Tipos abstractos de datos básicos y nociones de corrección

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025
Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

Repaso: Abstracciones de datos

- En esta teoría, especificamos e implementamos nuestra primera abstracción de datos: `TicketMachine`
- Una abstracción de datos define un conjunto de objetos, y un conjunto de operaciones aplicables a los objetos
  - Abstracción de datos = (objetos, operaciones)
- Notar que una abstracción de datos es un tipo de datos
- Pero nos abstraemos de la representación interna de los objetos
- Requerimos que los clientes usen las operaciones de los objetos, basándose en sus especificaciones (abstracción por especificación)
  - En lugar de modificar directamente la representación interna de los objetos

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 */
public class TicketMachine
{
    /**
     * @pre 'cost' > 0.
     * @post Create a machine that issues tickets
     *   with a price of 'cost'.
     */
    public TicketMachine(int cost)
    /**
     * @post Returns the price of a ticket.
     */
    public int getPrice()
    /**
     * @post Returns the amount of money already inserted
     * next ticket.
     */
    public int getBalance()
    /**
     * @pre 'amount' > 0.
     * @post Receives an amount of money from a customer.
     */
    public void insertMoney(int amount)
    /**
     * @post If enough money has been inserted, print a ti
     * and reduce the current balance by the ticket price.
     * successful; otherwise, it does nothing and returns
     */
    public boolean printTicket()
    /**
     * @post Returns the money in the balance and clears
     * the balance.
     */
    public int refundBalance()
}
```

Repaso: Abstracciones de datos

- De esta manera, los programas que utilizan las abstracciones se vuelven independientes de la representación interna de los objetos
  - Podemos cambiar la implementación sin afectar a las clases que la utilizan
- Diseñar y usar buenas abstracciones de datos resulta en programas bien modularizados
  - Lo que facilita la modificación, la reparación, y la extensión de los programas, y los hace más fáciles de entender
- Las abstracciones de datos son la base de la programación orientada a objetos, y uno de los conceptos centrales de la materia

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 */
public class TicketMachine
{
    /**
     * @pre 'cost' > 0.
     * @post Create a machine that issues tickets
     *   with a price of 'cost'.
     */
    public TicketMachine(int cost)
    /**
     * @post Returns the price of a ticket.
     */
    public int getPrice()
    /**
     * @post Returns the amount of money already inserted
     * next ticket.
     */
    public int getBalance()
    /**
     * @pre 'amount' > 0.
     * @post Receives an amount of money from a customer.
     */
    public void insertMoney(int amount)
    /**
     * @post If enough money has been inserted, print a ti
     * and reduce the current balance by the ticket price.
     * successful; otherwise, it does nothing and returns
     */
    public boolean printTicket()
    /**
     * @post Returns the money in the balance and clears
     * the balance.
     */
    public int refundBalance()
}
```

Tipos abstractos de datos

- Un tipo abstracto de datos (TAD) es una descripción abstracta de un tipo
  - En general, en términos de alguna estructura matemática conocida, como secuencias, conjuntos, maps, etc.
- Hasta ahora trabajamos con TADs desde la perspectiva del cliente
  - Usamos listas, conjuntos, y maps en nuestras aplicaciones
  - Destacamos que la mayoría de las aplicaciones actuales usan TADs de alguna u otra forma
- A partir de ahora vamos a trabajar con TADs desde la perspectiva del desarrollador: vamos a ver como hacer implementaciones eficientes de TADs
- Esto es útil por varios motivos:
  - Sirve para entender las características de eficiencia de cada implementación de TADs
  - Nos permite elegir el TAD y la implementación más apropiada para cada aplicación particular
  - Si las bibliotecas disponibles no son del todo satisfactorias, podemos desarrollar nuestras propias implementaciones de TADs

TAD Stack

- Un Stack modela una secuencia de elementos que se comporta como una pila
  - Vamos a denotar esto diciendo que para un objeto de la clase Stack:
    `this = [o1, o2,..., on]`

```
/**
 * Stack represents unbounded, last-in-first-out (LIFO)
 * stack of objects of type T.
 *
 * A typical Stack is a sequence [o1, o2,..., on]; we
 * denote this by: this = [o1, o2,..., on].
 *
 * The methods use equals to determine equality of elements.
 */
public interface Stack<T>
```

- Sus operaciones típicas son: agregar un elemento en el tope, quitar el elemento del tope, y consultar el elemento en el tope
- Decimos que la política de inserción y eliminación de las pilas es last-in-first-out (el último elemento que entra es el primero que sale)

TAD Stack

- Un Stack modela una secuencia de elementos que se comporta como una pila
  - Vamos a denotar esto diciendo que para un objeto de la clase Stack:
    `this = [o1, o2,..., on]`

```
/**
 * Stack represents unbounded, last-in-first-out (LIFO)
 * stack of objects of type T.
 *
 * A typical Stack is a sequence [o1, o2,..., on]; we
 * denote this by: this = [o1, o2,..., on].
 *
 * The methods use equals to determine equality of elements.
```

Vamos a especificar todos los TADs definiendo interfaces para los mismos

- del tope, y consultar el elemento en el tope
- Decimos que la política de inserción y eliminación de las pilas es last-in-first-out (el último elemento que entra es el primero que sale)

TAD Stack

- Un Stack modela una secuencia de elementos que se comporta como una pila
  - Vamos a denotar esto diciendo que para un objeto de la clase Stack:
    `this = [o1, o2,..., on]`

```
/**
 * Stack represents unbounded, last-in-first-out (LIFO)
 * stack of objects of type T.
 *
 * A typical Stack is a sequence [o1, o2,..., on]; we
 * denote this by: this = [o1, o2,..., on].
 *
 * The methods use equals to determine equality of elements.
```

Vamos a especificar todos los TADs definiendo interfaces para los mismos

- del tope, y consultar el elemento en el tope

Esto además nos permitirá usar distintas implementaciones, y agregar nuevas implementaciones sin afectar a los clientes

- t

TAD Stack: Operaciones típicas

- Como una pila es una secuencia, podemos especificar (algunas propiedades) de sus operaciones en términos de secuencias
  - Vamos a tomar al último elemento de la secuencia como el tope de la pila
- Para toda variable `v`, vamos a usar la siguiente convención en las especificaciones:
  - el operador `old(v)` retorna el valor de `v` antes de ejecutar el método
  - `v` denota el valor de la variable al finalizar la ejecución del método
    - De esta manera, la especificación de `push` nos dice que: `this = old(this) ++ [e]`
      - Donde `++` es la operación abstracta de concatenación de secuencias
      - Ej.: si ejecutamos `push(4)` sobre la pila `[1, 2, 3]`, obtenemos `[1, 2, 3, 4]`

```java
/**
 * @post Adds element e to the top of the stack.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void push(T e);
```

TAD Stack: Operaciones típicas

- Como una pila es una secuencia, podemos especificar (algunas propiedades) de sus operaciones en términos de secuencias
  - Vamos a tomar al último elemento de la secuencia como el tope de la pila
- Para toda variable `v`, vamos a usar la siguiente convención en las especificaciones:
  - el operador `old(v)` retorna el valor de `v` antes de ejecutar el método
  - `v` denota el valor de la variable al finalizar la ejecución del método
    - De esta manera, la especificación de `push` nos dice que: `this = old(this) ++ [e]`
      - Donde `++` es la operación abstracta de concatenación de secuencias
      - Ej.: si ejecutamos `push(4)` sobre la pila `[1, 2, 3]`, obtenemos `[1, 2, 3, 4]`

```
/**
 * @post Adds element e
 *   More formally, it
 */
public void push(T e);
```

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Removes and returns the item at the top of the stack.
 *   More formally, it satisfies:
 *     let old(this) = s1 ++ [e] |
 *       this = s1 && result = e.
 */
public T pop();
```

TAD Stack: Operaciones típicas

- Como una pila es una secuencia, podemos especificar (algunas propiedades) de sus operaciones en términos de secuencias
  - Vamos a tomar al último elemento de la secuencia como el tope de la pila
- Para toda variable `v`, vamos a usar la siguiente convención en las especificaciones:
  - el operador `old(v)` retorna el valor de `v` antes de ejecutar el método
  - `v` denota el valor de la variable al finalizar la ejecución del método
- De esta manera, la especificación de `push` nos dice que: `this = old(this) ++ [e]`
  - Donde `++` es la operación abstracta de concatenación de secuencias
  - Ej.: si ejecutamos `push(4)` sobre la pila `[1, 2, 3]`, obtenemos `[1, 2, 3, 4]`

```
/**
 * @post Adds element e
 *   More formally, it
 */
public void push(T e);
```

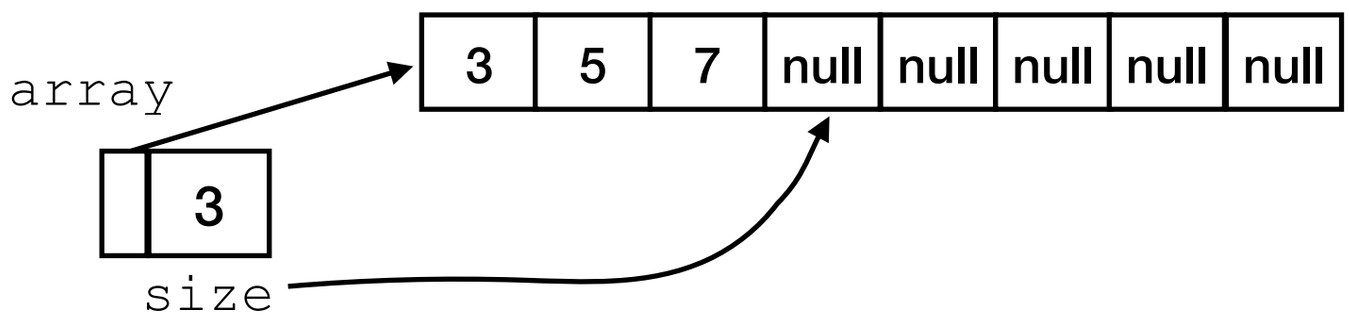
```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Removes and returns the item at the top of the stack.
 *   More formal
 *     let old(t
 *       this =
 */
public T pop();
```

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Returns the item at the top of the stack.
 *   More formally, it satisfies:
 *     let this = s1 ++ [e] | result = e.
 */
public T top();
```

Implementación de Stack con arreglos

```java
public class ArrayStack<T> implements Stack<T>, Iterable<T>
{
    // initial capacity of underlying resizing array
    private static final int INIT_CAPACITY = 8;

    protected T[] array;          // array of items
    protected int size;
```

- Una representación de pilas consiste en llevar un arreglo para guardar los elementos de la pila (`array`), y una variable entera (`size`) que cuente la cantidad de elementos almacenados
  - Los elementos de la pila son los elementos de `array` entre las posiciones `[0,..,size-1]`
  - El elemento en la posición `size-1` es el tope de la pila



- Inicialmente, el arreglo tiene tamaño 8, pero lo iremos agrandando/achicando bajo demanda en las operaciones

Implementación de Stack con arreglos: Operaciones

```java
/**
 * @post Creates an empty stack.
 *   More formally, it satisfies: this = [].
 */
public ArrayStack()
{
    array = (T[]) new Object[INIT_CAPACITY];
    size = 0;
}
```

```java
/**
 * @post Adds element e to the top of the stack.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void push(T item)
{
    if (size == array.length)
        resize(2*array.length);    // double size of array

    array[size++] = item;          // add item
}
```

Implementación de Stack con arreglos: Operaciones

```java
/**
 * @post Creates an empty stack.
 *   More formally, it satisfies: this = [].
 */
public ArrayStack()
{
    array = (T[]) new Object[INIT_CAPACITY];
    size = 0;
}
```

```java
/**
 * @post Adds element e to the
 *   More formally, it satisfie
 */
public void push(T item)
{
    if (size == array.length)
        resize(2*array.length);

    array[size++] = item;
}
```

```java
/**
 * @pre capacity > size() (throws IllegalArgumentException).
 * @post Resize the underlying array to the given capacity.
 */
private void resize(int capacity)
{
    if (capacity <= size())
        throw new IllegalArgumentException("The new array" +
          "must be larger than the current size: " + size());

    T[] copy = (T[]) new Object[capacity];
    for (int i = 0; i < size; i++) {
        copy[i] = array[i];
    }
    array = copy;
}
```

Implementación de Stack con arreglos: Operaciones

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Removes and returns the item at the top of the stack.
 *   More formally, it satisfies:
 *     let old(this) = s1 ++ [e] |
 *       this = s1 && result = e.
 */
public T pop()
{
    if (isEmpty())
        throw new NoSuchElementException("Stack underflow");

    T item = array[size-1];
    array[size-1] = null;
    size--;
    // shrink size of array if necessary
    if (size > 0 && size == array.length/4)
        resize(array.length/2);

    return item;
}
```

Implementación de Stack con arreglos: Operaciones

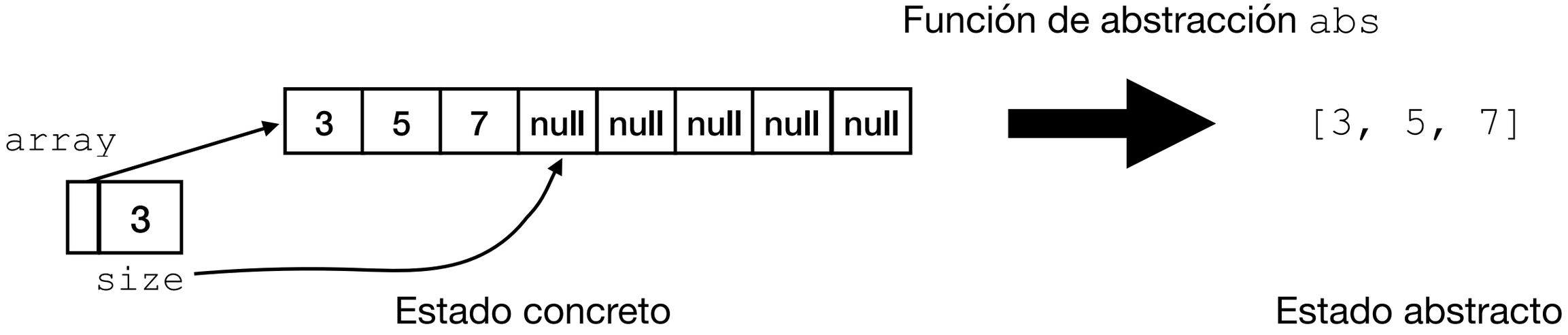
```java
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Returns the item at the top of the stack.
 *   More formally, it satisfies:
 *     let this = s1 ++ [e] | result = e.
 */
public T top()
{
    if (isEmpty())
        throw new NoSuchElementException("Stack underflow");

    return array[size-1];
}
```

Implementación de Stack con arreglos: Operaciones

```java
/**
 * @post Returns true iff the stack contains no elements.
 *   More formally, it satisfies: result = (this = []).
 */
public boolean isEmpty()
{
    return size == 0;
}

/**
 * @post Returns the number of elements in the stack.
 *   More formally, it satisfies: result = #this.
 */
public int size()
{
    return size;
}
```

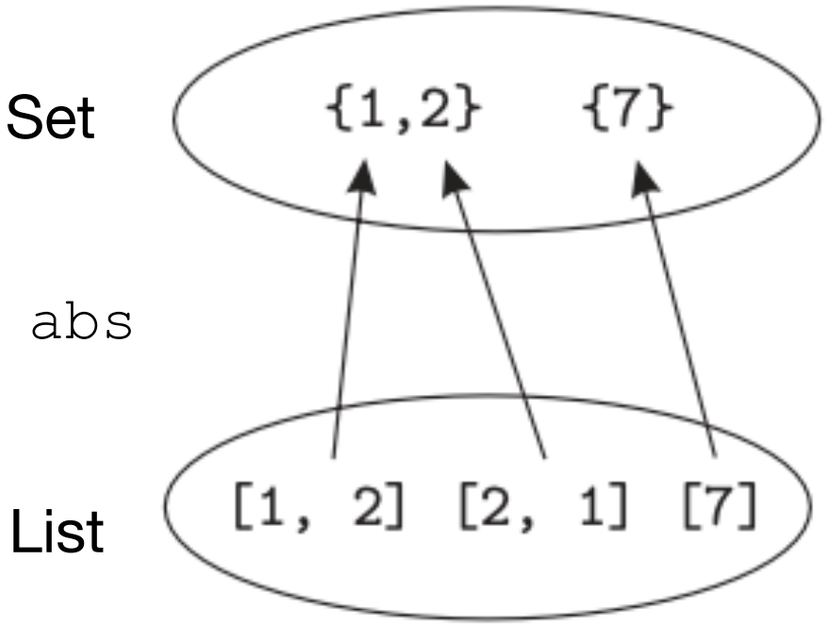
Función de abstracción

- Cuando implementamos un TAD elegimos una representación particular
  - En nuestro ejemplo, la representación consiste de un arreglo `array` y un entero `size`
- Y siempre tenemos en cuenta la relación que hay entre la representación y los elementos del TAD
- Esta relación puede definirse de manera precisa mediante la función de abstracción `abs: C -> A`
  - `abs` mapea elementos de la representación (`C`) en elementos del tipo abstracto (`A`)
- Para nuestro ejemplo, `abs` mapea un estado concreto a la secuencia de elementos en `array` entre `0` y `size-1`:



Función de abstracción

- La función de abstracción puede relacionar varios elementos concretos a un mismo elemento abstracto (es una relación many-to-one)
  - Abstrae el objeto concreto y pierde detalles
- La figura muestra la función de abstracción para una implementación de conjuntos con listas, que mapea cada lista al conjunto de sus elementos
- Al abstraer una lista para convertirla en un conjunto perdemos la noción de orden y repetición, por lo que las listas [1, 2] y [2, 1] representan el mismo conjunto {1, 2}

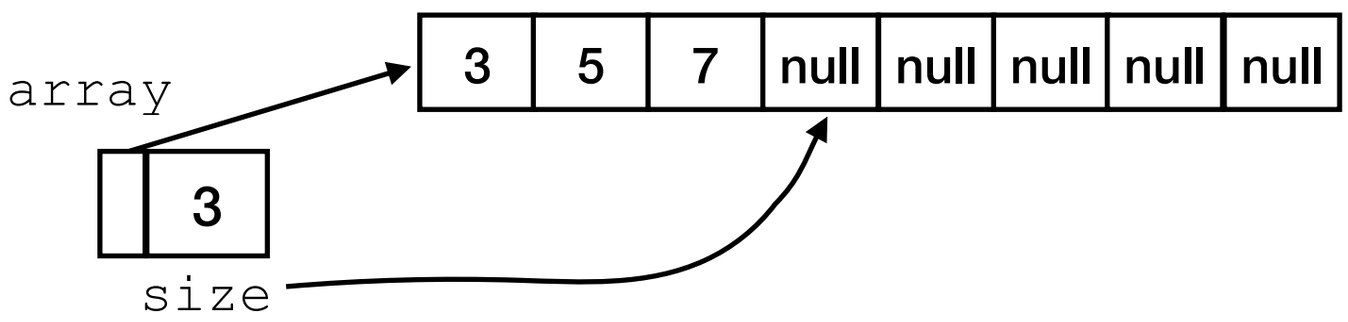


- La función de abstracción tiene información crucial sobre la implementación: define la manera en la que los objetos de la clase implementan los objetos abstractos
  - Por lo tanto, es de gran ayuda para entender la implementación

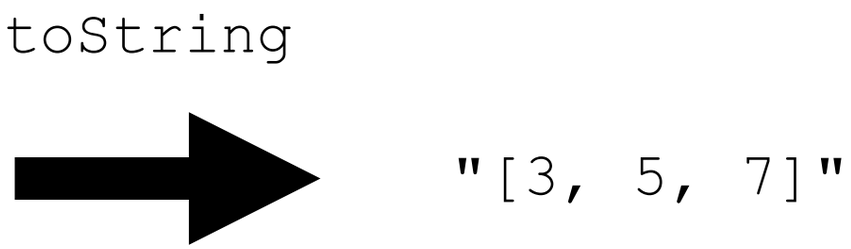
Función de abstracción

- La función de abstracción puede proveerse informalmente en un comentario en la clase
- O podemos implementarla en el método `toString` de la clase
- Vamos a seguir este último enfoque, ya que nos será de mucha utilidad para testear nuestras implementaciones

```java
/**
 * @post Returns a string representation of the stack. Implements
 *   the abstraction function. Hence, it represents the stack as a
 *   sequence "[o1, o2, ..., on]".
 */
public String toString() {
    String res = "[";
    for (int i = 0; i < size; i++)
    {
        res += array[i].toString();
        if (i < size-1)
            res += ", ";
    }
    res += "]";
    return res;
}
```



Estado concreto



Representación como `String` del estado abstracto

Función de abstracción y testing

- Como vimos, las operaciones de nuestra implementación se especifican en términos del TAD
- Por lo tanto, podemos usar la función de abstracción, las especificaciones y nuestros conocimientos sobre las operaciones del TAD para escribir tests para aumentar nuestra confianza en la corrección de la implementación
- Ej.: sabemos que el constructor de `LinkedStack` genera un objeto que representa la secuencia vacía `[]`

```
/**
 * @post Creates an empty stack.
 *   More formally, it satisfies: this = [].
 */
public LinkedStack()
```

- Podemos testear que nuestra implementación satisface la especificación del TAD con los siguientes pasos: construir un objeto, usar el `toString` para generar la secuencia abstracta que representa, y verificar que es efectivamente la cadena "`[]`"

```
@Test
public void testConstructorCreatesEmptyStack()
{
    Stack<Integer> stack = new ArrayStack<Integer>();
    String abstractSeq = stack.toString();
    assertEquals("[]", abstractSeq);
}
```

Función de abstracción y testing

- Similarmente, sabemos de la especificación de `push` que la operación agrega un objeto al final de la secuencia abstracta que representa el `Stack`

```
/**
 * @post Adds element e to the top of the stack.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void push(T item)
```

- Por lo tanto, podemos testear que si partimos del `Stack` vacío, y luego hacemos `push(1)` y `push(2)`, obtenemos la secuencia abstracta `"[1, 2]"`

```
@Test
public void testPushTwoElements()
{
    Stack<Integer> stack = new ArrayStack<Integer>();
    stack.push(1);
    stack.push(2);
    String abstractSeq = stack.toString();
    assertEquals("[1, 2]", abstractSeq);
}
```

Invariante de representación

- No siempre los objetos de una clase son representaciones legítimas de los elementos de un TAD
  - Por ejemplo, si `size < 0 || size > array.length` tendríamos objetos inválidos (que no representan elementos del tipo abstracto de datos secuencia)
- El invariante de representación, `repOK`, especifica las propiedades que todos los objetos legítimos de la clase deben satisfacer
  - Formalmente, `repOK` es una función que mapea objetos del tipo concreto `C` en booleanos:
    - `repOK: C -> boolean`
  - `repOK(obj) == true` si y sólo si `obj` es un objeto legítimo de tipo `C`
- `repOK` da información valiosa para comprender la implementación: define cuáles son las características de los objetos válidos
  - Y qué objetos no son considerados representantes válidos de los elementos del TAD

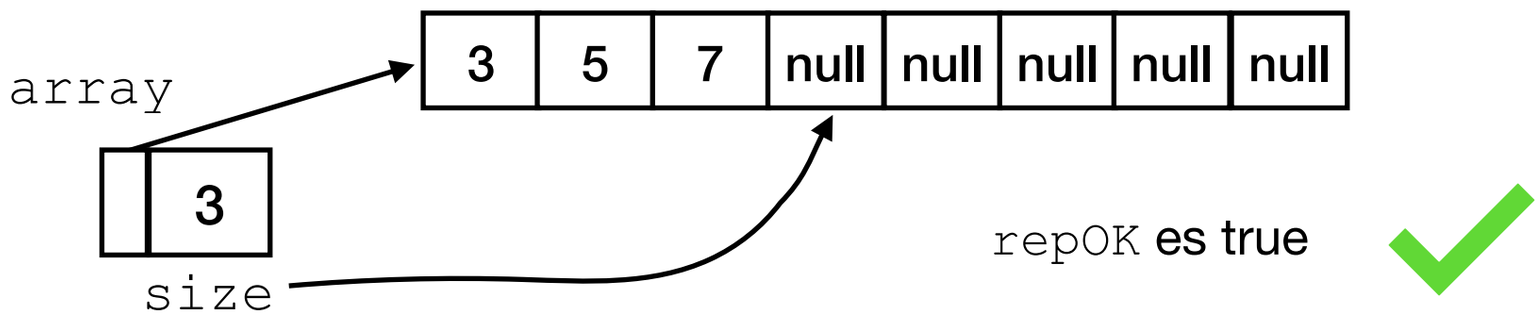
Invariante de representación

- Además, `repOK` nos da información adicional para verificar la corrección de la implementación:
  - Los constructores de la clase deben crear objetos que satisfacen `repOK`:
    - `{pre} o = new C(); {post && o.repOK()}`
  - Todos los métodos de la clase deben preservar `repOK`:
    - `{pre && o.repOK()} o.M(); {post && o.repOK()}`

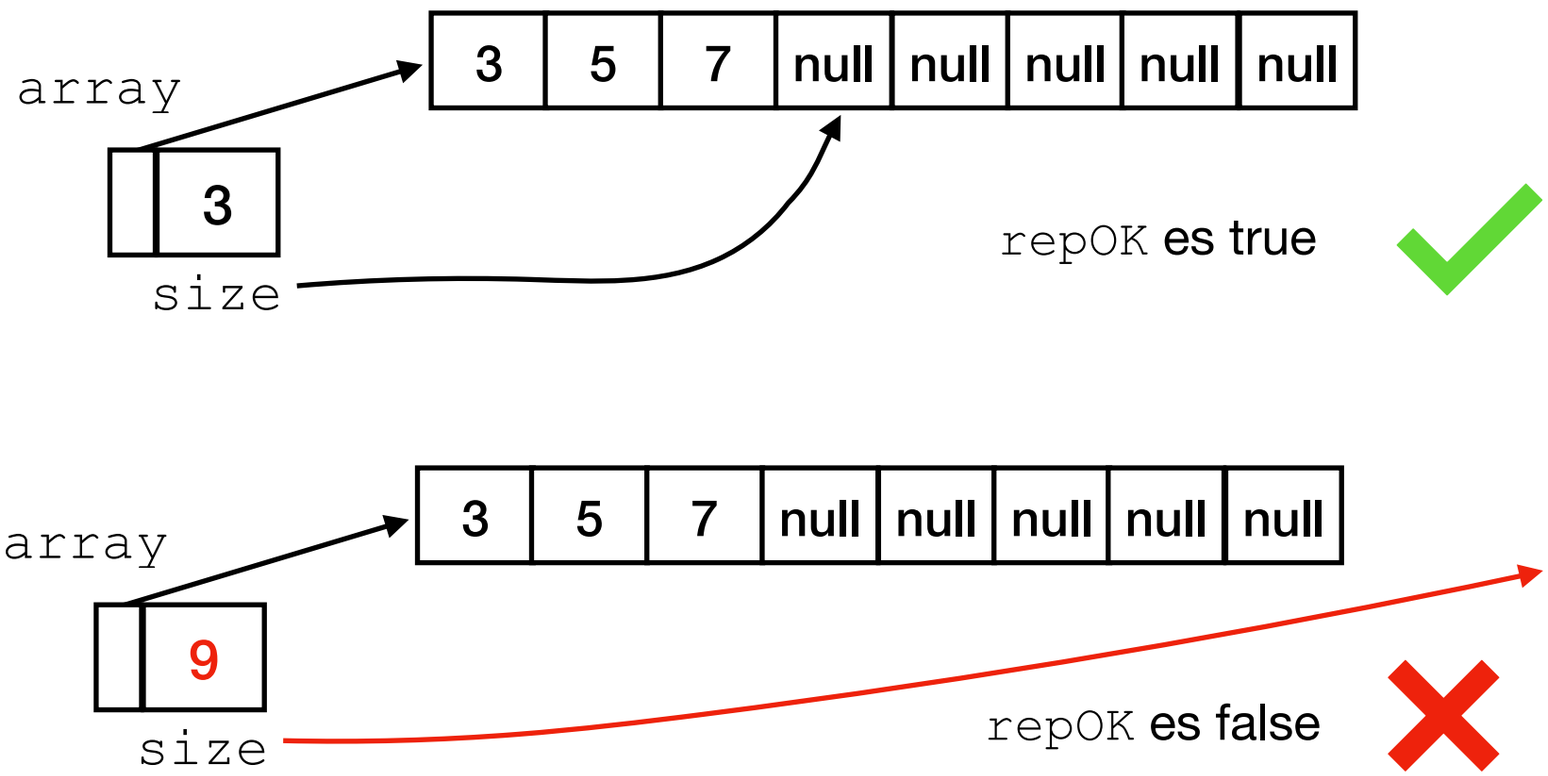
- Podemos definir `repOK` informalmente en un comentario de clase, o podemos implementarlo mediante un método booleano en la clase
- Seguiremos este último enfoque, que nos será de utilidad para testear mejor nuestras implementaciones
  - Usaremos `repOK` como aserción adicional durante el testing para verificar que los objetos creados para la clase son válidos

```java
/**
 * @post Returns true if and only if the structure is a
 *   valid stack.
 */
public boolean repOK() {
    if (size < 0 || size > array.length)
        return false;
    // The stack does not store null as an element
    for (int i = 0; i < size; i++) {
        if (array[i] == null)
            return false;
    }

    return true;
}
```

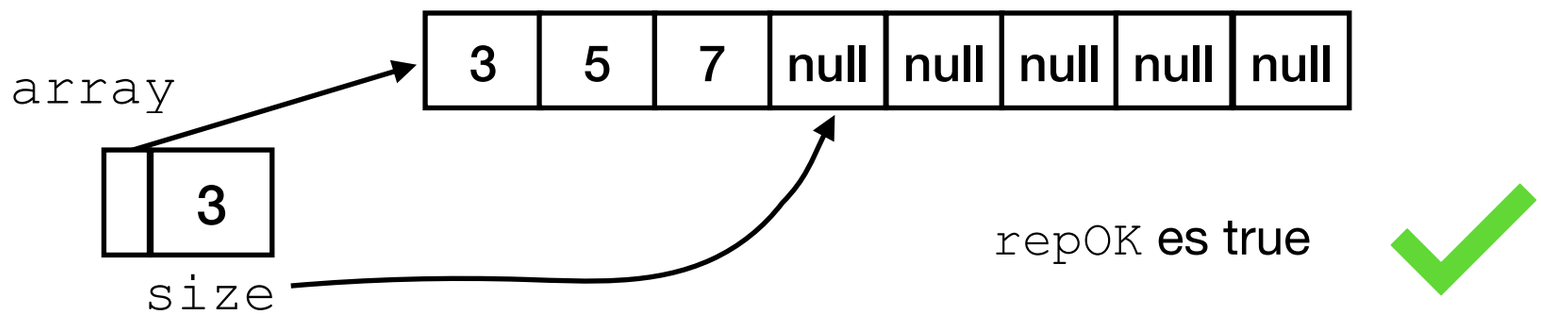
Invariante de representación

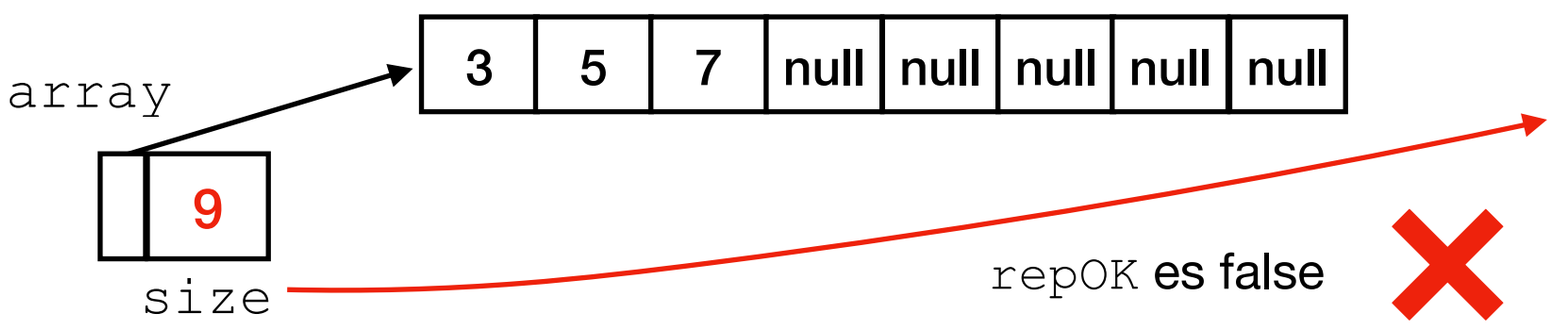


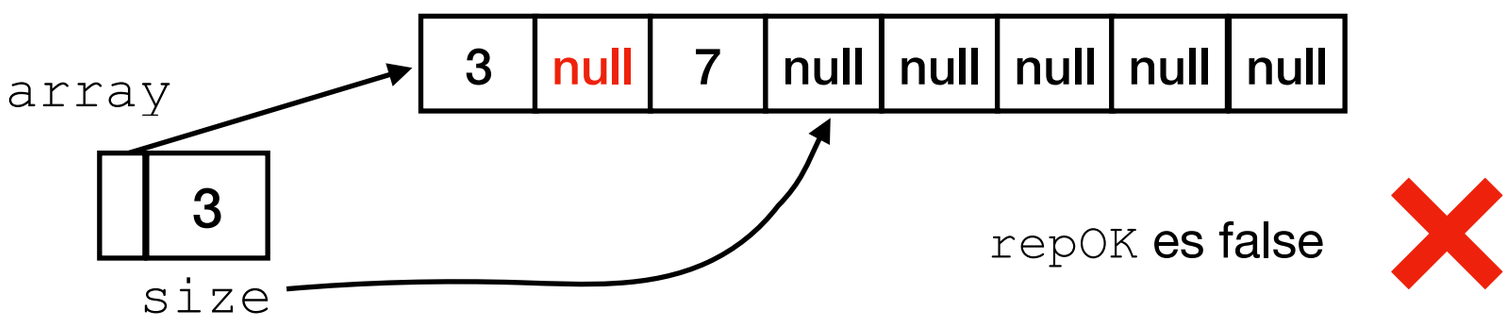
Invariante de representación



Invariante de representación







Invariante de representación y testing

- Ejemplos: Usar `repOK` como aserción en el testing para verificar que los métodos siempre crean objetos que representan elementos válidos del TAD

```java
@Test
public void testConstructorCreatesEmptyStackRepOK()
{
    Stack<Integer> stack = new ArrayStack<Integer>();
    String abstractSeq = stack.toString();
    assertEquals("[]", abstractSeq);
    assertTrue(stack.repOK());
}
```

Invariante de representación y testing

- Ejemplos: Usar `repOK` como aserción en el testing para verificar que los métodos siempre crean objetos que representan elementos válidos del TAD

```java
@Test
public void testConstructorCreatesEmptyStackRepOK()
{
    Stack<Integer> stack = new ArrayStack<Integer>();
    String abstractSeq = stack.toString();
    assertEquals("[]", abstractSeq);
    assertTrue(stack.repOK());
}
```

```java
@Test
public void testPushTwoElementsRepOK()
{
    Stack<Integer> stack = new ArrayStack<Integer>();
    stack.push(1);
    stack.push(2);
    String abstractSeq = stack.toString();
    assertTrue(stack.repOK());
}
```

Pilas sobre arreglos: Ventajas y desventajas

- Las operaciones implementadas son eficientes:
  - `top`, `size`, `isEmpty` son muy eficientes (de tiempo constante)
  - `push` y `pop` son bastante eficientes: la mayoría de las veces son de tiempo constante, salvo en unos pocos casos en los que deben agrandar/achicar el arreglo
    - Podríamos hacer `pop` constante si nunca achicamos el arreglo, a expensas de usar más memoria
      - Usualmente, la decisión de qué implementación es más conveniente depende de la aplicación particular
- Se puede desperdiciar memoria teniendo arreglos más grandes de lo necesario

Paquetes y visibilidad

- Java permite organizar el código en paquetes, de manera que es posible para el programador agrupar clases relacionadas en un mismo paquete
  - Esto es muy útil para organizar proyectos grandes, cuando tenemos decenas o cientos de clases
- El nombre del paquete al que pertenece la clase se define en la primera línea del archivo, por ejemplo, la siguiente línea indica que la clase pertenece a `tads.stack`:
  - `package tads.stack;`
- Para usar una clase desde otro paquete tenemos que importarla con `import`:
  - `import tads.stack.Stack;`
- La tabla a la derecha resume los distintos niveles de acceso para cada modificador, de acuerdo a si el elemento pertenece a la misma clase (Class), al mismo paquete (Package), es una subclase (Subclass), o está en otro paquete (World)
- Las clases en diferentes paquetes tienen otras restricciones de visibilidad que las clases de un mismo paquete
  - Ej.: elementos sin modificador (visibilidad de paquete) pueden accederse en clases del mismo paquete, pero no en clases de paquetes distintos

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Registros

- Un registro es simplemente una colección de campos
- Simplemente se usan para guardar valores, no hay lógica alguna para operar sobre los campos, ni restricciones para modificarlos
- Un registro es lo opuesto a una abstracción de datos (no hay representación para esconder, no hay métodos para operar)
- Podemos declarar registros en clases clases con campos públicos, o de visibilidad a nivel paquete
- Pero cuidado, si al usar el registro notamos que su operación requiere de alguna lógica, debemos convertirlo en una abstracción de datos

```
class Pair {
    // OVERVIEW: A record type
    int coeff;
    int exp;
    Pair(int c, int n) { coeff = c; exp = n; }
}
```

```
package tads.stack;

/**
 * Record that implements nodes of the linked stack.
 */
class Node<T>
{
    T item;
    Node<T> next;
}
```

Registros

- El ejemplo, tomado del libro Clean Code [2], ilustra que para clases que implementan registros no hay mucho para ganar usando ocultamiento de información
- En cambio, se obtiene un código más extenso, y no necesariamente más legible

Listing 6-7
address.java

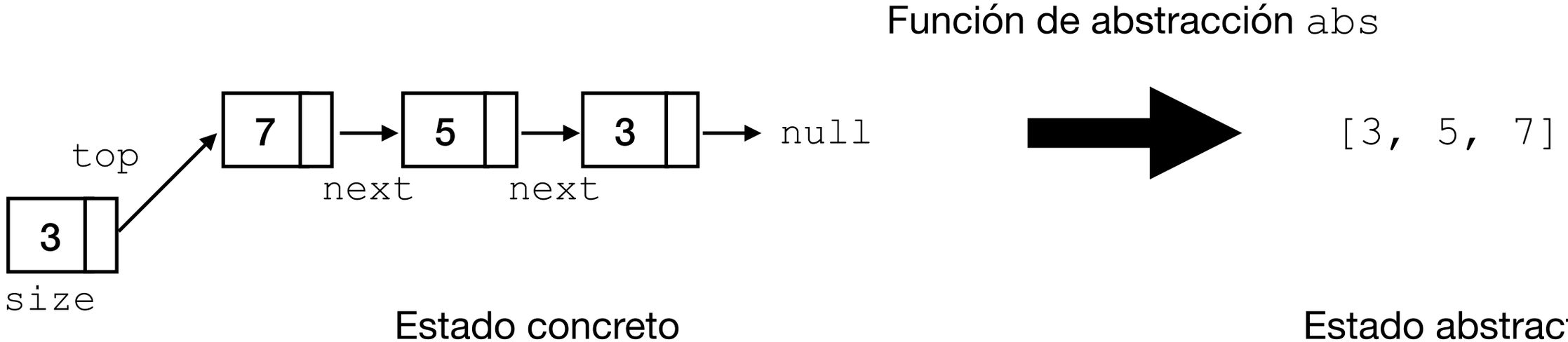
```
public class Address {
private String street;
private String streetExtra;
private String city;
private String state;
private String zip;

public Address(String street, String streetExtra,
               String city, String state, String zip) {
  this.street = street;
  this.streetExtra = streetExtra;
  this.city = city;
  this.state = state;
  this.zip = zip;
}

public String getStreet() {
  return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

[2] Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. 2008.

Implementación de Stack con listas enlazadas

```
public class LinkedStack<T> implements Stack<T>, Iterable<T>
{
    protected int size;      // size of the stack
    protected Node<T> top;   // top of stack
```

- Otra representación de pilas consiste en llevar una lista simplemente encadenada para guardar los elementos de la pila, y una referencia al primer nodo de esta lista (`top`)
- También llevaremos una variable entera (`size`) para contar la cantidad de elementos almacenados y hacer que la operación `size()` sea eficiente



Implementación de Stack con listas enlazadas: Operaciones

```java
/**
 * @post Creates an empty stack.
 *   More formally, it satisfies: this = [].
 */
public LinkedStack()
{
    top = null;
    size = 0;
}
```

Implementación de Stack con listas enlazadas: Operaciones

```java
/**
 * @post Creates an empty stack.
 *   More formally, it satisfies: this = [].
 */
public LinkedStack()
{
    top = null;
    size = 0;
}
```

```java
/**
 * @post Returns true iff the stack contains no elements.
 *   More formally, it satisfies: result = (this = []).
 */
public boolean isEmpty()
{
    return top == null;
}
```

Implementación de Stack con listas enlazadas: Operaciones

```java
/**
 * @post Creates an empty stack.
 *   More formally, it satisfies: this = [].
 */
public LinkedStack()
{
    top = null;
    size = 0;
}
```

```java
/**
 * @post Returns true iff the stack contains no elements.
 *   More formally, it satisfies: result = (this = []).
 */
public boolean isEmpty()
{
    return top == null;
}
```

```java
/**
 * @post Returns the number of elements in the stack.
 *   More formally, it satisfies: result = #this.
 */
public int size()
{
    return size;
}
```

Implementación de Stack con listas enlazadas: Operaciones

```java
/**
 * @post Adds element e to the top of the stack.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void push(T item)
{
    Node oldtop = top;
    top = new Node();
    top.item = item;
    top.next = oldtop;
    size++;
}
```

Implementación de Stack con listas enlazadas: Operaciones

```java
/**
 * @post Adds element e to the top of the stack.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void push(T item)
{
    Node oldtop = top;
    top = new Node();
    top.item = item;
    top.next = oldtop;
    size++;
}
```

```java
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Returns the item at the top of the stack.
 *   More formally, it satisfies:
 *     let this = s1 ++ [e] | result = e.
 */
public T top()
{
    if (isEmpty())
        throw new NoSuchElementException("Stack underflow");

    return top.item;
}
```

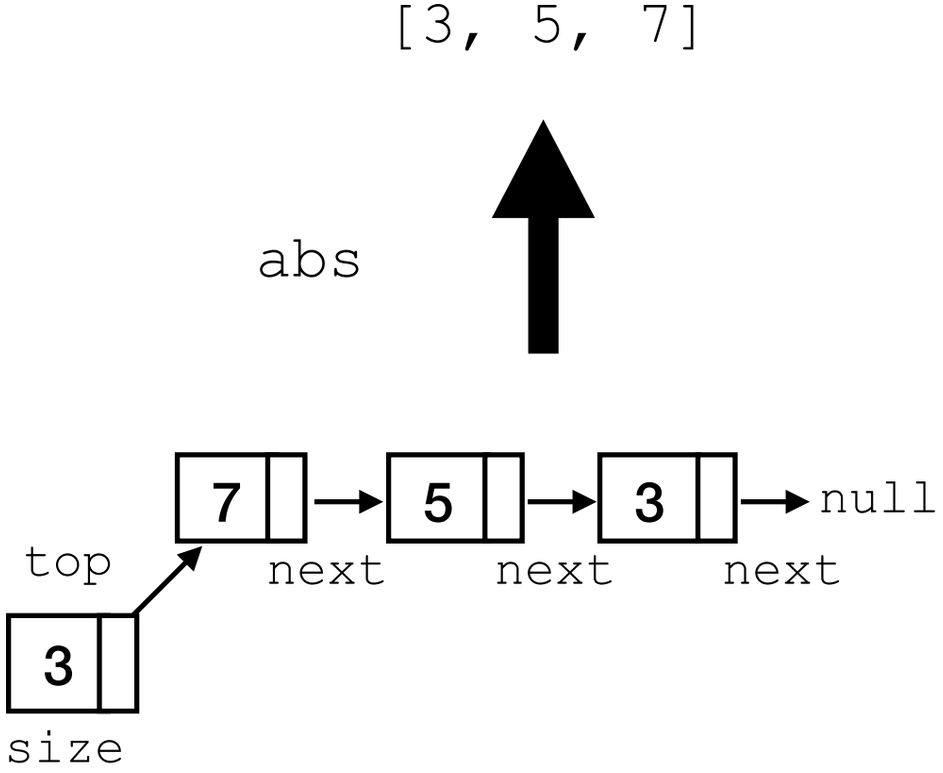
Implementación de Stack con listas enlazadas: Operaciones

```
/**
 * @post Adds element e to the top of the stack.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void push(T item)
{
    Node oldtop = top;
    top = new Node();
    top.item = item;
    top.next = oldtop;
    size++;
}
```

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Returns the item at the top of the stack.
 *   More formally, it satisfies:
 *     let this
 */
public T top()
{
    if (isEmpty
        throw n

    return top.
}
```

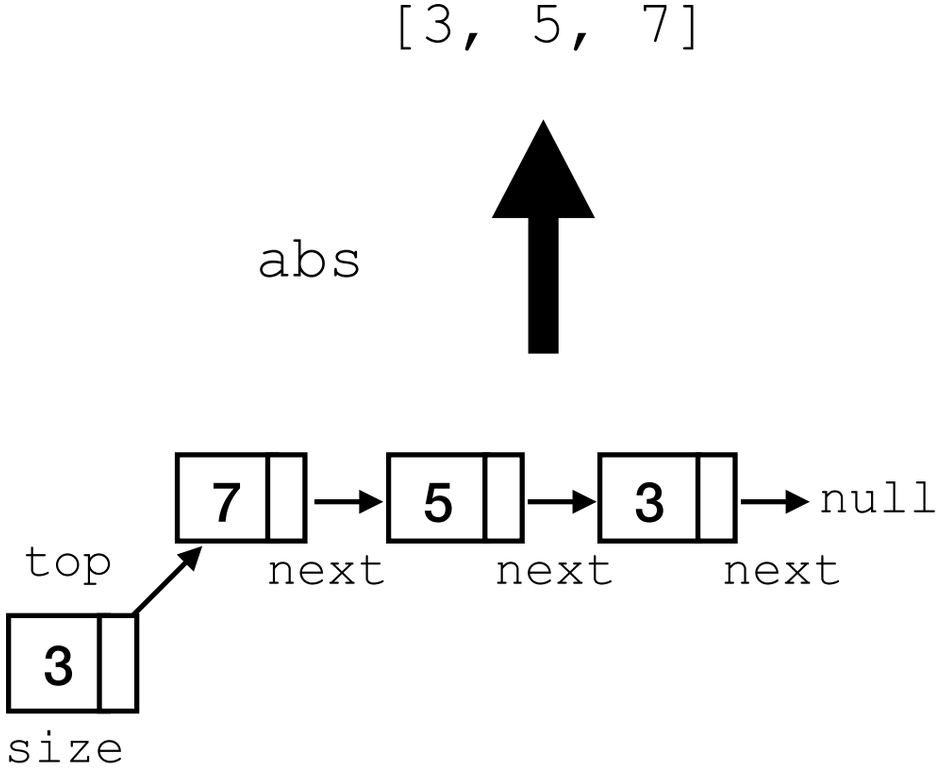
```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Removes and returns the item at the top of the stack.
 *   More formally, it satisfies:
 *     let old(this) = s1 ++ [e] |
 *       this = s1 && result = e.
 */
public T pop()
{
    if (isEmpty())
        throw new NoSuchElementException("Stack underflow");

    T item = top.item;          // save item to return
    top = top.next;             // delete top node
    size--;
    return item;                // return the saved item
}
```

Función de abstracción




```java
/**
 * @post Returns a string representation of the stack. Implements
 *   the abstraction function. Hence, it represents the stack as a
 *   sequence "[o1, o2,..., on]".
 */
public String toString() {
    // We start iterating from the top, which should be the last
    // element in string. Thus, we first use an aux stack to invert
    // the order of the elements.
    // TODO: This method traverses the stack twice and is not as
    // efficient as it should be. The efficiency can be improved by
    // using a doubly linked list implentation.
    Stack<T> aux = new LinkedStack<>();
    for (T item: this)
        aux.push(item);

    // Now print the aux stack.
    String res = "[";
    boolean first = true;
    for (T item: aux)
    {
        if (!first)
          res += ", ";
        res += item.toString();
        first = false;
    }
    res += "]";
    return res;
}
```

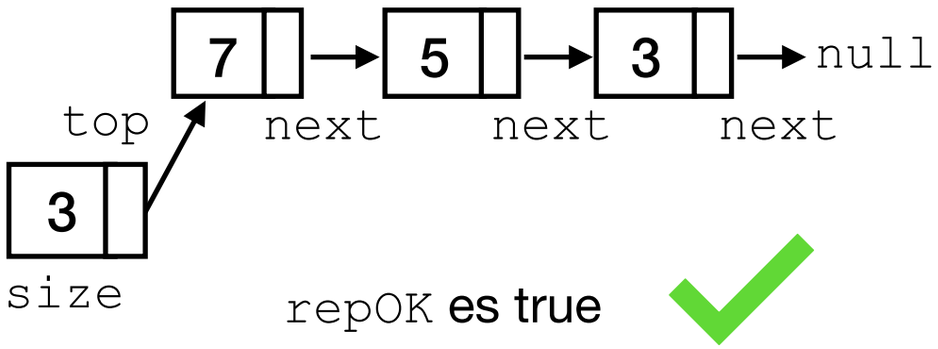
Función de abstracción




```java
/**
 * @post Returns a string representation of the stack. Implements
 *   the abstraction function. Hence, it represents the stack as a
 *   sequence "[o1, o2,..., on]".
 */
public String toString() {
    // We start iterating from the top, which should be the last
    // element in string. Thus, we first use an aux stack to invert
    // the order of the elements.
    // TODO: This method traverses the stack twice and is not as
    // efficient as it should be. The efficiency can be improved by
    // using a doubly linked list implentation.
    Stack<T> aux = new LinkedStack<>();
    for (T item: this)
        aux.push(item);

    // Now print the aux stack.
    String res = "[";
    boolean first = true;
    for (T item: aux)
    {
        if (!first)
            res += ", ";
        res += "]";
        return res;
}
```

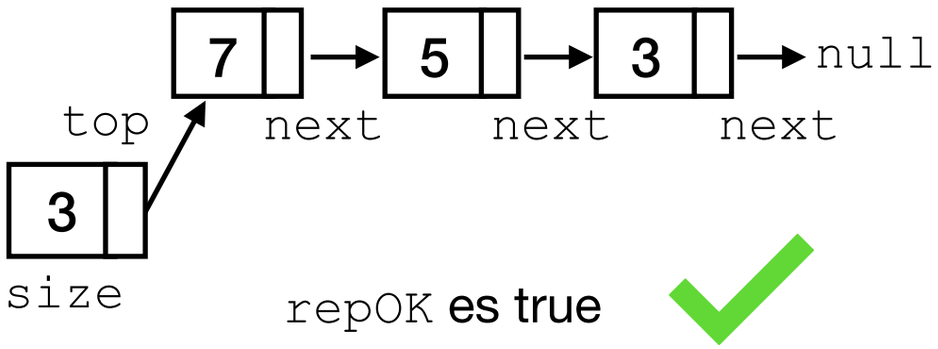
Ejercicio: ¿Cómo haría para evitar recorrer la pila dos veces? Ayuda: Piense en posibles modificaciones de la estructura encadenada

Invariante de representación

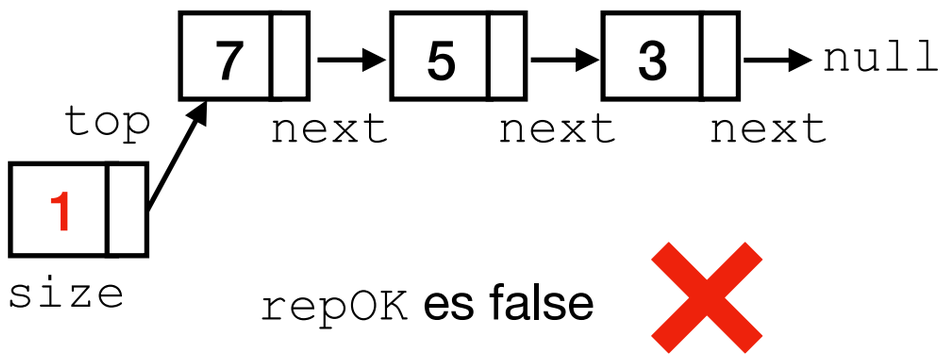



```java
/**
 * @post Returns true if and only if the structure is a
 *   valid stack.
 */
public boolean repOK() {
    if (size < 0)
        return false;
    // check internal consistency of instance variable size,
    // and that the stack is null terminated
    int numberOfNodes = 0;
    Node curr = top;
    while (curr != null && numberOfNodes <= size) {
        numberOfNodes++;
        curr = curr.next;
    }
    if (numberOfNodes != size || curr != null)
        return false;

    return true;
}
```

Invariante de representación



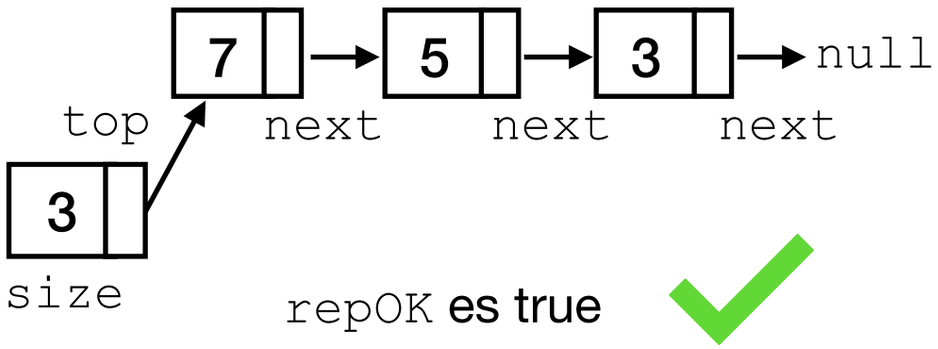
`repOK` es true

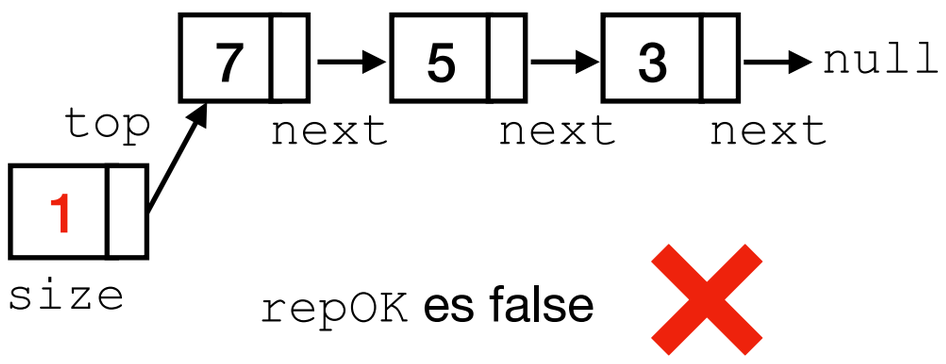


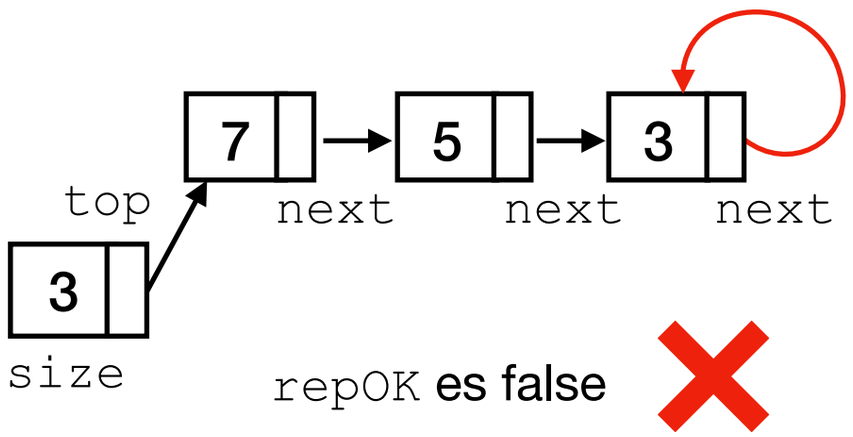
`repOK` es false

```java
/**
 * @post Returns true if and only if the structure is a
 *   valid stack.
 */
public boolean repOK() {
    if (size < 0)
        return false;
    // check internal consistency of instance variable size,
    // and that the stack is null terminated
    int numberOfNodes = 0;
    Node curr = top;
    while (curr != null && numberOfNodes <= size) {
        numberOfNodes++;
        curr = curr.next;
    }
    if (numberOfNodes != size || curr != null)
        return false;

    return true;
}
```

Invariante de representación








```java
/**
 * @post Returns true if and only if the structure is a
 *   valid stack.
 */
public boolean repOK() {
    if (size < 0)
        return false;
    // check internal consistency of instance variable size,
    // and that the stack is null terminated
    int numberOfNodes = 0;
    Node curr = top;
    while (curr != null && numberOfNodes <= size) {
        numberOfNodes++;
        curr = curr.next;
    }
    if (numberOfNodes != size || curr != null)
        return false;

    return true;
}
```

Stack con listas enlazadas: Tests

- Debido a que escribimos los tests basándonos en el TAD secuencia, los tests para la nueva implementación son casi idénticos a los de la implementación anterior
  - Y los podemos reutilizar para testear la nueva implementación
- Sólo debemos reemplazar la construcción de los objetos de `ArrayStack` por objetos de `LinkedStack`

```java
@Test
public void testPushTwoElementsRepOK()
{
    Stack<Integer> stack = new ArrayStack<Integer>();
    stack.push(1);
    stack.push(2);
    String abstractSeq = stack.toString();
    assertTrue(stack.repOK());
}
```

```java
@Test
public void testPushTwoElementsRepOK()
{
    Stack<Integer> stack = new LinkedStack<Integer>();
    stack.push(1);
    stack.push(2);
    String abstractSeq = stack.toString();
    assertTrue(stack.repOK());
}
```

- Es posible evitar la duplicación de código creando tests parametrizados [3]

[3] https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

Pilas sobre listas enlazadas: Ventajas y desventajas

- Las operaciones implementadas son eficientes:
  - `top`, `size`, `isEmpty`, `push` y `pop` son todas de tiempo constante
  - Se guarda solo la memoria necesaria para los elementos almacenados en la pila
- Cada elemento ocupa más espacio (hay que guardar el valor y un puntero al siguiente)

Aplicaciones de las Pilas

- Las pilas tienen diversas aplicaciones en computación, por ejemplo:
  - Implementar la llamada a rutinas en la mayoría de los lenguajes de programación (eso hace posible la recursión)
  - Chequear el balanceo de símbolos (paréntesis, etc) se puede hacer fácilmente utilizando una pila.
  - Los algoritmos para evaluar expresiones pueden ser implementados usando pilas.

TAD Queue

- Una Queue modela una secuencia de elementos que se comporta como una cola
  - Es decir, para una Queue se cumple que: $this = [o_1, o_2, ..., o_n]$

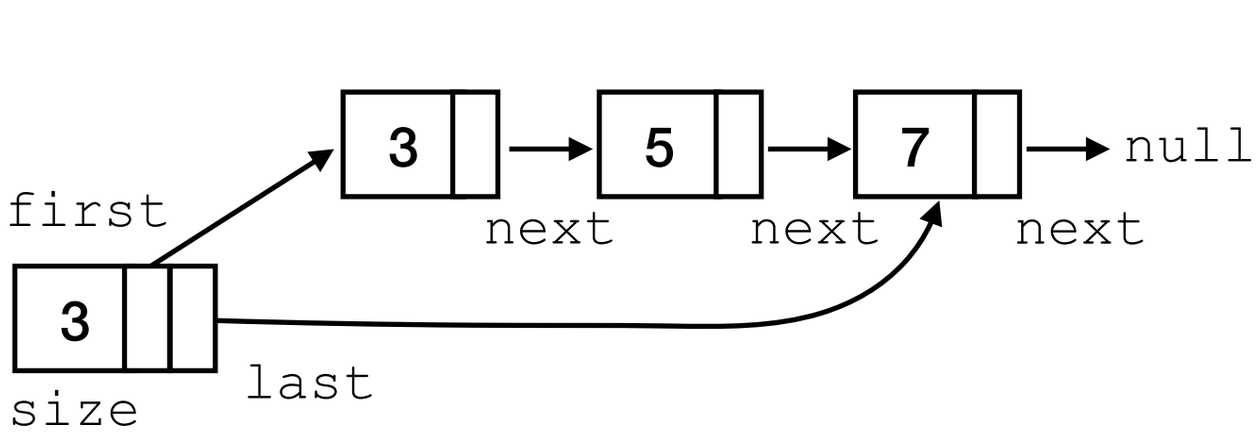
```java
/**
 * Queue represents unbounded, first-in-first-out (FIFO)
 * queue objects of type T.
 *
 * A typical Queue is a sequence [o1, o2,..., on]; we
 * denote this by: this = [o1, o2,..., on].
 *
 * The methods use equals to determine equality of elements.
 */
public interface Queue<T> extends Iterable<T>
```

- Sus operaciones típicas son: agregar un elemento al final de la cola, quitar el elemento al inicio de la cola, y consultar el elemento al inicio
- Decimos que la política de inserción y eliminación de las colas es first-in-first-out (el primer elemento que entra es el primero que sale)

Implementación de colas con listas enlazadas

- Como dijimos anteriormente, las operaciones de colas requieren insertar en un extremo de la cola (al inicio), y eliminar en el otro (al final)
- Entonces, para implementar las operaciones eficientemente tenemos que llevar referencias al primer y al último elemento de la cola
  - `first` y `last`, respectivamente

```
public class LinkedQueue<T> implements Queue<T>
{
    protected int size;          // number of elements on queue
    protected Node<T> first;     // beginning of queue
    protected Node<T> last;      // end of queue
```



Estado concreto

Función de abstracción `abs`

`[3, 5, 7]`

Estado abstracto

Implementación de colas con listas enlazadas: Operaciones

```java
/**
 * @post Creates an empty queue.
 *   More formally, it satisfies: this = [].
 */
public LinkedQueue()
{
    first = null;
    last = null;
    size = 0;
}
```

Implementación de colas con listas enlazadas: Operaciones

```java
/**
 * @post Creates an empty queue.
 *   More formally, it satisfies: this = [].
 */
public LinkedQueue()
{
    first = null;
    last = null;
    size = 0;
}
```

```java
/**
 * @post Returns the number of elements in the queue.
 *   More formally, it satisfies: result = #this.
 */
public int size() {
    return size;
}
```

Implementación de colas con listas enlazadas: Operaciones

```java
/**
 * @post Creates an empty queue.
 *   More formally, it satisfies: this = [].
 */
public LinkedQueue()
{
    first = null;
    last = null;
    size = 0;
}
```

```java
/**
 * @post Returns the number of elements in the queue.
 *   More formally, it satisfies: result = #this.
 */
public int size() {
    return size;
}
```

```java
/**
 * @post Returns true iff the queue contains no elements.
 *   More formally, it satisfies: result = #this = 0.
 */
public boolean isEmpty() {
    return first == null;
}
```

Implementación de colas con listas enlazadas: Operaciones

```java
/**
 * @post Adds element e to the end of the queue.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void enqueue(T e) {
    Node oldlast = last;
    last = new Node();
    last.item = e;
    last.next = null;
    if (isEmpty())
        first = last;
    else
        oldlast.next = last;
    size++;
}
```

Implementación de colas con listas enlazadas: Operaciones

```
/**
 * @post Adds element e to the end of the queue.
 *   More f
 */
public void
    Node ol
    last =
    last.it
    last.ne
    if (isE
        fir
    else
        old
    size++;
}
```

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Removes and returns the item at the beggining of the queue.
 *   More formally, it satisfies:
 *     let old(this) = [e] ++ s1 |
 *         this = s1 && result = e.
 */
public T dequeue() {
    if (isEmpty())
        throw new NoSuchElementException("Queue underflow");

    T item = first.item;
    first = first.next;
    size--;
    if (isEmpty())
        last = null;

    return item;
}
```

Implementación de colas con listas enlazadas: Operaciones

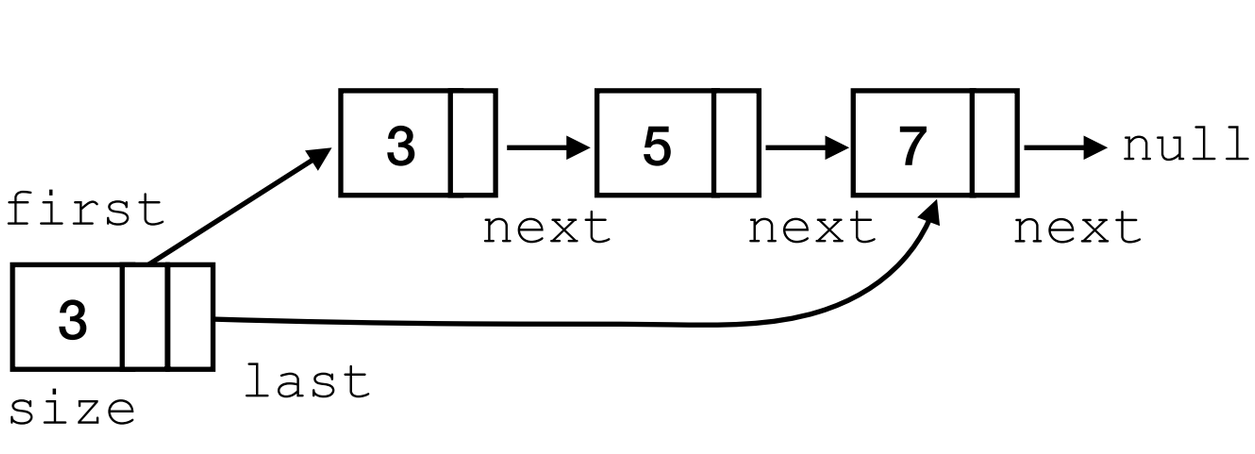
```
/**
 * @post Adds element e to the end of the queue.
 *   More f
 */
public void
    Node ol
    last =
    last.it
    last.ne
    if (isE
        fir
    else
        old
    size++;
}
```

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Removes and returns the item at the beggining of the queue.
 *   More formally, it satisfies:
 *     let old(this) = [e] ++ s1 |
 *         this = s1 && result = e.
 */
public T dequeue() {
    if (isEmpty())
        throw new NoSu

    T item = first.ite
    first = first.next
    size--;
    if (isEmpty())
        last = null;

    return item;
}
```

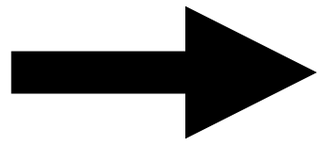
```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Returns the item at the beggining of the queue.
 *   More formally, it satisfies:
 *     let this = [e] ++ s1 | result = e.
 */
public T peek() {
    if (isEmpty())
        throw new NoSuchElementException("Queue underflow");

    return first.item;
}
```

Función de abstracción

```java
/**
 * @post Returns a string representation of the queue. Implements
 *   the abstraction function. Hence, it represents the queue as a
 *   sequence "[o1, o2,..., on]".
 */
public String toString() {
    String res = "[";
    Node<T> curr = first;
    while (curr != null) {
        res += curr.item.toString();
        if (curr.next != null)
          res += ", ";
        curr = curr.next;
    }
    res += "]";

    return res;
}
```



Función de abstracción `abs`

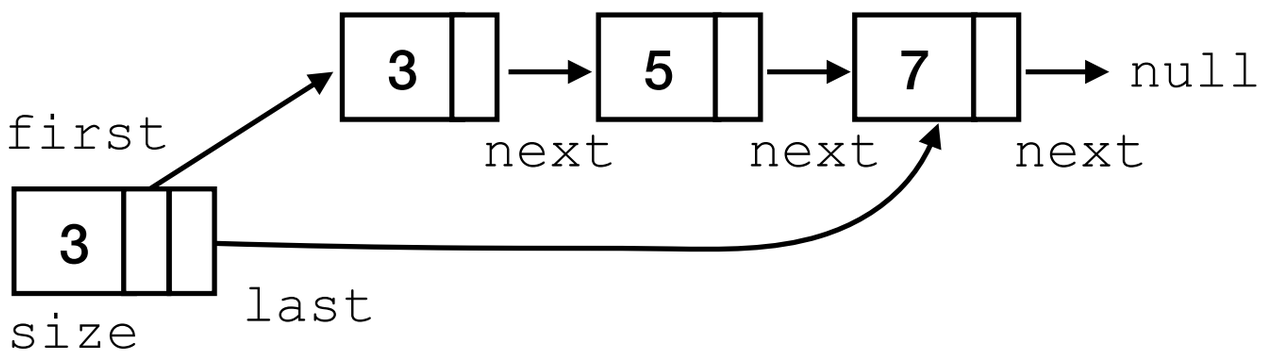


`[3, 5, 7]`

Invariante de representación

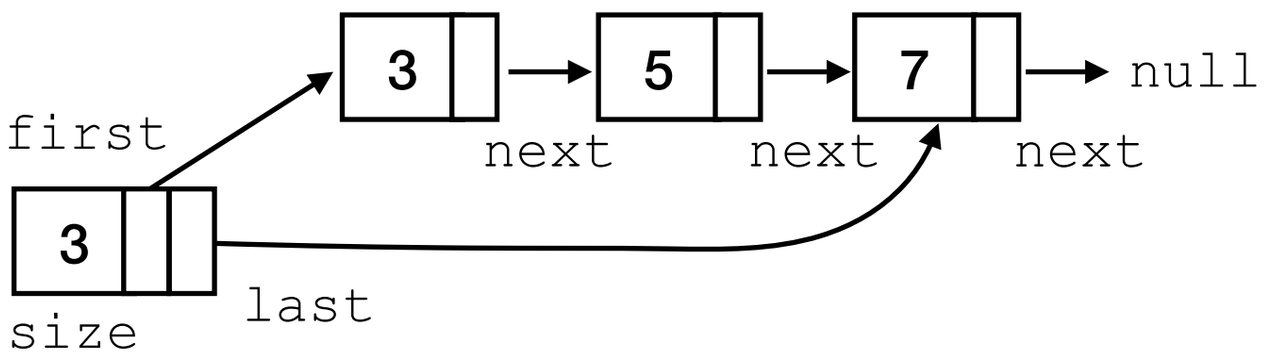
```java
/**
 * @post Returns true if and only if the structure is a
 *   valid queue.
 */
public boolean repOK() {
    if (size < 0)
        return false;

    // check internal consistency of instance variables size, last,
    // and that the list is null terminated
    int numberOfNodes = 0;
    Node<T> prev = null, curr = first;
    while (curr != null && numberOfNodes <= size && prev != last) {
        prev = curr;
        curr = curr.next;
        numberOfNodes++;
    }

    if (numberOfNodes != size || curr != null || prev != last)
        return false;

    return true;
}
```

Invariante de representación

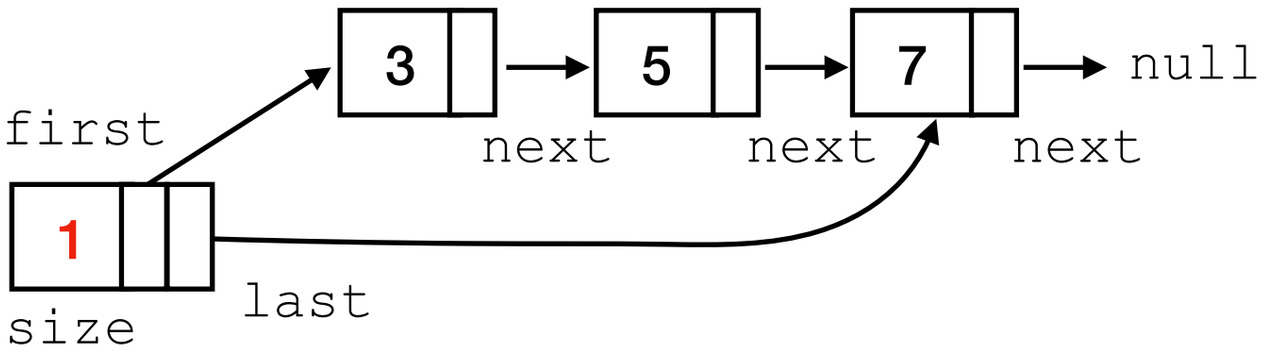


`repOK` es true

Invariante de representación

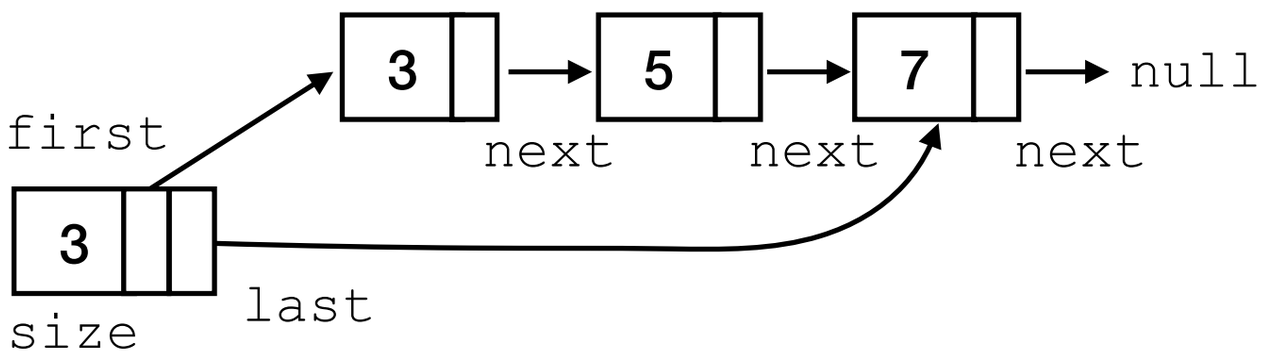


`repOK` es true

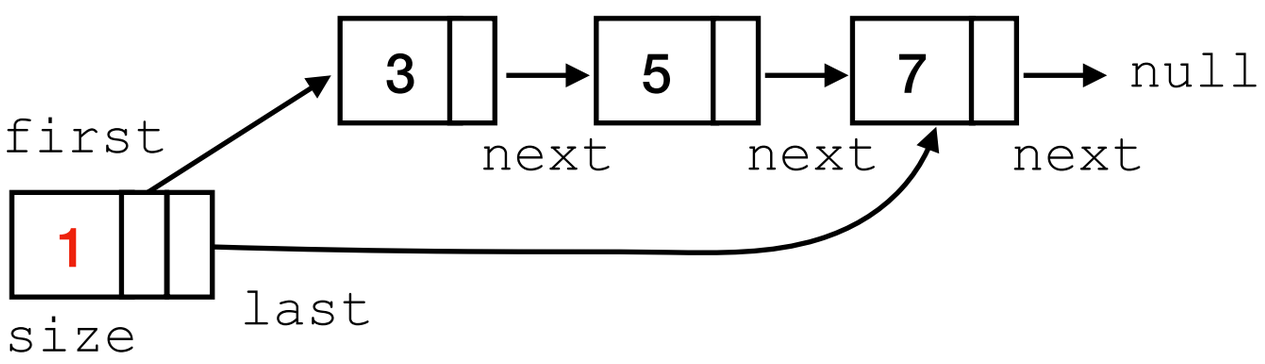


`repOK` es false

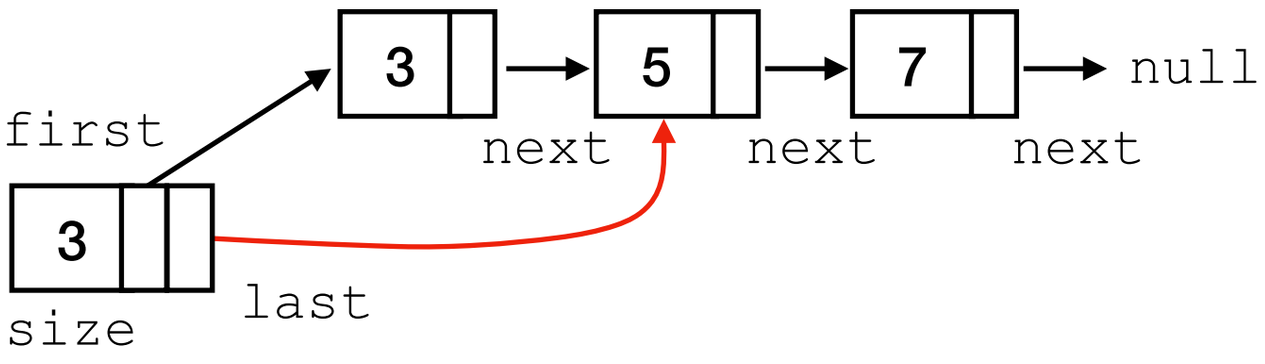
Invariante de representación



`repOK` es true



`repOK` es false



`repOK` es false

Colas con listas enlazadas: Tests

```java
@Test
public void testConstructorCreatesEmptyQueue()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    assertEquals("[]", queue.toString());
    assertTrue(queue.repOK());
}
```

Colas con listas enlazadas: Tests

```java
@Test
public void testConstructorCreatesEmptyQueue()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    assertEquals("[]", queue.toString());
    assertTrue(queue.repOK());
}
```

```java
@Test
public void testEnqueueTwoElements()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    assertEquals("[1, 2]", queue.toString());
    assertTrue(queue.repOK());
}
```

Colas con listas enlazadas: Tests

```java
@Test
public void testConstructorCreatesEmptyQueue()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    assertEquals("[]", queue.toString());
    assertTrue(queue.repOK());
}
```

```java
@Test
public void testEnqueueTwoElements()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    assertEquals("[1, 2]", queue.toString());
    assertTrue(queue.repOK());
}
```

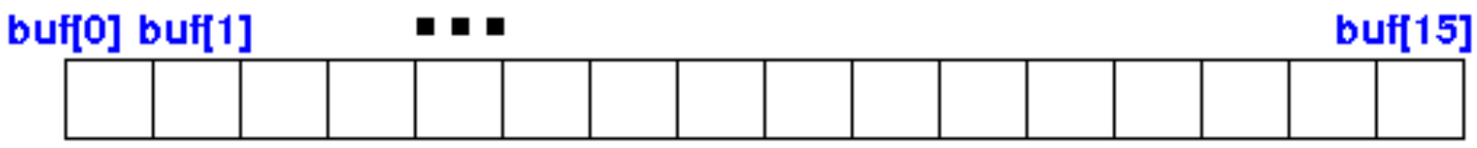
```java
@Test
public void testDequeueWithTwoElements()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    queue.dequeue();
    assertEquals("[2]", queue.toString());
    assertTrue(queue.repOK());
}
```

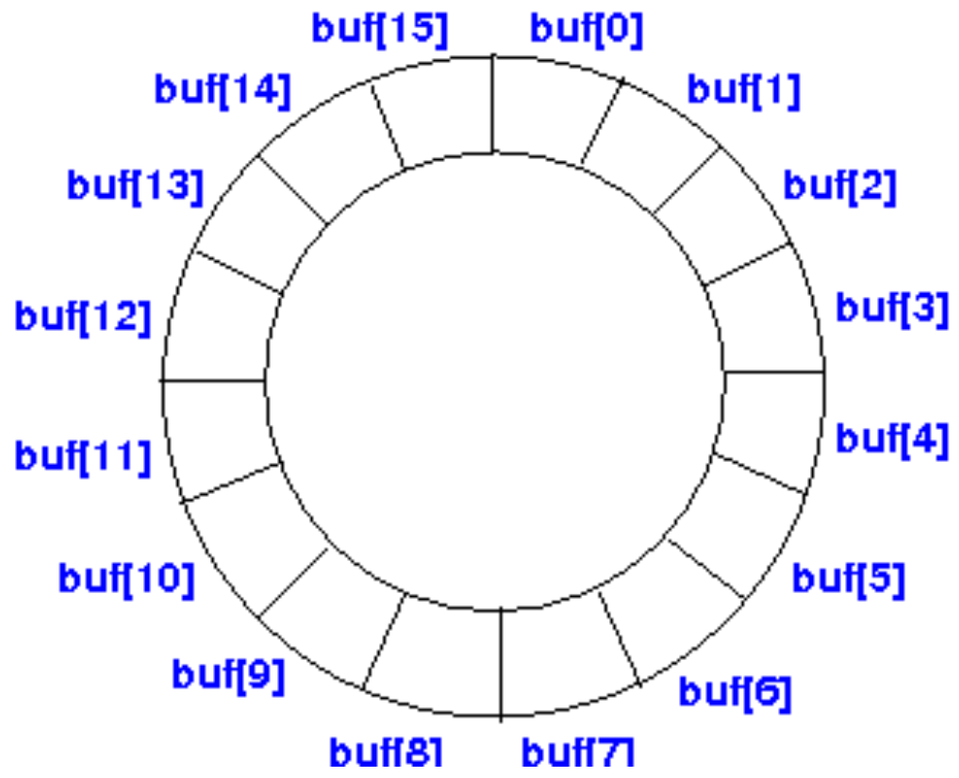
Colas sobre listas enlazadas: Ventajas y desventajas

- Todas las operaciones implementadas son de tiempo constante
- Se guarda solo la memoria necesaria para los elementos almacenados en la cola
- Cada elemento ocupa más espacio (hay que guardar el valor y un puntero al siguiente)

Implementación de colas con arreglos circulares

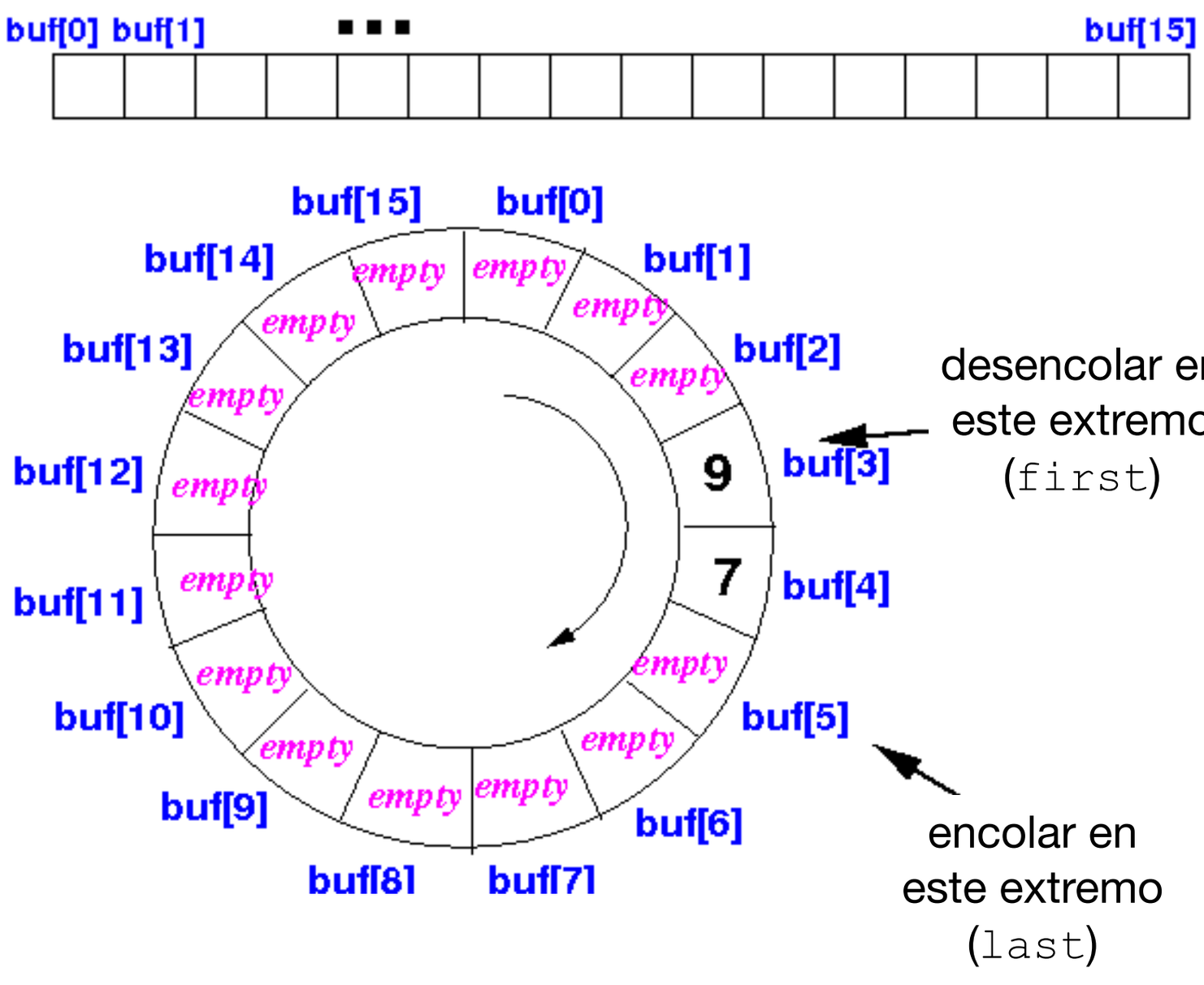
- Otra opción para implementar colas eficientemente es usar un arreglo, y tratarlo como si fuera circular (ver Figura)
- Al igual que con las pilas, vamos a ir agrandando el arreglo circular a medida que sea necesario para insertar más elementos





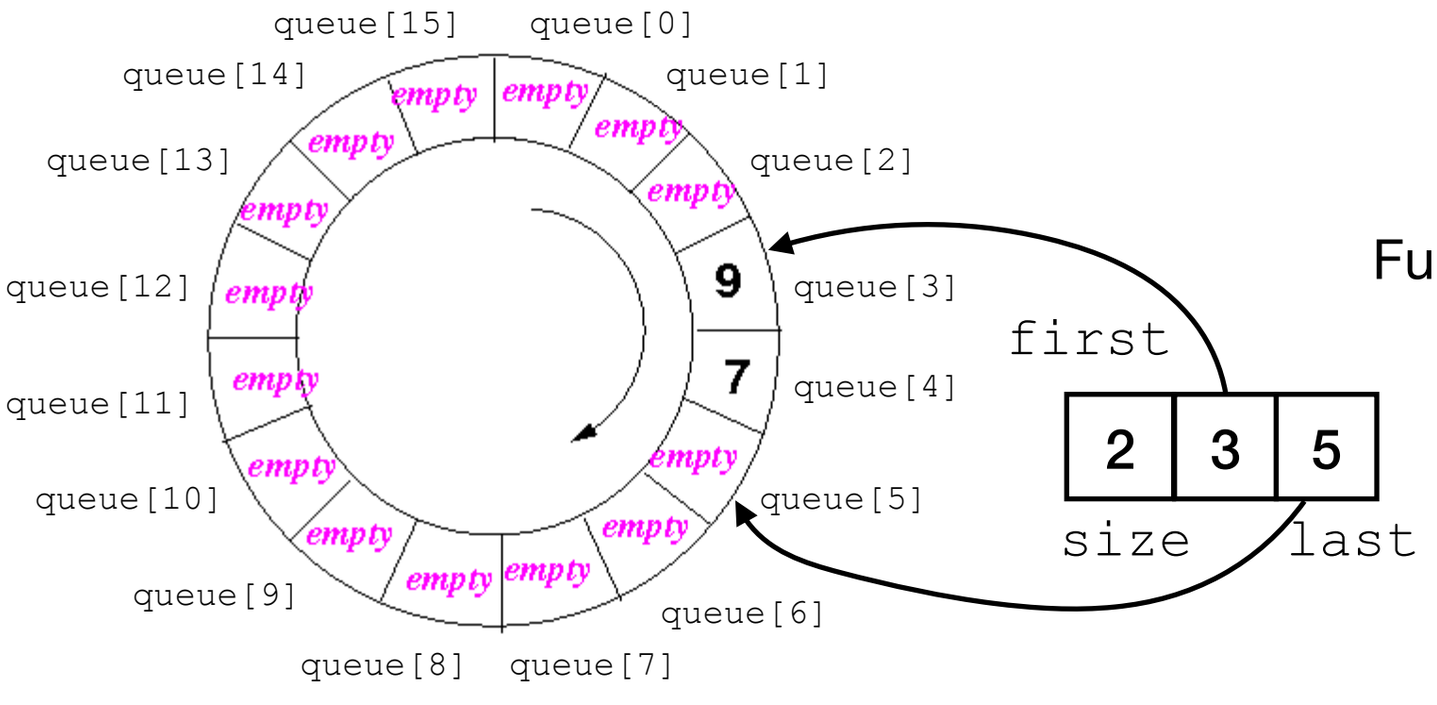
Implementación de colas con arreglos circulares

- Otra opción para implementar colas eficientemente es usar un arreglo, y tratarlo como si fuera circular (ver Figura)
- Vamos a llevar dos índices, `first` y `last`
- La secuencia reprsentada por la cola consiste de los elementos entre `first` y `last - 1`
- Encolamos un elemento en el índice `last`
- Desencolamos un elemento en el índice `first`



Implementación de colas con arreglos circulares

```java
public class CircularQueue<T> implements Queue<T> {
    private static final int INIT_CAPACITY = 8;

    protected T[] queue;        // queue elements
    protected int size;           // number of elements on queue
    protected int first;       // index of first element of queue
    protected int last;        // index of next available slot
```



Estado concreto

Función de abstracción `abs`

`[9, 7]`

Estado abstracto

Implementación de colas con arreglos circulares: Operaciones

```java
/**
 * @post Creates an empty queue.
 *   More formally, it satisfies: this = [].
 */
public CircularQueue() {
    queue = (T[]) new Object[INIT_CAPACITY];
    size = 0;
    first = 0;
    last = 0;
}
```

Implementación de colas con arreglos circulares: Operaciones

```java
/**
 * @post Creates an empty queue.
 *   More formally, it satisfies: this = [].
 */
public CircularQueue() {
    queue = (T[]) new Object[INIT_CAPACITY];
    size = 0;
    first = 0;
    last = 0;
}
```

```java
/**
 * @post Returns true iff the queue contains no elements.
 *   More formally, it satisfies: result = #this = 0.
 */
public boolean isEmpty() {
    return size == 0;
}
```

Implementación de colas con arreglos circulares: Operaciones

```java
/**
 * @post Creates an empty queue.
 *   More formally, it satisfies: this = [].
 */
public CircularQueue() {
    queue = (T[]) new Object[INIT_CAPACITY];
    size = 0;
    first = 0;
    last = 0;
}
```

```java
/**
 * @post Returns true iff the queue contains no elements.
 *   More formally, it satisfies: result = #this = 0.
 */
public boolean isEmpty() {
    return size == 0;
}
```

```java
/**
 * @post Returns the number of elements in the queue.
 *   More formally, it satisfies: result = #this.
 */
public int size() {
    return size;
}
```

Implementación de colas con arreglos circulares: Operaciones

```
/**
 * @post Adds element e to the end of the queue.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void enqueue(T item) {
    // double size of array if necessary and recopy to front of array
    if (size == queue.length)
        resize(2* queue.length);   // double size of array if necessary

    queue[last++] = item;                          // add item
    if (last == queue.length)
        last = 0;           // wrap-around
    size++;
}
```

Implementación de colas con arreglos circulares: Operaciones

```java
/**
 * @post Adds element e to the end of the queue.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void enqueue(T item) {
    // double size of array if n
    if (size == queue.length)
        resize(2* queue.length);

    queue[last++] = item;
    if (last == queue.length)
        last = 0;                // wr
    size++;
}
```

```java
/**
 * @pre capacity > size() (throws IllegalArgumentException).
 * @post Resize the underlying array to the given capacity.
 */
private void resize(int capacity) {
    if (capacity <= size())
        throw new IllegalArgumentException("The new array" +
                "must be larger than the current size: " + size());

    T[] copy = (T[]) new Object[capacity];
    for (int i = 0; i < size; i++) {
        copy[i] = queue[(first + i) % queue.length];
    }
    queue = copy;
    first = 0;
    last  = size;
}
```

Implementación de colas con arreglos circulares: Operaciones

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Removes and returns the item at the beggining of the queue.
 *   More formally, it satisfies:
 *     let old(this) = [e] ++ s1 |
 *       this = s1 && result = e.
 */
public T dequeue() {
    if (isEmpty())
        throw new NoSuchElementException("Queue underflow");

    T item = queue[first];
    queue[first] = null;                    // to avoid loitering
    size--;
    first++;
    if (first == queue.length)
        first = 0;           // wrap-around
    // shrink size of array if necessary
    if (size > 0 && size == queue.length/4)
        resize(queue.length/2);
    return item;
}
```

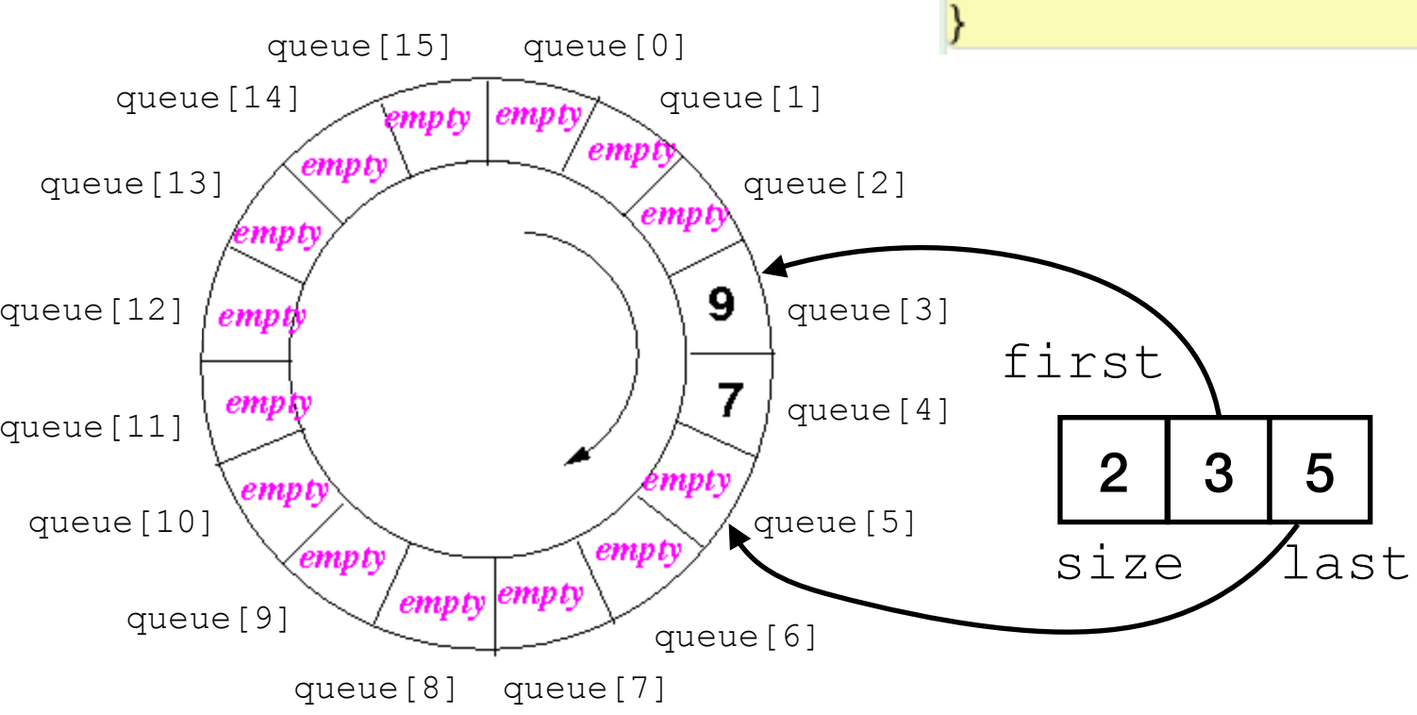
Implementación de colas con arreglos circulares: Operaciones

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Removes and returns the item at the beggining of the queue.
 *   More formally, it satisfies:
 *     let old(this) = [e] ++ s1 |
 *       this = s1 && resu
 */
public T dequeue() {
    if (isEmpty())
        throw new NoSuchEl

    T item = queue[first];
    queue[first] = null;
    size--;
    first++;
    if (first == queue.length)
        first = 0;           // wrap-around
    // shrink size of array if necessary
    if (size > 0 && size == queue.length/4)
        resize(queue.length/2);
    return item;
}
```

```
/**
 * @pre !isEmpty() (throws NoSuchElementException)
 * @post Returns the item at the beggining of the queue.
 *   More formally, it satisfies:
 *     let this = [e] ++ s1 | result = e.
 */
public T peek() {
    if (isEmpty()) throw new NoSuchElementException("Queue underflow");
    return queue[first];
}
```

Función de abstracción

```java
/**
 * @post Returns a string representation of the queue. Implements
 *   the abstraction function. Hence, it represents the queue as a
 *   sequence "[o1, o2,..., on]".
 */
public String toString() {
    String res = "[";
    for (int i = 0; i < size; i++) {
        res += queue[(first + i) % queue.length].toString();
        if (i < size -1)
            res += ", ";
    }
    res += "]";
    return res;
}
```



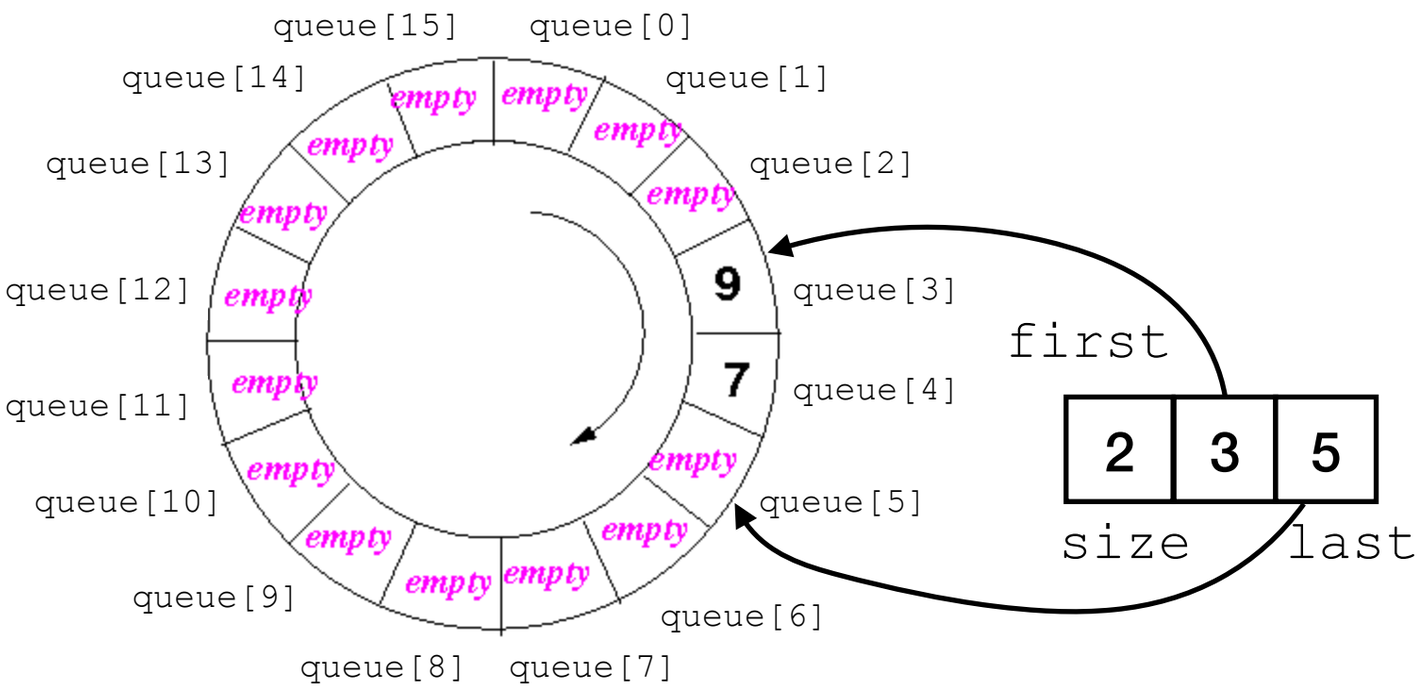
toString

"[9, 7]"

Invariante de representación

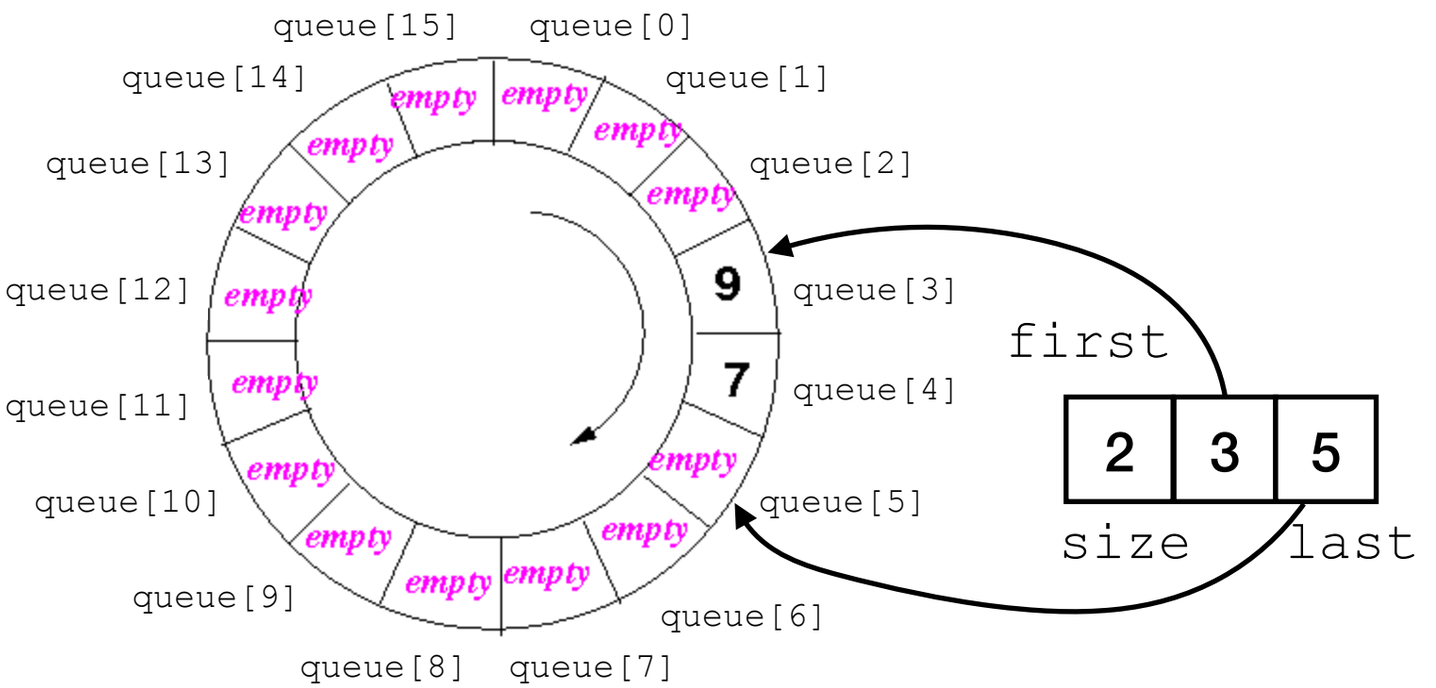
```java
/**
 * @post Returns true if and only if the structure is a
 *   valid queue.
 */
public boolean repOK() {
    if (first < 0 || first >= queue.length)
        return false;
    if (last < 0 || last >= queue.length)
        return false;
    if (size < 0 || size > queue.length)
        return false;

    int i = first;
    int count = size;
    while (count > 0) {
        if (queue[i] == null)
            return false;
        i = (i+1) % queue.length;
        count--;
    }
    if (i != last)
        return false;

    return true;
}
```

Invariante de representación

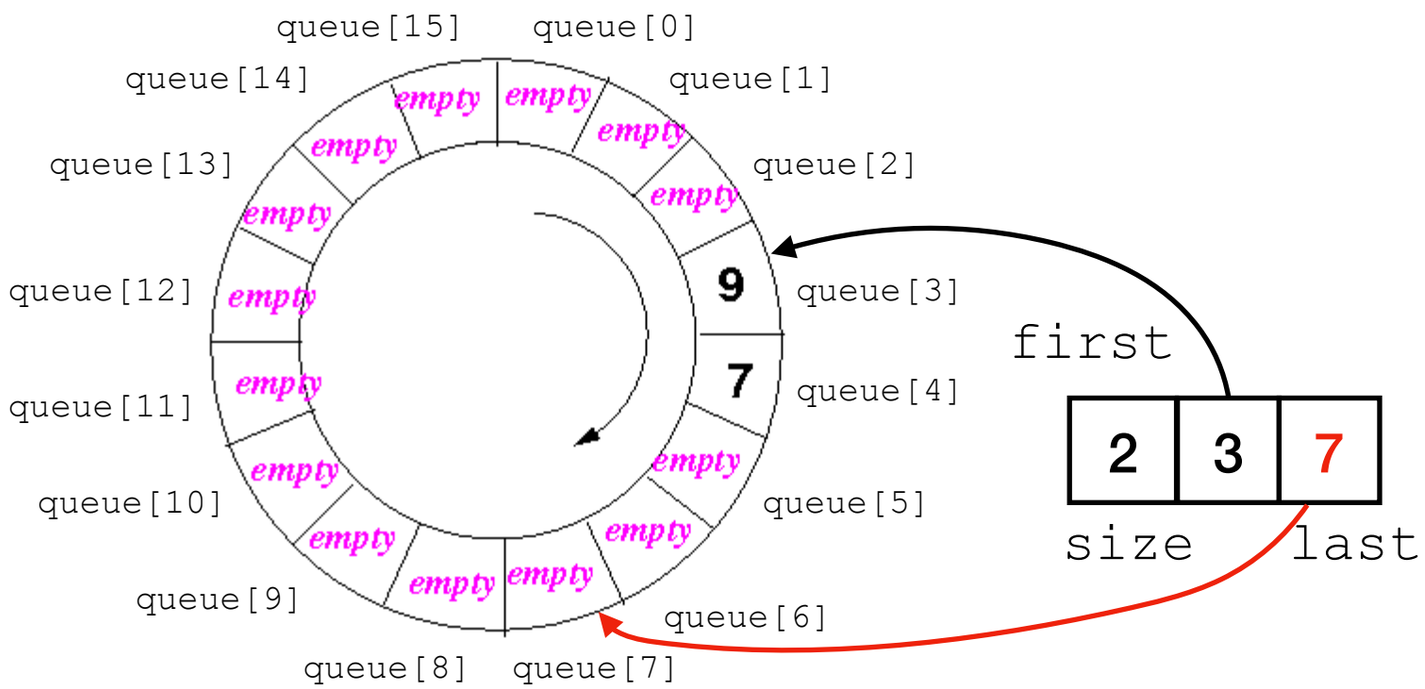


`repOK` es true

Invariante de representación



`repOK` es true ✓



`repOK` es false ✗

Colas con arreglos circulares: Tests

```java
@Test
public void testConstructorCreatesEmptyQueue()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    assertEquals("[]", queue.toString());
    assertTrue(queue.repOK());
}
```

Colas con arreglos circulares: Tests

```java
@Test
public void testConstructorCreatesEmptyQueue()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    assertEquals("[]", queue.toString());
    assertTrue(queue.repOK());
}
```

```java
@Test
public void testEnqueueTwoElements()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    assertEquals("[1, 2]", queue.toString());
    assertTrue(queue.repOK());
}
```

Colas con arreglos circulares: Tests

```java
@Test
public void testConstructorCreatesEmptyQueue()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    assertEquals("[]", queue.toString());
    assertTrue(queue.repOK());
}
```

```java
@Test
public void testEnqueueTwoElements()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    assertEquals("[1, 2]", queue.toString());
    assertTrue(queue.repOK());
}
```

```java
@Test
public void testDequeueWithTwoElements()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    queue.dequeue();
    assertEquals("[2]", queue.toString());
    assertTrue(queue.repOK());
}
```

Colas con arreglos circulares: Tests

- Debido a que escribimos los tests basándonos en el TAD secuencia, los tests para la nueva implementación son casi idénticos a los de la implementación anterior
  - Y los podemos reutilizar para testear la nueva implementación
- Sólo debemos reemplazar la construcción de los objetos de `LinkedQueue` por objetos de `CircularQueue`

```
@Test
public void testDequeueWithTwoElements()
{
    Queue<Integer> queue = new CircularQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    queue.dequeue();
    assertEquals("[2]", queue.toString());
    assertTrue(queue.repOK());
}
```

Colas con arreglos circulares: Tests

- Debido a que escribimos los tests basándonos en el TAD secuencia, los tests para la nueva implementación son casi idénticos a los de la implementación anterior
  - Y los podemos reutilizar para testear la nueva implementación
- Sólo debemos reemplazar la construcción de los objetos de `LinkedQueue` por objetos de `CircularQueue`

```java
@Test
public void testDequeueWithTwoElements()
{
    Queue<Integer> queue = new LinkedQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    queue.dequeue();
    assertEquals("[2]", queue.toString());
    assertTrue(queue.repOK());
}
```

```java
@Test
public void testDequeueWithTwoElements()
{
    Queue<Integer> queue = new CircularQueue<Integer>();
    queue.enqueue(1);
    queue.enqueue(2);
    queue.dequeue();
    assertEquals("[2]", queue.toString());
    assertTrue(queue.repOK());
}
```

Colas sobre arreglos circulares: Ventajas y desventajas

- Las operaciones implementadas son eficientes:
  - `top`, `size`, `isEmpty` son de tiempo constante
  - `enqueue` y `dequeue` son bastante eficientes: la mayoría de las veces son de tiempo constante, salvo en unos pocos casos en los que deben agrandar/achicar el arreglo
    - Podríamos hacer `dequeue` constante si nunca achicamos el arreglo, a expensas de usar más memoria
      - Usualmente, la decisión de qué implementación es más conveniente depende de la aplicación particular
- Se puede desperdiciar memoria teniendo arreglos más grandes de lo necesario

TAD List

- `List` modela una secuencia de elementos
  - Esto es, `this = [o1, o2,..., on]`

```java
/**
 * List represents unbounded sequences of objects of type T.
 *
 * A typical List is a sequence [o1, o2,..., on]; we
 * denote this by: this = [o1, o2,..., on].
 *
 * The methods use equals to determine equality of elements.
 */
public interface List<T> {
```

- Las listas tienen operaciones más generales que las pilas y colas: Permiten insertar en cualquier posición, eliminar de cualquier posición, obtener el elemento de una posición dada, etc.
- Veamos algunas de sus operaciones más típicas

TAD List: Operaciones

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Replaces the element at position index with e, and returns
 *   the element that was replaced.
 *   More formally, it satisfies:
 *     this[index].equals(e) && #this = #old(this) &&
 *     result.equals(old(this)[index]).
 */
public T set(int index, T e);
```

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Returns the element at position index in the list,
 *   More formally, it satisfies: result = this[index].
 */
public T get(int index);
```

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Inserts the element at position index with e.
 *   More formally, it satisfies:
 *   this[index].equals(e) && #this = #old(this) +1.
 */
public void add(int index, T e);
```

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Removes the element at position index.
 *   More formally, it satisfies:
 *   result = old(this)[index] && #this = #old(this) -1.
 */
public T remove(int index);
```

TAD List: Algunas operaciones

```
/**
 * @post Returns true iff the list contains element e.
 *   More formally, it satisfies:
 *   result = exists o | o in this && e.equals(o).
 */
public boolean contains(T e);
```

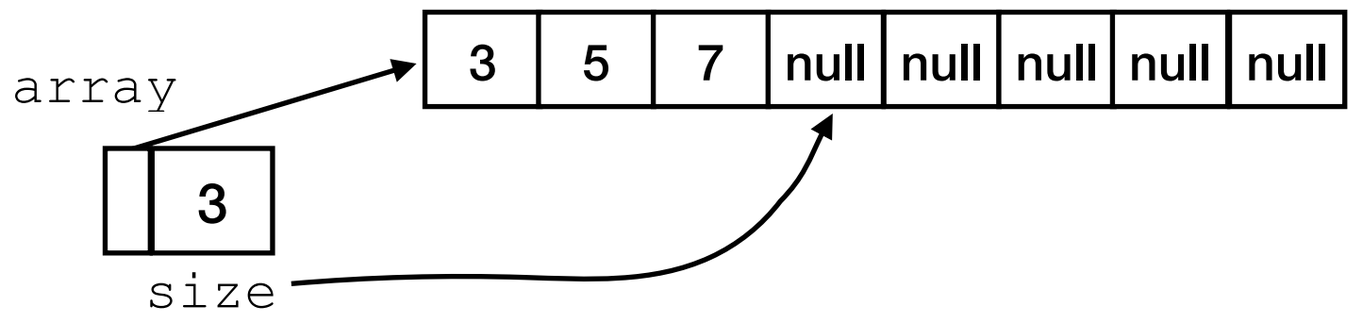
```
/**
 * @post Appends element e to the end of this list.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void add(T e);
```

```
/**
 * @post Removes the first occurrence of e from this list.
 *   If e is not in the list it does not modify the list.
 *   Returns true iff e is removed (result = e in old(list)).
 */
public boolean remove(T e);
```

```
/**
 * @post Returns the index of the first occurrence of e
 *   in the list, or -1 if this list does not contain e.
 *   More formally, it satisfies:
 *     result = -1 -> !(e in this) &&
 *     result != -1 -> this[result].equals(e).
 */
public int indexOf(T e);
```

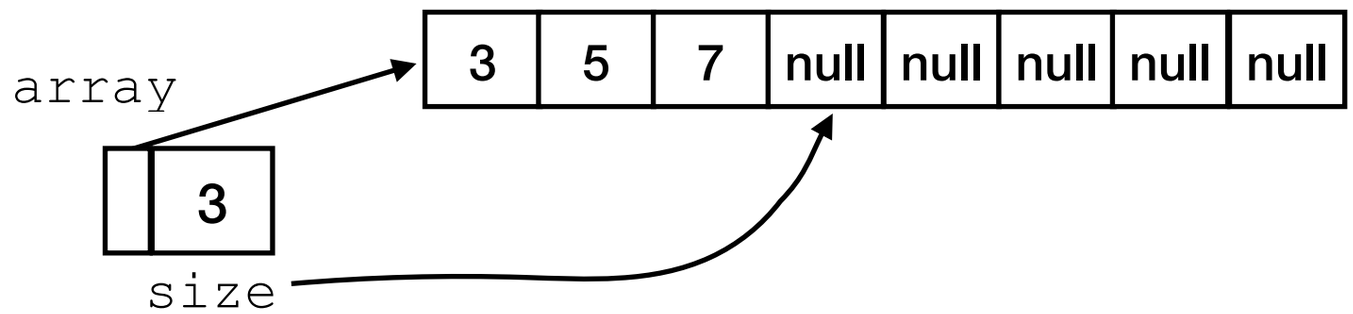
Implementación de listas con arreglos

- Para implementar listas con arreglos usamos la misma representación que para pilas: un arreglo que crece su tamaño bajo demanda



Implementación de listas con arreglos

- Para implementar listas con arreglos usamos la misma representación que para pilas: un arreglo que crece su tamaño bajo demanda



Ejercicio: Implementar listas con arreglos, con las operaciones mencionadas anteriormente, y la función de abstracción y el invariante de representación

Implementación de listas con arreglos

- La mayor ventaja de esta implementación es que `set` y `get` en una posición dada son operaciones de tiempo constante
  - Porque tenemos acceso directo a la posición `index` del arreglo
- En cambio, `add` y `remove` en una posición dada son menos eficientes: en el peor caso deben recorrer toda la lista
  - Cuando se agrega (elimina) un elemento en la posición 0 del arreglo hay que mover todos los elementos de la lista un lugar a la derecha (a la izquierda)

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Replaces the element at position index with e, and returns
 *   the element that was replaced.
 *   More formally, it satisfies:
 *     this[index].equals(e) && #this = #old(this) &&
 *     result.equals(old(this)[index]).
 */
public T set(int index, T e);
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Returns the element at position index in the list,
 *   More formally, it satisfies: result = this[index].
 */
public T get(int index);
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Inserts the element at position index with e.
 *   More formally, it satisfies:
 *   this[index].equals(e) && #this = #old(this) +1.
 */
public void add(int index, T e);
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Removes the element at position index.
 *   More formally, it satisfies:
 *   result = old(this)[index] && #this = #old(this) -1.
 */
public T remove(int index);
```

Implementación de listas con arreglos

- Agregar un elemento al final de la lista (`add`) es también de tiempo constante
- En cambio, dado un elemento `e`, ver si `e` pertenece o no a la lista, remover `e`, y obtener el índice en el que aparece `e`, son todas operaciones que requieren recorrer toda la lista en el peor caso (cuando el elemento no está)

```
/**
 * @post Appends element e to the end of this list.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void add(T e);
```

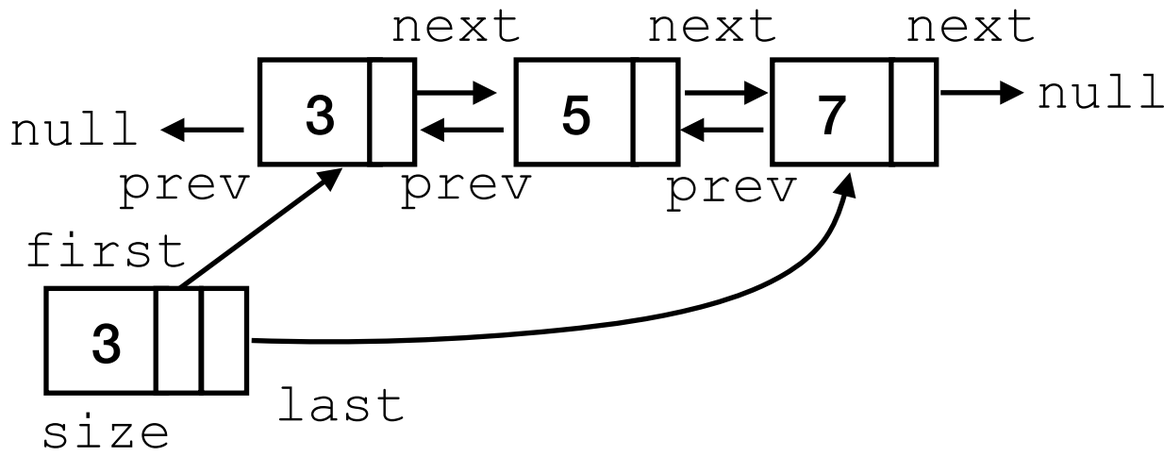
```
/**
 * @post Returns true iff the list contains element e.
 *   More formally, it satisfies:
 *   result = exists o | o in this && e.equals(o).
 */
public boolean contains(T e);
```

```
/**
 * @post Removes the first occurrence of e from this list.
 *   If e is not in the list it does not modify the list.
 *   Returns true iff e is removed (result = e in old(list)).
 */
public boolean remove(T e);
```

```
/**
 * @post Returns the index of the first occurrence of e
 *   in the list, or -1 if this list does not contain e.
 *   More formally, it satisfies:
 *     result = -1 -> !(e in this) &&
 *     result != -1 -> this[result].equals(e).
 */
public int indexOf(T e);
```

Implementación de List con listas doblemente encadenadas

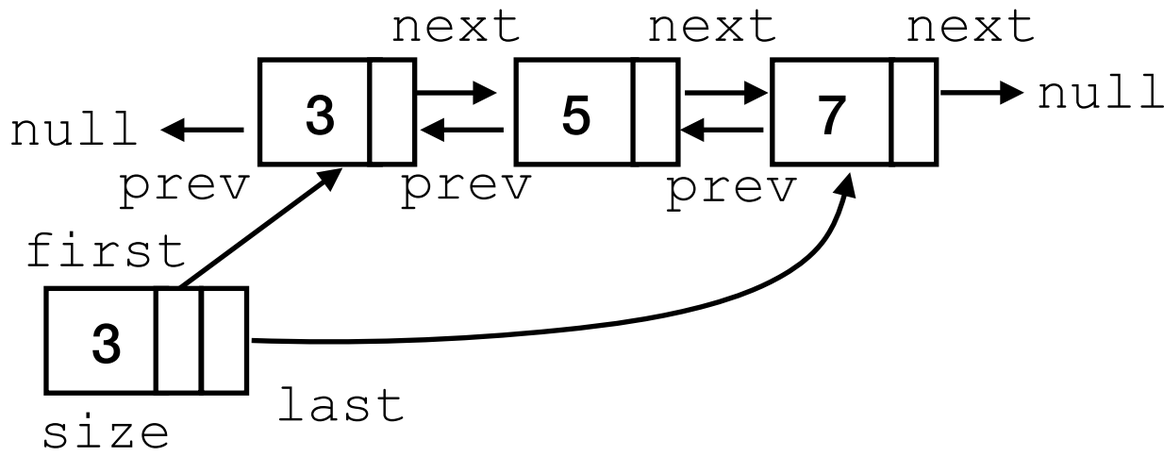
- Una alternativa interesante para implementar listas es usar listas doblemente encadenadas: es decir, cada elemento tiene un referencia al elemento siguiente y al elemento previo
- Vamos a mantener también una referencia al inicio de la lista y otra al final



- La ventaja de las listas doblemente encadenadas (vs. las simplemente encadenadas) es que podemos recorrer la lista en ambas direcciones
  - Esto permite implementar una operación que retorna los elementos de la lista en orden inverso (`reverse`) recorriendo la lista una única vez
    - En lugar de tener que recorrerla dos veces, por ejemplo, usando una pila para invertir el orden de la lista

Implementación de List con listas doblemente encadenadas

- Una alternativa interesante para implementar listas es usar listas doblemente encadenadas: es decir, cada elemento tiene un referencia al elemento siguiente y al elemento previo
- Vamos a mantener también una referencia al inicio de la lista y otra al final



- La ventaja de las listas doblemente encadenadas (vs. las simplemente encadenadas) es que podemos recorrer la lista en ambas direcciones

Ejercicio: Implementar List con listas doblemente encadenadas, con las operaciones mencionadas anteriormente, las operaciones de concatenar dos listas y reverse, la función de abstracción y el invariante de representación

Implementación de List con listas doblemente encadenadas

- La mayor ventaja de esta implementación es que `set`, `get`, `add` y `remove` en una posición dada deben recorrer la mitad de la lista en el peor caso
  - Podemos empezar el recorrido desde el inicio o desde el final dependiendo de la posición dada
  - El peor caso sería cuando el índice está cerca del medio
- Vamos a ver que recorrer la mitad de la lista no da una gran ganancia respecto de recorrerla entera (el tiempo de ejecución en el peor caso es lineal)

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Replaces the element at position index with e, and returns
 *   the element that was replaced.
 *   More formally, it satisfies:
 *     this[index].equals(e) && #this = #old(this) &&
 *     result.equals(old(this)[index]).
 */
public T set(int index, T e);
```

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Returns the element at position index in the list,
 *   More formally, it satisfies: result = this[index].
 */
public T get(int index);
```

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Inserts the element at position index with e.
 *   More formally, it satisfies:
 *   this[index].equals(e) && #this = #old(this) +1.
 */
public void add(int index, T e);
```

```
/**
 * @pre 0 <= index < size() (throws an IndexOutOfBoundsException)
 * @post Removes the element at position index.
 *   More formally, it satisfies:
 *   result = old(this)[index] && #this = #old(this) -1.
 */
public T remove(int index);
```

Implementación de List con listas doblemente encadenadas

- Agregar un elemento al final de la lista (`add`) es de tiempo constante (y agregar al inicio también, si existiera una operación `addFirst`)
- En cambio, dado un elemento `e`, ver si `e` pertenece o no a la lista, remover `e`, y obtener el índice en el que aparece `e`, son todas operaciones que requieren recorrer toda la lista en el peor caso (cuando el elemento no está)

```
/**
 * @post Appends element e to the end of this list.
 *   More formally, it satisfies: this = old(this) ++ [e].
 */
public void add(T e);
```

```
/**
 * @post Returns true iff the list contains element e.
 *   More formally, it satisfies:
 *     result = exists o | o in this && e.equals(o).
 */
public boolean contains(T e);
```

```
/**
 * @post Removes the first occurrence of e from this list.
 *   If e is not in the list it does not modify the list.
 *   Returns true iff e is removed (result = e in old(list)).
 */
public boolean remove(T e);
```

```
/**
 * @post Returns the index of the first occurrence of e
 *   in the list, or -1 if this list does not contain e.
 *   More formally, it satisfies:
 *     result = -1 -> !(e in this) &&
 *     result != -1 -> this[result].equals(e).
 */
public int indexOf(T e);
```

Actividades

- Leer el capítulo 5 del libro "Program Development in Java - Abstraction, Specification, and Object-Oriented Design". B. Liskov & J. Guttag. Addison-Wesley. 2001

- Leer el capítulo 1.3 del libro "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016

Bibliografía

- "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016
- "Program Development in Java - Abstraction, Specification, and Object-Oriented Design". B. Liskov & J. Guttag. Addison-Wesley. 2001
- Clean Code: A Handbook of Agile Software Craftsmanship (1st. ed.)". Robert C. Martin. "Prentice Hall. 2008.