

# ESTRUCTURAS LINEALES

Secuencias





# Estructuras Lineales

---

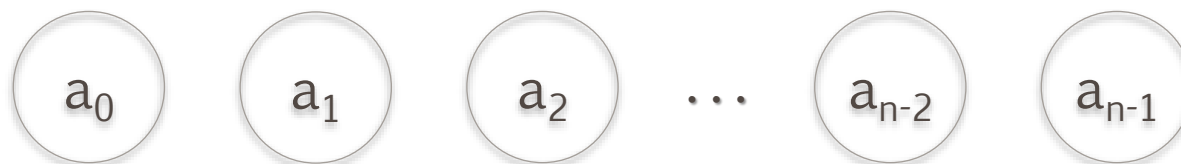
- Tipos de datos para secuencias de elementos
  - Listas (list)
  - Listas especializadas
    - Colas (queue): cola LIFO (pila), cola FIFO (cola), etc.
    - Dobles colas (double-ended queue )

# Listas (1)

---

- Listas

- Conjunto de elementos ordenados en sucesión (o secuencia). Admiten repeticiones de elementos



- Interfaz *List* $\langle E \rangle$

- Extiende *Collection* $\langle E \rangle$  (*Iterable* $\langle E \rangle$ )
- Incluye operaciones de acceso, inserción y borrado por índice (posición en la secuencia)
- Vistas (operación *subList* e iteradores)

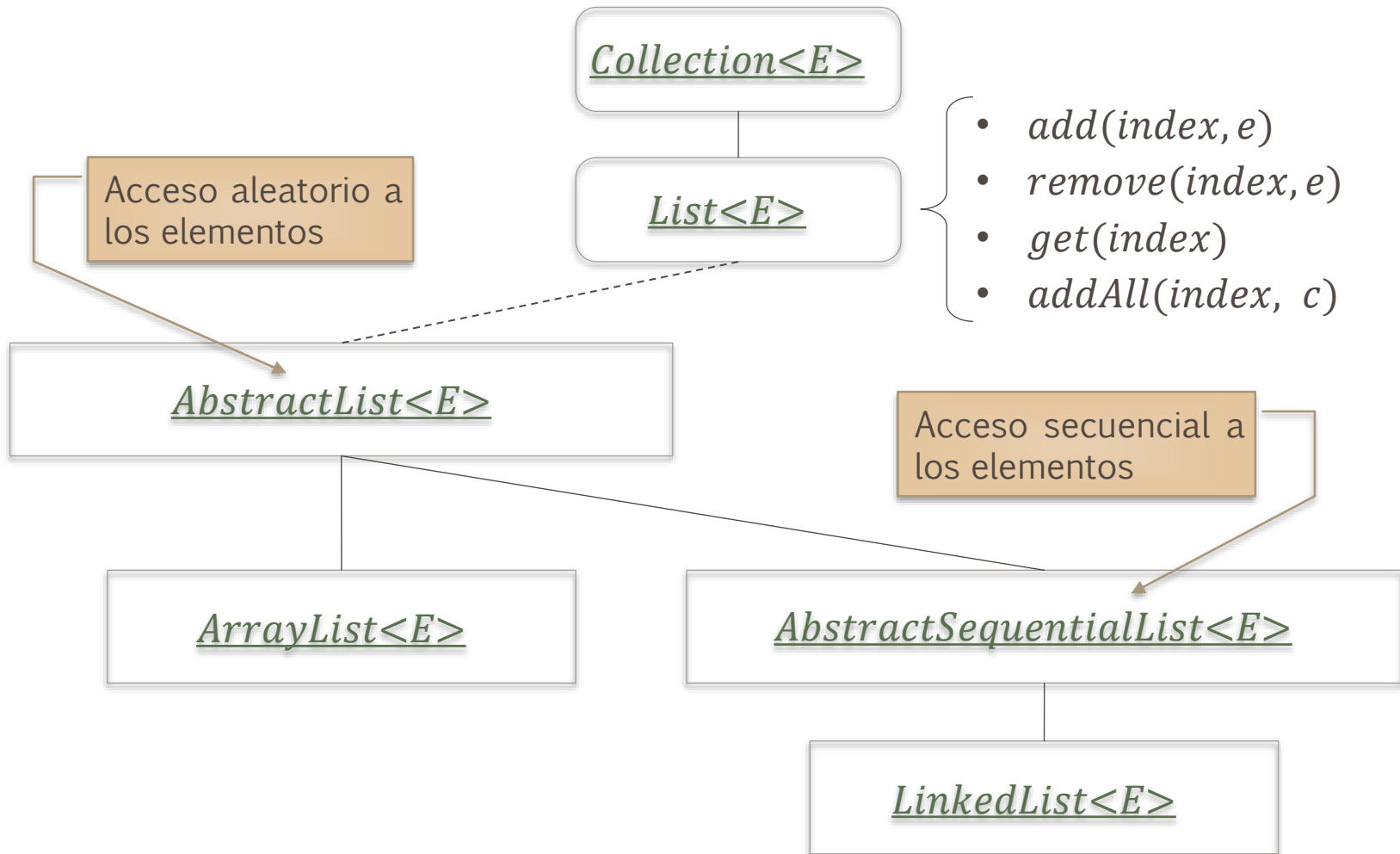


## Listas (2)

Operaciones adicionales	Descripción
<code>boolean add(int index, E e)</code>	Añade el elemento <code>e</code> en la posición <code>index</code> de la lista (opcional)
<code>E get(int index)</code>	Retorna el elemento que se encuentra en la posición <code>index</code> de la lista
<code>boolean remove(int index)</code>	Quita, el elemento que está en la posición <code>index</code> de la lista (opcional)
<code>E set(int index, E e)</code>	Reemplaza el elemento que se encuentra en la posición <code>index</code> de la lista por el especificad. Retorna el elemento reemplazado (opcional)
<code>int indexOf(Object o)</code>	Retorna la posición que ocupa la primera ocurrencia del objeto <code>o</code> en la lista. Si no se encuentra retorna <code>-1</code>
<code>boolean addAll(int index, Collection&lt;?&gt; c)</code>	(opcional)
<code>List&lt;E&gt; subList(int fromIndex, int toIndex)</code>	



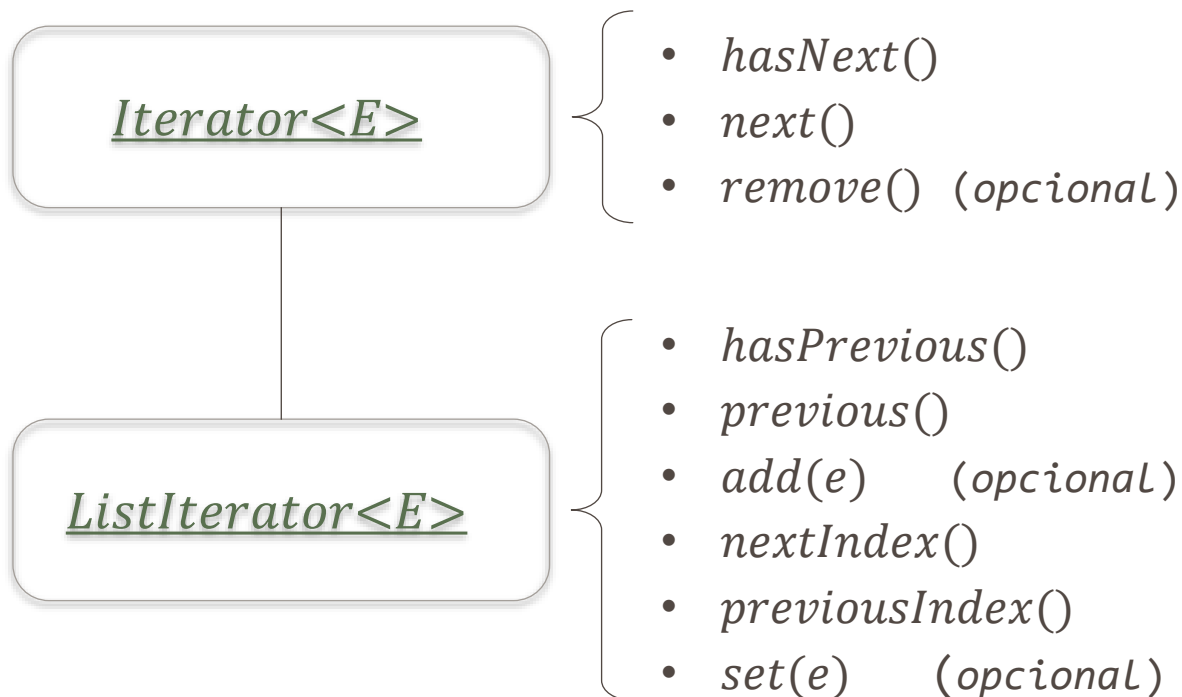
## Listas (3)



## Listas (4)

### ■ Iteradores

- Las interfaces *Collection<E>* y *List<E>* incluyen los métodos: *iterator()* y *listIterator()*, respectivamente, para crear los iteradores



## Listas (5)

---

- Ejemplo de uso de iteradores
  - Igualdad de listas

```
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
  
    if (!(obj instanceof List<?>)) {  
        return false;  
    }  
  
    List<?> other = (List<?>) obj;
```

## Listas (5)

---

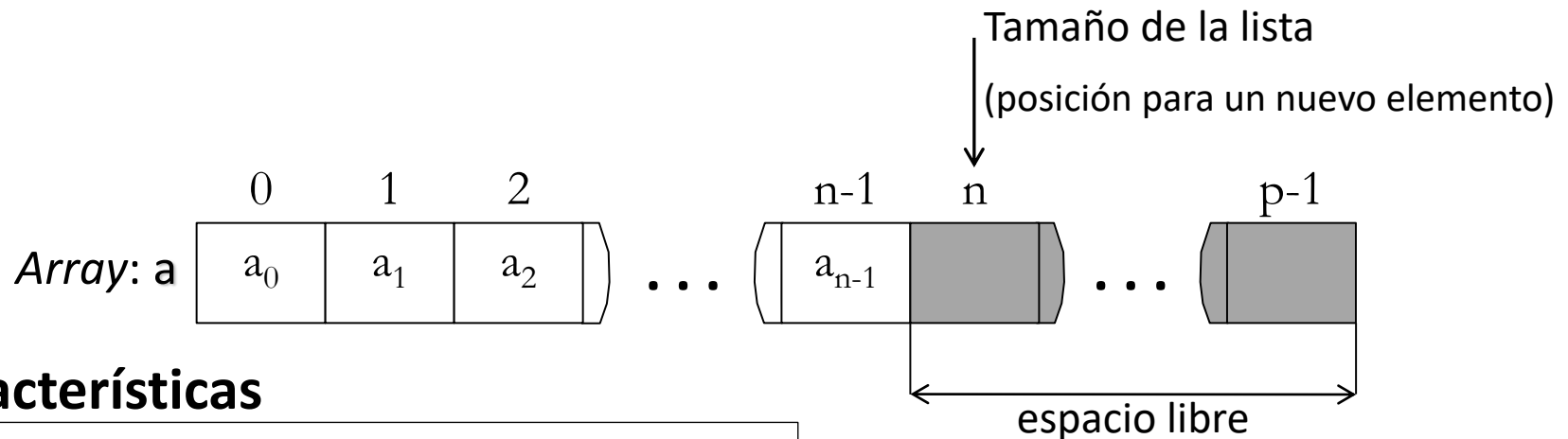
```
Iterator<E> itr1 = iterator();
Iterator<?> itr2 = other.iterator();
while (itr1.hasNext() && itr2.hasNext()) {
    E e = itr1.next();
    Object o = itr2.next();
    if (!(e == null ? o == null : e.equals(o))) {
        return false;
    }
}

// comprobar que ambos iteradores han finalizado
return itr1.hasNext() == itr2.hasNext();
}
```



# Listas con *arrays* (1)

## ■ Representación con *arrays* dinámicos



### Características

- *Espacio de memoria: compacto (contiguo)*
- *Acceso a un elemento:  $\Theta(1)$*
- *Inserción al final: ¿  $\Theta(1)$  ?*
- *Borrado del último elemento:  $\Theta(1)$*
- *Resto de operaciones:  $O(n)$*
- *Tamaño de la secuencia, ¿ lista vacía ? :  $\Theta(1)$*

### Representación

- *Un array* : contiene los elementos de la secuencia
- *Un entero*: tamaño de la secuencia



## Listas con *arrays* (2)

---

- Coste de la operación de inserción al final
  - Caso 1. El tamaño del *array* siempre coincide con el tamaño de la secuencia (*no es una buena idea*)

***Adición al final de la secuencia  $a_0 a_1 \dots a_{i-1}$  del elemento  $a_i$***

Nuevo *array* de tamaño  $i+1$  en el que deben introducirse todos los elementos de la secuencia y  $a_i$ : Coste  $i+1$

Coste temporal de la inserción de  $n$  elementos:  $O(n^2)$




## Listas con *arrays* (3)

---

- Caso 2. El tamaño del *array* es mayor que el tamaño de la secuencia. Cuando es necesario se duplica el espacio.

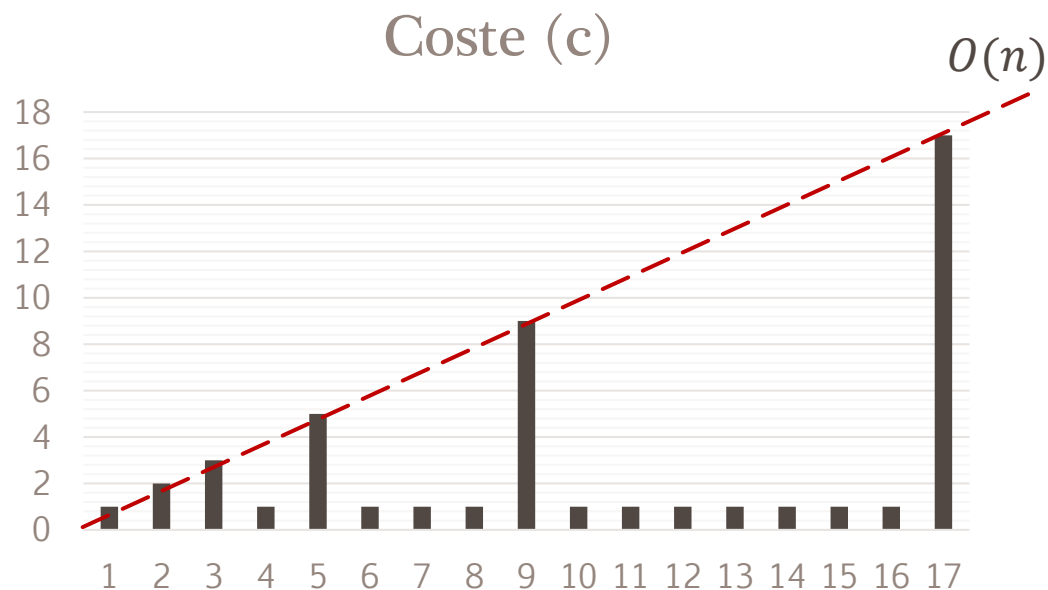
### ***Análisis de coste amortizado***

1. *Método de agregación* 
2. *Método de cuenta o del banquero*
3. *Método del potencial*



## Listas con *arrays* (4)

Nº Ítems (i)	Capacidad (s)	Coste (c)
1	1	1
2	2	2
3	4	3
4	4	1
5	8	5
6	8	1
7	8	1
8	8	1
9	16	9
10	16	1
11	16	1
12	16	1
13	16	1
14	16	1
15	16	1
16	16	1
17	32	17



## Listas con *arrays* (5)

---

Coste de añadir el elemento  $i$

$$c_i = \begin{cases} i & \text{si } i - 1 \text{ es una potencia de 2} \\ 1 & \text{en otro caso} \end{cases}$$

Coste de doblar el tamaño del *array*

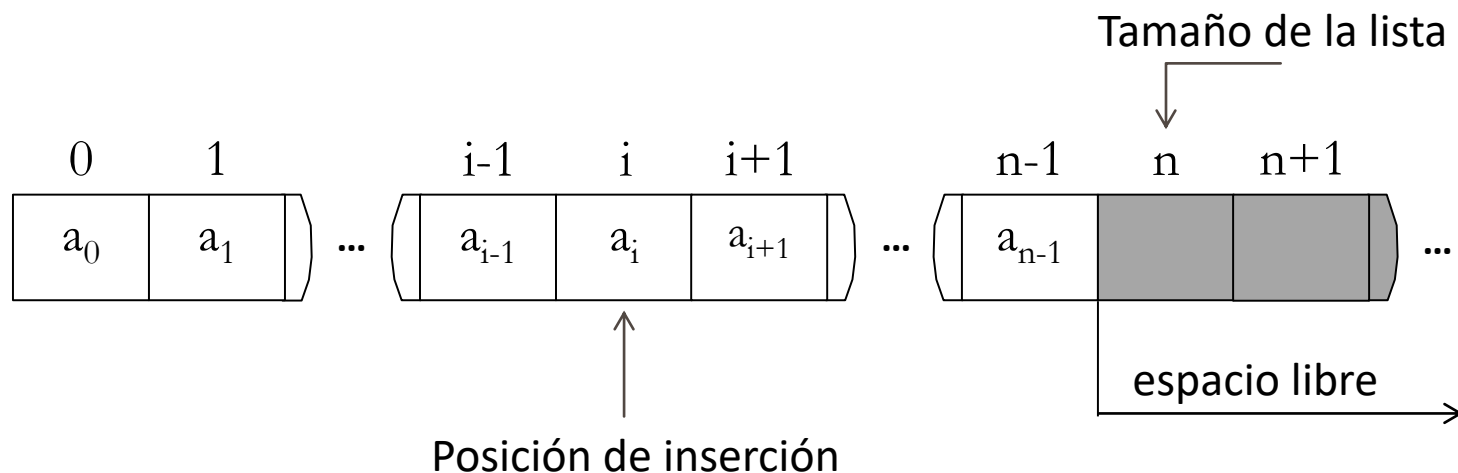
$$d_i = \begin{cases} i - 1 & \text{si } i - 1 \text{ es una potencia de 2} \\ 0 & \text{en otro caso} \end{cases}$$

La operación de inserción se ejecuta en *tiempo amortizado constante*.

Coste temporal de la inserción de  $n$  elementos:  $O(n)$

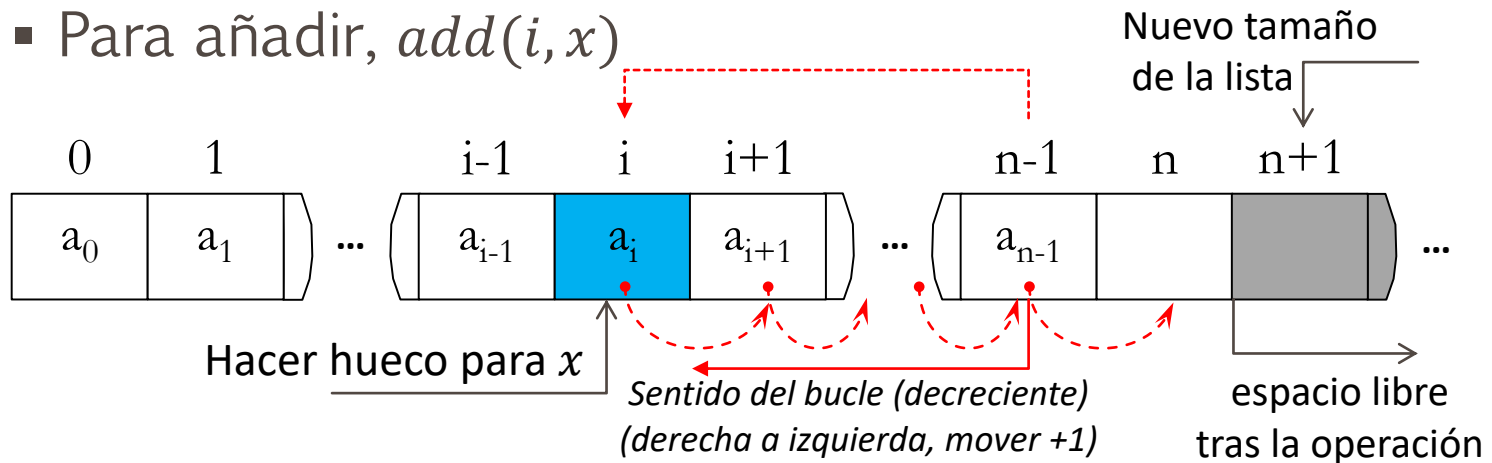
## Listas con *arrays* (6)

- Operaciones de inserción y borrado al principio o en medio de la lista
  - Requieren desplazar los elementos
    - Para hacer un hueco donde insertar el nuevo elemento
    - Para eliminar el hueco que deja el elemento a quitar
- La lista antes de realizar las operaciones

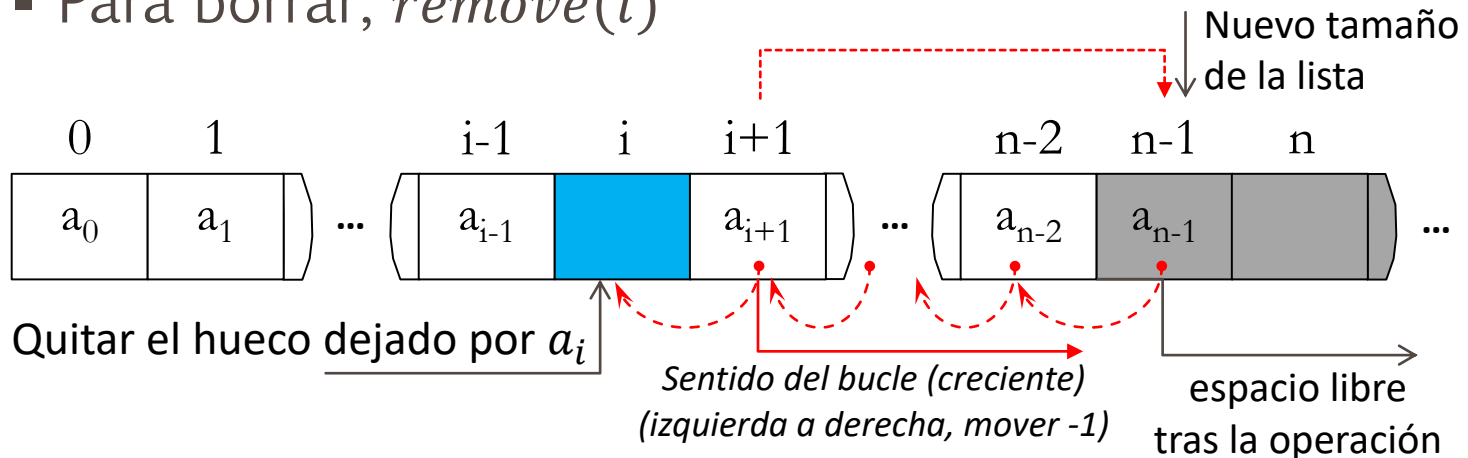


## Listas con *arrays* (7)

- Para añadir,  $add(i, x)$



- Para borrar,  $remove(i)$



## Listas con *arrays* (8)

---

- La clase *ArrayList<E>*
  - En Java la clase *ArrayList<E>* es una clase concreta que extiende la clase *AbstractList<E>* para listas con acceso aleatorio a los elementos y que está implementada de forma análoga a la vista previamente.
  - A priori, la única diferencia puede estar en la forma en que crece la memoria cuando es necesario. No está indicado como se lleva a cabo tal crecimiento y, por otra parte, existe un método (*ensureCapacity*) para indicar la **capacidad mínima deseada**. En todo caso, en la especificación se indica que *la operación de inserción (add(e)) se ejecuta en tiempo amortizado constante*.





# Listas enlazadas (1)

---

- Representación mediante estructuras enlazadas
  - Características
    - Memoria no compacta. La lista crece (o decrece) dinámicamente sin necesidad de reorganización.
    - Acceso secuencial a los elementos (coste temporal  $O(n)$ ).
    - La inserción y borrado de un elemento en una posición *ya dada* (por ejemplo, mediante un **iterador**), es de tiempo constante (no hay necesidad de desplazar el resto de los elementos como en el caso previo).

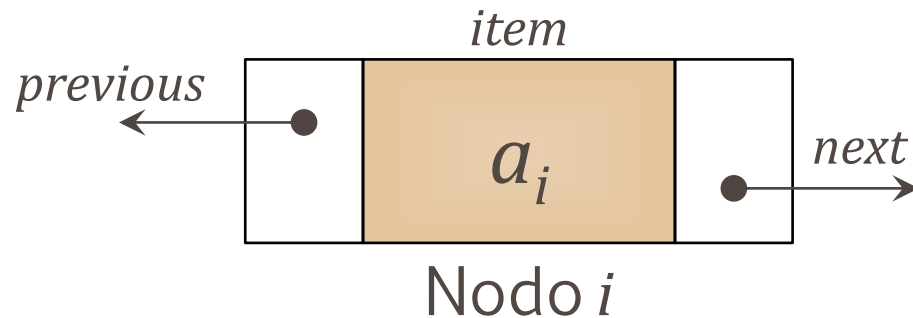
# Listas doblemente enlazadas (1)

---

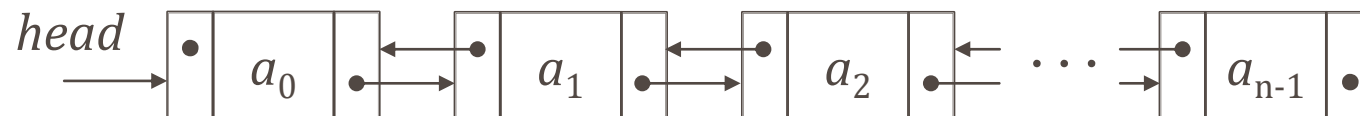
- Nodos de la lista
  - En este caso los nodos incluyen un campo adicional: el **nodo previo**

```
private static class Node<E> { // clase interna
    E item;                    // elemento
    Node<E> previous;         // nodo previo
    Node<E> next;             // nodo siguiente
    Node(E e, Node<E> previous, Node<E> next) {
        item = e;
        this.previous = previous;
        this.next = next;
    }
}
```

## Listas doblemente enlazadas (2)



- Representación gráfica

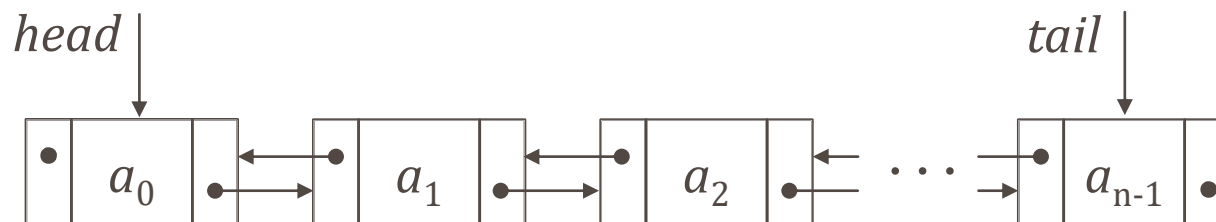


- Representación

```
// área de datos  
private Node<E> head;
```

## Listas doblemente enlazadas (3)

- Clase *MutableList<E>*
  - Listas mutables implementadas extendiendo la clase abstracta *AbstractSequentialList<E>*
  - Tanto las operaciones en los extremos de ambas listas, como la operación *size()* deberán ser de tiempo constante ( $O(1)$ )
- Representación gráfica





## Listas doblemente enlazadas (4)

---

```
public class MutableList<E> implements AbstractSequentialList {  
    // área de datos  
    private Node<E> head;    // primer nodo de una lista  
    private Node<E> tail;    // último nodo de una lista  
    private int size;        // número de elementos de la lista  
    private static class Node<E> { // clase interna  
        ...  
    }  
    public MutableList() { ... }  
    public MutableList(Collection<? extends E> c) { ... }  
    @Override  
    public int size() { ... }  
    @Override  
    public ListIterator<E> listIterator(int index) {  
        return new MutableListIterator(index);  
    }  
}
```



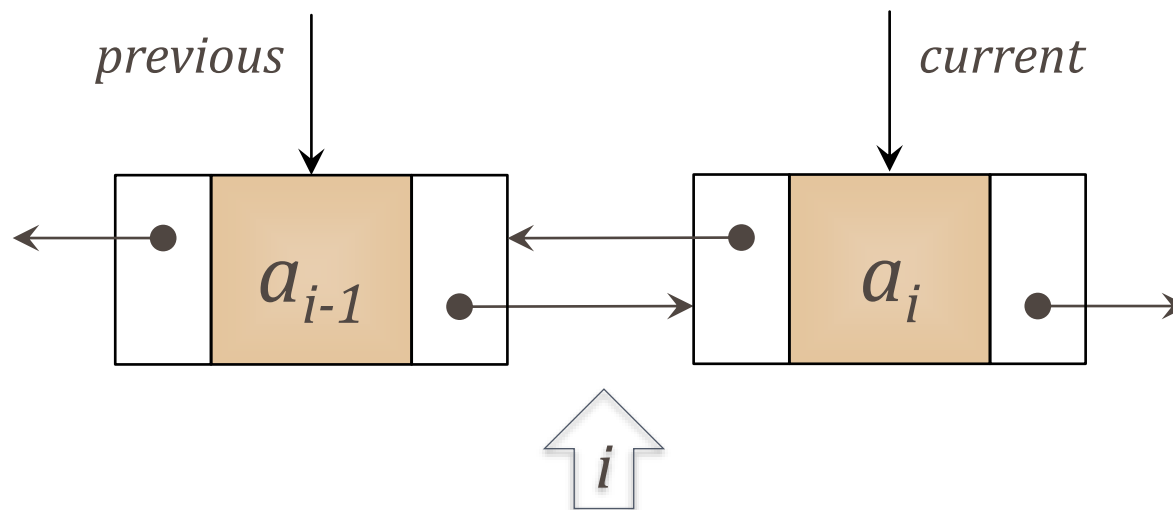
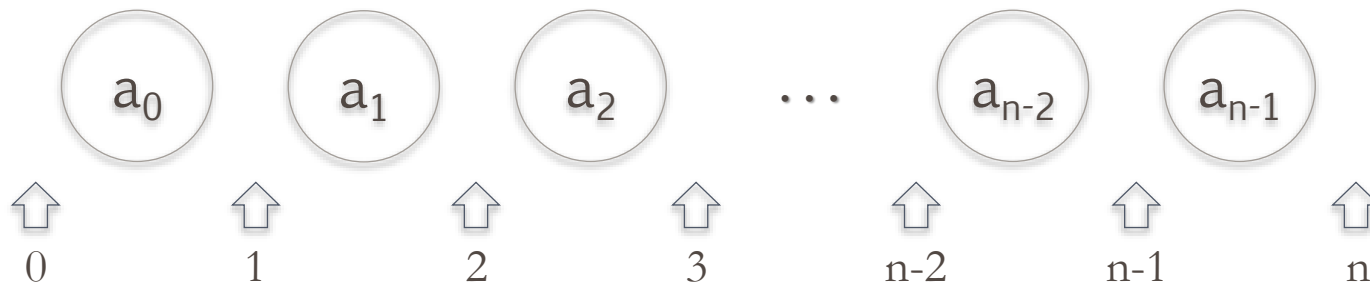
## Lista doblemente enlazadas (5)

---

- Iterador *MutableListIterator*
  - Clase interna que implementa *ListIterator<E>*
  - Campos requeridos
    - El nodo *current* que contiene el elemento que retornará la operación *next()*
    - El nodo *previous* que contiene el elemento que retornará la operación *previous()*
    - El nodo *lastReturned()* que contiene el último elemento retornado por *next()* o *previous()*. Su valor es *null* al principio y también tras una operación *remove()*.
    - El índice *currentIndex* correspondiente a la posición del elemento del nodo *current* en la lista



## Listas doblemente enlazadas (6)



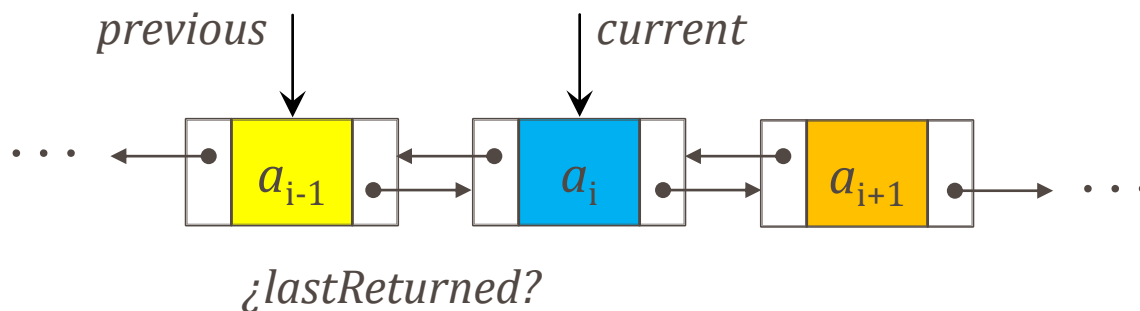
# Listas doblemente enlazadas (7)

---

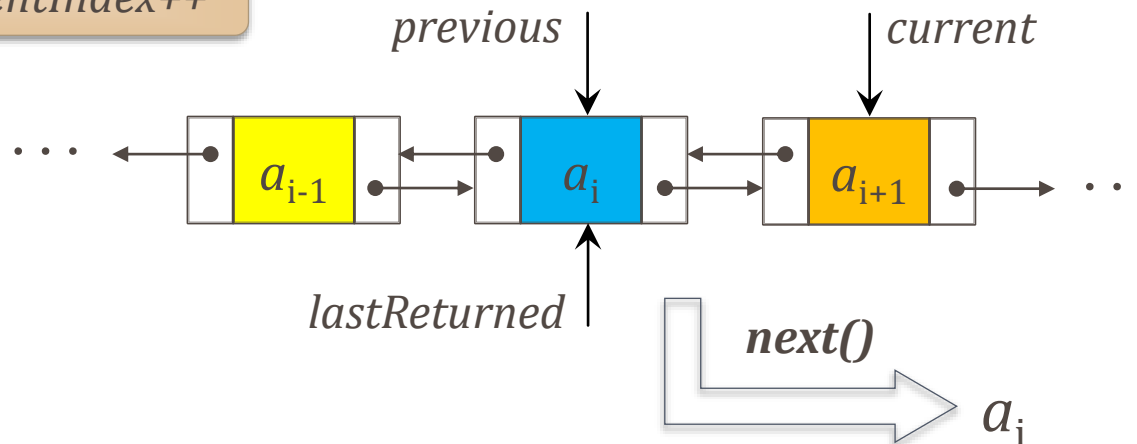
- Constructor *MutableListIterator(index)*
  - Debe ser privado, ya que el iterador se crea con la operación *listIterator()* de listas.
  - Inicializar los campos correctamente
    - Al respecto, debe recordarse que las operaciones en ambos extremos de la lista deben ser de tiempo constante. Esto sólo podrá ser así, si el coste temporal tanto de *MutableListIterator(0)* como de *MutableListIterator(size())* es de  $O(1)$
- Operación *next()*
  - Si hay siguiente elemento retornará el elemento del nodo *current*, pero previamente habrá que restablecer toda la información.
  - *lastReturned* puede ser *null*



## Listas doblemente enlazadas (8)

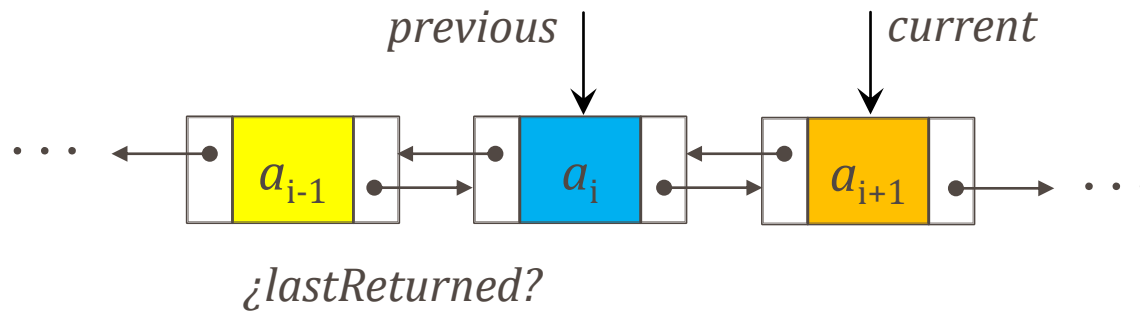


*currentIndex++*

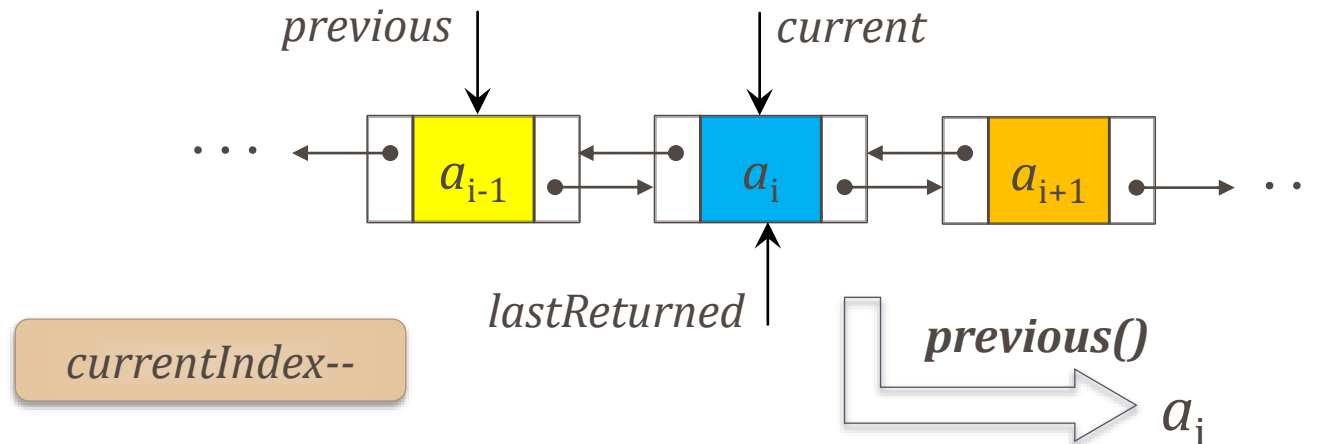


# Listas doblemente enlazadas (9)

- Operación *previous()*
  - Si hay elemento anterior retornará el elemento del nodo *previous*, pero previamente habrá que restablecer toda la información.
  - *lastReturned* puede ser *null*



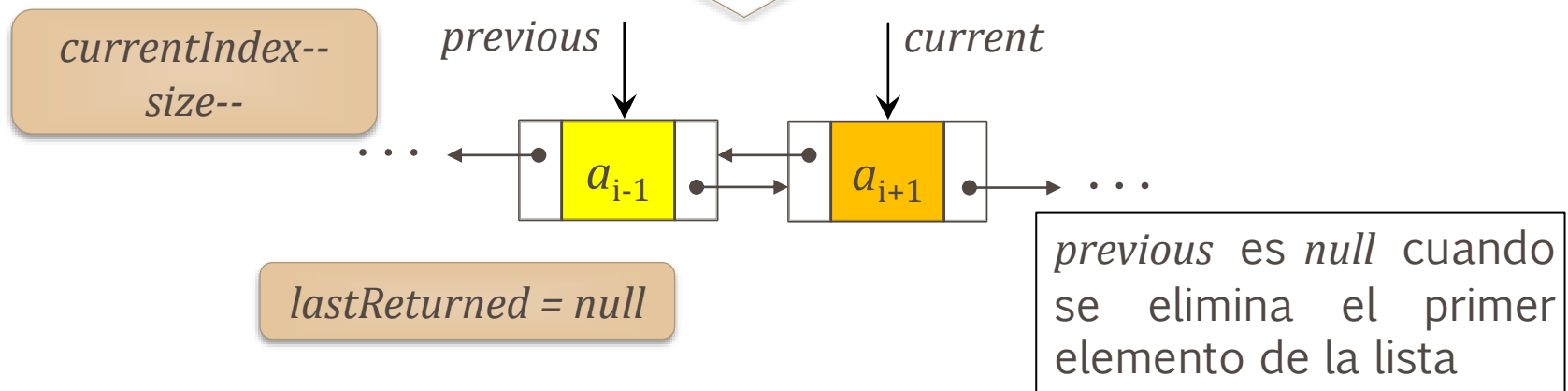
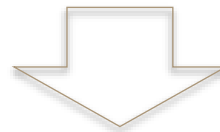
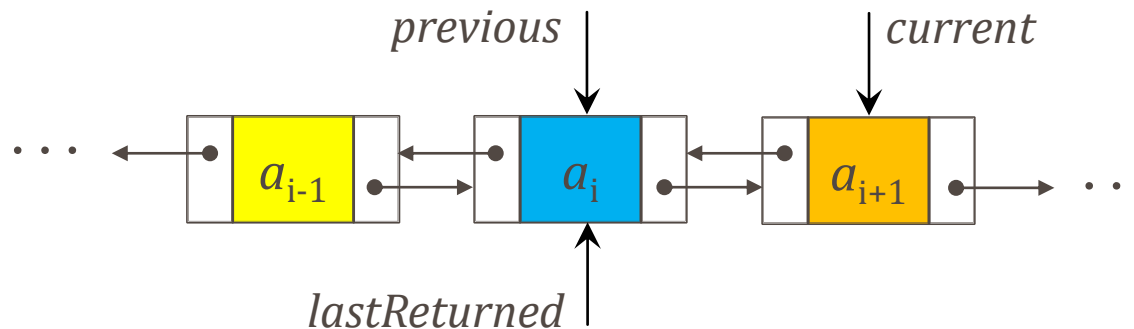
# Listas doblemente enlazadas (10)



- Operación *remove()*
  - Se elimina el elemento del nodo *lastReturned*, que no puede ser *null*. Si es *null* se lanza la excepción *IllegalStateException*
  - Si *previous* = *lastReturned*, previamente se realizó una operación *next()* y si *current* = *lastReturned* la operación *previous()*
  - Es necesario tener en cuenta los casos particulares de eliminación del primer nodo y del último

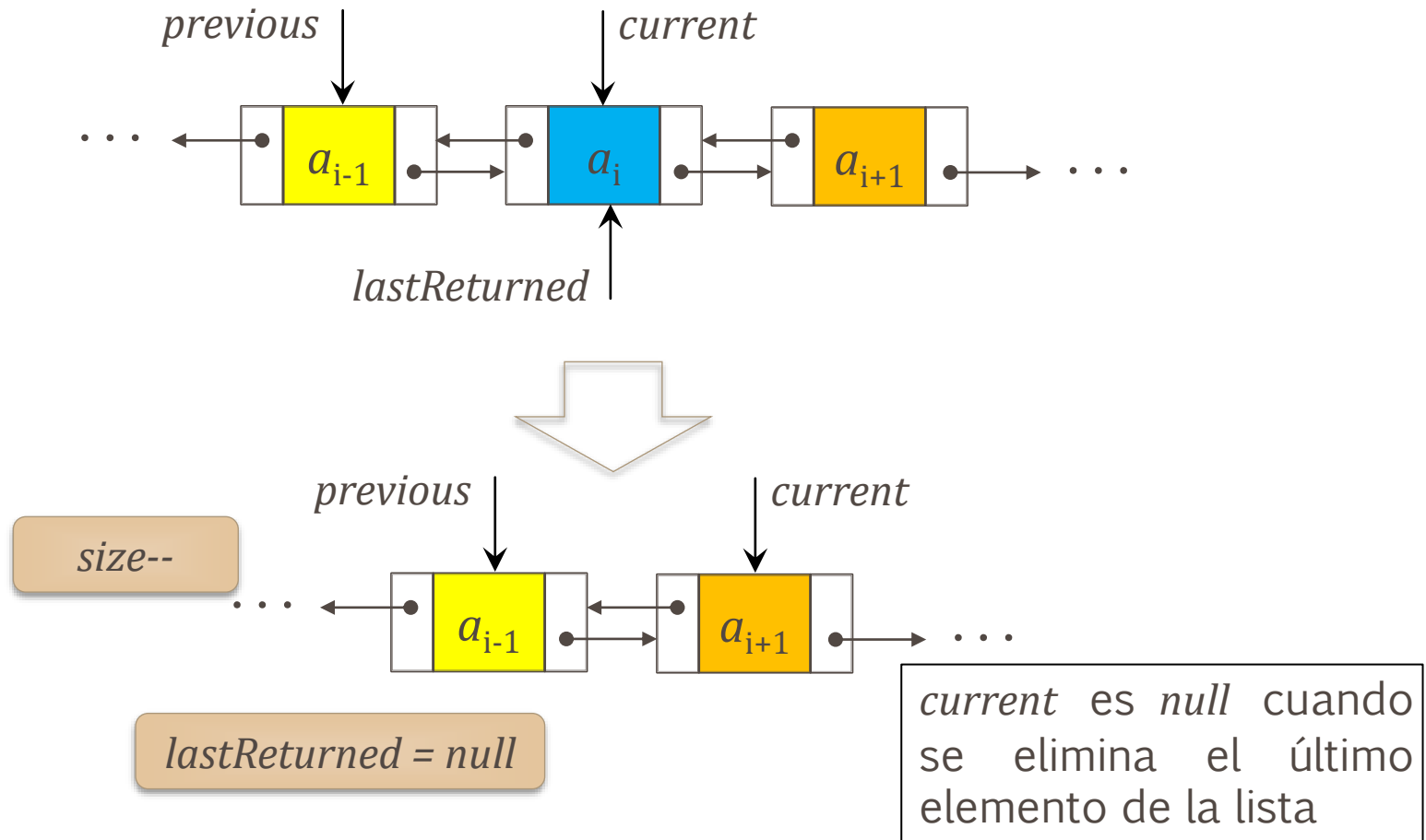
# Listas doblemente enlazadas (11)

- Caso 1 (viene de una operación *next()*)



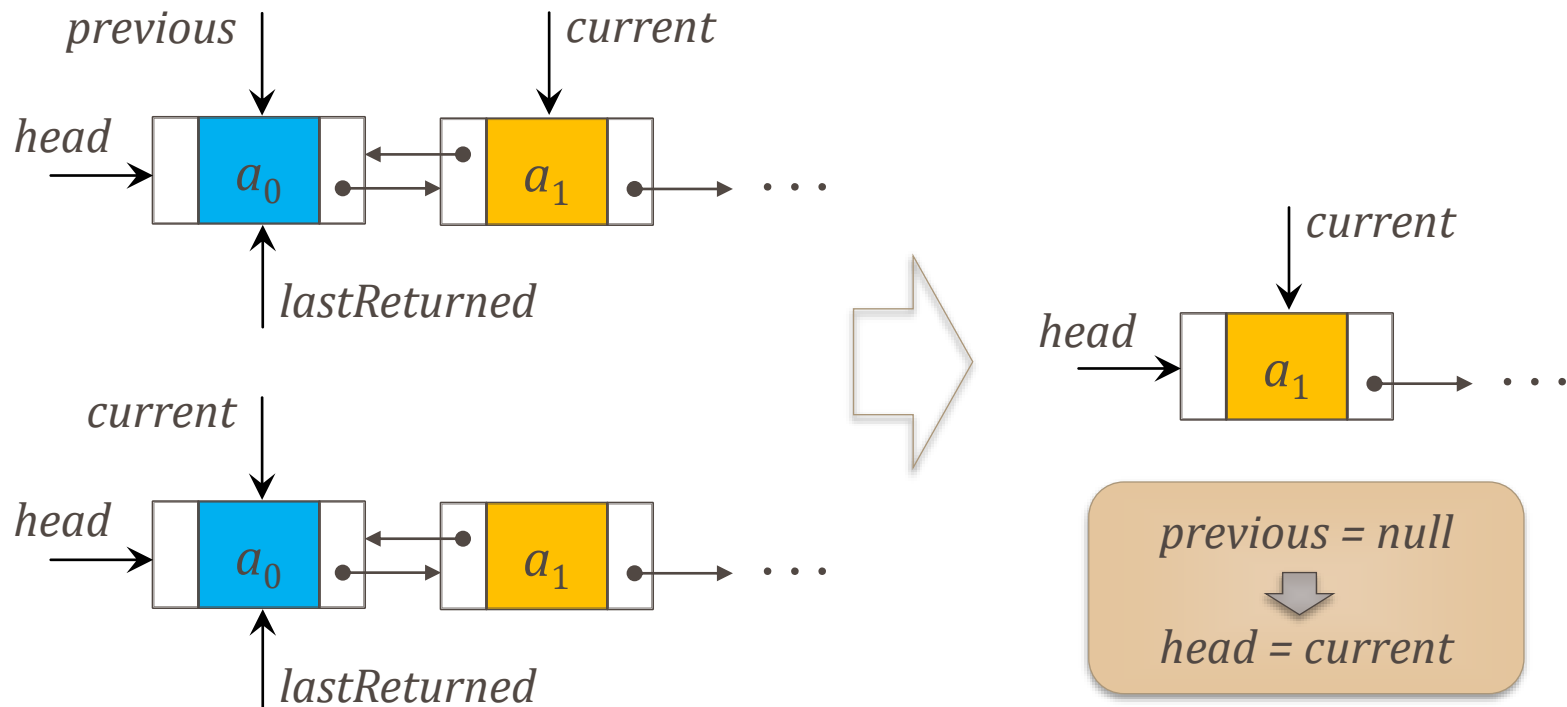
# Listas doblemente enlazadas (12)

- Caso 2 (viene de una operación *previous()*)

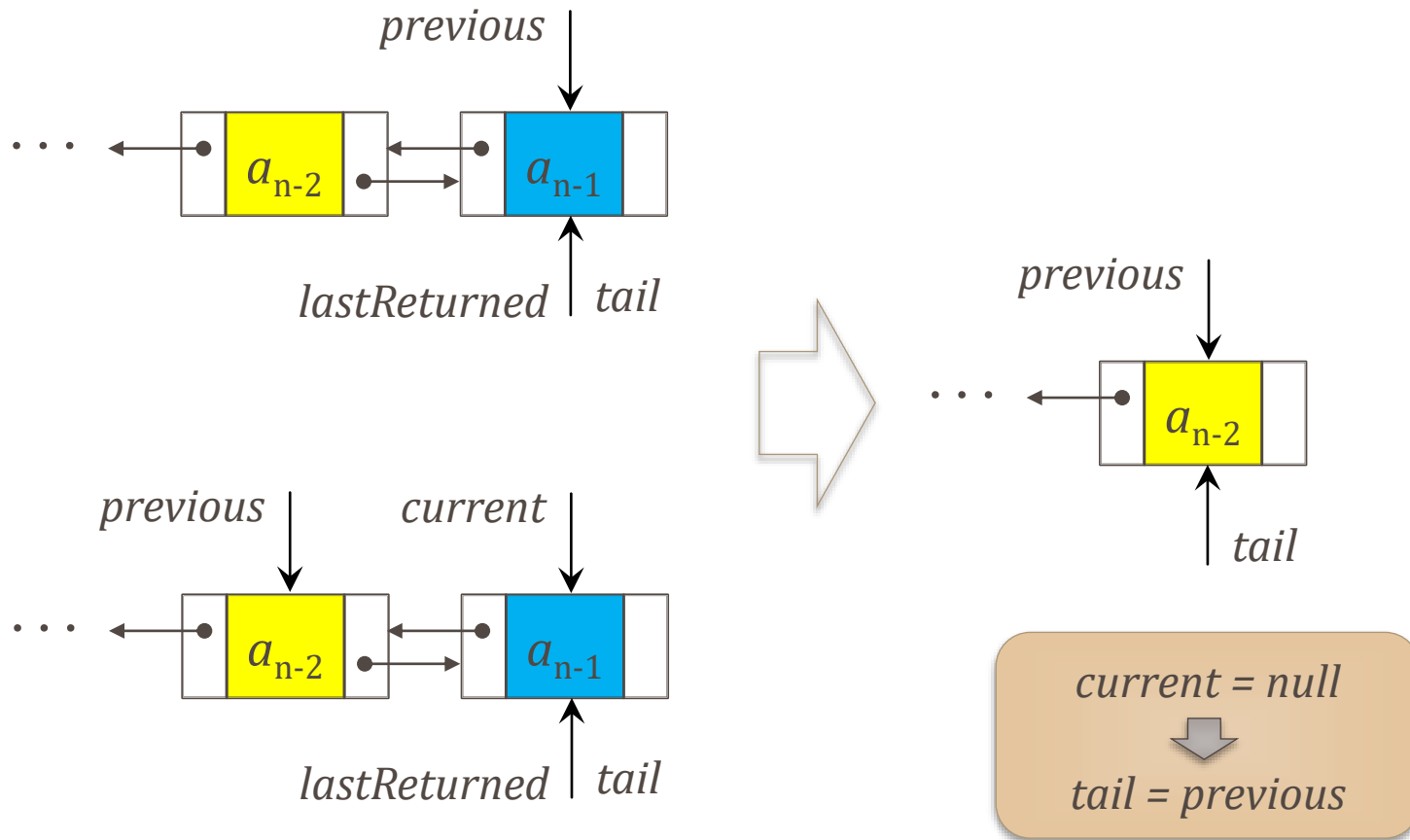


# Listas doblemente enlazadas (13)

- Casos particulares
  - Tanto en el caso 1 como en el 2, es necesario restaurar *head* o *tail* cuando se elimina el primer nodo o el último, respectivamente.

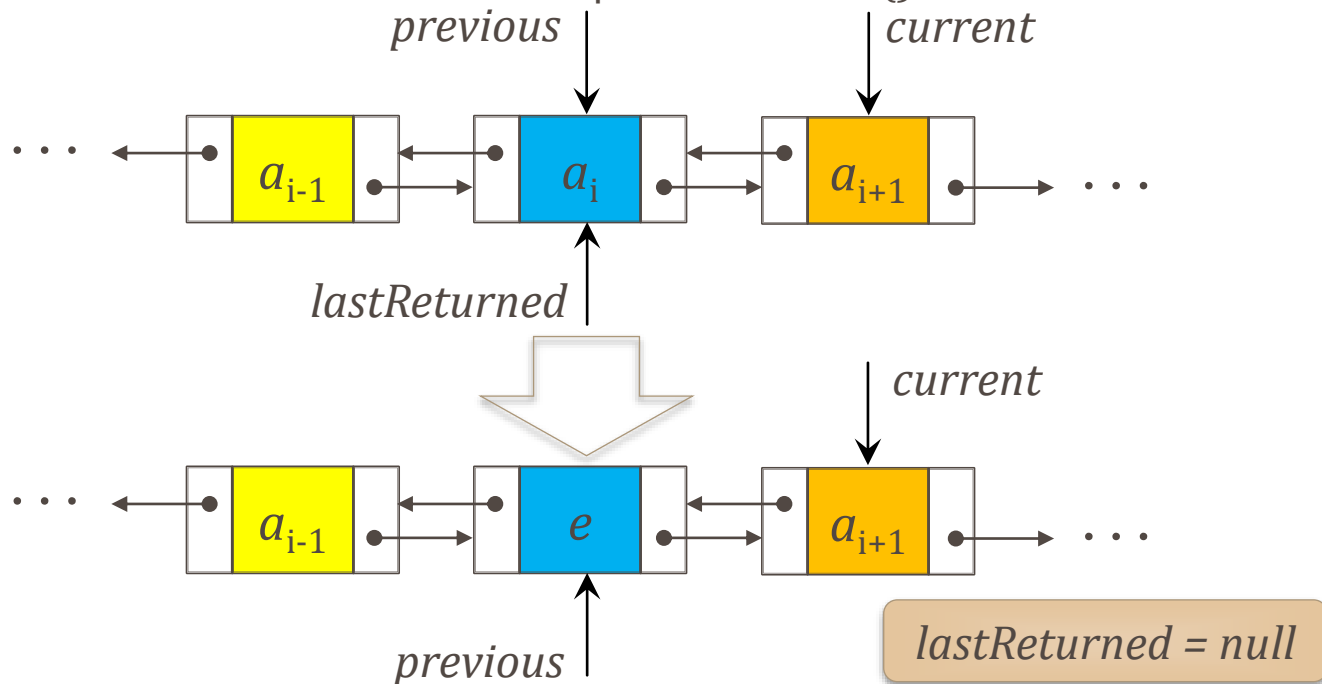


# Listas doblemente enlazadas (14)



# Listas doblemente enlazadas (15)

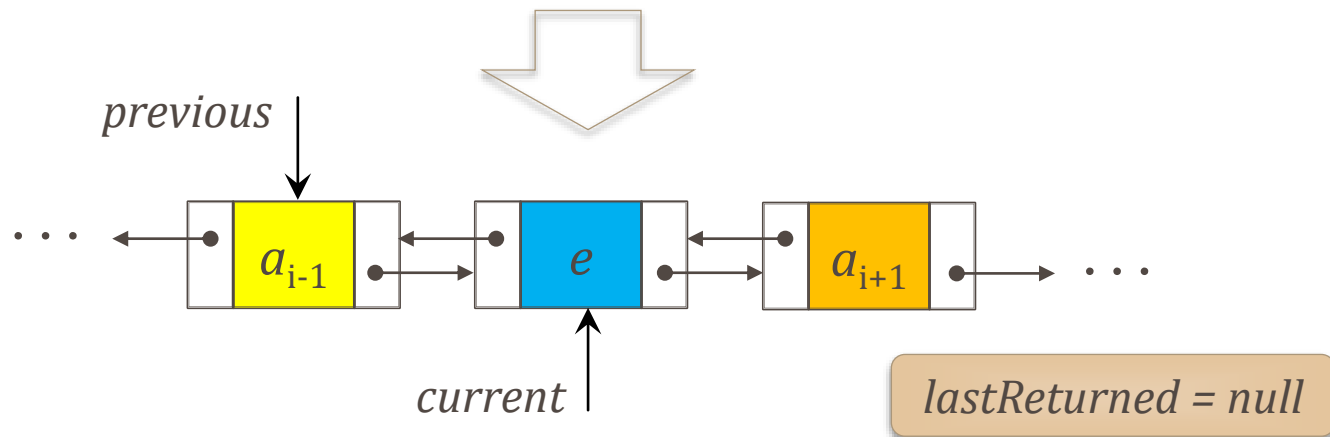
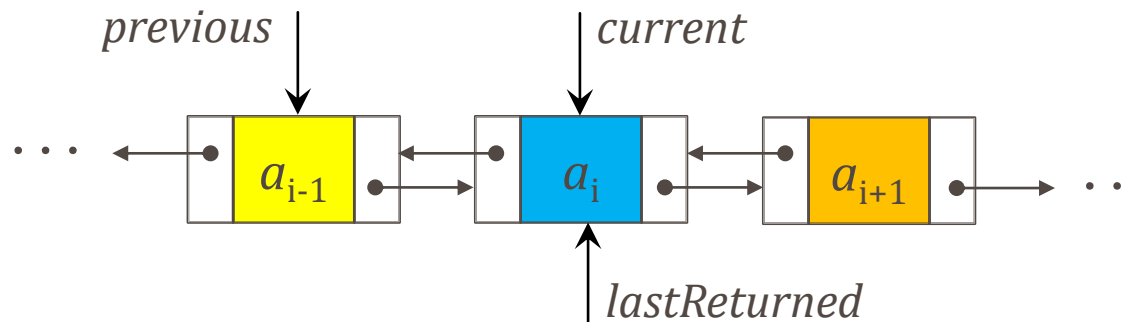
- Operación *set(e)*
  - El elemento del nodo *lastReturned* se cambia por el especificado (*e*). Si *lastReturned* = *null* se lanza la excepción *IllegalStateException*
  - Caso 1 (viene de una operación *next()*)





# Listas doblemente enlazadas (16)

- Caso 2 (viene de una operación *previous()*)





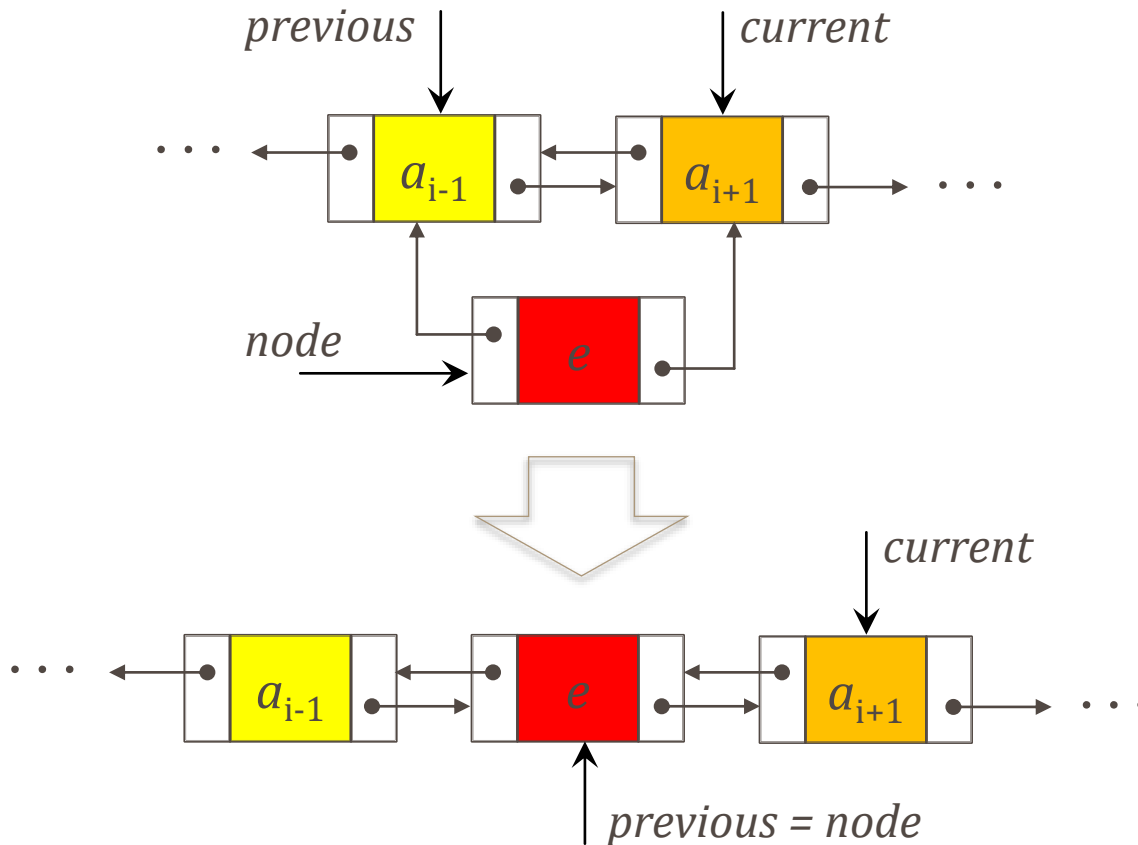
## Listas doblemente enlazadas (17)

---

- Operación *add(e)*
  - Hay que crear un nuevo nodo entre *previous* y *current*  
 $Node<E> \text{ node} = \text{new Node}<>(e, \text{previous}, \text{current})$
  - Las inserciones al principio (*previous* = *null*) y al final de la lista (*current* = *null*) hay que tratarlas aparte y deben ser de tiempo constante
  - *lastReturned* podría ser *null*

# Listas doblemente enlazadas (18)

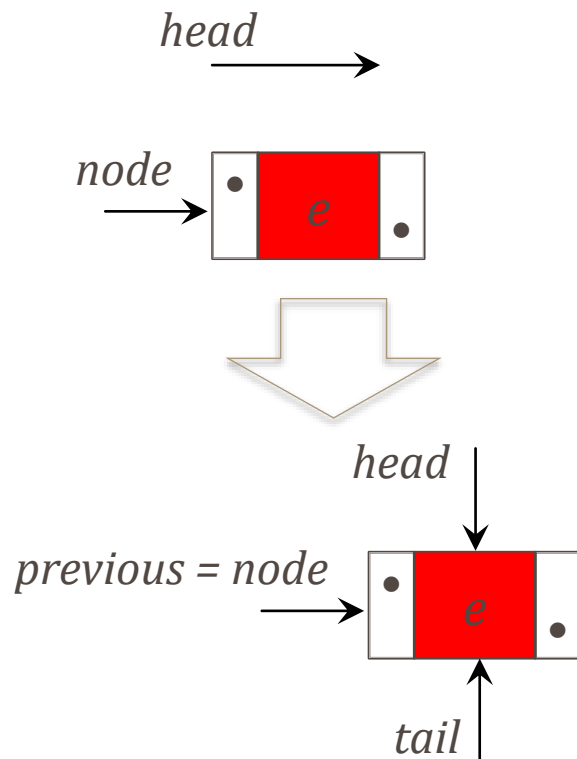
- Caso general (*previous*  $\neq$  null y *current*  $\neq$  null)



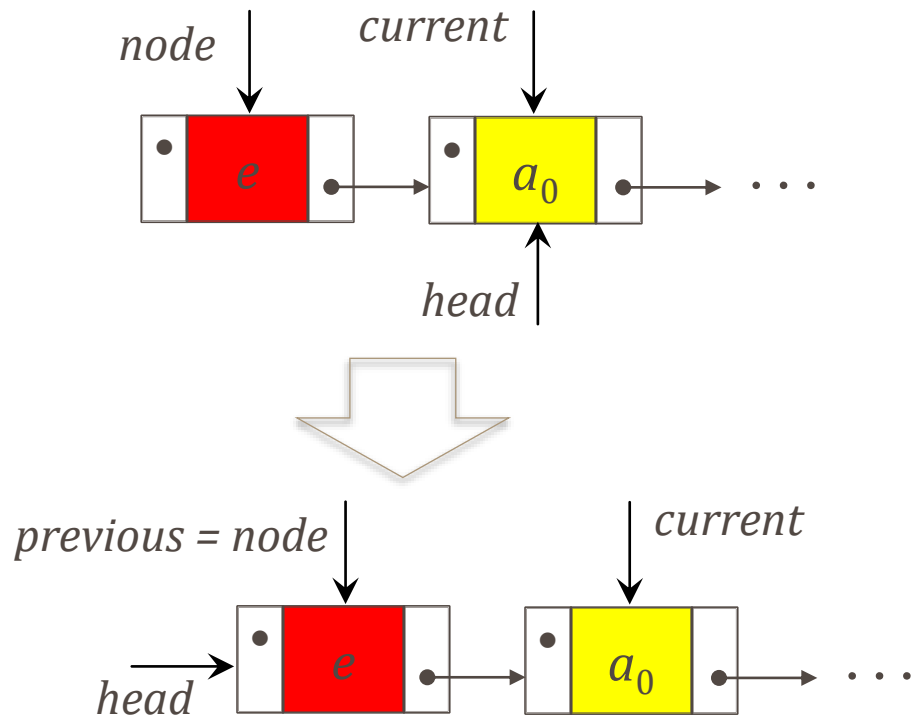
# Listas doblemente enlazadas (19)

- Inserción al principio (*previous = null*)

*Lista vacía*

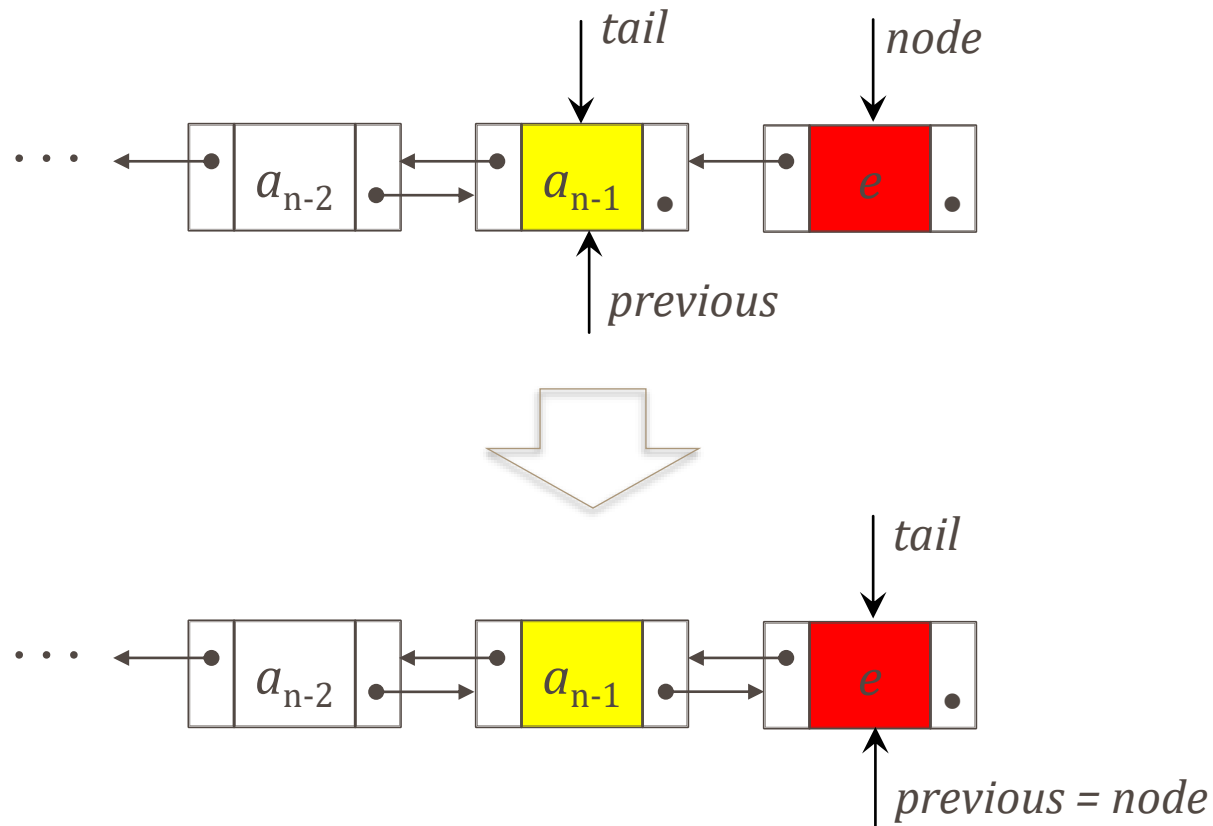


*Lista no vacía*



# Listas doblemente enlazadas (20)

- Inserción al final ( $current = null$ )





## Listas doblemente enlazadas (21)

---

- Tanto en el caso general como en los particulares (inserción al principio y al final), es necesario hacer:

*currentIndex++*

*size++*

*lastReturned = null*

# Listas simplemente enlazadas (1)

---

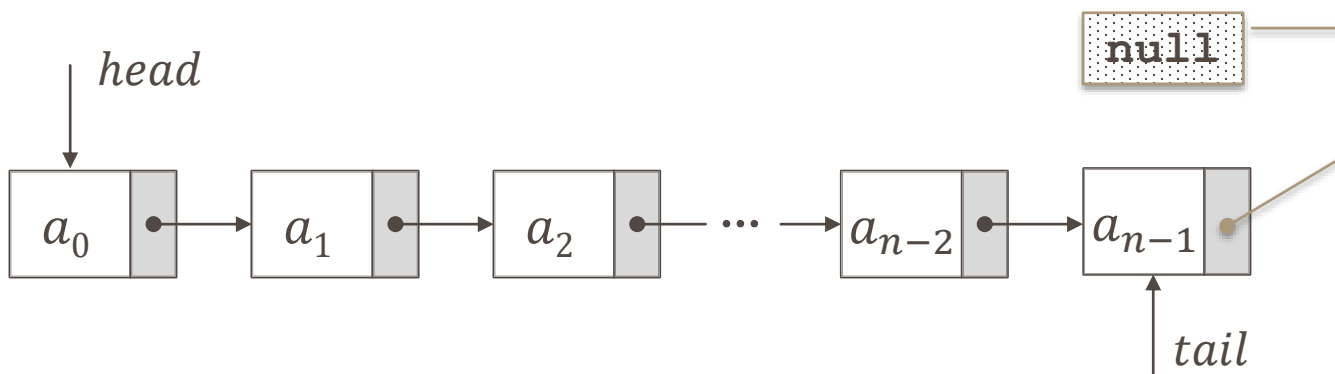
- Clase *SingleEndedList*<E>
  - Una implementación de listas simplemente enlazadas con el último nodo también conocido que sería análoga a la vista previamente. Pero debe tenerse en cuenta que los nodos no tienen información sobre el nodo previo (no existe el campo *previous*). Como consecuencia:
    - Desaparece cualquier referencia al campo *previous* de los nodos
    - La operación *previous()* del iterador pasa a tener un coste lineal y lo mismo ocurre con la operación *remove()* cuando el elemento eliminado es el último retornado por *previous()*.
      - En ambos casos, es necesario restablecer el nodo *previous* del iterador, que pasa a ser el nodo anterior a éste, y la única forma de obtenerlo es hacer una búsqueda secuencial.

## Listas simplemente enlazadas (2)

- Representación:

*Node<E> head; // primer nodo*

*Node<E> tail; // último nodo*

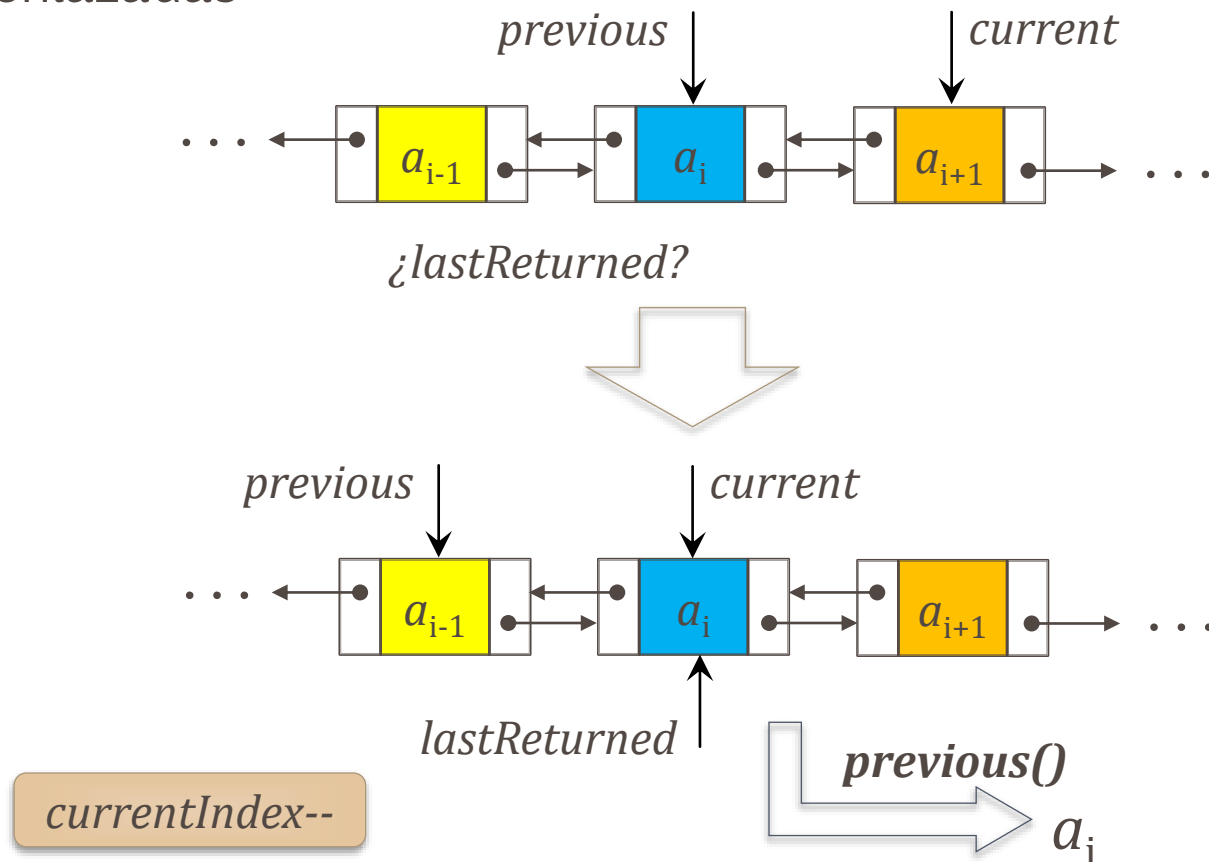


- El segundo campo (*tail*) es prescindible, pero se incluye para que la inserción al final sea de tiempo constante



## Listas simplemente enlazadas (3)

- Operación *previous()* del iterador en listas doblemente enlazadas

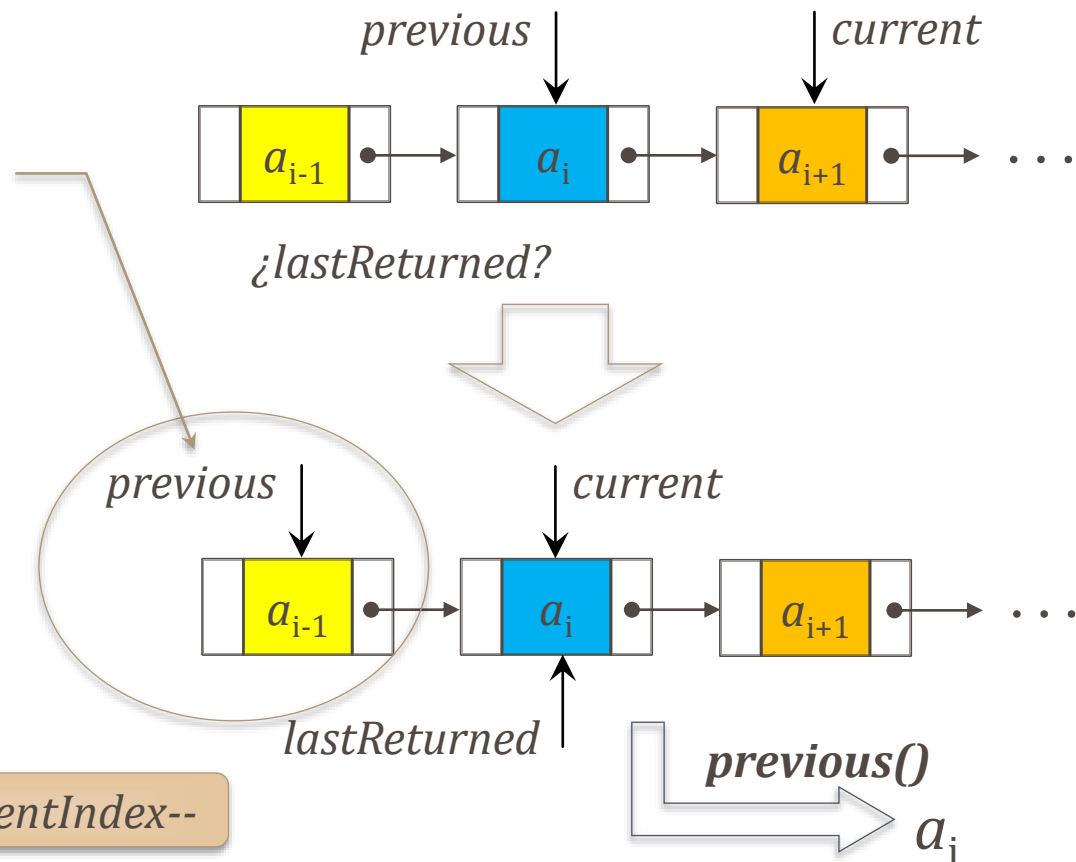


## Listas simplemente enlazadas (4)

- Para listas simplemente enlazadas

Sólo se puede obtener realizando una búsqueda secuencial desde el primer elemento de la lista.

*previous = backward(previous)*





## Listas simplemente enlazadas (5)

---

- Por tanto, resulta conveniente incluir en el iterador la siguiente operación interna:

```
/* Retorna el nodo anterior al nodo especificado.  
* @param node el nodo dado  
* @return el nodo de la lista anterior al  
* especificado  
*/  
private Node<E> backward(Node<E> node) {  
    if (node == SingleEndedList.this.head) {  
        return null;  
    }  
  
    Node<E> result = SingleEndedList.this.head;  
    while (result.next != node) {  
        result = result.next;  
    }  
  
    return result;  
}
```



# Listas enlazadas en Java

---

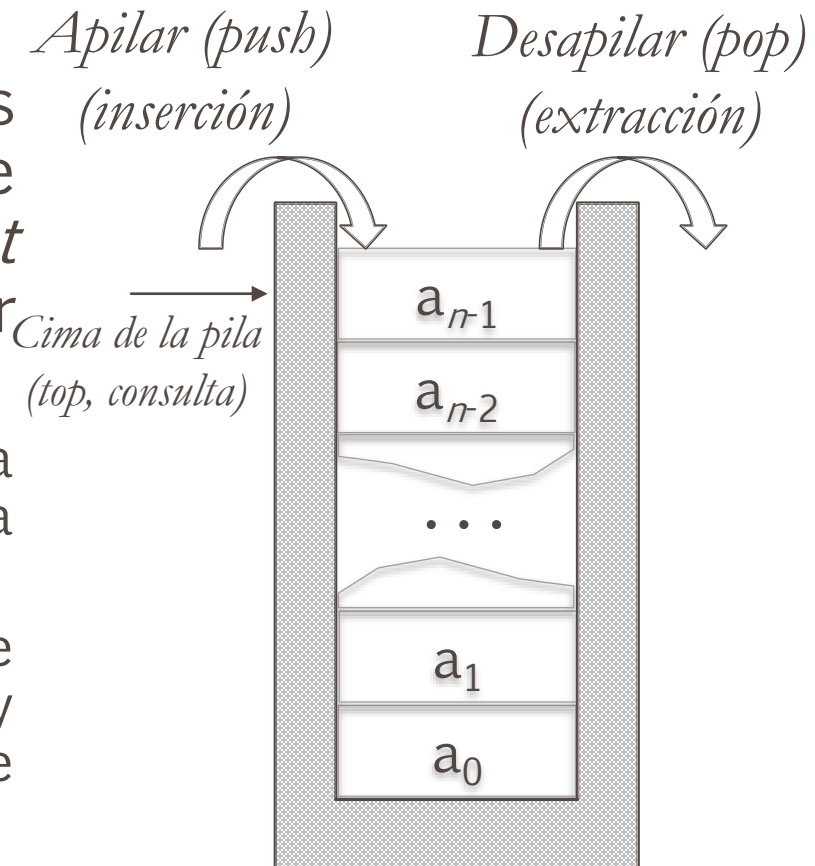
- Clase *LinkedList<E>*
  - Clase concreta de listas enlazadas proporcionada por la librería de Java. Extiende la clase abstracta *AbstractSequentialList<E>*
    - Implementa todas las operaciones opcionales y permite todos los elementos (incluido **null**).
    - Implementa las interfaces *List<E>* y *Deque<E>*
      - Al implementar la interfaz *Deque<E>*, los métodos para obtener, extraer e insertar elementos al principio y al final de la lista son de tiempo constante.
    - Implementación basada en listas doblemente enlazadas
      - La operación *previous()* de *ListIterator<E>* es de  $O(1)$

# Pilas (1)

## ■ Pila

- Secuencia de elementos con modo de acceso de tipo LIFO (*Last In First Out*), último en entrar primero en salir.

- Sólo se tiene acceso a la parte superior de la pila (cima de la pila, *top*).
- Las operaciones de inserción, extracción y consulta del elemento de la cima deben ser de  $\Theta(1)$ .





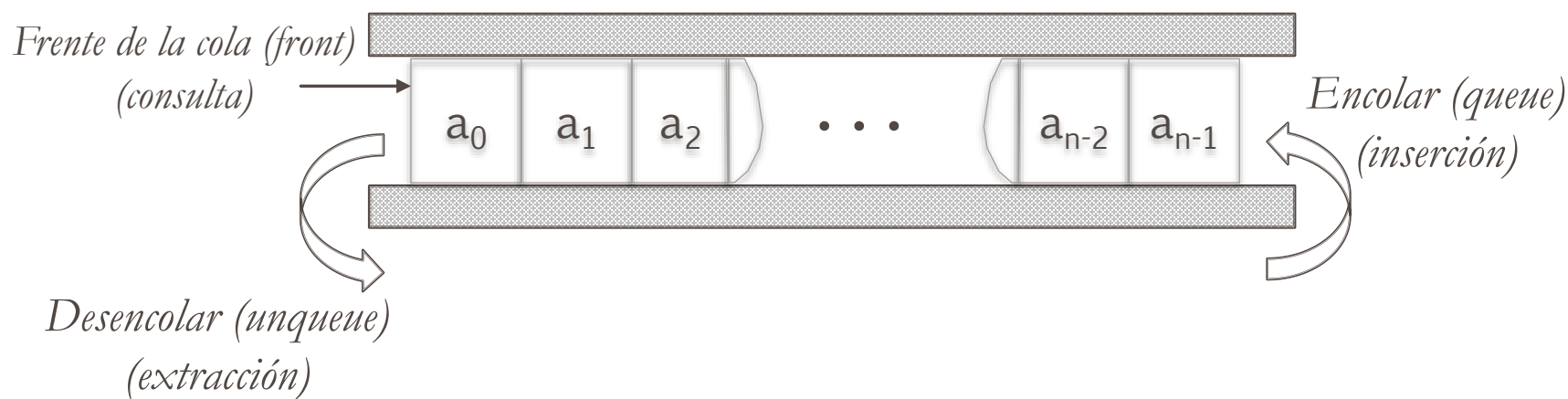
## Pilas (2)

---

- Diversa bibliografía de Estructuras de Datos trata las pilas independientemente de las colas y tienen una **interfaz** propia (habitualmente de nombre *Stack*).
  - Así ocurre en diversas bibliotecas, incluso en la biblioteca estándar de algunos lenguajes de programación (por ejemplo, en C++). Sin embargo, en diversos campos ambas estructuras se denotan simplemente como colas (por ejemplo, en Sistemas Operativos), considerándose las pilas como un caso particular de éstas: las colas LIFO.
  - En este sentido Java dispone de una única interfaz *Queue<E>* para distintos tipos de cola: FIFO, LIFO y otras.

# Colas (1)

- Cola
  - Secuencia de elementos con modo de acceso de tipo FIFO (*First In First Out*), primero en entrar primero en salir.
    - Las operaciones de inserción y extracción se realizan en extremos opuestos de la secuencia y deben ser de  $\Theta(1)$ .



## Colas (2)

- La interfaz Java *Queue<E>*
  - La interfaz incluye las operaciones de inserción, extracción y consulta que se indican en la tabla siguiente:

Operación	Lanzan excepción	Retornan un valor especial	
Inserción	<i>add(e)</i>	<i>offer(e)</i>	<b>false</b>
Extracción	<i>remove()</i>	<i>poll()</i>	<b>null</b>
Consulta	<i>element()</i>	<i>peek()</i>	<b>null</b>

- El valor espacial que se retorna es **null** o **false**, dependiendo de la operación.





## Colas (3)

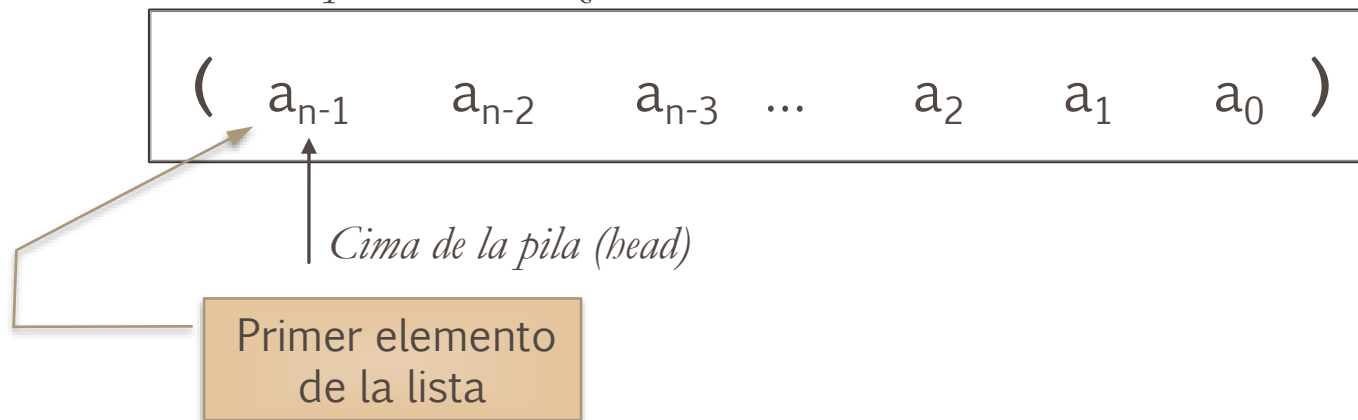
---

- Se denomina cabeza de la cola (*head*) a la posición del elemento que se extraerá mediante las operaciones *remove()* o *poll()*.
- El orden de los elementos en la cola, y por tanto cómo se realizan las operaciones de inserción, dependen del tipo de cola.
  - En una cola FIFO la inserción se realiza en el extremo opuesto a la cabeza (*tail*)
  - En una cola LIFO (o pila) la inserción se realiza en la cabeza de la cola (*head*)
  - En una cola de prioridad los elementos se ordenan según ésta y, por tanto, los elementos se insertan según su prioridad. Habitualmente, es el elemento de menor prioridad el que se encuentra en la cabeza de la cola (orden de prioridad creciente)

## Colas. Colas LIFO (4)

- Posibles representaciones para colas LIFO
  - Utilizando por composición una lista simplemente enlazada
    - En este caso la cima de la pila se posiciona al principio de la lista para que las operaciones puedan ser de tiempo constante (las operaciones tienen lugar en dicho extremo).

*Lista simplemente enlazada*



## Colas. Colas LIFO (5)

- Utilizando por composición las clases de Java que implementan la interfaz *List<E>* : *ArrayList<E>* y *LinkedList<E>*.
  - *ArrayList<E>*
    - Al igual que en el caso previo las operaciones sólo se pueden realizar en un extremo para que sean de tiempo constante. En este caso al final de la lista, que es el extremo que adjunta el espacio libre.

*ArrayList<E>*



Último elemento  
de la lista

Cima de la pila  
(head)

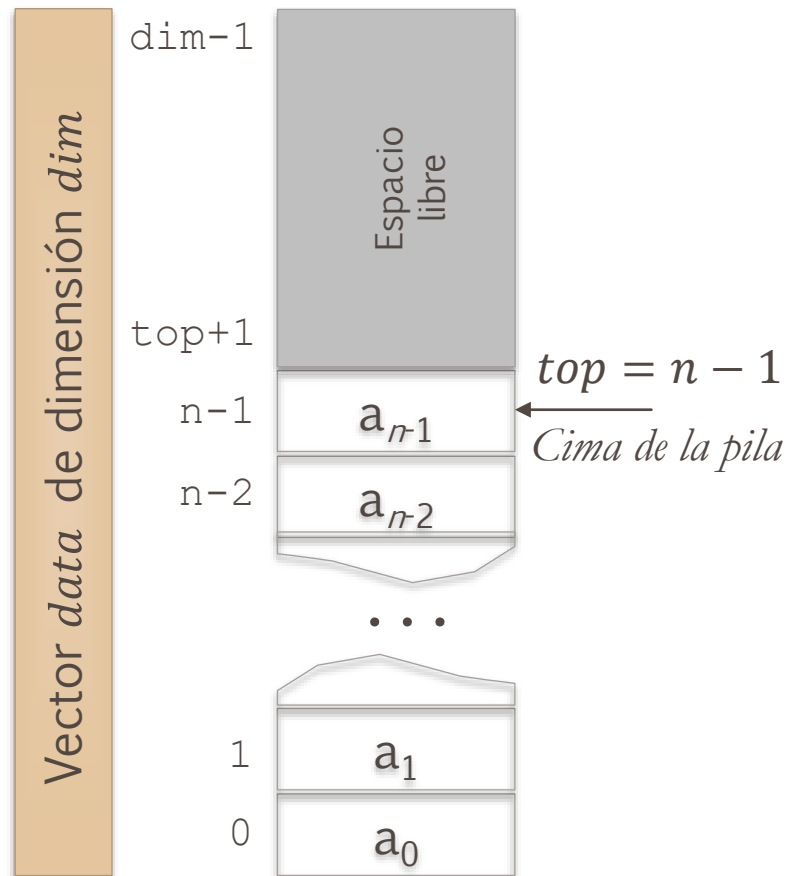


## Colas. Colas LIFO (6)

---

- *LinkedList<E>*
  - Estas listas están doblemente enlazadas y en la especificación se indica que dispone de operaciones eficientes para recuperar, insertar y borrar elementos en ambos extremos de la lista. En consecuencia, la cima de la pila (o cabeza de la cola) puede ubicarse en cualquiera de ambos extremos.
- Directamente mediante un *array*
  - Sería una representación análoga a la vista para la representación mediante una lista de tipo *ArrayList<E>*
    - Un vector de cierta dimensión, susceptible o no, de ser redimensionado en caso necesario. En caso de no serlo, estaríamos ante una cola LIFO (o pila) de capacidad fija.
    - Los elementos se apilan desde la posición 0 hasta el índice correspondiente a la cima de la pila (*top*), quedando el espacio libre en la parte superior del vector (tras la cima de la pila).

## Colas. Colas LIFO (6)



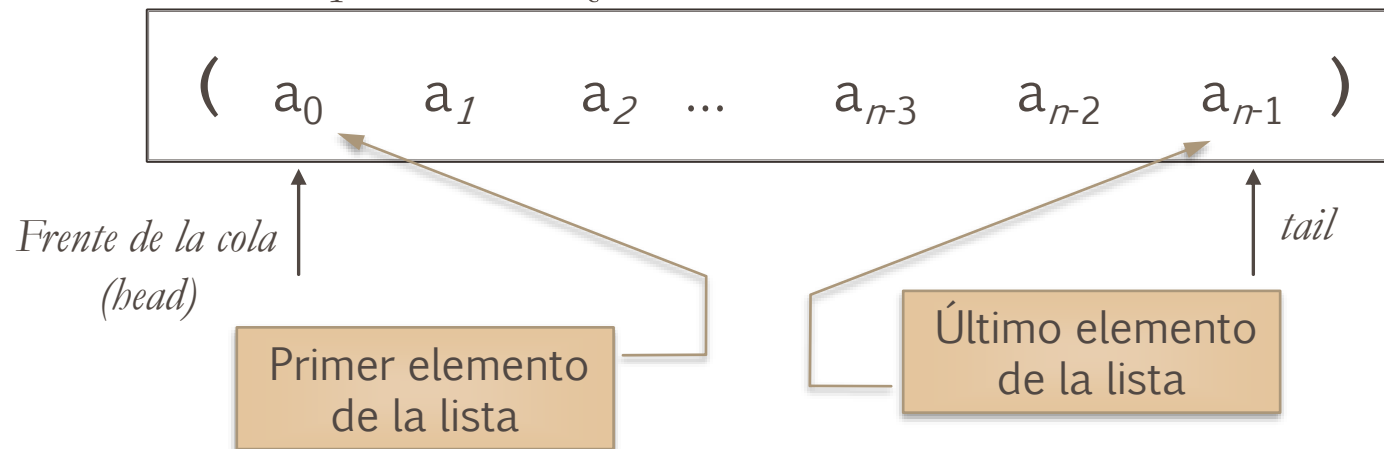
### Nota

Obsérvese que no puede utilizarse el mecanismo de herencia en ninguna de las representaciones dadas, ya que entonces se podrían realizar operaciones no aplicables a una pila. Por ejemplo, insertar o extraer elementos fuera de la cima de la pila.

## Colas. Colas FIFO (7)

- Posibles representaciones para colas FIFO
  - Utilizando por composición una lista simplemente enlazada que permita la inserción al final de la misma (operación *addLast(e)*) en tiempo constante (lista con centinela o posición del último nodo, *tail*, conocida).

*Lista simplemente enlazada con acceso aleatorio en ambos extremos*





## Colas (4)

---

- Utilizando por composición las clases de Java que implementan la interfaz *List<E>* : *LinkedList<E>*.
  - *ArrayList<E>*
    - Esta clase no se puede utilizar para implementar colas porque las operaciones de inserción y borrado en uno de sus extremos (al principio de la secuencia) son de  $\Theta(n)$ .
      - Más adelante, en dobles colas, se estudiará una representación basada en *arrays* que también se podría utilizar aquí.



## Colas (5)

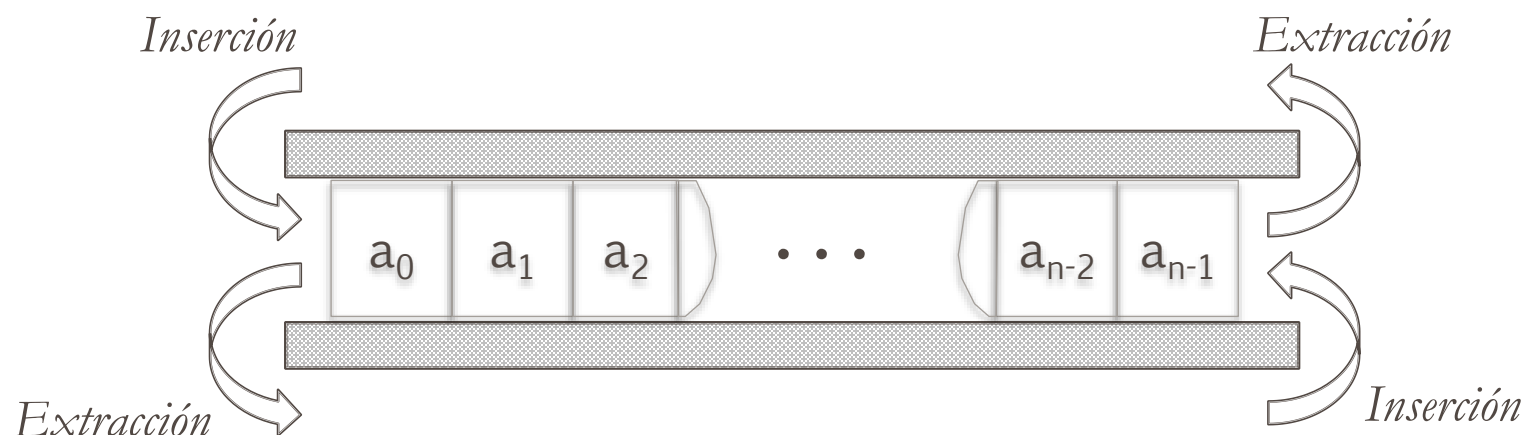
---

- *LinkedList<E>*
  - Las operaciones de inserción y extracción pueden realizarse en cualquiera de los extremos en tiempo constante. En consecuencia, el frente de la cola puede ubicarse en cualquiera de ellos.
    - De acuerdo con lo visto al principio de este apartado ([Colas \(1\)](#)), la operación de extracción se realizará en el extremo elegido para el frente de la cola (*head*) y la de inserción en el extremo opuesto (*tail*).



# Dobles colas (1)

- Doble cola
  - Secuencia de elementos en el que las operaciones de inserción, extracción y consulta, tienen lugar en ambos extremos de la misma (operaciones de  $\Theta(1)$ ).



## Dobles colas (2)

---

- La interfaz Java *Deque<E>*
  - Tipo de dato abstracto para colecciones lineales que soportan operaciones eficientes de inserción, consulta y extracción en ambos extremos. Representan dobles colas (también pilas y colas)
  - Extiende las interfaces siguientes: *Collection<E>*, *Iterable<E>* y *Queue<E>*
  - Además de la clase concreta *LinkedList<E>*, existe otra implementación concreta basada en *arrays* redimensionables, *ArrayDeque<E>*, que también se puede utilizar para colas.

## Dobles colas (3)

- Resumen de operaciones

	Primer Elemento ( <i>head</i> )		Último elemento ( <i>tail</i> )	
	<i>Lanza excepción</i>	<i>Valor especial</i>	<i>Lanza excepción</i>	<i>Valor especial</i>
Inserción	<i>addFirst(e)</i>	<i>offerFirst(e)</i>	<i>addLast(e)</i>	<i>offerLast(e)</i>
Extracción	<i>removeFirst()</i>	<i>pollFirst()</i>	<i>removeLast()</i>	<i>pollLast()</i>
Consulta	<i>getFirst()</i>	<i>peekFirst()</i>	<i>getLast()</i>	<i>peekLast()</i>

- Como esta interfaz extiende *Queue<E>*, también conviene conocer las equivalencias con las operaciones de ésta, las cuáles se indican en la tabla siguiente.

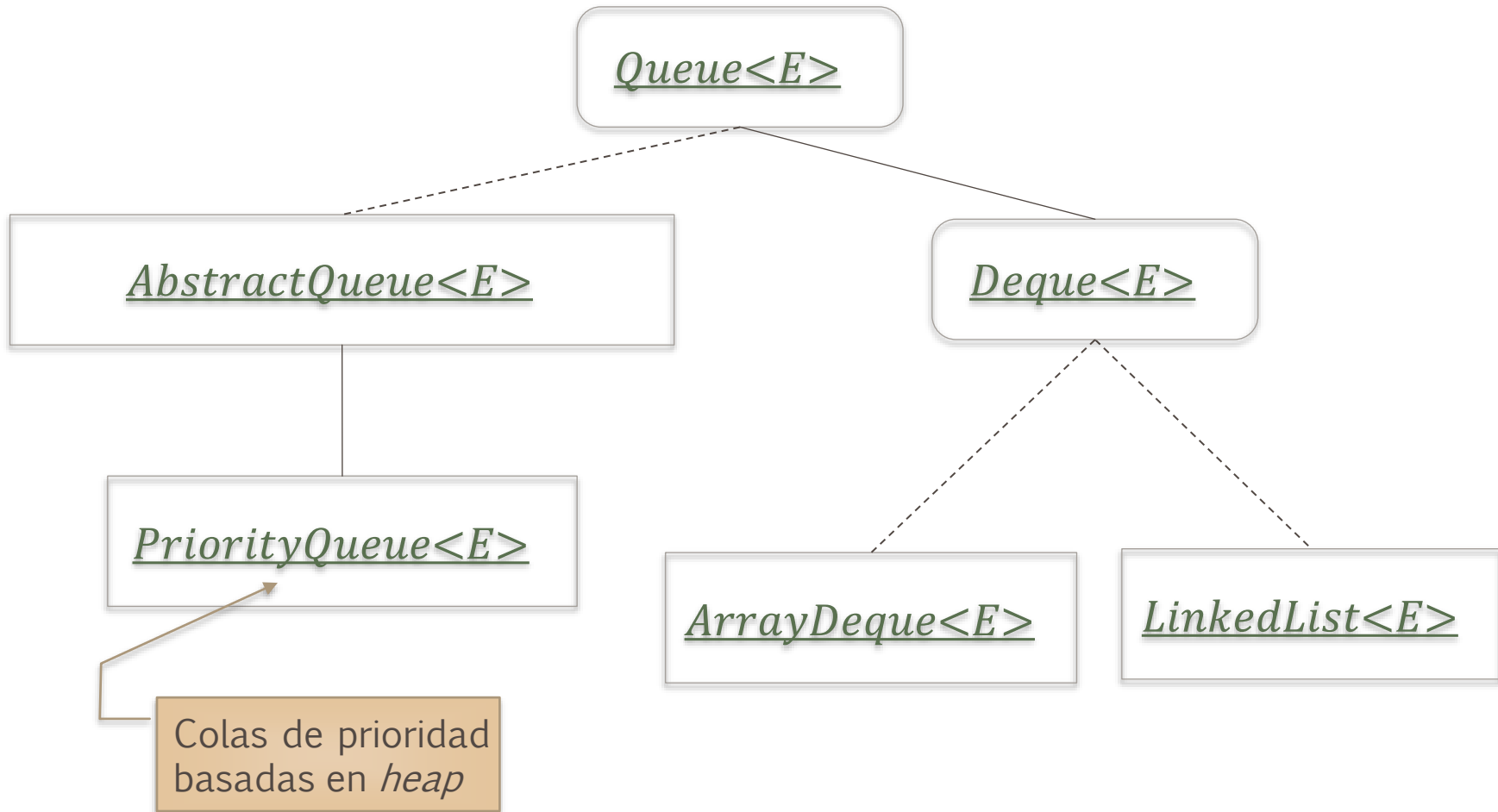
## Dobles colas (4)

Métodos de <i>Queue</i>	Métodos <i>Deque</i> equivalentes
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

- Obsérvese, que las equivalencias se corresponden con el manejo habitual de una cola FIFO.
  - Las operaciones de inserción por el final (*tail*)
  - Las operaciones de extracción y consulta por el principio (*head*)

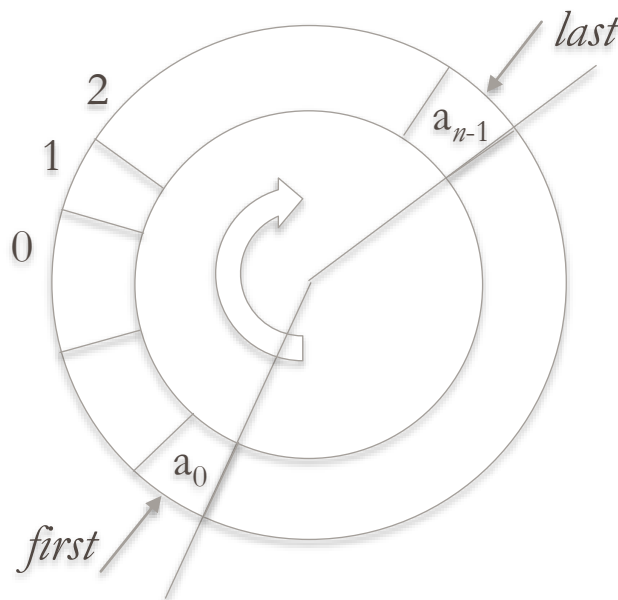


## Dobles colas (5)



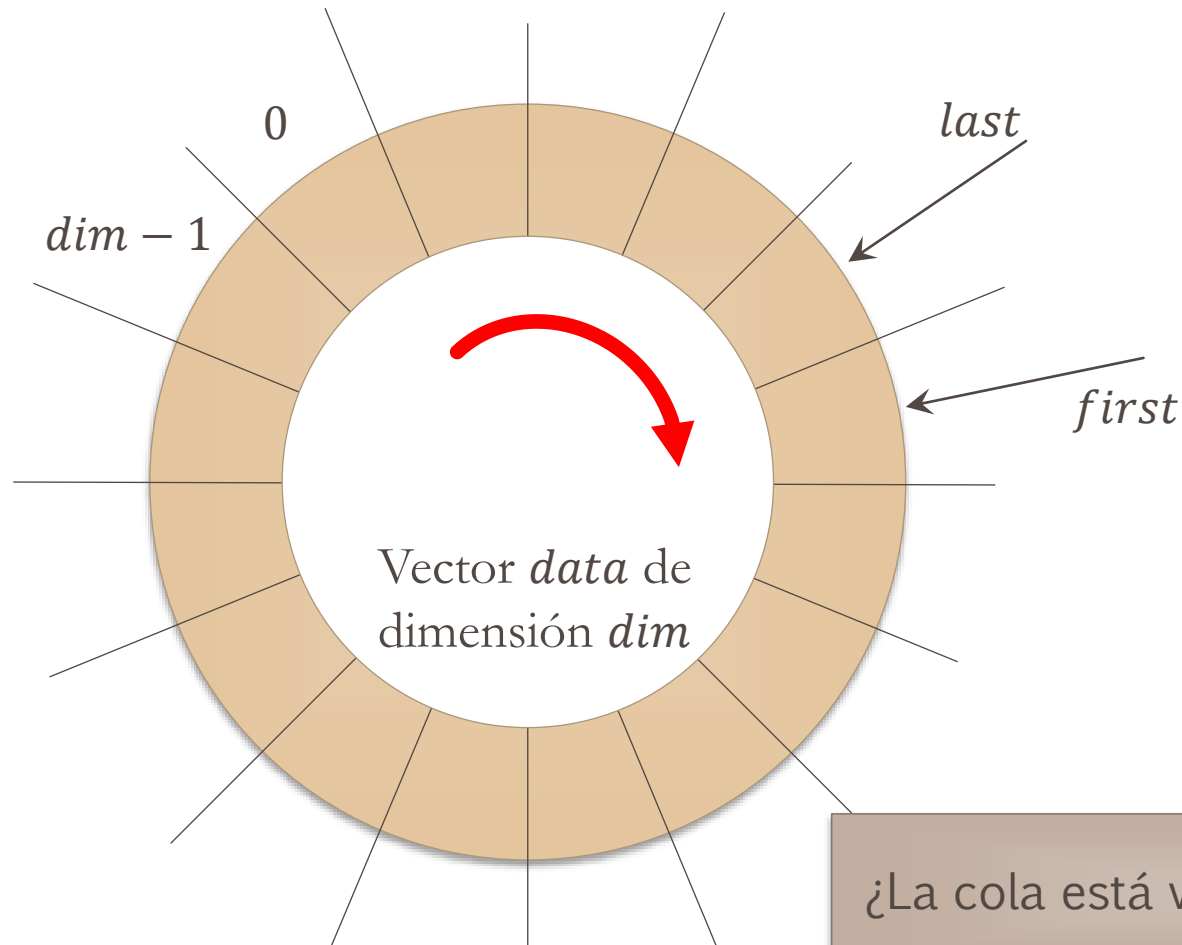
## Dobles colas (6)

- Posibles representaciones
  - Utilizando por composición la clase *LinkedList*<E>.
  - Utilizando por composición un *array* circular.



```
// área de datos  
E[] data; // el array  
int first; // posición del primer elemento  
int last; // posición del último elemento  
int size; // número de elementos
```

## Dobles colas (7)





# Resumen

