

Estructuras de Datos.

Grado en Ingeniería Informática, Ingeniería del Software e Ingeniería de
Computadores
ETSI Informática
Universidad de Málaga

3. Tipos Abstractos de Datos

José E. Gallardo, Francisco Gutiérrez, Pablo López
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga

Índice

- Tipos Abstractos de Datos. Especificaciones algebraicas
- Stack
 - Signatura y axiomas, descripción con quickCheck
 - Implementación lineal de Stack. (intermedio) Módulos
 - Evaluación de expresiones en polaca inversa via stack
 - Java: Interfaces e Implementaciones
 - Implementaciones de Stack en Java
- Queue
 - Especificación algebraica
 - Implementaciones en Haskell
 - El problema de Josephus
 - Implementaciones en Java
- List
 - Interfaz e implementaciones de List en Java
- Set
 - Especificación algebraica de Set
 - Implementaciones de Set en Haskell
 - Interfaz e implementaciones de Set en Java
- Iteradores/Plegados sobre TADs
 - Iteradores en Java. Modelo y uso
 - Plegado para Set
 - Iteradores para Set

Tipo Abstracto de Datos (TAD)

Interfaz/Signatura y Axiomas

- Objetivos:
 - Introducir la idea de TAD
 - Describir los mecanismos que Haskell y Java proporcionan para definirlos
 - Describir una colección de TADs elementales
- **Abstracción:** un mecanismo para comprender sistemas que involucran gran cantidad de detalles:
 - **destaca** los detalles relevantes,
 - **oculta** los irrelevantes.
- Un TAD es una abstracción:
 - destacamos las operaciones que podemos realizar con los datos,
 - ocultamos (los detalles de) la implementación de éstos.
- Un TAD se integra en el sistema de tipos del lenguaje como un tipo más
 - Mismos privilegios (asignación, paso de parámetros, etc.).
 - Es indistinguible de un tipo primitivo.

Tipo Abstracto de Datos (TAD) Interfaz/Signatura y Axiomas (II)

- Un TAD se describe (**especifica**) mediante:
 - las operaciones y el tipo de cada una (**interfaz/signatura**),
 - algunas propiedades que interrelacionan las operaciones (**axiomas**).
- La signatura permite decidir qué “mezcla” de operaciones tiene sentido.
- Los axiomas permiten operar con los datos en forma abstracta.

signatura + axiomas = especificación

- Un programador que usa un TAD solo conoce su especificación, pero no conoce cómo se “manipula internamente”.
- Solamente puede manipular el TAD en forma abstracta a través de la especificación.
- Estas **barreras de abstracción** son esenciales si en un proyecto trabajan varios programadores: se puede cambiar la implementación sin afectar al trabajo del resto de programadores.

Tipo Abstracto de Datos (TAD) Interfaz/Signatura y Axiomas (III)

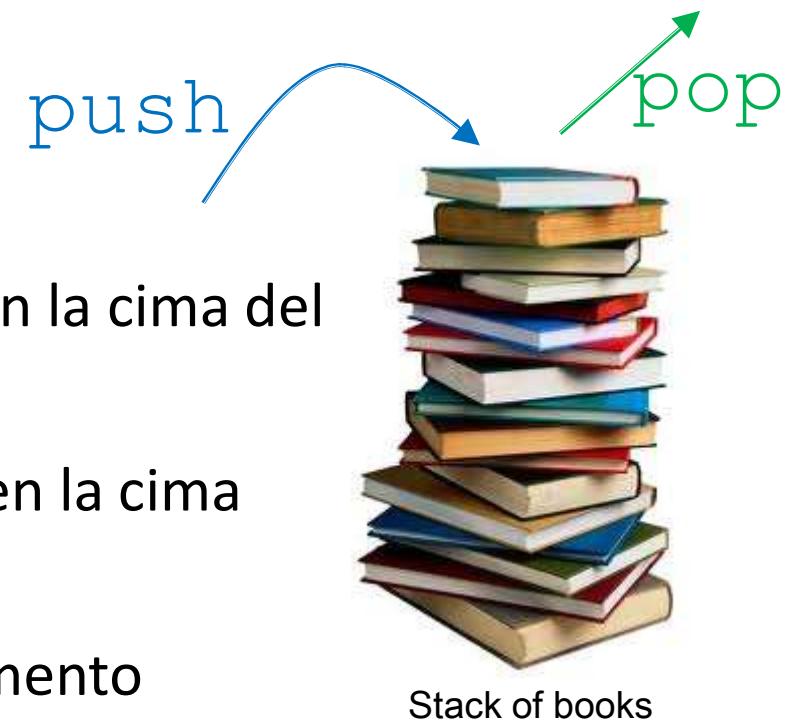
- Un ejemplo de TAD estándar: **Double**.
- El programador no necesita saber cómo se representa internamente un dato **Double**, ni cómo se opera con éstos internamente.
- Puede manipular los valores a través de operaciones, como la suma (+), y algunas constantes: 3.1415

- El programador usuario de **Double** desea una implementación eficiente del tipo y de sus operaciones, pero no está interesado en escribirlas.
- El programador implementador del tipo debe facilitar información sobre la eficiencia (complejidad) de las operaciones, que dependerán de la implementación.

El TAD Stack (pila)

- Un **Stack** es una estructura contenedora que organiza los elementos en una política **LIFO** (**Last In First Out**)

[http://en.wikipedia.org/wiki/Stack_\(data_structure\)](http://en.wikipedia.org/wiki/Stack_(data_structure))



- Operaciones básicas:

- **push**: coloca un nuevo elemento en la cima del Stack
- **pop**: elimina el elemento situado en la cima del Stack
- **top**: devuelve (sin eliminar) el elemento situado en la cima del Stack
- **isEmpty**: comprueba si el Stack no contiene elementos

Interfaz/Signatura del TAD Stack

-- Interfaz en Haskell

Selector

`isEmpty :: Stack a -> Bool`

Transformador

`push :: a -> Stack a -> Stack a`

Selector

`top :: Stack a -> a`

Transformador

`pop :: Stack a -> Stack a`

Constructor

`empty :: Stack a`

Transformador: devuelve
un dato transformado

Constructor: construye un dato

Selector: obtiene
información del dato

Un Ejemplo de Uso del TAD Stack

```
s1 :: Stack Int
s1 = push 3 (push 2 (push 1 empty))
```

```
size :: Stack a -> Int
size s
| isEmpty s = 0
| otherwise = 1 + size (pop s)
```



Descripción de Axiomas con QuickCheck

```
import DataStructures.Stack.LinearStack
import Test.QuickCheck

-- top debe devolver el último elemento apilado
ax1 x s = top (push x s) == x

-- Hacer pop tras push no modifica el stack
ax2 x s = pop (push x s) == s

-- un stack vacío está vacío
ax3      = isEmpty empty

-- push devuelve un stack no vacío
ax4 x s = not (isEmpty (push x s))
```

No es necesario escribir
`True ==>` en las
propiedades si no hay
precondición.
El tipo es `Bool` en vez de
`Property`

Main> quickCheck (ax2 :: Int -> Stack Int -> Bool)
+++ OK, passed 100 tests.

Representación de Stack como Tipo de Dato Recursivo

- El tipo a definir puede aparecer como argumento de alguno de sus constructores de datos.

```
data Stack a = Empty | Node a (Stack a) deriving Show
```

El segundo argumento del constructor `Node` debe ser un Stack

```
s1 :: Stack Int  
s1 = Node 2 (Node 1 Empty)
```

constructores

Este dato también es un Stack

```
top :: Stack a -> a  
top Empty      = error "top for empty stack"
```

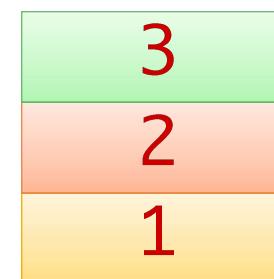
```
top (Node x s) = x
```

patrones: `x :: a`, `s :: Stack a`

```
size :: Stack a -> Int
```

```
size Empty      = 0
```

```
size (Node x s) = 1 + size s
```



(Intermedio) Módulos (I)

- Un módulo Haskell es una colección de declaraciones relacionadas: funciones, tipos, clases,...
- Dividir un programa en varios módulos tiene muchas ventajas:
 - Solo las funciones exportadas por el módulo pueden ser usadas por otro módulo (**bibliotecas**)
 - Los módulos facilitan la organización y el diseño de programas complejos
 - Diferentes programadores pueden trabajar simultáneamente sobre diferentes módulos

Módulos (II)

- En Haskell, cada módulo se implementa en un fichero distinto
- Los nombres de módulos son identificadores que comienzan por mayúsculas
- El nombre del fichero y del módulo deben coincidir (extensión .hs)
- El fichero del módulo debe comenzar con una **cabecera de módulo**:

```
module ModuleName ( Identifier1
                   , Identifier2
                   , ...
                   , Identifiern
                   ) where
```

Ésta es la colección de funciones, tipos, constructores y clases que el módulo proporciona (**exporta**)

```
import ... (justo detrás de la cabecera)
```

Esta es la **interfaz** para este módulo. Otros módulos que lo importen (**clientes**) podrán usar solo lo que aquí se proporciona

Definiciones de tipos, funciones, clases, instancias, ...

Todas las definiciones que no estén en la interfaz, se consideran definiciones **privadas**

Módulos (III)

- Para usar un módulo, un cliente debe importarlo
- La cláusula **import** debe estar al comienzo del módulo cliente, justo delante de su primera definición

```
import ModuleName
```

Permite acceder a los recursos exportados por el módulo ModuleName

```
import ModuleName(Identifier1, Identifier2, ..., Identifiern)
```

Permite acceder a un subconjunto de los recursos exportados por el módulo

```
import qualified ModuleName(Identifier1 ..., Identifiern) as Prefix
```

Los identificadores exportados por el módulo pueden usarse precedidos por **Prefix** y un punto. Esto evita **conflictos** con identificadores con el mismo identificador importados desde otros módulos

Módulos (y IV)

LinearStack.hs

```
module DataStructures.Stack.LinearStack (
    Stack
    , empty
    , ...
) where

data Stack a = ...
empty :: Stack a
empty = ...
...
```

LinearQueue.hs

```
module DataStructures.Queue.LinearQueue (
    Queue
    , empty
    , ...
) where

data Queue a = ...
empty :: Queue a
empty = ...
...
```

Prefijo para identificadores del módulo Stack

Client.hs

```
import qualified DataStructures.Stack.LinearStack as S
import qualified DataStructures.Queue.LinearQueue as Q
```

```
s1 :: S.Stack Int
s1 = S.empty
```

Se refiere a empty del módulo Stack

```
q1 :: Q.Queue Int
q1 = Q.empty
```

Se refiere a empty del módulo Queue

Prefijo para identificadores del módulo Queue

Organización en Módulos o Paquetes

Haskell

DataStructures

Stack

LinearStack

Queue

LinearQueue

TwoStacksQueue

Set

LinearSet

ListSet

...

Java

dataStructures

stack

Stack

LinkedListStack

ArrayStack

LinkedStack

queue

Queue

LinkedListQueue

ArrayQueue

LinkedQueue

set

Set

LinkedListSet

LinkedSet

...

Implementación Lineal de Stack

DataStructures.Stack.LinearStack

```
module DataStructures.Stack.LinearStack (
    Stack,
    empty,
    isEmpty,
    push,
    pop,
    top
) where
```

La implementación del tipo Stack queda oculta fuera del módulo

Estas son la únicas operaciones posibles para los clientes del módulo

```
import Data.List(intersperse)

data Stack a = Empty | Node a (Stack a)

empty :: Stack a
empty = Empty

isEmpty :: Stack a -> Bool
isEmpty Empty = True
isEmpty _ = False

push :: a -> Stack a -> Stack a
push x s = Node x s

top :: Stack a -> a
top Empty = error "top on empty stack"
top (Node x s) = x

pop :: Stack a -> Stack a
pop Empty = error "pop on empty stack"
pop (Node x s) = s
```

Los constructores de datos Empty y Node no se exportan, y por tanto son **privados**

Impl. DataStructures.Stack.LinearStack (II)

Se podría haber generado la igualdad estructural directamente:

```
data Stack a = Empty | Node a (Stack a) deriving Eq
```

```
instance (Eq a) => Eq (Stack a) where
```

```
Empty == Empty = True
```

```
(Node x s) == (Node x' s') = x==x' && s==s'
```

```
- == - = False
```

Determina cómo Haskell mostrará Stacks

```
instance (Show a) => Show (Stack a) where
```

```
show s ="LinearStack(" ++ concat (intersperse "," (stackToList s))  
++ ")"
```

```
stackToList :: (Show a) => Stack a -> [String]
```

```
stackToList Empty = []
```

```
stackToList (Node x s) = show x : stackToList s
```

Efecto de show

```
Main> push 3 (push 2 (push 1 empty))
```

```
LinearStack(3,2,1)
```

Efecto de Eq

```
Main> push 3 (push 2 (push 1 empty)) == push 1 empty
```

```
False
```

Impl. DataStructures.Stack.LinearStack (y IV)

Eficiencia de la implementación:

Operación	Orden	
push	$O(1)$	
pop	$O(1)$	
top	$O(1)$	
isEmpty	$O(1)$	
empty	$O(1)$	

A blue rectangular callout box with a white background and a thin blue border. Inside, the text "Eficiencia Óptima" is written in black, accompanied by a green smiley face emoji.

Un Ejemplo de Uso de Stack

```
import DataStructures.Stack.LinearStack

s1 :: Stack Int
s1 = push 3 (push 2 (push 1 empty))

size :: Stack a -> Int
size s
| isEmpty s = 0
| otherwise = 1 + size (pop s)

s2 :: Stack Int
s2 = push (size s1) s1
```

No podemos usar los constructores de datos de Stack en el código del cliente. Su implementación fue **ocultada (privada)**

Evaluación de una Expresión en Polaca Inversa usando una Pila (I)

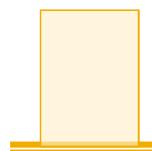
La expresión:

$$5 + (6 - 2) * 3$$

En Polaca Inversa es:

$$5 \ 6 \ 2 \ - \ 3 \ * \ +$$

$$5 \ 6 \ 2 \ - \ 3 \ * \ +$$



$$6 \ 2 \ - \ 3 \ * \ +$$

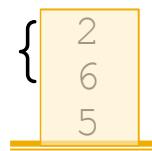


$$2 \ - \ 3 \ * \ +$$

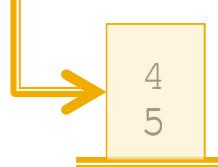


$$- \ 3 \ * \ +$$

(-) {

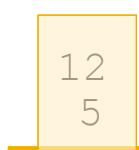


$$3 \ * \ +$$



$$* \ +$$

$$+$$



Un Módulo para Representar Expresiones (I)

```
module Expression (
    Item(..),
    Expression,
    value,
    showExpr,
    sample1, sample2
) where
```

La implementación del tipo
Item se muestra fuera del
módulo

```
data Item = Add | Dif | Mul | Value Integer | LeftP | RightP
          deriving Show
type Expression = [Item]

-- sample1 corresponde con 5 + (6 - 2) * 3
sample1 = [ Value 5, Add, LeftP, Value 6
           , Dif, Value 2, RightP, Mul, Value 3 ]

-- sample2 es la expresión anterior en Polaca Inversa
sample2 = [ Value 5, Value 6, Value 2, Dif, Value 3, Mul, Add ]
```

Un módulo para representar expresiones (y II)

```
value :: Item -> Integer -> Integer -> Integer
value Add x y = x + y
value Dif x y = x - y
value Mul x y = x * y

showExpr :: Expression -> String
showExpr (Value x : ts) = ' ' : show x ++ Expr ts
showExpr (Add : ts)      = '+' : Expr ts
showExpr (Dif : ts)      = '-' : Expr ts

...
```

Evaluación de una Expresión en Polaca Inversa usando una Pila (II)

```
*PostFix> sample2
[Value 5, Value 6, Value 2,Dif, Value 3,Mul,Add]
*PostFix> show' sample2
“ 5 6 2 - 3 * +”
*PostFix> postFixEval sample2
17
```

```
module PostFix(postFixEval) where

import DataStructures.Stack.LinearStack
import Expression

postFixEval :: Expression -> Integer
postFixEval e = eval e empty

eval :: Expression -> Stack Integer -> Integer
eval []           s      = top s
eval (Value x : ts) s      = eval ts (push x s)
eval (op:ts)       v2v1s = eval ts (push (value op v1 v2) s)
  where v2 = top v2v1s -- segundo arg
        v1s = pop v2v1s -- el stack sin segundo arg
        v1  = top v1s   -- primer arg
        s   = pop v1s   -- el stack sin los dos args
```

Java: Interfaces e Implementaciones

- Separamos la interfaz de las implementaciones de las estructuras de datos
 - **Interfaz**: signaturas que describen las operaciones soportadas por la estructura
 - **Implementación**: código correspondiente a una realización de la estructura. Normalmente, un interfaz ofrece diferentes implementaciones (con distintas eficiencias de las operaciones)
 - **Cliente**: un programa que usa la estructura de datos a través de su interfaz
- Ventajas:
 - Proporcionamos una interfaz **abstracta** para manipular la estructura. Los clientes no necesitan conocer los detalles de la implementación
 - Un cliente puede elegir la implementación más apropiada de la estructura de datos para resolver un problema concreto

Interfaz Stack en Java

Aunque el paquete `java.util` ya contiene una implementación de `Stack`, aquí consideraremos otra diferente basado en la siguiente interfaz:

```
package dataStructures.stack;  
  
public interface Stack<T> {  
    void push(T elem);  
    T top();  
    void pop();  
    boolean isEmpty();  
}
```

T denota el tipo base:
tipo de elementos en
la pila

Se trata de una
implementación **mutable**:
algunos métodos
modifican el objeto
receptor

- La operación `empty` (que vimos en Haskell) se corresponde en Java con el constructor de la clase que implementa la interfaz

Implementaciones de Stack en Java

Estudiaremos tres implementaciones:

- Basada en `java.util.LinkedList` de Java:
`LinkedListStack`
- Basada en arrays: `ArrayStack`
- Basada en nodos enlazados: `LinkedStack`

Un ejemplo de uso del TAD Stack

```
package dataStructures.stack;

public class Example {
    static public void main(String [ ] args) {
        Stack<Integer> stack = new ArrayStack<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);
        while (!stack.isEmpty( )) {
            System.out.println(stack.top( ));
            stack.pop( );
        }
    }
}
```

Interfaz

Implementación



Excepciones en Stack

Las operaciones inválidas lanzarán esta excepción:

```
package dataStructures.stack;

public class EmptyStackException extends RuntimeException {

    public EmptyStackException() {
        super();
    }

    public EmptyStackException(String msg) {
        super(msg);
    }
}
```

Como RuntimeException,
pero varía el nombre de la
excepción

Implementación de Stack por Delegación sobre java.util.LinkedList (I)

- La clase `java.util.LinkedList<T>`, perteneciente a la biblioteca de colecciones de Java, dispone de los siguientes métodos útiles para implementar un `Stack<T>`
 - `public boolean isEmpty()`
 - `public void addFirst(T elem)`
 - `public T getFirst()`
 - `public T removeFirst()`
- Todas estas operaciones tienen complejidad $O(1)$

Implementación de Stack por Delegación sobre java.util.LinkedList (II)

```
package dataStructures.stack;

import java.util.*;

public class LinkedListStack<T> implements Stack<T> {
    protected LinkedList<T> elements;

    public LinkedListStack() {
        elements = new LinkedList<>();
    }

    public void push(T elem) {
        elements.addFirst(elem);
    }

    public boolean isEmpty() {
        return elements.isEmpty();
    }

    public T top() {
        if (isEmpty())
            throw new EmptyStackException("top on empty stack");
        return elements.getFirst();
    }

    public void pop() {
        if (isEmpty())
            throw new EmptyStackException("pop on empty stack");
        elements.removeFirst();
    }
}
```

El constructor crea la lista interna

push llama a addFirst de LinkedList

isEmpty llama a isEmpty de LinkedList

top llama a getFirst de LinkedList

pop llama a removeFirst de LinkedList

Implementación de Stack por Delegación sobre java.util.LinkedList (III)

La eficiencia de esta implementación depende de la complejidad de las operaciones de LinkedList usadas:

Operación	Orden
push	O(1)
pop	O(1)
top	O(1)
isEmpty	O(1)
LinkedListStack	O(1)

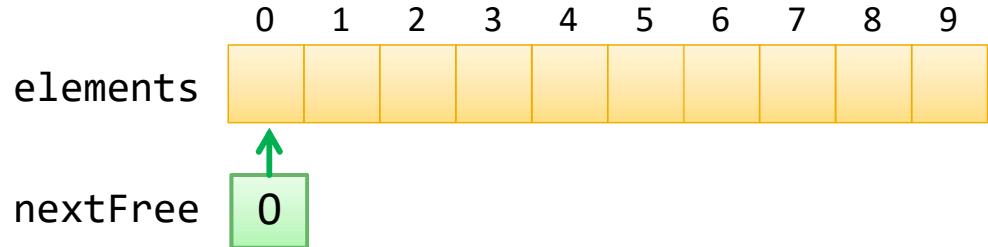
Implementación de Stack vía arrays (I)

Utilizaremos un array para almacenar los elementos del Stack.

Necesitamos:

- Crear el array con un tamaño inicial dado por defecto
 - Usaremos una constante
 - Mantener la siguiente posición libre en el array
 - Redimensionar el array si nos falta espacio
-
- Solución muy compacta: consume poca memoria 😊
 - Si el array se queda pequeño para albergar los datos, se debe expandir (método Arrays .copyOf)

Impl. `dataStructures.stack.ArrayStack` vía arrays (II)

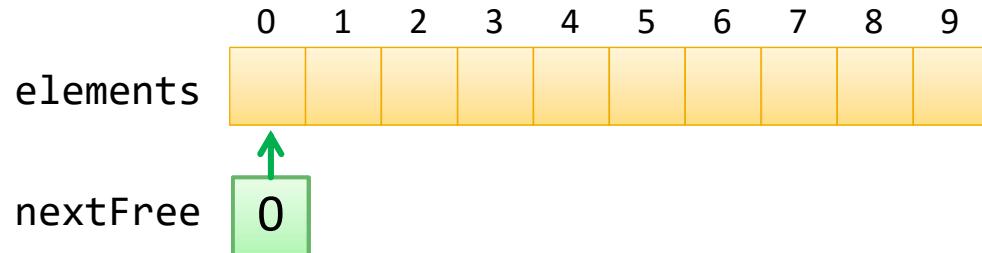


```
public class ArrayStack<T> implements Stack<T> {  
    protected T[] elements;  
    protected int nextFree;  
  
    private final int INITIAL_CAPACITY = 10;  
  
    public ArrayStack() {  
        elements = (T[]) new Object[INITIAL_CAPACITY];  
        nextFree = 0;  
    }  
  
    private void ensureCapacity() {  
        if (nextFree == elements.length)  
            elements = Arrays.copyOf(elements, 2*elements.length);  
    }
```

Si no cabe el nuevo elemento,
se dobla el tamaño del array

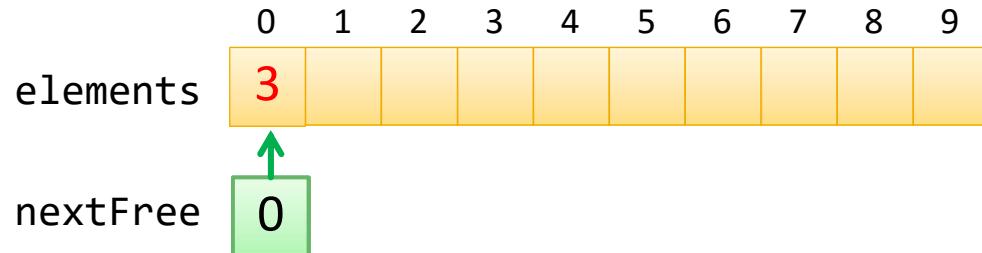
Impl. `dataStructures.stack.ArrayStack` vía arrays (III)

```
public void push(T elem) {  
    ensureCapacity();  
    elements[nextFree] = elem;  
    nextFree++;  
}
```



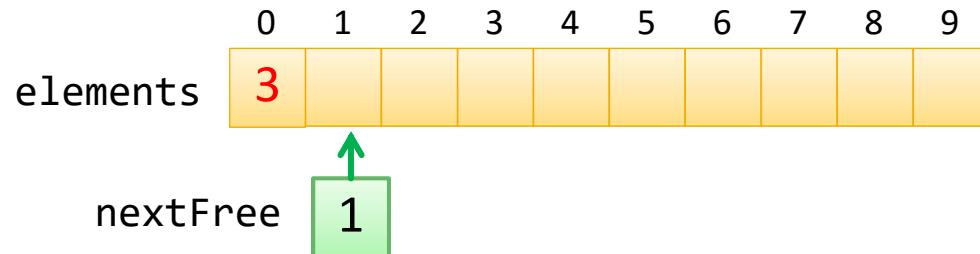
Impl. `dataStructures.stack.ArrayStack` vía arrays (IV)

```
public void push(T elem) {  
    ensureCapacity();  
    elements[nextFree] = elem;  
    nextFree++;  
}
```



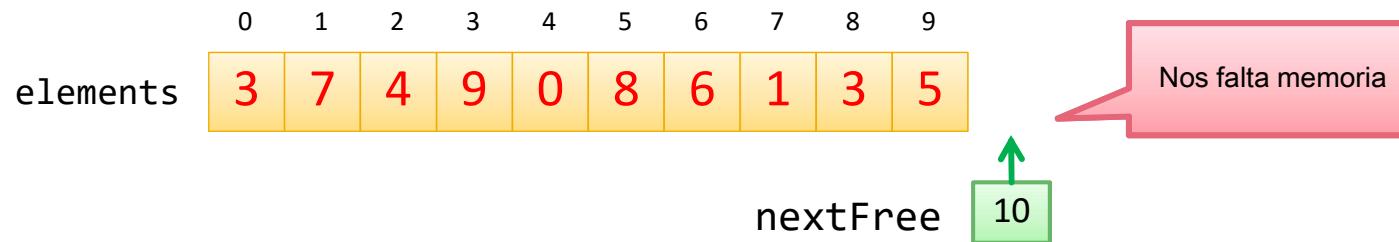
Impl. `dataStructures.stack.ArrayStack` vía arrays (V)

```
public void push(T elem) {  
    ensureCapacity();  
    elements[nextFree] = elem;  
    nextFree++;  
}
```



Impl. `dataStructures.stack.ArrayStack` vía arrays (VI)

```
public void push(T elem) {  
    ensureCapacity();  
    elements[nextFree] = elem;  
    nextFree++;  
}
```



Impl. `dataStructures.stack.ArrayStack` vía arrays (VII)

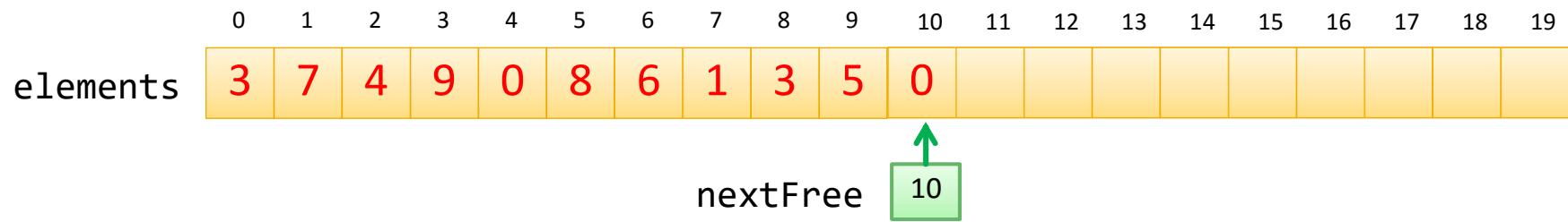
```
public void push(T elem) {  
    ensureCapacity();  
    elements[nextFree] = elem;  
    nextFree++;  
}
```



```
private void ensureCapacity() {  
    if (nextFree == elements.length)  
        elements = Arrays.copyOf(elements, 2*elements.length);  
}
```

Impl. `dataStructures.stack.ArrayStack` vía arrays (VIII)

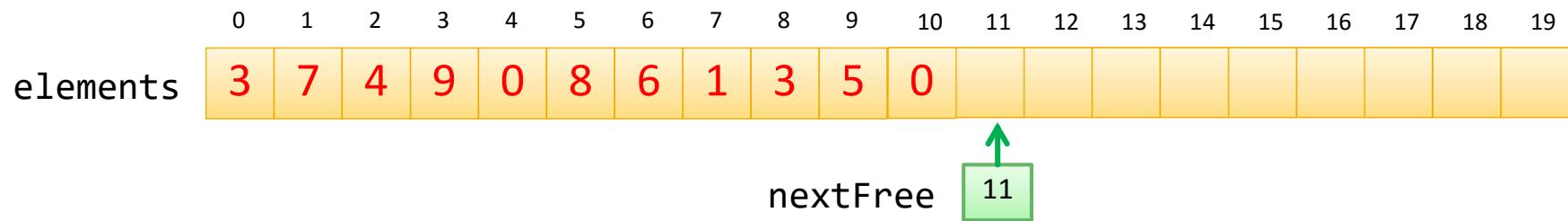
```
public void push(T elem) {  
    ensureCapacity();  
    elements[nextFree] = elem;  
    nextFree++;  
}
```



```
private void ensureCapacity() {  
    if (nextFree == elements.length)  
        elements = Arrays.copyOf(elements, 2*elements.length);  
}
```

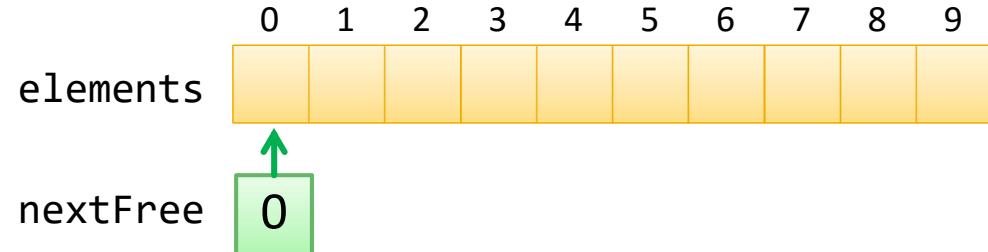
Impl. `dataStructures.stack.ArrayStack` vía arrays (IX)

```
public void push(T elem) {  
    ensureCapacity();  
    elements[nextFree] = elem;  
    nextFree++;  
}
```



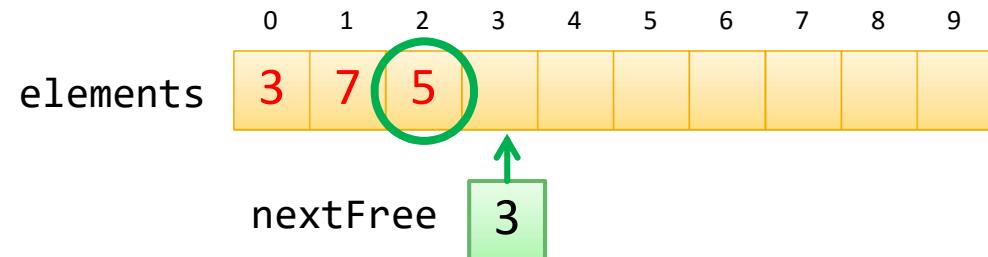
```
private void ensureCapacity() {  
    if (nextFree == elements.length)  
        elements = Arrays.copyOf(elements, 2*elements.length);  
}
```

Impl. `dataStructures.stack.ArrayStack` vía arrays (X)



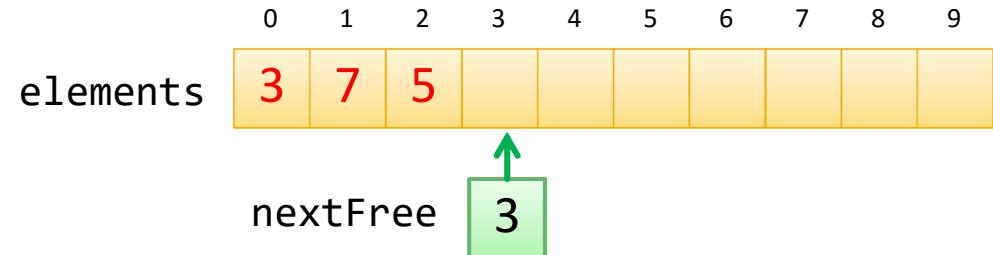
```
public boolean isEmpty() {  
    return nextFree == 0;  
}
```

Impl. `dataStructures.stack.ArrayStack` vía arrays (XI)



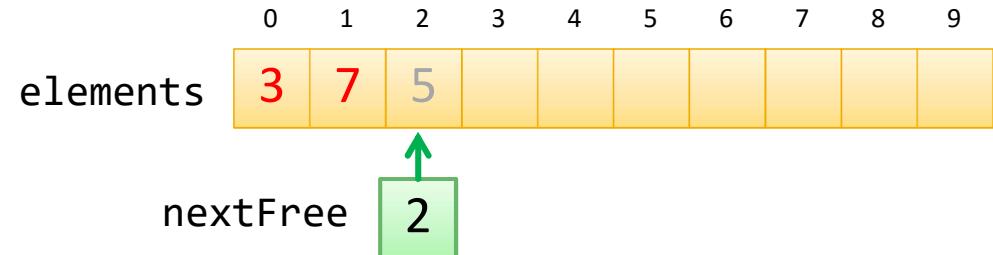
```
public void top() {  
    if(isEmpty())  
        throw new EmptyStackException("top on empty stack");  
    else  
        return elements[nextFree - 1];  
}
```

Impl. `dataStructures.stack.ArrayStack` vía arrays (XII)



```
public void pop() {  
    if(isEmpty())  
        throw new EmptyStackException("pop on empty stack");  
    else  
        nextFree --;  
}
```

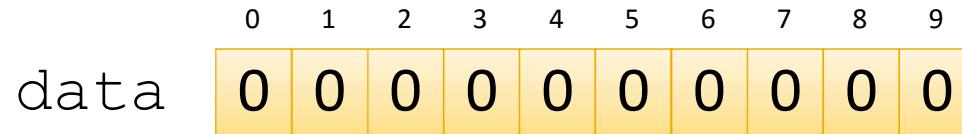
Impl. `dataStructures.stack.ArrayStack` vía arrays (XIII)



```
public void pop() {  
    if(isEmpty())  
        throw new EmptyStackException("pop on empty stack");  
    else  
        nextFree --;  
}
```

Impl. `dataStructures.stack.ArrayStack` vía arrays. Análisis (XIV)

```
int[] data = new int[10];
```



- Crear un array de n elementos toma $O(n)$ pasos en Java debido a la **inicialización** (otros lenguajes toman $O(1)$)
- Acceder a un elemento de un array toma $O(1)$ pasos
`int n = data[2];`
- Modificar un elemento de un array toma $O(1)$ pasos
`data[2] = 8;`
- El acceso a los elementos de un arrays es directo (**random access**) ☺

Impl. `dataStructures.stack.ArrayStack` vía arrays. Análisis (y XV)

Eficiencia de la implementación:

Operación	Orden
push	$O(1)$, $O(n)^*$
pop	$O(1)$
top	$O(1)$
isEmpty	$O(1)$
ArrayStack	$O(1)$

Constructor de la clase (genera un array de tamaño inicial **constante**)

* Si es necesario redimensionar, la operación puede ser $O(n)$

Implementación de Stack vía nodos enlazados (I)

- Un stack vacío:



```
Stack<Integer> s = new LinkedStack<>();
```

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (II)

- Con un elemento:



```
Stack<Integer> s = new LinkedStack<>();
```

```
s.push(1);
```

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (III)

- Un stack con dos elementos:

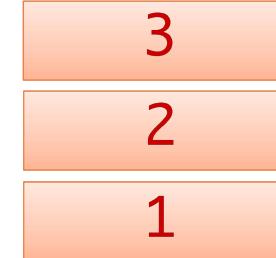
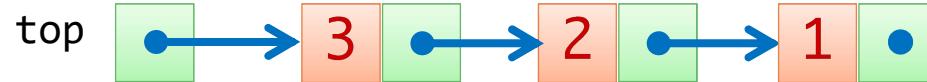


```
Stack<Integer> s = new LinkedStack<>();
```

```
s.push(1);  
s.push(2);
```

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (IV)

- Un stack con tres elementos:



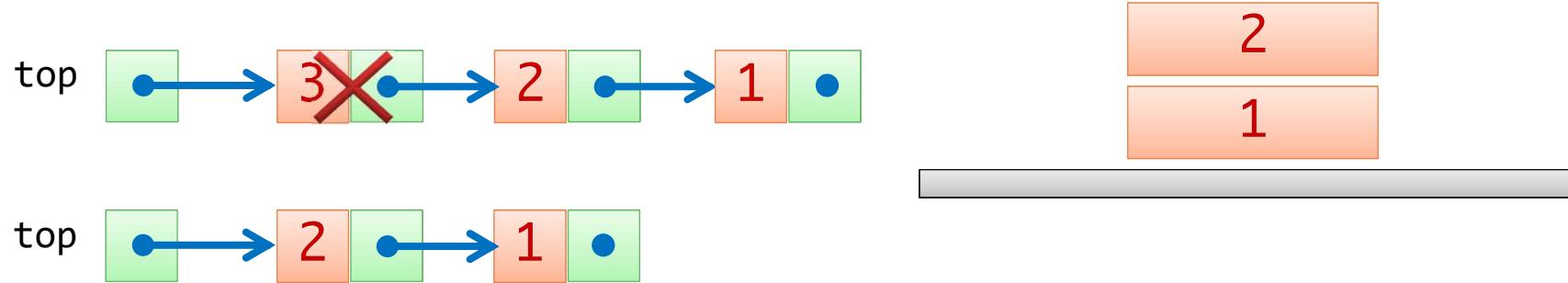
El elemento de la
cima es el de la
izquierda

```
Stack<Integer> s = new LinkedStack<>();
```

```
s.push(1);  
s.push(2);  
s.push(3);
```

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (V)

- Un stack con dos elementos:



```
Stack<Integer> s = new LinkedStack<>();
```

```
s.push(1);  
s.push(2);  
s.push(3);  
s.pop();
```

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (VI)

■ Implementando un stack:

```
public class LinkedStack<T> implements Stack<T> {  
  
    static private class Node<E> {  
        E elem;  
        Node<E> next;  
  
        public Node(E x, Node<E> node) {  
            elem = x;  
            next = node;  
        }  
    }  
  
    private Node<T> top;  
  
    public LinkedStack() {  
        top = null;  
    }  
  
    public boolean isEmpty() {  
        return top == null;  
    }
```

Un nodo:



Enlace al nodo
situado en la cima
del stack



El stack está
vacío inicialmente

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (VII)

- Apilando un elemento en el stack:

`push(3)`

```
public void push(T x) {  
    Node<T> node = new Node<>(x,top);  
    top = node;  
}
```

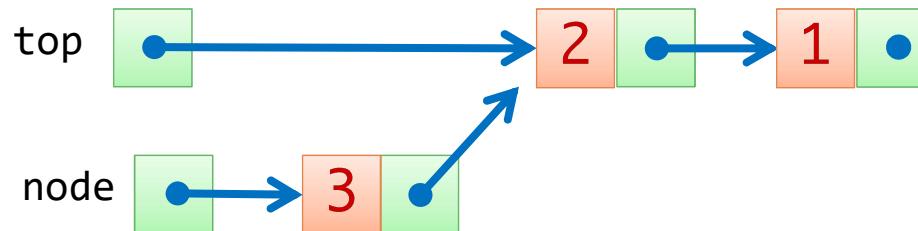


Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (VIII)

- Apilando un elemento en el stack:

`push(3)`

```
public void push(T x) {  
    Node<T> node = new Node<>(x, top);  
    top = node;  
}
```

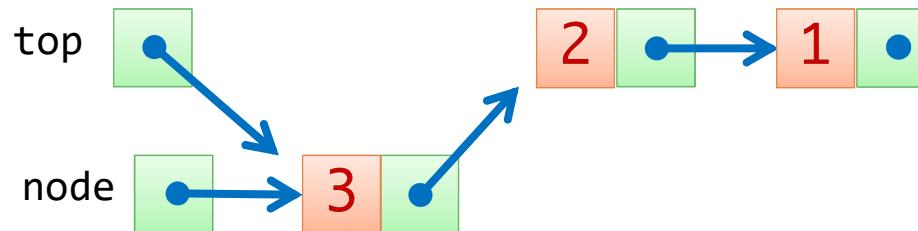


Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (IX)

- Apilando un elemento en el stack:

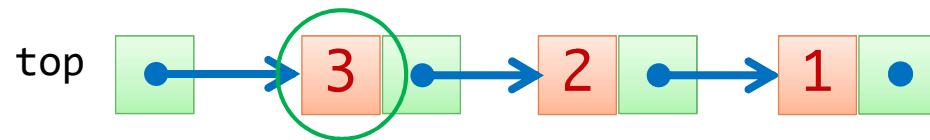
`push(3)`

```
public void push(T x) {  
    Node<T> node = new Node<>(x,top);  
    top = node;  
}
```



Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (X)

- Devolviendo la cima del stack:



```
public T top() {  
    if(isEmpty())  
        throw new EmptyStackException("top on empty stack");  
    else  
        return top.elem;  
}
```

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (XI)

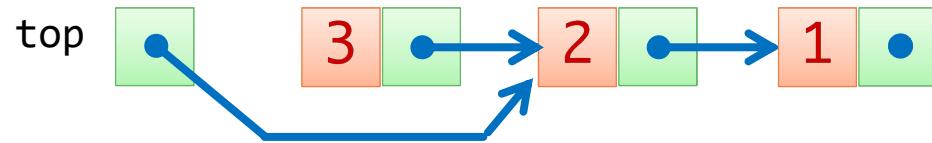
- Eliminando la cima del stack:



```
public void pop() {  
    if(isEmpty())  
        throw new EmptyStackException("pop on empty stack");  
    else  
        top = top.next;  
}
```

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (XII)

Eliminando la cima de stack:



```
public void pop() {  
    if(isEmpty())  
        throw new EmptyStackException("pop on empty stack");  
    else  
        top = top.next;  
}
```

Impl. `dataStructures.stack.LinkedStack` vía nodos enlazados (y XIII)

Eficiencia de la implementación:

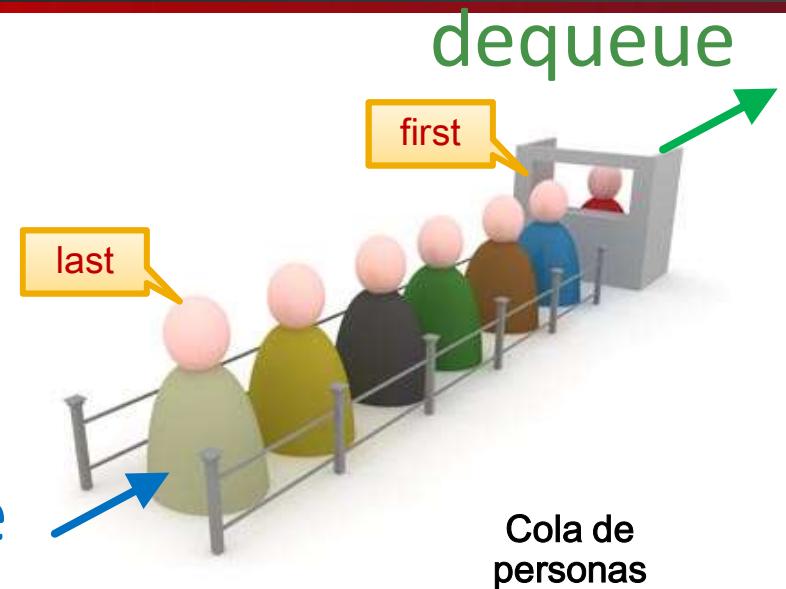
Operación	Orden
push	O(1)
pop	O(1)
top	O(1)
isEmpty	O(1)
LinkedStack	O(1)

LinkedStack vs ArrayStack

- Test experimental
- Hemos medido el tiempo de ejecución para realizar 10 millones de operaciones aleatorias (push o pop) sobre un stack inicialmente vacío.
- Usando una CPU Intel i7 860:
 - ArrayStack fue alrededor de dos veces más rápido que LinkedStack ☺

Tipo Abstracto de Datos Queue

- Una **Queue** (cola) es un TAD que organiza los elementos de tal manera que el primer elemento introducido es el primero en salir (**FIFO**, First In First Out)
- Operaciones básicas:
 - **enqueue**: coloca un nuevo elemento al final de la cola
 - **dequeue**: extrae el elemento que se encuentra al principio de la cola
 - **first**: devuelve el elemento que se encuentra al principio de la cola (sin extraerlo)
 - **isEmpty**: comprueba si la cola está vacía



Signatura del TAD Queue

-- Interfaz en Haskell

isEmpty :: Queue a -> Bool

enqueue :: a -> Queue a -> Queue a

dequeue :: Queue a -> Queue a

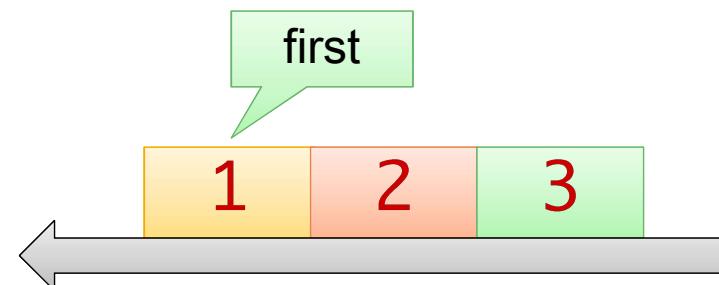
first :: Queue a -> a

empty :: Queue a

Un Ejemplo de Uso de Queue

```
s1 :: Queue Int  
s1 = enqueue 3 (enqueue 2 (enqueue 1 empty))
```

```
size :: Queue a -> Int  
size s  
| isEmpty s = 0  
| otherwise = 1 + size (dequeue s)
```



Especificación de Queue y Pruebas con QuickCheck

```
module DataStructures.Queue.QueueAxioms
  (ax1,ax2,ax3,ax4,ax5,ax6,queueCheckAxioms) where

import DataStructures.Queue.LinearQueue      -- Implementation to use in tests
import Test.QuickCheck

ax1 x = first (enqueue x empty) == x
ax2 x q = not (isEmpty q) ==> first (enqueue x q) == first q
ax3 x q = not (isEmpty q) ==> dequeue (enqueue x q) == enqueue x (dequeue q)
ax4 x =
ax5 =
ax6 x q = isEmpty empty
           not (isEmpty (enqueue x q))
```



```
type Elem = Int -- Tipo a usar en las pruebas

queueCheckAxioms = do
  quickCheck (ax1 :: Elem -> Bool)
  quickCheck (ax2 :: Elem -> Queue Elem -> Property)
  ...
  quickCheck (ax6 :: Elem -> Queue Elem -> Bool)
```

Implementaciones de Queue en Haskell

- Algunas implementaciones posibles son:
 - Lineal: LinearQueue
 - Basada en dos pilas (ejercicio par casa)
 - Basada en dos listas: TwoListsQueue (Complementario)

Implementación Lineal de Queue (I)

```
data Queue a = Empty | Node a (Queue a)
```

- Una cola de 0 elementos:

Empty



- Una cola de 1 elemento:

Node 1 Empty



- Una cola de 2 elementos:

Node 1 (Node 2 Empty)



- Una cola de 3 elementos:

Node 1 (Node 2 (Node 3 Empty))



Impl. DataStructures.Queue.LinearQueue (II)

```
module DataStructures.Queue.LinearQueue
```

```
( Queue  
, empty  
, isEmpty  
, enqueue  
, dequeue  
, first  
) where
```

Implementation of
data type is **private**

```
import Data.List(intercalate)
```

```
data Queue a = Empty | Node a (Queue a)
```

```
empty :: Queue a  
empty = Empty
```

```
isEmpty :: Queue a -> Bool  
isEmpty Empty = True  
isEmpty _       = False
```

Impl. DataStructures.Queue.LinearQueue (III)

```
data Queue a = Empty | Node a (Queue a)
```

enqueue :: a -> Queue a -> Queue a
enqueue x Empty = Node x Empty
enqueue x (Node y q) = Node y (enqueue x q)

Operación O(n)



first :: Queue a -> a
first Empty = error "first on empty queue"
first (Node x q) = x

dequeue :: Queue a -> Queue a
dequeue Empty = error "dequeue on empty queue"
dequeue (Node x q) = q

Impl. DataStructures.Queue.LinearQueue (y IV)

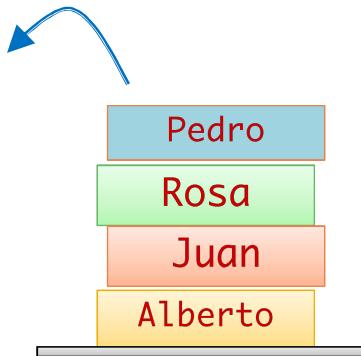
Eficiencia de la implementación:

Operación	Orden
enqueue	$O(n)$
dequeue	$O(1)$
first	$O(1)$
isEmpty	$O(1)$
empty	$O(1)$

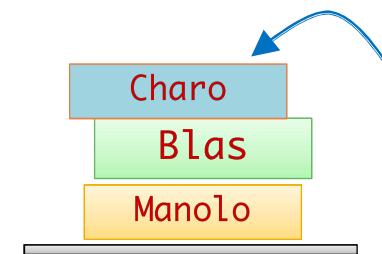
Veremos que puede
mejorarse con otra
implementación

Implementación de Queue con dos Stacks (I)

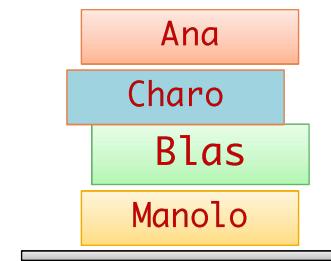
El **Juez** extrae expedientes de la cima de su pila:



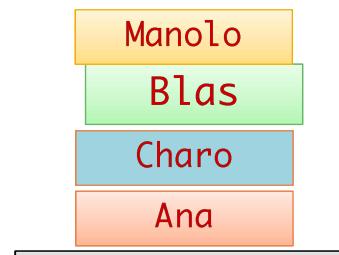
La **Secretaria** añade expedientes en la cima de su pila:



Si la pila del **Juez** queda vacía ...



.... la Secretaria le pasa su pila, pero invertida; ahora el **Juez** tiene una nueva pila, y la de la Secretaria queda vacía:



Implementación de Queue con dos Stacks (y II)

Eficiencia de la implementación:

Ejercicio para casa:
realiza la
implementación

Operación	Orden
enqueue	O(1)
dequeue	O(1), O(n) *
first	O(1)
isEmpty	O(1)
empty	O(1)

* El **coste amortizado** es O(1): sólamente invertimos una pila de n elementos después de encolar n elementos

El Problema de Josephus

En época de superpoblación mundial se decide reducir la población.

De cada M personas, deben morir todas menos una.

La elección de los que mueren se hace mediante un juego

Se colocan las M personas en un círculo

Se elige un número N y se procede desde el principio:

Se saltan N posiciones y se elimina la persona que ocupa ese lugar.

Se sigue así hasta que quede una persona

Por ejemplo con M = 7 y N = 2

0 1 2 3 4 5 6	inicio
3 4 5 6 0 1	Se elimina el 2
6 0 1 3 4	Se elimina el 5
3 4 6 0	Se elimina el 1
0 3 4	Se elimina el 6
0 3	Se elimina el 4
3	Se elimina el 0

Queda viva la persona que ocupa la posición número 3

La lista de eliminaciones resultante es [2,5,1,6,4,0,3]

Un programa enseñaría a Josephus dónde colocarse para salvarse siempre

El Problema de Josephus en Haskell

```
module Josephus where
import DataStructures.Queue.LinearQueue

createQueue :: Int -> Queue Int
createQueue m = foldr enqueue empty [m-1,m-2..0]

skip :: Int -> Queue Int -> Queue Int
skip 0 q = q
skip n q = skip (n-1) (enqueue x q')
  where x = first q
        q' = dequeue q

josephus :: Int -> Int -> [Int]
josephus m n = josephus' n (createQueue m)

josephus' :: Int -> Queue Int -> [Int]
josephus' n q
| isEmpty q = []
| otherwise = x : josephus' n q''
  where q' = skip n q
        x = first q'
        q'' = dequeue q'
```

La lista de los sacrificados y en ese orden. El último es el superviviente

Interfaz Queue en Java

Aunque Java ya dispone de una interfaz Queue en `java.util`, vamos a definir nuestra propia interfaz:

```
package dataStructures.queue;  
  
public interface Queue<T> {  
    void enqueue(T elem);  
    void dequeue();  
    T first();  
    boolean isEmpty();  
}
```

La operación `empty` se implementa con en el constructor en Java

Implementaciones del TAD Queue en Java

Presentaremos tres implementaciones:

1. Basada en la `java.util.LinkedList`: `LinkedListQueue`
2. Basada en nodos enlazados: `LinkedQueue`
3. Basada en array circular: `ArrayQueue`

Excepciones para Queue

Las operaciones inválidas lanzarán esta excepción:

```
package dataStructures.queue;

public class EmptyQueueException extends RuntimeException {

    public EmptyQueueException() {
        super();
    }

    public EmptyQueueException(String msg) {
        super(msg);
    }
}
```

Similar a RuntimeException,
pero varía el nombre de la
excepción

Implementación de Queue por Delegación sobre java.util.LinkedList (I)

La clase `java.util.LinkedList<T>`, perteneciente a la biblioteca de colecciones de Java, dispone de los siguientes métodos útiles para implementar una `Queue<T>`

- `public boolean isEmpty()`
- `public void addLast(T elem)`
- `public T getFirst()`
- `public T removeFirst()`
- Todas estas operaciones tienen complejidad $O(1)$

Implementación de Queue por Delegación sobre java.util.LinkedList (II)

```
package dataStructures.queue;

import java.util.*;

public class LinkedListQueue<T> implements Queue<T> {
    protected LinkedList<T> elements;

    public LinkedListQueue() {
        elements = new LinkedList<>();
    }

    public void enqueue(T elem) {
        elements.addLast(elem);
    }

    public boolean isEmpty() {
        return elements.isEmpty();
    }

    public T first() {
        if (isEmpty())
            throw new EmptyQueueException("first on empty queue");
        return elements.getFirst();
    }

    public void dequeue() {
        if (isEmpty())
            throw new EmptyQueueException("dequeue on empty queue");
        elements.removeFirst();
    }
}
```

El constructor crea la lista interna

enqueue llama a addLast de LinkedList

isEmpty llama a isEmpty de LinkedList

first llama a getFirst de LinkedList

dequeue llama a removeFirst de LinkedList

Implementación de Queue por Delegación sobre java.util.LinkedList (y III)

La eficiencia de esta implementación depende de la complejidad de las operaciones de LinkedList usadas:

Operación	Orden
push	O(1)
pop	O(1)
top	O(1)
isEmpty	O(1)
LinkedListQueue	O(1)

Implementación de Queue vía Nodos Enlazados (I)

- La Queue mantiene información:
 - del nodo que está a la cabeza
 - (mediante la variable `first`)
 - y del último nodo introducido
 - (mediante la variable `last`)
- Cada nodo se enlaza con el siguiente

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (II)

- Una queue vacía:

first 

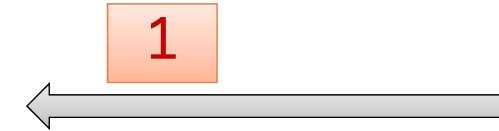
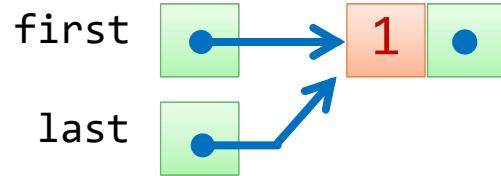


last 

```
Queue<Integer> q = new LinkedQueue<>();
```

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (III)

- Una queue con un elemento:

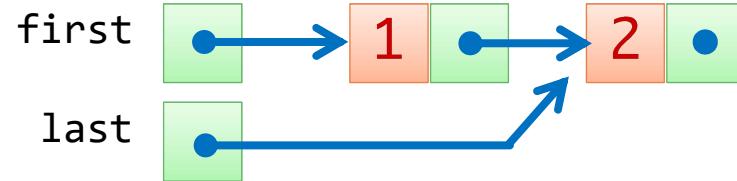


```
Queue<Integer> q = new LinkedQueue<>();
```

```
q.enqueue(1);
```

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (IV)

- Una queue con dos elementos:

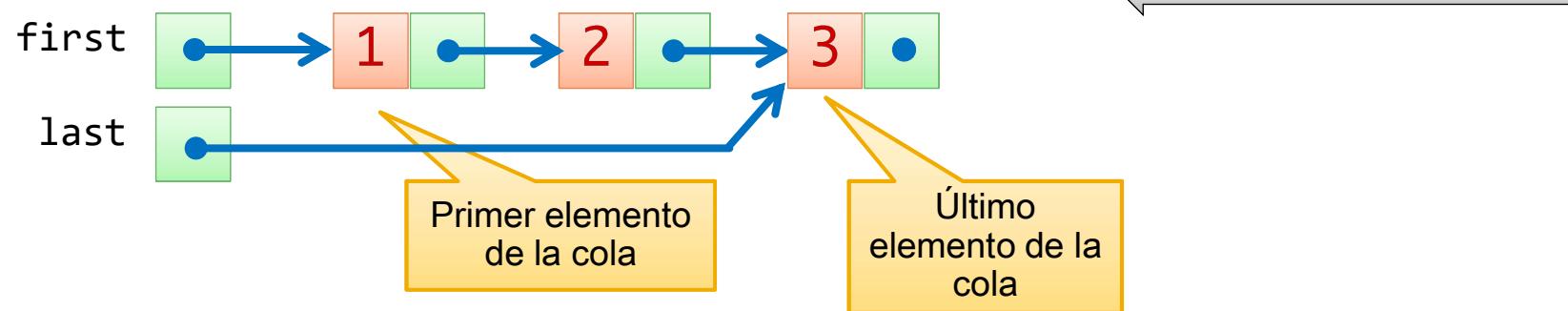


```
Queue<Integer> q = new LinkedQueue<>();
```

```
q.enqueue(1);  
q.enqueue(2);
```

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (V)

- Una queue con tres elementos:



```
Queue<Integer> q = new LinkedQueue<>();
```

```
q.enqueue(1);  
q.enqueue(2);  
q.enqueue(3);
```

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (VI)

■ Implementando una queue:

```
public class LinkedQueue<T> implements Queue<T> {  
  
    private static class Node<E> {  
        private E elem;  
        private Node<E> next;  
  
        public Node(E x, Node<E> node) {  
            elem = x;  
            next = node;  
        }  
    }  
  
    private Node<T> first, last;  
  
    public LinkedQueue() {  
        first = null;  
        last = null;  
    }  
  
    public boolean isEmpty() {  
        return first == null;  
    }
```

Un nodo:



Enlaces al primer y
último nodo de la
queue



La queue está
vacía inicialmente

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (VII)

- Insertando un elemento en una queue vacía:

```
public void enqueue(T x) {  
    Node<T> node = new Node<>(x,null);  
    if(first==null) { //queue was empty  
        first = node;  
        last = node;  
    } else {  
        last.next = node;  
        last = last.next;  
    }  
}
```

enqueue(1)

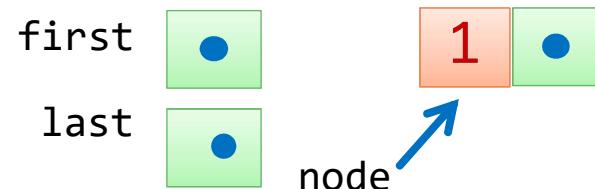


Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (VII)

- Insertando un elemento en una queue vacía:

```
public void enqueue(T x) {  
    Node<T> node = new Node<>(x,null);  
    if(first==null) { //queue was empty  
        first = node;  
        last = node;  
    } else {  
        last.next = node;  
        last = last.next;  
    }  
}
```

enqueue(1)

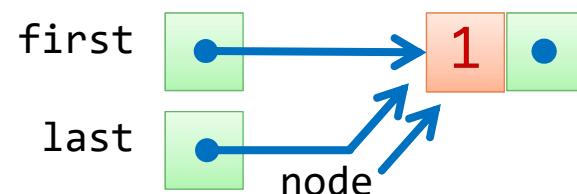


Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (IX)

- Insertando un elemento en una queue vacía:

```
public void enqueue(T x) {  
    Node<T> node = new Node<>(x,null);  
    if(first==null) { //queue was empty  
        first = node;  
        last = node;  
    } else {  
        last.next = node;  
        last = last.next;  
    }  
}
```

enqueue(1)



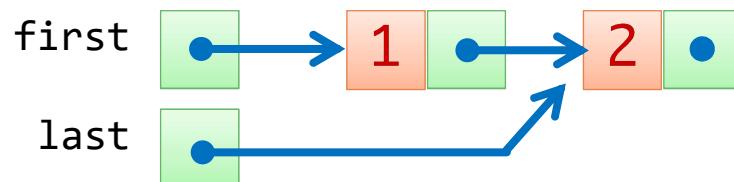
Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (X)

- Insertando un elemento en una queue no vacía:

```
public void enqueue(T x) {  
    Node<T> node = new Node<>(x,null);  
    if(first==null) { //queue was empty  
        first = node;  
        last = node;  
    } else {  
        last.next = node;  
        last = last.next;  
    }  
}
```

enqueue(3)

Supongamos que partimos de la siguiente situación:



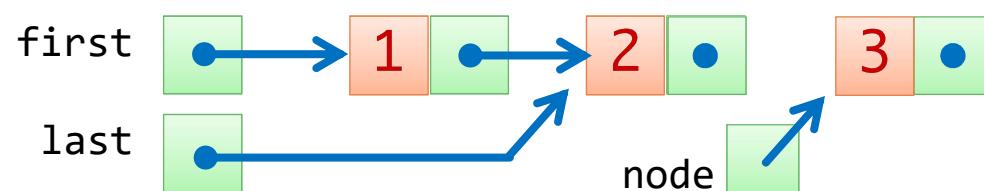
y ejecutamos enqueue(3)

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (XI)

- Insertando un elemento en una queue no vacía:

```
public void enqueue(T x) {  
    Node<T> node = new Node<>(x,null);  
    if(first==null) { //queue was empty  
        first = node;  
        last = node;  
    } else {  
        last.next = node;  
        last = last.next;  
    }  
}
```

enqueue(3)

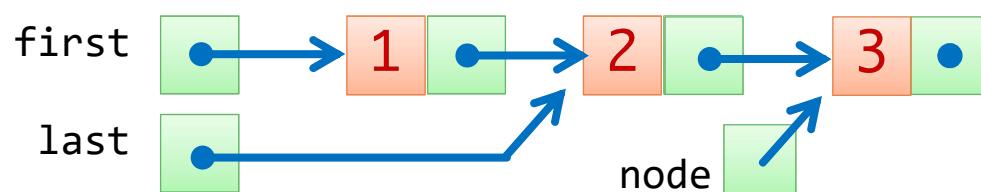


Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (XII)

- Insertando un elemento en una queue no vacía:

```
public void enqueue(T x) {  
    Node<T> node = new Node<>(x,null);  
    if(first==null) { //queue was empty  
        first = node;  
        last = node;  
    } else {  
        last.next = node;  
        last = last.next;  
    }  
}
```

enqueue(3)

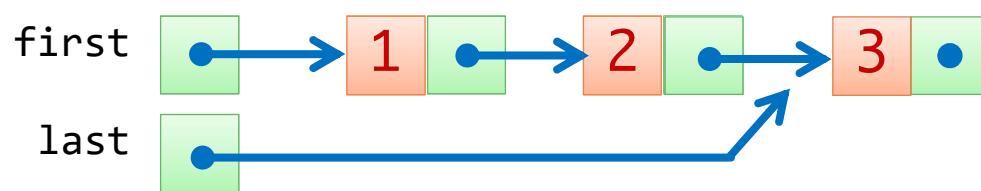


Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (XIII)

- Insertando un elemento en una queue no vacía:

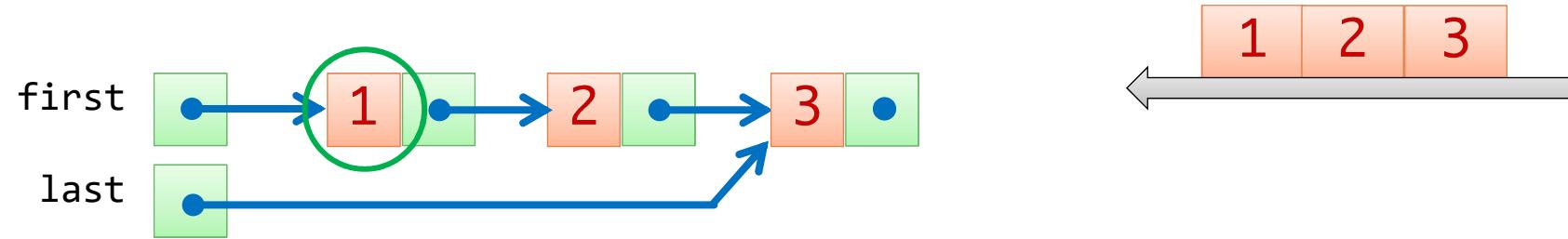
```
public void enqueue(T x) {  
    Node<T> node = new Node<>(x,null);  
    if(first==null) { //queue was empty  
        first = node;  
        last = node;  
    } else {  
        last.next = node;  
        last = last.next;  
    }  
}
```

enqueue(3)



Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (XIV)

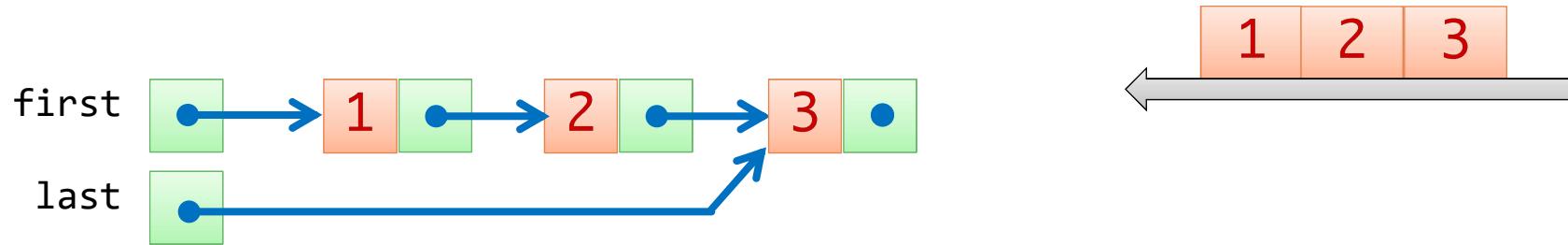
- Devolviendo el primer elemento de una queue:



```
public T first() {  
    if(isEmpty())  
        throw new EmptyQueueException("first on empty queue");  
    else  
        return first.elem;  
}
```

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (XV)

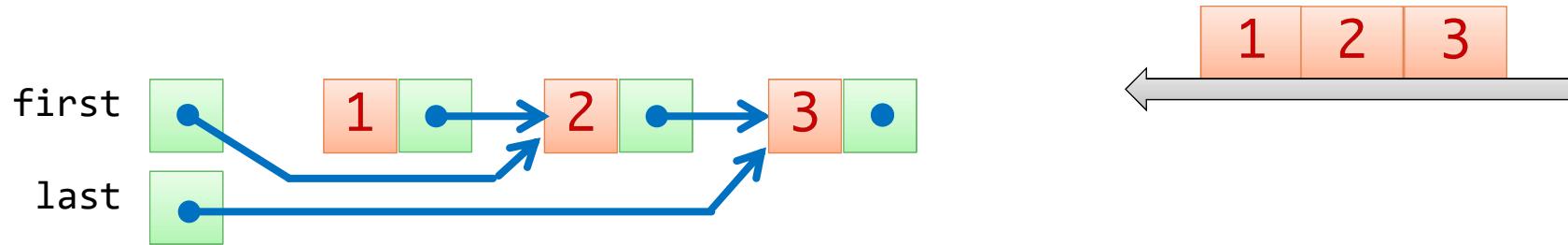
- Eliminando el primer elemento de una queue:



```
public void dequeue() {  
    if(isEmpty())  
        throw new EmptyQueueException("dequeue on empty queue");  
    else {  
        first = first.next;  
        if(first==null) //queue became empty  
            last = null;  
    }  
}
```

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (XVI)

- Eliminando el primer elemento de una queue:



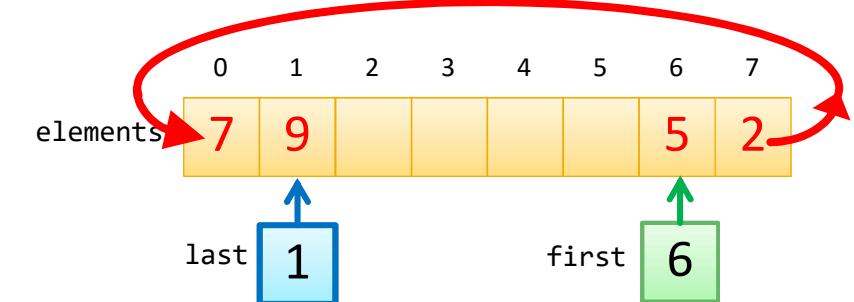
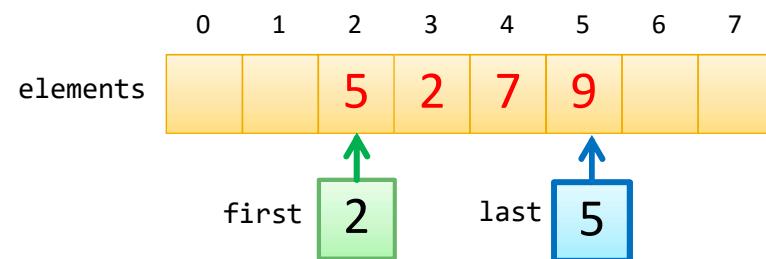
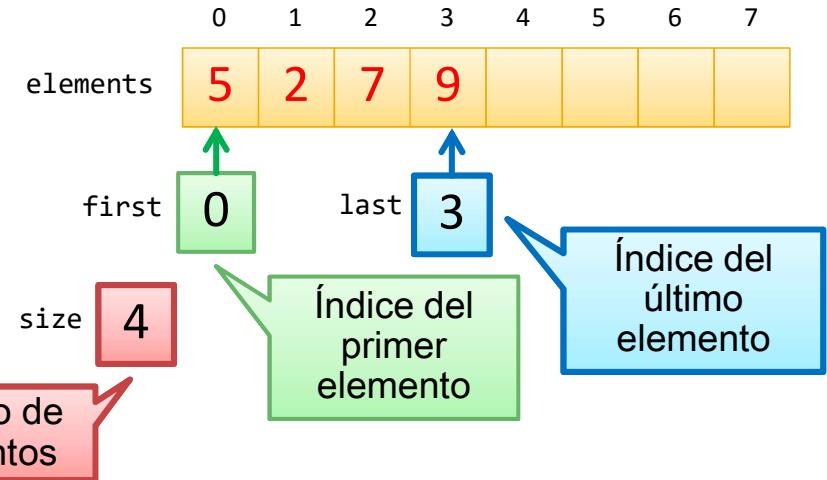
```
public void dequeue() {
    if(isEmpty())
        throw new EmptyQueueException("dequeue on empty queue");
    else {
        first = first.next;
        if(first==null) //queue became empty
            last = null;
    }
}
```

Impl. `dataStructures.queue.LinkedQueue` vía Nodos Enlazados (y XVII)

Eficiencia de la implementación:

Operación	Orden
enqueue	O(1)
dequeue	O(1)
first	O(1)
isEmpty	O(1)
LinkedQueue	O(1)

Implementación de Queue vía un Array Circular (I)

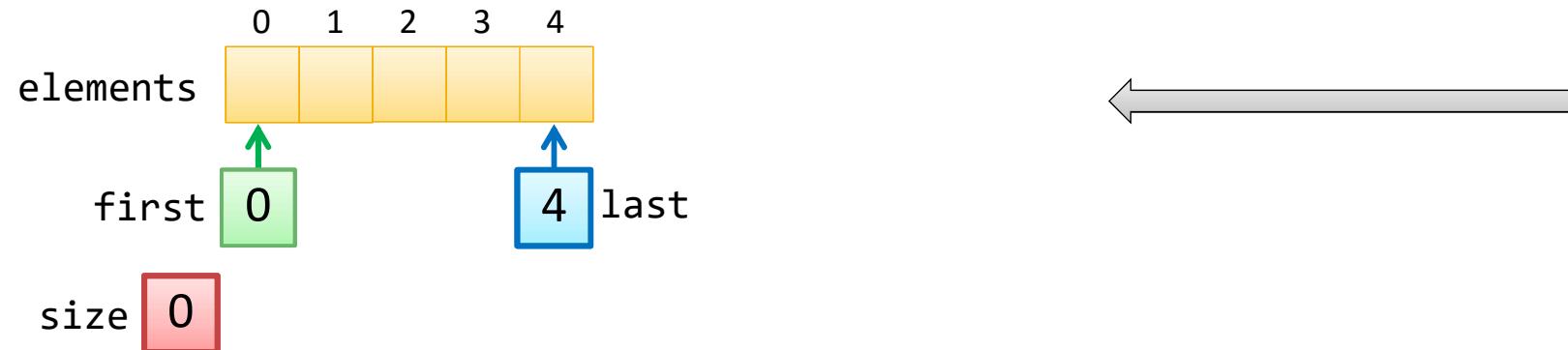


size 4

Al array es utilizado de forma circular

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (II)

- Una queue vacía:



```
Queue<Integer> q = new ArrayQueue<>();
```

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (III)

- Una queue con un elemento:



```
Queue<Integer> q = new ArrayQueue<>();  
q.enqueue(5);
```

enqueue incrementa (de forma circular) last y coloca el nuevo elemento en la casilla indexada por el nuevo valor de last

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (IV)

- Una queue con dos elementos:



```
Queue<Integer> q = new ArrayQueue<>();  
q.enqueue(5);  
q.enqueue(2);
```

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (V)

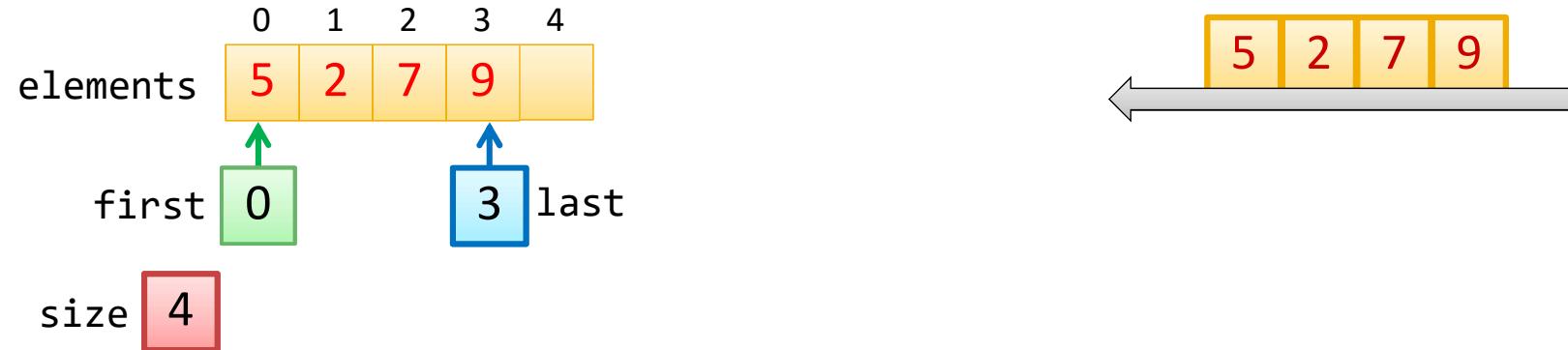
- Una queue con tres elementos:



```
Queue<Integer> q = new ArrayQueue<>();  
q.enqueue(5);  
q.enqueue(2);  
q.enqueue(7);
```

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (VI)

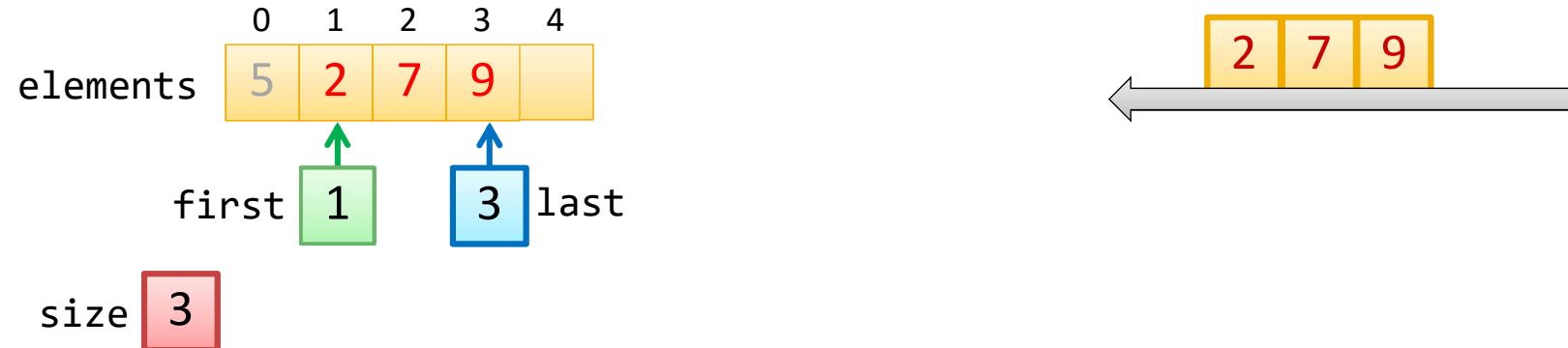
- Una queue con cuatro elementos:



```
Queue<Integer> q = new ArrayQueue<>();  
q.enqueue(5);  
q.enqueue(2);  
q.enqueue(7);  
q.enqueue(9);
```

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (VII)

- Una queue con tres elementos:

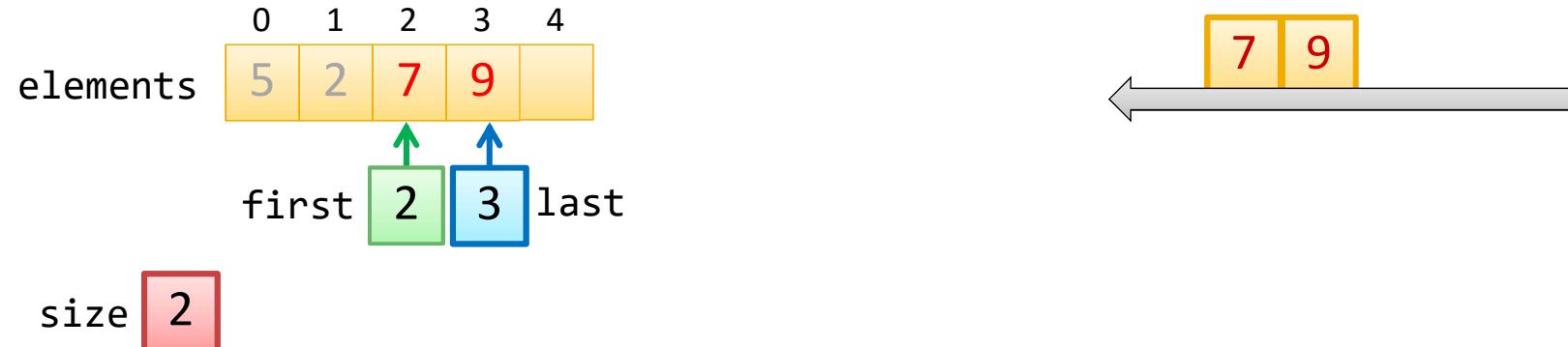


```
Queue<Integer> q = new ArrayQueue<>();  
q.enqueue(5);  
q.enqueue(2);  
q.enqueue(7);  
q.enqueue(9);  
q.dequeue();
```

dequeue incrementa (de forma circular) first

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (VIII)

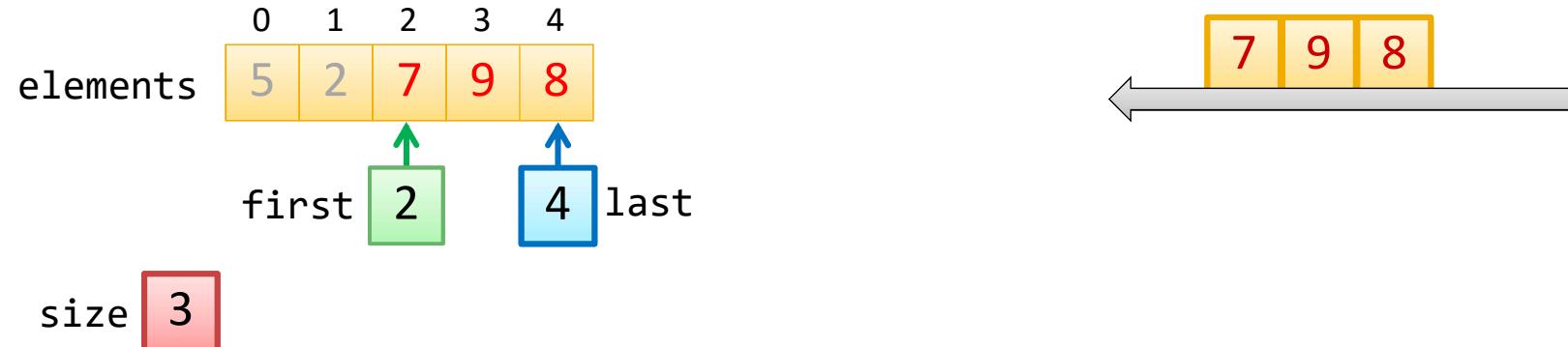
- Una queue con dos elementos:



```
Queue<Integer> q = new ArrayQueue<>();  
q.enqueue(5);  
q.enqueue(2);  
q.enqueue(7);  
q.enqueue(9);  
q.dequeue();  
q.dequeue();
```

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (IX)

- Una queue con tres elementos:



```
Queue<Integer> q = new ArrayQueue<>();  
q.enqueue(5);  
q.enqueue(2);  
q.enqueue(7);  
q.enqueue(9);  
q.dequeue();  
q.dequeue();  
q.enqueue(8);
```

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (X)

- Una queue con cuatro elementos:



```
Queue<Integer> q = new ArrayQueue<>();  
q.enqueue(5);  
q.enqueue(2);  
q.enqueue(7);  
q.enqueue(9);  
q.dequeue();  
q.dequeue();  
q.enqueue(8);  
q.enqueue(6);
```

Impl. dataStructures.queue.ArrayQueue vía un Array Circular (XI)

```
package dataStructures.queue;
public class ArrayQueue<T> implements Queue<T> {
    protected T[] elements;
    protected int first, last, size;

    private final int INITIAL_CAPACITY = 10;

    public ArrayQueue() {
        elements = (T[]) new Object[INITIAL_CAPACITY];
        size = 0;
        first = 0;
        last = elements.length - 1;
    }

    private int advance(int i) {
        return (i+1) % elements.length;
    }

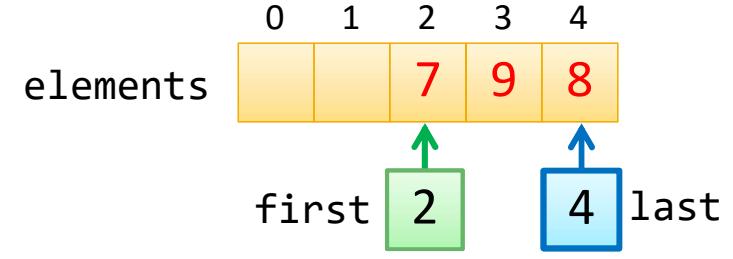
    private void ensureCapacity() {
        if (size == elements.length) {
            T[] extension = (T[]) new Object[2 * elements.length];
            for(int i=0; i<size; i++) {
                extension[i] = elements[first];
                first = advance(first);
            }
            elements = extension;
            first = 0;
            last = size - 1;
        }
    }
}
```

Implementa el incremento circular:
`advance(elements.length-1) => 0`

Si se llena el array, creamos otro de capacidad el doble y lo copiamos

Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (XII)

```
public boolean isEmpty() {  
    return size == 0;  
}  
  
public void enqueue(T x) {  
    ensureCapacity();  
    last = advance(last);  
    elements[last] = x;  
    size++;  
}  
  
public T first() {  
    if (isEmpty())  
        throw new EmptyQueueException("first on empty queue");  
    else  
        return elements[first];  
}  
  
public void dequeue() {  
    if (isEmpty())  
        throw new EmptyQueueException("dequeue on empty queue");  
    else {  
        first = advance(first);  
        size--;  
    }  
}
```



Impl. `dataStructures.queue.ArrayQueue` vía un Array Circular (y XIII)

Eficiencia de la implementación:

Operación	Orden
enqueue	$O(n)$, $O(1)$
dequeue	$O(1)$
first	$O(1)$
isEmpty	$O(1)$
ArrayQueue	$O(1)$

$O(n)$ cuando se expande el array, y $O(1)$ en el resto de los casos



Constructor de la clase (genera un array de tamaño inicial **constante**)

Queue con Nodos Enlazados vs Queue con Arrays

- Prueba experimental
- Hemos realizado 10 millones de operaciones aleatorias(enqueue o dequeue) sobre una cola inicialmente vacía.
- Usando una CPU Intel i7 860:
 - La Queue sobre arrays fue 1.6 más rápida que la que usa nodos enlazados ☺

Listas

[http://en.wikipedia.org/wiki/List_\(computing\)](http://en.wikipedia.org/wiki/List_(computing))

- Una lista es una colección homogénea de valores
- Cada valor tiene asociada una posición ([index](#)), usualmente un número natural (0 para el primer elemento, 1 para el segundo, etc.)
- Un mismo valor puede aparecer varias veces en distintas posiciones
- En Haskell las listas están predefinidas. Ver la documentación del módulo [Data.List](#) en



<http://haskell.org/ghc/docs/latest/html/libraries/base-4.4.0.0/Data-List.html>

Las listas en Haskell

- De forma algebraica sería muy sencillo definirlas:

```
data List a = Nil | Cons a (List a)
```

```
lista :: List Int
```

```
lista = Cons 1 (Cons 2 (Cons 1 Nil))
```

Listas en Java

```
package dataStructures.list;

public interface List<T> {
    boolean isEmpty();
    int size();
    T get(int i);
    void set(int i, T elem);
    void insert(int i, T elem);
    void remove(int i);
}
```

Excepciones para List

Las operaciones inválidas lanzarán esta excepción:

```
package dataStructures.list;

public class ListException implements RuntimeException {

    public ListException() {
        super();
    }

    public ListException(String msg) {
        super(msg);
    }
}
```

Similar a RuntimeException,
pero varía el nombre de la
excepción

Implementaciones del TAD List en Java

Estudiaremos dos implementaciones:

1. Basada en arrays: ArrayList
2. Basada en nodos enlazados: LinkedList

Implementación de List vía Arrays (I)

- Una List puede ser implementada manteniendo sus elementos en un array

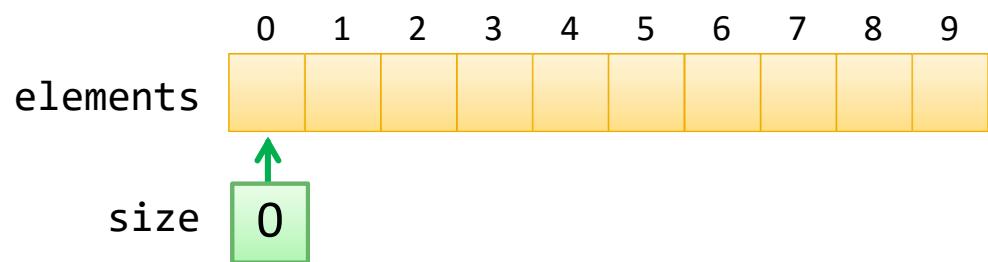
```
package dataStructures.list;
public class ArrayList<T> implements List<T> {
    protected T[] elements;
    protected int size;

    private final int INITIAL_CAPACITY = 10;

    public ArrayList() {
        elements = (T[]) new Object[INITIAL_CAPACITY];
        size = 0;
    }

    public int size() {
        return size;
    }

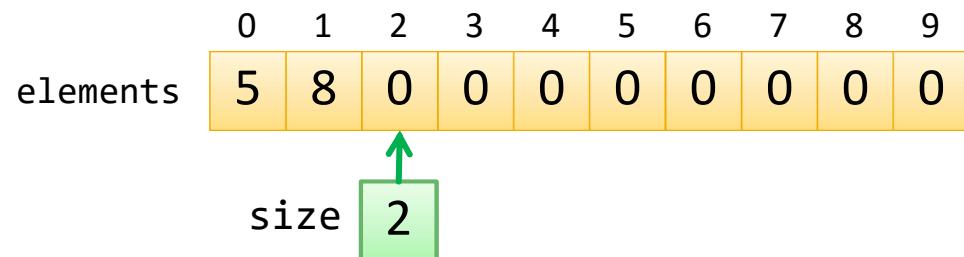
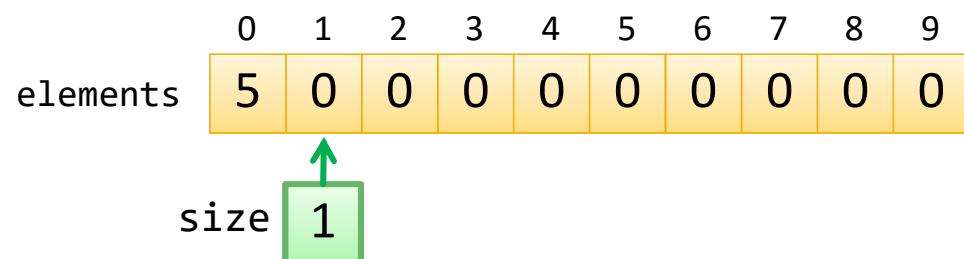
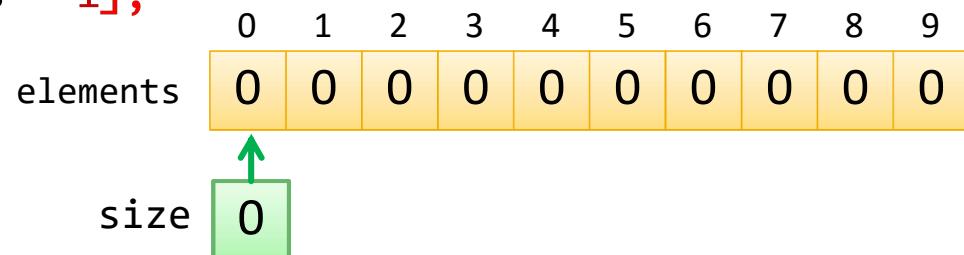
    public boolean isEmpty() {
        return size == 0;
    }
}
```



Impl. `dataStructures.list.ArrayList` vía Arrays (II)

```
public void insert(int i, T elem) {  
    ensureCapacity();  
    if(i != size) {  
        validateIndex(i);  
        for(int pos = size; pos > i; pos--)  
            elements[pos] = elements[pos - 1];  
    }  
    elements[i] = elem;  
    size++;  
}
```

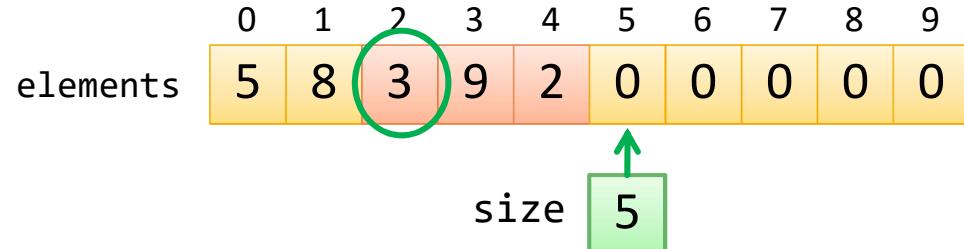
```
List<Integer> xs =  
    new ArrayList<>();  
  
xs.insert(0,5);  
xs.insert(1,8);
```



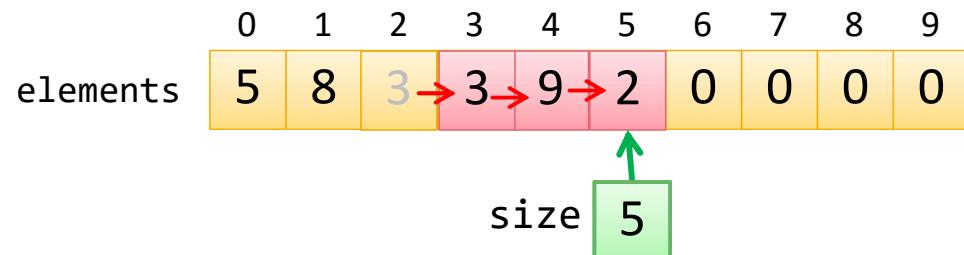
Impl. `dataStructures.list.ArrayList` via Arrays (III)

```
public void insert(int i, T elem) {  
    ensureCapacity();  
    if (i != size) {  
        validateIndex(i);  
        for (int pos = size; pos > i; pos--)  
            elements[pos] = elements[pos - 1];  
    }  
    elements[i] = elem;  
    size++;  
}
```

Se hace sitio: Mueve una posición a la derecha todos los elementos tras el punto de inserción



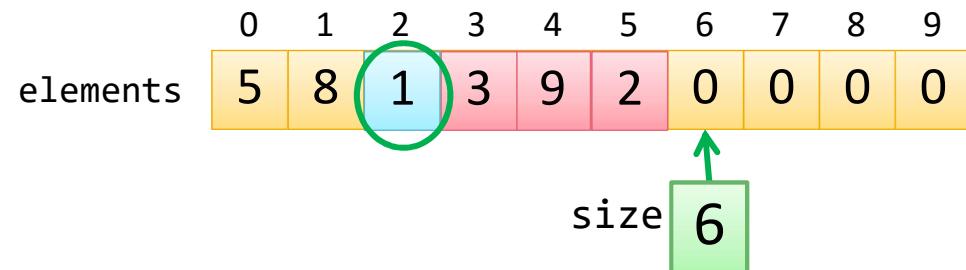
```
List<Integer> xs =  
    new ArrayList<>();  
  
xs.insert(0,5);  
xs.insert(1,8);  
xs.insert(2,3);  
xs.insert(3,9);  
xs.insert(4,2);  
xs.insert(2,1);
```



Impl. `dataStructures.list.ArrayList` via Arrays (IV)

```
public void insert(int i, T elem) {  
    ensureCapacity();  
    if (i != size) {  
        validateIndex(i);  
        for (int pos = size; pos > i; pos--)  
            elements[pos] = elements[pos - 1];  
    }  
    elements[i] = elem;  
    size++;  
}
```

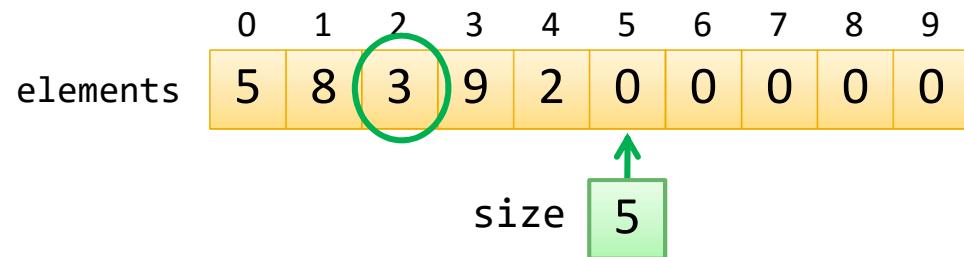
```
List<Integer> xs =  
    new ArrayList<>();  
  
xs.insert(0,5);  
xs.insert(1,8);  
xs.insert(2,3);  
xs.insert(3,9);  
xs.insert(4,2);  
xs.insert(2,1);
```



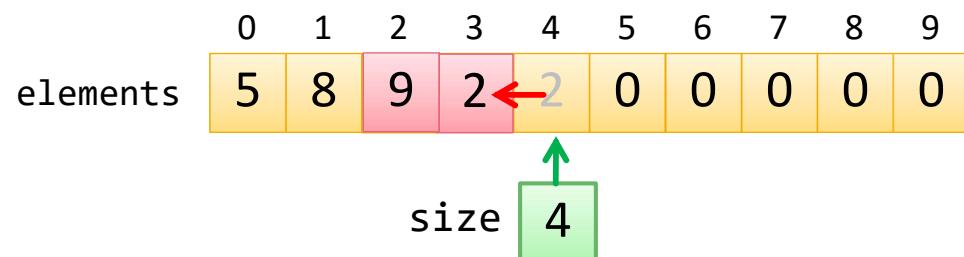
Impl. `dataStructures.list.ArrayList` via Arrays (V)

```
public void remove(int i) {  
    validateIndex(i);  
    for (int pos = i; pos < size - 1; pos++) {  
        elements[pos] = elements[pos + 1];  
    }  
    size--;  
}
```

Mueve una posición a la izquierda todos los elementos tras el punto de eliminación



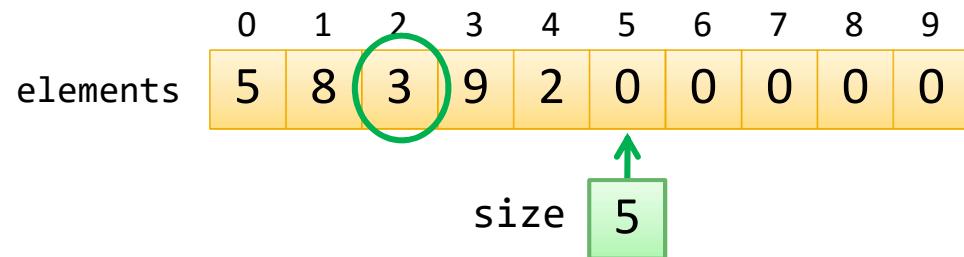
```
List<Integer> xs =  
    new ArrayList<>();  
  
xs.insert(0,5);  
xs.insert(1,8);  
xs.insert(2,3);  
xs.insert(3,9);  
xs.insert(4,2);  
xs.remove(2);
```



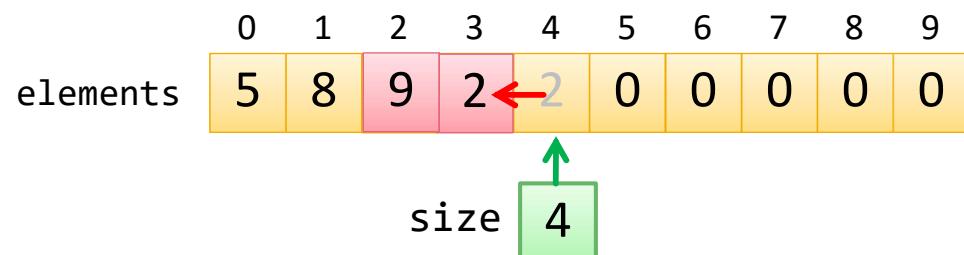
Impl. `dataStructures.list.ArrayList` via Arrays (V)

```
public void remove(int i) {  
    validateIndex(i);  
    for (int pos = i; pos < size - 1; pos++) {  
        elements[pos] = elements[pos + 1];  
    }  
    size--;  
}
```

Mueve una posición a la izquierda todos los elementos tras el punto de inserción



```
List<Integer> xs =  
    new ArrayList<>();  
  
xs.insert(0,5);  
xs.insert(1,8);  
xs.insert(2,3);  
xs.insert(3,9);  
xs.insert(4,2);  
xs.remove(2);
```



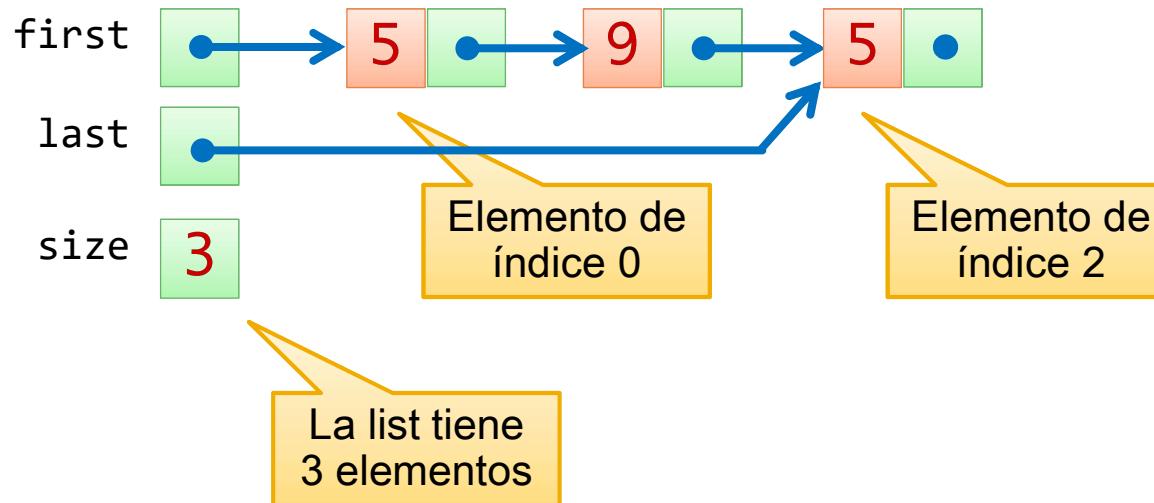
Impl. `dataStructures.list.ArrayList` vía Arrays (y VI)

Eficiencia de la implementación:

Operación	Orden
get	O(1)
set	O(1)
insert	O(n)
remove	O(n)
isEmpty	O(1)
size	O(1)
ArrayList	O(1)

Implementación de List en Java vía Nodos Enlazados (I)

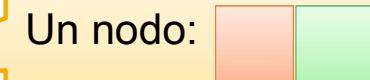
- Una List con 3 elementos:



```
List<Integer> xs = new LinkedList<>();  
  
xs.insert(0,5);  
xs.insert(1,9);  
xs.insert(2,5);
```

Impl. dataStructures.list.linkedList vía Nodos Enlazados (II)

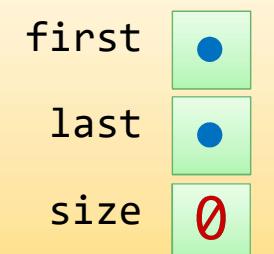
```
package dataStructures.list;
public class LinkedList<T> implements List<T> {
    private static class Node<E> {
        private E elem;
        private Node<E> next;
        public Node(E x, Node<E> node) {
            elem = x;
            next = node;
        }
    }
    private Node<T> first, last;
    private int size;
    public LinkedList() {
        first = null;
        last = null;
        size = 0;
    }
    public int size() {
        return size;
    }
    public boolean isEmpty() {
        return size == 0;
    }
}
```



Enlaces al primer y último nodo de la list

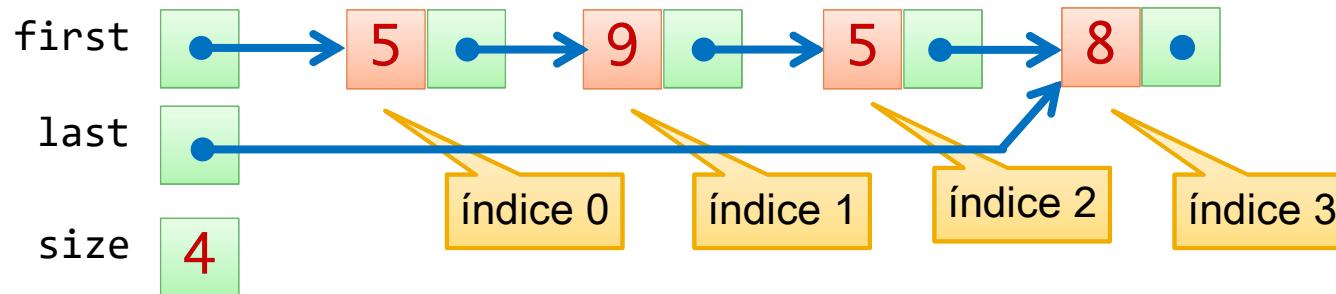
Número de elementos de la list

Una list está inicialmente vacía



Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (III)

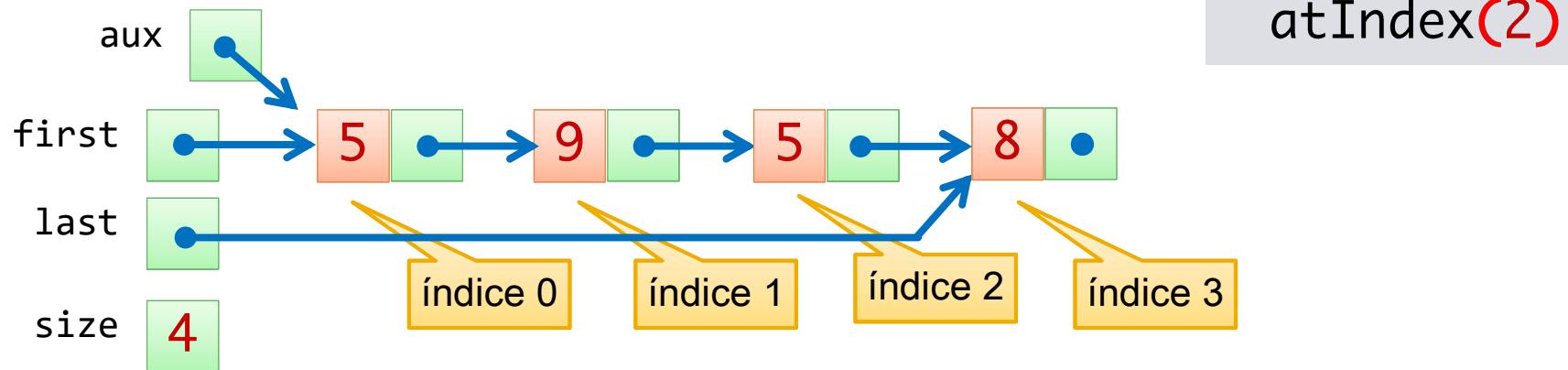
- Localizando un elemento por su índice:



```
private Node<T> atIndex(int i) {  
    Node<T> aux = first;  
    for (int pos = 0; pos < i; pos++)  
        aux = aux.next;  
  
    return aux;  
}
```

Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (IV)

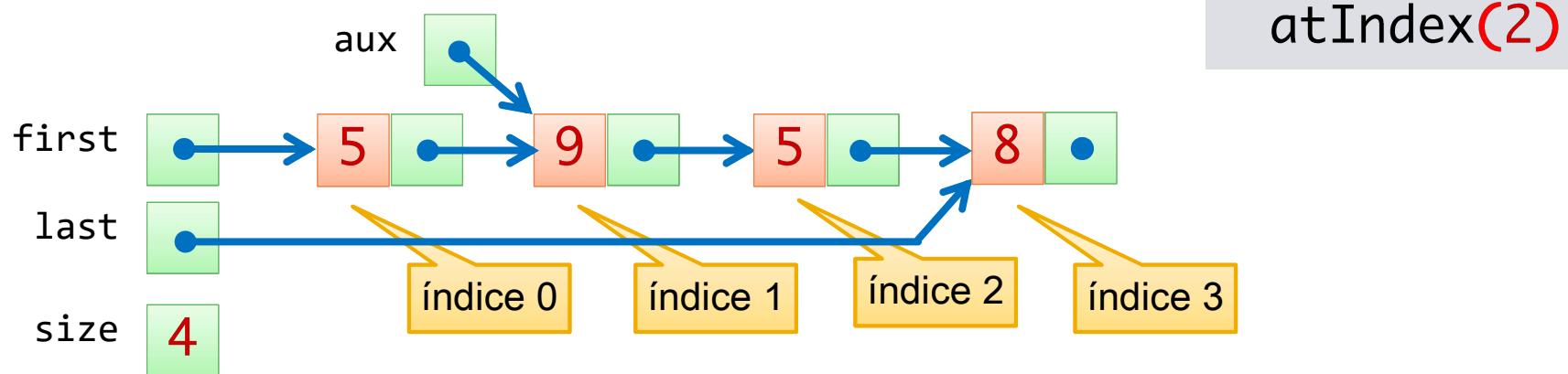
- Localizando un elemento por su índice:



```
private Node<T> atIndex(int i) {  
    Node<T> aux = first;  
    for (int pos = 0; pos < i; pos++)  
        aux = aux.next;  
  
    return aux;  
}
```

Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (V)

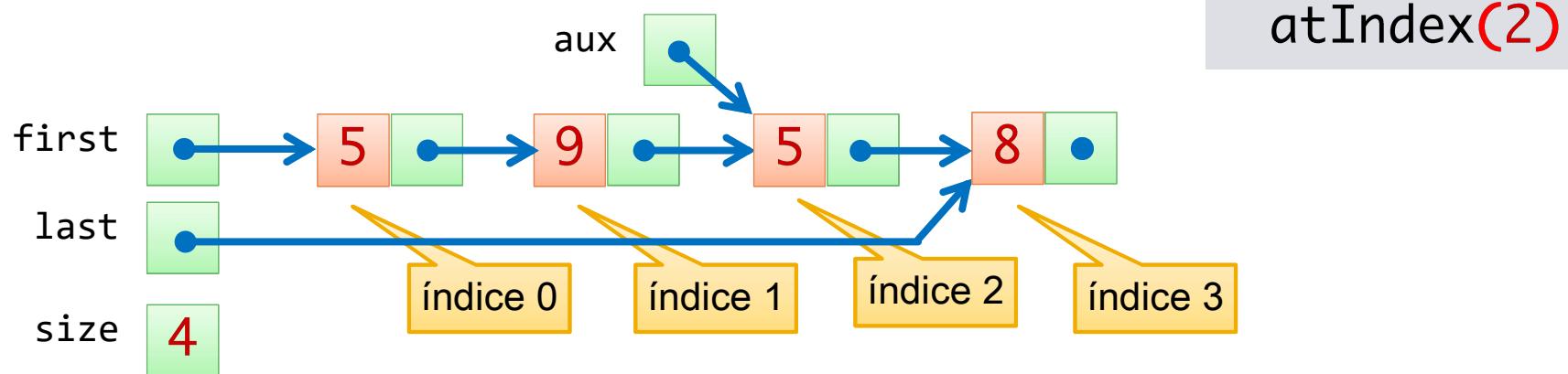
- Localizando un elemento por su índice:



```
private Node<T> atIndex(int i) {  
    Node<T> aux = first;  
    for (int pos = 0; pos < i; pos++)  
        aux = aux.next; pos = 0  
    return aux;  
}
```

Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (VI)

- Localizando un elemento por su índice:



```
private Node<T> atIndex(int i) {  
    Node<T> aux = first;  
    for (int pos = 0; pos < i; pos++)  
        aux = aux.next;    pos = 1  
    return aux;  
}
```

Impl. dataStructures.list.linkedList vía Nodos Enlazados (VII)

```
private void validateIndex(int i) {  
    if (i < 0 || i >= size())  
        throw new ListException("Invalid position " + i);  
}
```

Test $0 \leq i < \text{size}$

```
private Node<T> atIndex(int i) {  
    Node<T> aux = first;  
    for (int pos = 0; pos < i; pos++)  
        aux = aux.next;  
  
    return aux;  
}
```

Devuelve una
referencia al nodo i

```
public T get(int i) {  
    validateIndex(i);  
    return atIndex(i).elem;  
}
```

Devuelve el elemento
de índice i

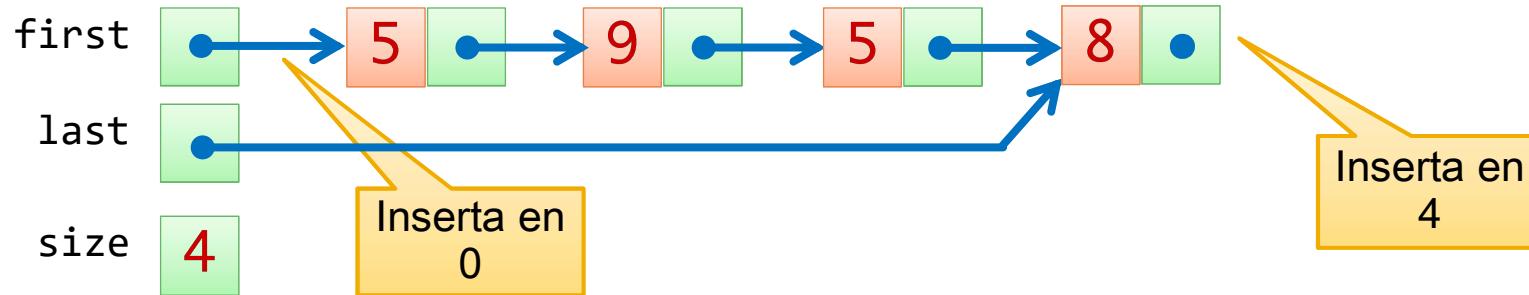
```
public void set(int i, T elem) {  
    validateIndex(i);  
    atIndex(i).elem = elem;  
}
```

Modifica el elemento
de índice i

referencia al Nodo situado en la posición i

Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (VIII)

- `void insert(int i, T elem);`

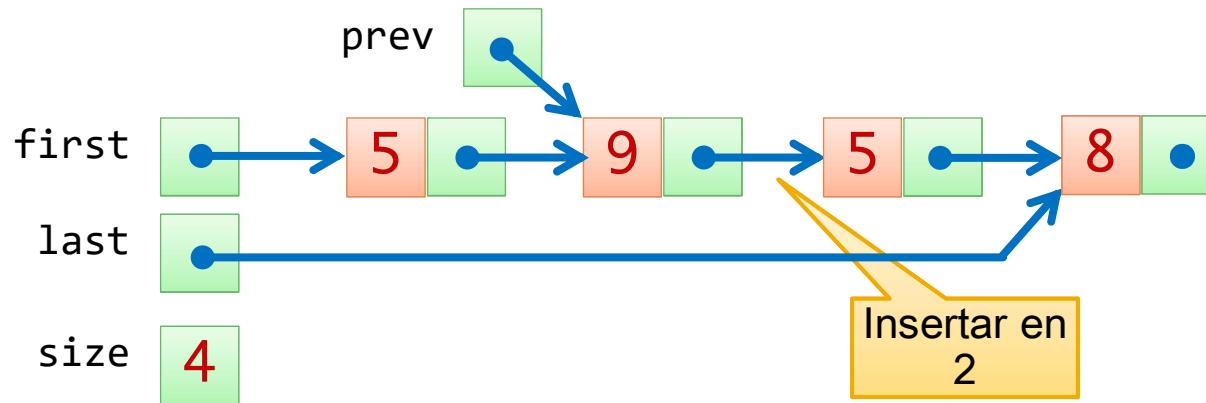


- Insertar en la posición `0` es similar al `push` de `Stack`
- Insertar en la posición `size` es similar al `encolar` en `Queue`
- En otro caso:
 - Localizar la posición del nodo anterior (de índice `i-1`) (`prev`)
 - Crear un nuevo nodo y enlazarlo con `prev.next`
 - Hacer que el enlace `prev.next` enlace al nuevo nodo
 - Incrementar `size`

Impl. dataStructures.list.linkedList vía Nodos Enlazados (IX)

- `void insert(int i, T elem);`

`insert(2,0)`

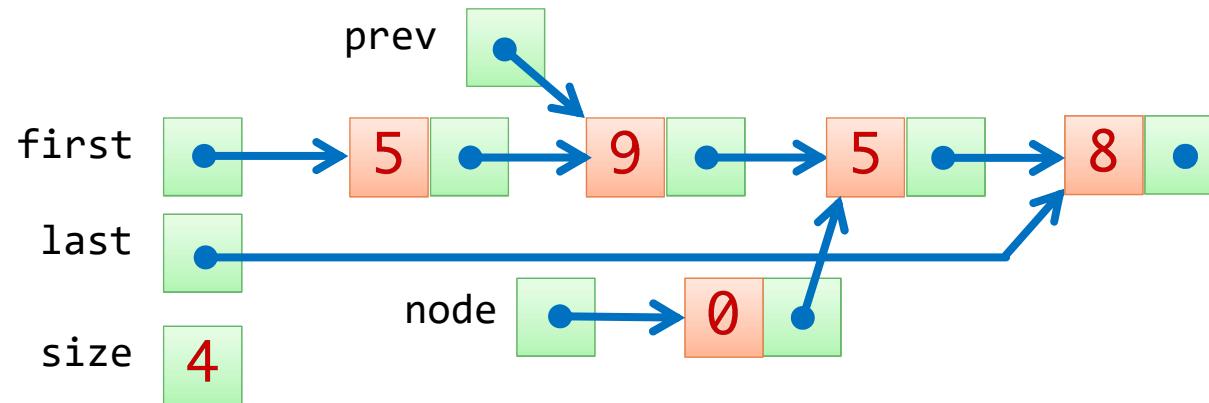


- Insertar en la posición **0** es similar al push de Stack
- Insertar en la posición **size** es similar al encolar en Queue
- En otro caso:
 - Localizar la posición del nodo anterior (de índice **i-1**) (**prev**)
 - Crear un nuevo nodo y enlazarlo con **prev.next**
 - Hacer que el enlace **prev.next** enlace al nuevo nodo
 - Incrementar **size**

Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (X)

- `void insert(int i, T elem);`

`insert(2,0)`

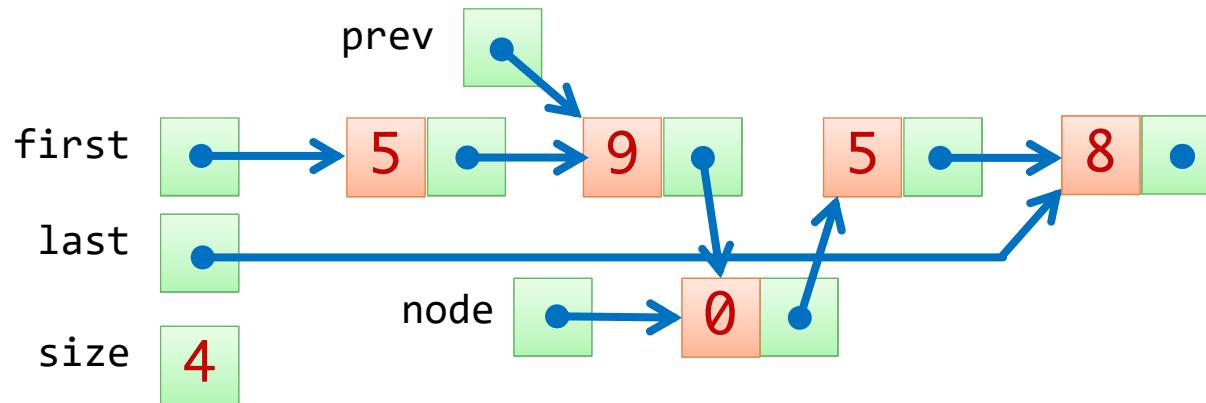


- Insertar en la posición `0` es similar al `push` de Stack
- Insertar en la posición `size` es similar al encolar en Queue
- En otro caso:
 - Localizar la posición del nodo anterior (de índice `i-1`) (`prev`)
 - Crear un nuevo nodo y enlazarlo con `prev.next`
 - Hacer que el enlace `prev.next` enlace al nuevo nodo
 - Incrementar `size`

Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (XI)

- `void insert(int i, T elem);`

`insert(2,0)`

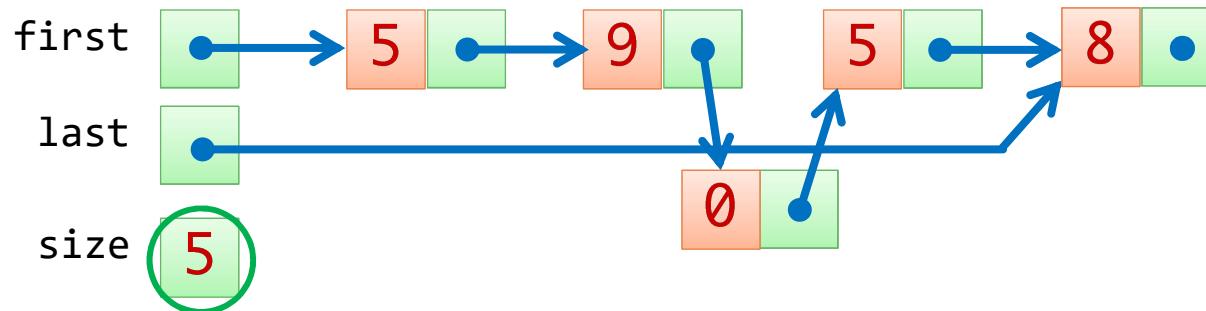


- Insertar en la posición `0` es similar al `push` de Stack
- Insertar en la posición `size` es similar al encolar en Queue
- En otro caso:
 - Localizar la posición del nodo anterior (de índice `i-1`) (`prev`)
 - Crear un nuevo nodo y enlazarlo con `prev.next`
 - Hacer que el enlace `prev.next` enlace al nuevo nodo
 - Incrementar `size`

Impl. dataStructures.list.linkedList vía Nodos Enlazados (XII)

- void insert(int i, T elem);

insert(2,0)



- Insertar en la posición 0 es similar al push de Stack
- Insertar en la posición size es similar al encolar en Queue
- En otro caso:
 - Localizar la posición del nodo anterior (de índice $i-1$) (`prev`)
 - Crear un nuevo nodo y enlazarlo con `prev.next`
 - Hacer que el enlace `prev.next` enlace al nuevo nodo
 - Incrementar `size`

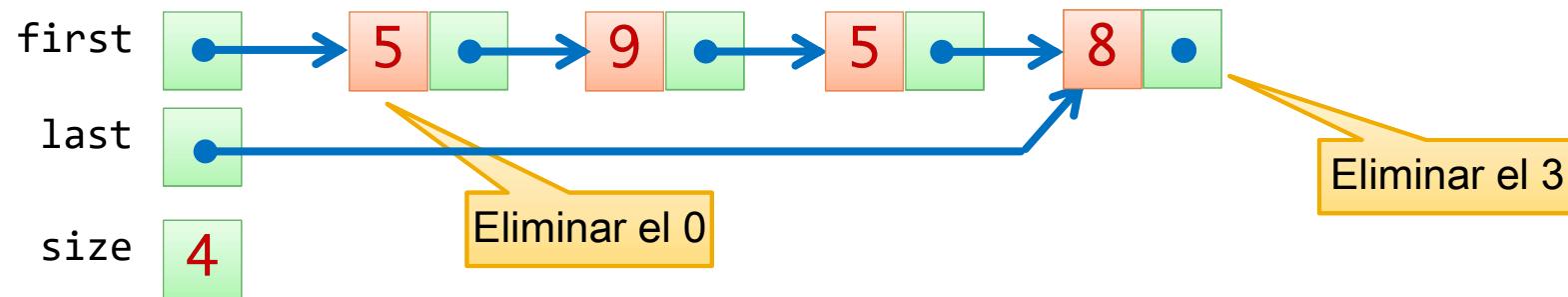
Impl. dataStructures.list.linkedList vía Nodos Enlazados (XIII)

■ Implementación de insert:

```
public void insert(int i, T elem) {  
    if (i == size) { // insertion after last element  
        Node<T> node = new Node<>(elem, null);  
        if (size == 0) // was list empty?  
            first = node;  
        else  
            last.next = node;  
        last = node;  
    } else if (i == 0) { // insertion at head, and list was not empty  
        first = new Node<>(elem, first);  
    } else { // internal insertion  
        validateIndex(i);  
        Node<T> prev = atIndex(i-1);  
        prev.next = new Node<>(elem, prev.next);  
    }  
    size++;  
}
```

Impl. dataStructures.list.linkedList vía Nodos Enlazados (XIV)

- `void remove(int i);`

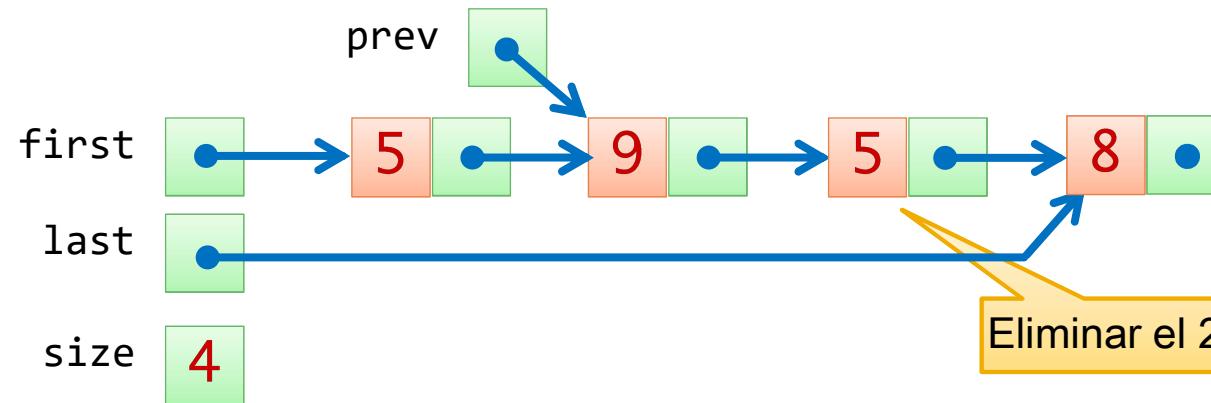


- Eliminar la posición **0** es similar a pop de Stack
- En otro caso:
 - Localizar el nodo previo (de índice **i-1**) (`prev`)
 - Hacer que el enlace `prev.next` enlace a `prev.next.next`
 - Decrementar `size`
- Si el nodo eliminado es el último, `last` debe ser actualizado

Impl. dataStructures.list.linkedList vía Nodos Enlazados (XVI)

- `void remove(int i);`

remove(2)

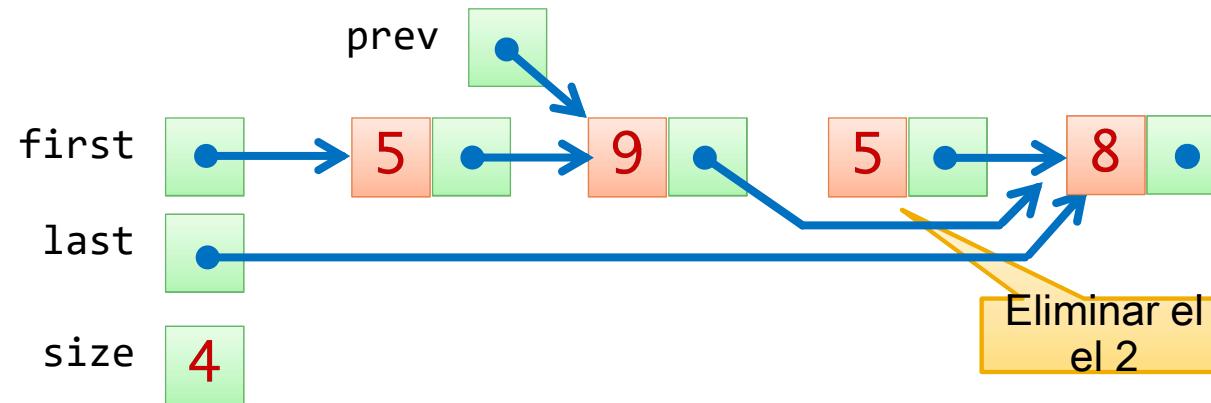


- Eliminar la posición **0** es similar a pop de Stack
- En otro caso:
 - Localizar el nodo previo (de índice **i - 1**) (`prev`)
 - Hacer que el enlace `prev.next` enlace a `prev.next.next`
 - Decrementar `size`
- Si el nodo eliminado es el último, `last` debe ser actualizado

Impl. dataStructures.list.linkedList vía Nodos Enlazados (XVII)

- `void remove(int i);`

remove(2)

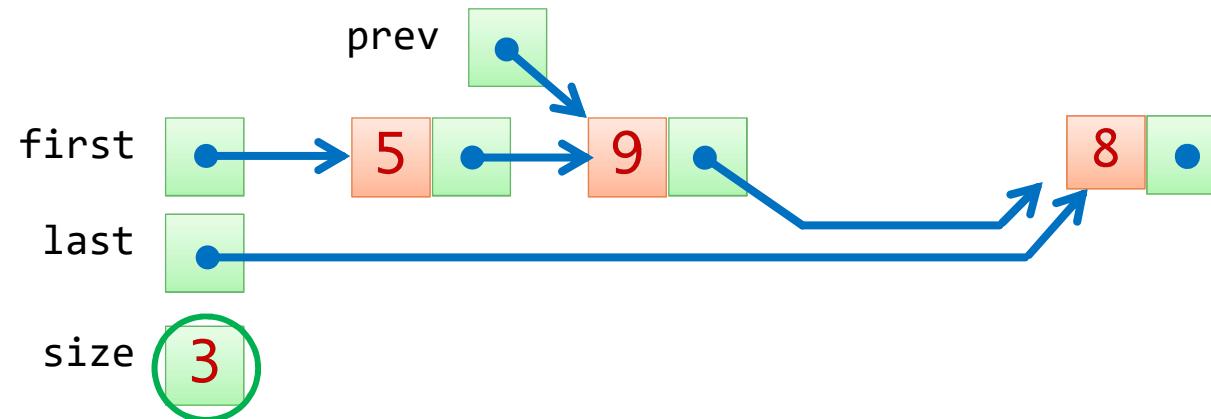


- Eliminar la posición 0 es similar a pop de Stack
- En otro caso:
 - Localizar el nodo previo (de índice $i-1$) (prev)
 - Hacer que el enlace `prev.next` enlace a `prev.next.next`
 - Decrementar size
- Si el nodo eliminado es el último, `last` debe ser actualizado

Impl. dataStructures.list.linkedList vía Nodos Enlazados (XVIII)

- `void remove(int i);`

remove(2)



- Eliminar la posición 0 es similar a pop de Stack
- En otro caso:
 - Localizar el nodo previo (de índice $i-1$) (`prev`)
 - Hacer que el enlace `prev.next` enlace a `prev.next.next`
 - Decrementar `size`
- Si el nodo eliminado es el último, `last` debe ser actualizado

Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (XIX)

■ Implementación de remove:

```
public void remove(int i) {  
    validateIndex(i);  
    if (i == 0) { // removing first element  
        first = first.next;  
        if (first == null) // was also the last element?  
            last = null;  
    } else {  
        Node<T> prev = atIndex(i-1);  
        prev.next = prev.next.next;  
        if (i == size-1) // was last element?  
            last = prev;  
    }  
    size--;  
}
```

Impl. `dataStructures.list.linkedList` vía Nodos Enlazados (y XVIII)

Eficiencia de la implementación:

Operación	Orden
get	$O(n)$
set	$O(n)$
insert	$O(n)$
remove	$O(n)$
isEmpty	$O(1)$
size	$O(1)$
LinkedList	$O(1)$

Linked List vs List con Arrays

- Test experimental
- Medimos el tiempo de ejecución al realizar 10000 operaciones aleatorias (`insert`, `remove` o `get`) en una lista inicialmente vacía.
- Usando un procesador Intel i7 860:
 - Las listas con arrays fueron más del doble de rápidas 😊

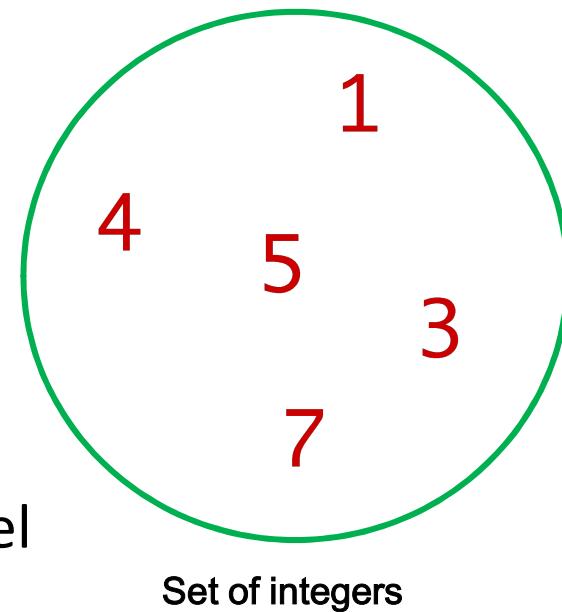
Implementaciones de Stack y Queue por delegación sobre `dataStructures.list.LinkedList`

- Ya se comentó la posibilidad de implementar un Stack y una Queue por delegación sobre `java.util.LinkedList`
- También es posible implementar un Stack y una Queue delegando sobre `dataStructures.list.LinkedList`

(Ejercicio para casa: realiza las implementaciones y analiza sus eficiencias)

Tipo Abstracto de Datos Set

- Un **Set** (conjunto) es un TAD que organiza sus elementos sin que éstos tengan un índice asociado y sin elementos repetidos
- Se corresponden con **conjuntos finitos** en matemáticas
- Operaciones básicas:
 - **isElem**: comprueba si un elemento está en el conjunto
 - **insert**: introduce un nuevo elemento en el conjunto. No hace nada si ya está
 - **delete**: elimina un elemento dado del conjunto. No hace nada si no está
 - **isEmpty**: comprueba si el conjunto es vacío



Signatura del TAD Set

-- Interfaz en Haskell

empty :: Set a

isEmpty :: Set a -> Bool

insert :: Eq a => a -> Set a -> Set a

delete :: Eq a => a -> Set a -> Set a

isElem :: Eq a => a -> Set a -> Bool

Puede ser (Ord a) si la implementación mantiene los elementos ordenados

Un Ejemplo de Uso de Set

```
s1 :: Set Int
s1 = insert 1 (insert 2 (insert 1 empty))
```

```
Main> s1
SortedLinearSet(1,2)

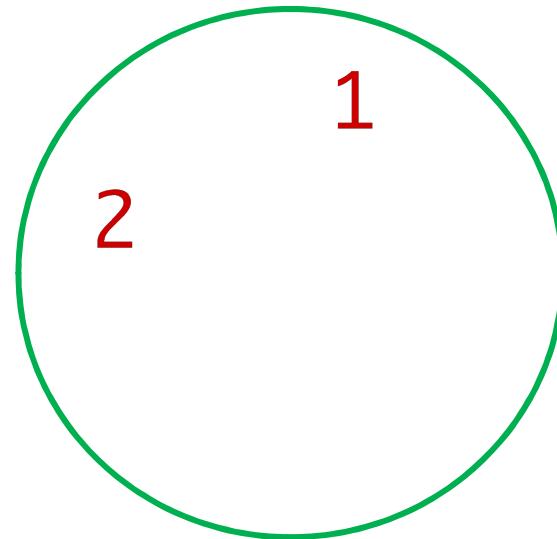
Main> isElem 3 s1
False

Main> isElem 7 s1
False

Main> delete 1 s1
SortedLinearSet(2)

Main> delete 1 (delete 1 s1)
SortedLinearSet(2)

Main> insert 10 s1
SortedLinearSet(1,2,10)
```



Especificación Algebraica de Set usando QuickCheck

- Cualquier implementación de Set debe verificar los siguientes axiomas:

```
module dataStructures.Set.SetAxioms
  (ax1,ax2,ax3,ax4,ax5,ax6,ax7,ax8,ax9,ax10, setAxioms) where

  ax1      = True ==> isEmpty empty
  ax2 x s = True ==> not (isEmpty (insert x s))

  ax3 x     = True ==> not (isElem x empty )
  ax4 x y s = True ==> isElem x (insert y s) == (x==y) || isElem x s

  ax5      = True           ==> size (empty :: Set Elem) == 0
  ax6 x s = isElem x s    ==> size (insert x s) == size s
  ax7 x s = not (isElem x s) ==> size (insert x s) == 1 + size s

  ax8 x     = True ==> delete x empty == empty
  ax9 x y s = x==y ==>
              delete x (insert y s) == delete x s
  ax10 x y s = x/=y ==>
              delete x (insert y s) == insert y (delete x s)
```

Implementaciones de Set en Haskell

- Algunas implementaciones posibles:
 - Lineal:
 - Mantiene los elementos del conjunto **sin repeticiones**
(véase `DataStructures.Set.LinearSet`)
 - Lineal y ordenada: `DataStructures.Set.SortedLinearSet`
 - Mantiene los elementos del conjunto **sin repeticiones** y **ordenados**
 - El orden puede mejorar la eficiencia promedio de las operaciones 😊, aunque no la del peor caso 😞
 - Por delegación sobre listas : `ListSet`
(véase `DataStructures.Set.ListSet`)
- En temas posteriores se verán otras implementaciones mucho más eficientes

Impl. DataStructures.Set.SortedLinearSet (I)

```
module DataStructures.Set.SortedLinearSet
  ( Set
  , empty
  , isEmpty
  , size
  , insert
  , isElem
  , delete
  , fold
  , union
  , intersection
  , difference
  ) where
```

The code block shows a brace grouping the operations from `Set` to `difference`. Two green callout boxes point to this brace:

- The top box contains the text: "La implementación del tipo Set queda oculta" (The implementation of the Set type remains hidden).
- The bottom box contains the text: "Estas son las únicas operaciones disponibles para los clientes" (These are the only operations available to clients).

```
import Data.List(intercalate)
import Test.QuickCheck
```

```
data Set a = Empty | Node a (Set a)
```

```
empty :: Set a
empty = Empty
```

Impl. DataStructures.Set.SortedLinearSet (II)

...

Los elementos están en orden.

Es una simplificación de

(Eq a, Ord a)

```
insert :: (Ord a) => a -> Set a -> Set a
insert x Empty = Node x Empty
insert x (Node y s)
| x < y      = Node x (Node y s)
| x == y     = Node y s
| otherwise   = Node y (insert x s)
```

...

La siguiente secuencia de operaciones:

insert 7 (insert 3 (insert 1 (insert 3 empty)))

Da lugar a la siguiente representación:

Node 1 (Node 3 (Node 7 Empty))

Impl. DataStructures.Set.SortedLinearSet (y III)

Eficiencia de la implementación:

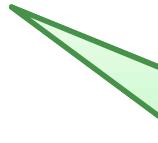
Operación	Orden
isElem	$O(n)$
insert	$O(n)$
delete	$O(n)$
isEmpty	$O(1)$
empty	$O(1)$

Interfaz Set en Java

Aunque Java ya dispone de una interfaz Set en `java.util`, definimos nuestra propia interfaz:

```
package dataStructures.set;

public interface Set<T> extends Iterable<T> {
    boolean isElem(T el);
    void insert(T el);
    void delete(T el);
    boolean isEmpty();
    int size();
}
```



Cualquier implementación de Set tiene que proporcionar un método iterator

La operación `empty` se implementa en el constructor de la clase en Java

Implementaciones del TAD Set en Java

Algunas implementaciones posibles:

- Basada en nodos enlazados **sin repeticiones**:
(LinkedSet)
- Basada en nodos enlazados **sin repeticiones** y con los elementos **en orden**:
(véase SortedLinkedList)
- Por delegación sobre listas de Java (`java.util.LinkedList`):
(LinkedListSet)
- Usando un array **sin repeticiones**:
(ArraySet)
- Usando un array **sin repeticiones** y con los elementos **en orden**:
(SortedArrayList)

Implementación de Set vía Nodos Enlazados Ordenados (I)

- Un Set mantiene información
 - del nodo que está a la cabeza
 - (mediante la variable `first`)
- Cada nodo se enlaza con el siguiente
- **No hay repeticiones**
- Los elementos en los nodos están **ordenados**



```
Set<Integer> s =  
    new SortedLinkedList<>();  
  
s.insert(8);  
s.insert(5);  
s.insert(6);  
s.insert(8);  
s.insert(9);
```

Impl. `dataStructures.set.SortedLinkedList` vía Nodos Enlazados Ordenados (II)

```
package dataStructures.set;

public class SortedLinkedList<T extends Comparable<? super T>> implements Set<T> {
    static private class Node<E> {
        E elem;
        Node<E> next;
        Node(E x, Node<E> node) {
            elem = x;
            next = node;
        }
    }
    private Node<T> first; // we keep linked list sorted by "elem"
    private int size;

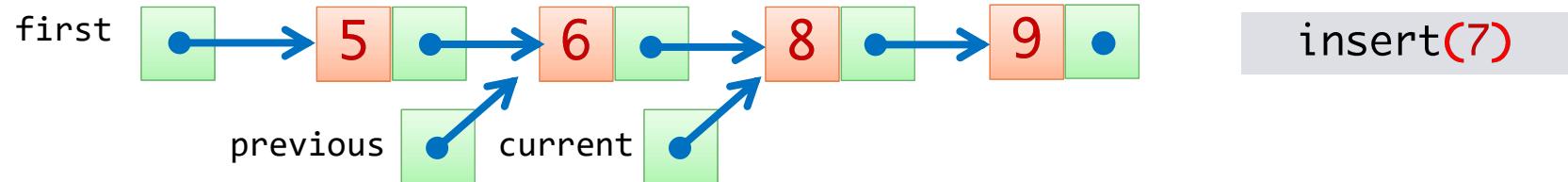
    public boolean isEmpty() {
        return size == 0;
    }

    public int size() { return size; }

    ...
}
```

Los elementos del Set
deben definir una relación
de orden (`compareTo`)

Impl. `dataStructures.set.SortedLinkedList` vía Nodos Enlazados Ordenados (III)



```
public void insert(T item) {
    Node<T> previous = null;
    Node<T> current = first;

    while (current != null && current.elem.compareTo(item) < 0) {
        previous = current;
        current = current.next;
    }

    boolean found = (current != null) && current.elem.equals(item);
    if (!found) {
        if (previous == null) // Insert in first position
            first = new Node<T>(item, first);
        else // Insert after previous
            previous.next = new Node<T>(item, current);
        size++;
    }
}
```

Impl. `dataStructures.set.SortedLinkedList` vía Nodos Enlazados Ordenados (y IV)

Eficiencia de la implementación:

Operación	Orden
<code>isElem</code>	$O(n)$
<code>insert</code>	$O(n)$
<code>delete</code>	$O(n)$
<code>isEmpty</code>	$O(1)$
<code>SortedLinkedList</code>	$O(1)$

Iteradores vs Plegados

- Un **iterador** es un patrón de diseño que abstrae el proceso de recorrer los elementos de una colección.
 - Los elementos de la colección pueden:
 - Formar parte de una estructura de datos, o
 - Ser computados por el iterador conforme se necesiten.
 - Ejemplos
 - Un iterador que recorra todos los elementos de una lista
 - Un iterador que recorra los números del intervalo [5..9]

Iteradores vs Plegados (II)

- Un iterador es útil en algunos tipos abstractos de datos:
 - Set
 - List
 - Tree
 - etc.
- Haskell
 - Es necesario definir una función de **plegado** (fold) para la estructura (similar a la función foldr de listas)
 - La **evaluación perezosa** permite acceder a los elementos **bajo demanda**.
- Java. Se deben crear dos objetos (pertenecientes a dos clases diferentes)
 - Un objeto que implemente la interfaz Iterable<T> que creará un **iterador**.
 - Este iterador debe implementar la interfaz Iterator<T>, proporcionando los métodos next y hasNext, que permiten acceder secuencialmente a los elementos.

Un Plegado para DataStructures.Set.SortedLinearSet en Haskell

```
module DataStructures.Set.SortedLinearSet
( Set
, empty
, ...
, fold
) where
```

Este plegado permite a un cliente iterar sobre todos los elementos del conjunto

```
data Set a = Empty | Node a (Set a)
```

```
fold :: (a -> b -> b) -> b -> Set a -> b
fold f z s = fun s
```

where

```
fun Empty      = z
fun (Node x s) = x `f` (fun s)
```

Un Plegado para

DataStructures.Set.SortedLinearSet en Haskell (II)

```
fold :: (a -> b -> b) -> b -> Set a -> b
fold f z s = fun s
  where
    fun Empty      = z
    fun (Node x s) = x `f` (fun s)
```

Sea la representación de un conjunto S la siguiente:

Node 1 (Node 3 (Node 7 Empty))

add x y = x + y

fold add 0 s => 1 `add` (3 `add` (7 `add` 0)) => ... => 11

```
*Main> fold (+) 0 s
11
Iterar y sumar elementos

*Main> fold (*) 1 s
21
Iterar y multiplicar elementos

*Main> fold (:) [] s
[1,3,7]
Iterar y almacenar en
una lista los elementos

*Main> fold (const (+1)) 0 s
3
Iterar y contar elementos
```

Iteradores en Java

- Un iterador que permite recorrer los elementos de una colección de objetos de tipo T es un objeto que implementa la interfaz :

```
public interface Iterator<T> {  
    boolean hasNext();  
    T next();  
    default void remove() {  
        throw new UnsupportedOperationException();  
    }  
}
```

Esta operación falla por defecto
y no la implementaremos

- Una clase que proporcione un iterador debe implementar la interfaz:

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

- Los bucles `foreach` pueden usarse con objetos de cualquier clase que implemente la interfaz `Iterable<T>`

Implementando Iterable

```
public class MyClass implements Iterable<Type> {  
    ...
```

Este es el tipo concreto
de los elementos
devueltos al iterar

```
    public Iterator<Type> iterator() {  
        return new MyClassIterator();  
    }
```

Método que crea un
iterador (MyClassIterator
implementa Iterator)

```
private class MyClassIterator implements Iterator<Type> {  
    ...  
    public boolean hasNext() {  
        ...  
    }  
  
    public Type next() {  
        ...  
    }  
}
```

Clase anidada que implementa el
iterador para recorrer los
elementos de MyClass

La clase anidada puede
acceder a los atributos de
MyClass 😊

Implementando un Iterador para dataStructures.set.SortedLinkedList

```
package dataStructures.set;

public interface Set<T> extends Iterable<T> {
    boolean isElem(T el);
    ...
}

public class SortedLinkedList<T extends Comparable<? super T>> implements Set<T> {

    static private class Node<E> {
        E elem;
        Node<E> next;

        Node(E x, Node<E> node) {
            elem = x;
            next = node;
        }
    }

    private Node<T> first;
    ...
}
```

Cualquier implementación de Set tiene que proporcionar un método iterator

Implementando un Iterador para dataStructures.set.SortedLinkedList (y II)

```
// Este código va dentro de la clase SortedLinkedList<T>

public Iterator<T> iterator() {
    return new LinkedSetIterator();
}

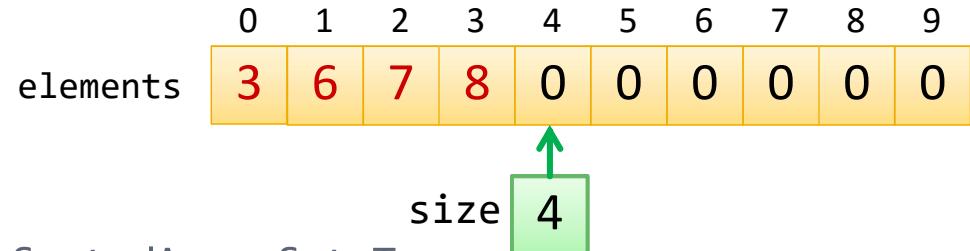
private class LinkedSetIterator implements Iterator<T> {
    Node<T> current; // Esta variable recuerda el nodo a visitar

    public LinkedSetIterator() {
        current = first; // Comenzamos visitando el primer nodo
    }

    public boolean hasNext() {
        return current != null; // Hemos visitado ya todos los nodos?
    }

    public T next() {
        if (!hasNext())
            throw new NoSuchElementException();
        T x = current.elem; // Visitamos el nodo que toca
        current = current.next; // Y avanzamos al siguiente
        return x;
    }
}
```

Implementando un Iterador para dataStructures.set.SortedArraySet



```
// Este código va dentro de la clase SortedArraySet<T>
public Iterator<T> iterator() {
    return new ArraySetIterator();
}

private class ArraySetIterator implements Iterator<T> {
    int current; // Esta variable recuerda el elemento a visitar

    public ArraySetIterator(){
        current = 0; // Comenzamos visitando el primer elemento
    }

    public boolean hasNext(){
        return current != size; // Hemos visitado ya todos los elementos?
    }

    public T next(){
        if (!hasNext())
            throw new NoSuchElementException();
        T x = elements[current]; // Visitamos el elemento que toca
        current++; // Y avanzamos al siguiente
        return x;
    }
}
```

Iterando sobre un Iterable

- Usando un iterador explícito (iterator, hasNext y next):

```
MyClass coll = new MyClass(...);  
Iterator<Type> it = coll.iterator();  
while (it.hasNext())  
    System.out.println(it.next());
```

- Usando un bucle foreach:

```
MyClass coll = new MyClass(...);  
for (Type x : coll)  
    System.out.println(x);
```

Iterando sobre un `dataStructures.Set`

- Usando un iterador explícito (`iterator`, `hasNext` y `next`):

```
Set<Integer> s = new SortedLinkedSet<>();
s.insert(7);
s.insert(3);
...
```

```
Iterator<Integer> it = s.iterator();
while(it.hasNext())
    System.out.println(it.next());
```

- Usando un bucle `foreach`:

```
Set<Integer> s = new SortedArraySet<>();
s.insert(7);
s.insert(3);
...
for(Integer x : s)
    System.out.println(x);
```

Material Complementario

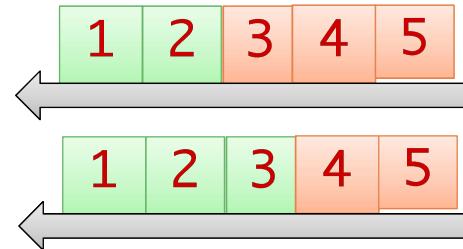
Para profundizar

Implementación de Queue vía dos listas (I)

`data Queue a = Q [a] [a]`

Los elementos de la cola `Q xs ys` son los de la lista `xs++reverse ys`

- Ejemplos:
 - `Q [1,2] [5,4,3]` corresponde a
 - `Q [1,2,3] [5,4]` es la misma
- La misma cola puede representarse de diversas formas
- Pero para cualquier cola `Q xs ys`, si la lista `xs` es vacía también lo será `ys`:
 - `Q [] []` es válida 😊
 - `Q [1,2] []` es válida 😊
 - `Q [] [5,4,3]` es inválida 😥. La implementación evitará esto



Invariante

Impl. DataStructures.Queue.TwoListsQueue vía dos listas (II)

```
module DataStructures.Queue.TwoListsQueue
```

```
( Queue
, empty
, isEmpty
, enqueue
, first
, dequeue
) where
```

```
data Queue a = Q [a] [a]
```

```
empty :: Queue a
empty = Q [] []
```

```
isEmpty :: Queue a -> Bool
isEmpty (Q [] _) = True
isEmpty _         = False
```

Por el invariante, la
segunda lista es vacía

Impl. DataStructures.Queue.TwoListsQueue vía dos listas (III)

```
mkValid :: [a] -> [a] -> Queue a  
mkValid [] ys = Q (reverse ys)  
mkValid xs ys = Q xs ys
```

privada al módulo

Construye una cola a partir de dos listas manteniendo el **invariante**

```
enqueue :: a -> Queue a -> Queue a  
enqueue x (Q xs ys) = mkValid xs (x:ys)
```

Se encola un nuevo elemento manteniendo el invariante

```
first :: Queue a -> a  
first (Q [] _) = error "first on empty queue"  
first (Q (x:xs) ys) = x
```

El primero está en la cabeza de la primera lista

```
dequeue :: Queue a -> Queue a  
dequeue (Q [] _) = error "dequeue on empty queue"  
dequeue (Q (x:xs) ys) = mkValid xs ys
```

El primer elemento se elimina manteniendo e invariante

Impl. DataStructures.Queue.TwoListsQueue vía dos listas (IV). enqueue

```
mkValid :: [a] -> [a] -> Queue a
mkValid [] ys = Q (reverse ys) []
mkValid xs ys = Q xs ys
```

```
enqueue :: a -> Queue a -> Queue a
enqueue x (Q xs ys) = mkValid xs (x:ys)
```

- Si xs no es vacía se usa la segunda ecuación de mkValid , por tanto enqueue es $O(1)$
- Si xs es vacía, ys también lo es (**invariante**), por tanto $(x:ys)$ es una lista con un elemento, y enqueue es $O(1)$
- Finalmente, enqueue es siempre $O(1)$ 😊

Impl. DataStructures.Queue.TwoListsQueue vía dos listas (V). dequeue

```
mkValid :: [a] -> [a] -> Queue a
mkValid [] ys = Q (reverse ys) []
mkValid xs ys = Q xs ys
```

- La llamada a reverse es $O(n)$

dequeue :: Queue a -> Queue a
dequeue (Q []) = error "dequeue on empty queue"
dequeue (Q (x:xs) ys) = mkValid xs ys

- Si $(x:xs)$ tiene más de un elemento, xs no es vacía y se usa la segunda ecuación de mkValid : dequeue es $O(1)$
- Si $(x:xs)$ tiene un solo elemento, xs es vacía y se usa la primera ecuación de mkValid : dequeue es $O(n)$ (siendo n la longitud de ys), pero la primera lista no volverá a ser vacía en las próximas $n-1$ llamadas a dequeue
- Por tanto dequeue es como mínimo $O(1)$ y a lo sumo $O(n)$, pero el coste amortizado es $O(1)$ 😊

Impl. DataStructures.Queue.TwoListsQueue vía dos listas (y VI)

Eficiencia de la implementación:

Operación	Orden
enqueue	O(1)
dequeue	O(1), O(n) *
first	O(1)
isEmpty	O(1)
empty	O(1)

* El **coste amortizado** es O(1): invertimos una lista de n elementos después de encolar n elementos

Secuencias Aritméticas como Iterables

```
public class ArithmeticSeq implements Iterable<Integer> {  
    private int from, to, step;
```

```
    public ArithmeticSeq(int n, int m) {  
        from = n;  
        to = m;  
        step = n < m ? 1 : -1;  
    }
```

new ArithmeticSeq(1,4) representa 1,2,3,4
new ArithmeticSeq(4,1) representa 4,3,2,1

```
    public ArithmeticSeq(int n, int m, int s) {  
        from = n;  
        to = m;  
        step = s;  
    }
```

new ArithmeticSeq(1,7,2) representa 1,3,5,7

```
    public Iterator<Integer> iterator() {  
        return new ArithmeticIter();  
    }
```

...

Secuencias Aritméticas como Iterables (II)

```
public class ArithmeticSeq implements Iterable<Integer> {  
    ...  
    private class ArithmeticIter implements Iterator<Integer> {  
        int current; // holds value to return on invoking next method  
  
        public ArithmeticIter() {  
            current = from;  
        }  
  
        public boolean hasNext() {  
            return step > 0 ? current < to; // Una secuencia creciente  
        }  
  
        public Integer next() {  
            if (!hasNext()) {  
                throw new NoSuchElementException();  
            } else {  
  
                int x = current;  
                current += step;  
                return x;  
            }  
        }  
    }  
}
```

Una secuencia decreciente

Una secuencia creciente

Secuencias Aritméticas como Iterables (III)

```
public class ExplicitIterator {  
    public static void main(String[] args) {  
        Iterator<Integer> it = new ArithmeticSeq(5,9).iterator();  
        while (it.hasNext())  
            System.out.println(it.next());  
    }  
}
```

Salida:
5
6
7
8
9

```
public class ForEachLoop {  
    public static void main(String[] args) {  
        for (int i : new ArithmeticSeq(0,6,2))  
            System.out.println(i);  
    }  
}
```

Salida:
0
2
4
6