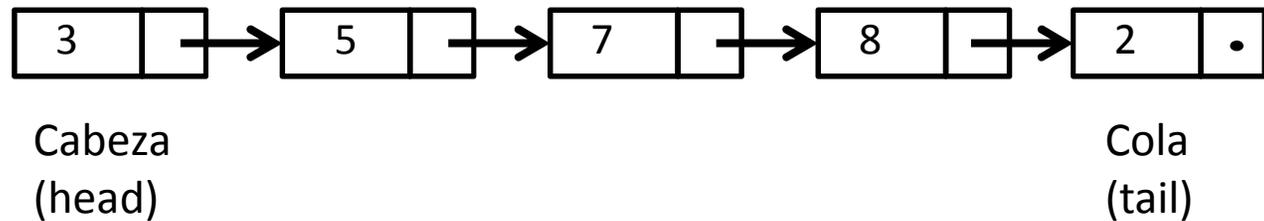
```
public class ColaEnlazada<E>
    implements Queue<E> {
    protected Nodo<E> head, tail;
    protected int tamaño;

    ....
}
```

Implementaremos operaciones en orden 1.

```
public void enqueue( E elem ) {
    Nodo<E> nodo = new Nodo<E>();
    nodo.setElemento( elem );
    nodo.setSiguiete( null );
    if ( tamaño == 0 )
        head = nodo;
    else
        tail.setSiguiete( nodo );
    tail = nodo;
    tamaño++;
}
```

Cola: Implementación con nodos enlazados



```
public class ColaEnlazada<E>
    implements Queue<E> {
    protected Nodo<E> head, tail;
    protected int tamaño;

    ....
}
```

```
public E dequeue() throws
EmptyQueueException {
    if( tamaño == 0 )
        throw new EmptyQueueException( "cola
vacía" );

    E tmp = head.getElemento();
    head = head.getNext();
    tamaño --;
    if( tamaño == 0 ) tail = null;
    return tmp;
}
```

Problema: Insertar los elementos 1, 2, 3, 4 en una cola y luego mostrar todos los elementos de la cola.

```
public class App
{
    public static void main( String [] args ) {
        try {
            Queue<Integer> q = new ColaEnlazada<Integer>();

            for( int i=1; i<=4; i++)
                q.enqueue( i );
            while( !q.isEmpty() )
                System.out.println( q.dequeue() );
        } catch( EmptyQueueException e ) {
            System.out.println( "e: " + e.getMessage() );
            e.printStackTrace();
        }
    }
}
```

Bibliografía

- Goodrich & Tamassia, Data Structures and Algorithms in Java, 4th edition, John Wiley & Sons, 2006, Capítulo 5
- Se puede profundizar la parte de listas enlazadas en la Sección 3.2 del libro