Estructuras de datos y algoritmos

UNIVERSIDAD DE CANTABRIA

1. Introducción
2. Estructuras de datos lineales
3. Estructuras de datos jerárquicas
4. Grafos y caminos
5. ***Implementación de listas, colas y pilas***
6. Implementación de mapas, árboles y grafos

© Michael González Harbour
17/nov/09

UNIVERSIDAD DE CANTABRIA

5. Implementación de listas, colas y pilas

- 5.1. Introducción
- 5.2. Pilas, colas y vectores implementados mediante arrays
- 5.3. Implementaciones con listas enlazadas simples
- 5.4. Listas enlazadas con cursores
- 5.5. Listas doblemente enlazadas

5.1. Introducción

UNIVERSIDAD DE CANTABRIA

Motivos para crear estructuras de datos propias

- *persistente*: estructura de datos que reside en disco
- *específicas de la aplicación*: por ejemplo, mapa con elementos no modificables, adaptación a los datos concretos de la aplicación
- *optimizadas*: en tiempo de ejecución o en espacio
- *funcionalidad extra*: por ejemplo ordenación especial, ...
- *comodidad*: por ejemplo dar implementaciones más simples o en otros idiomas, ...
- *adaptación* de otras estructuras ya realizadas

Implementación de estructuras de datos propias para colecciones Java

En las colecciones java se definen implementaciones de clases *abstractas*

- pensadas para facilitar la creación de estructuras de datos *propias*
- implementan parte de los métodos, para ahorrar trabajo
  - estos métodos se pueden redefinir si se cree conveniente

Las clases abstractas que se proporcionan son

- `AbstractCollection` — una Colección que no es ni un Conjunto ni una Lista
  - como mínimo hay que proporcionar el iterador y el método `size`
- `AbstractSet` — un Conjunto;
  - como mínimo hay que proporcionar el iterador y el método `size`

Implementación de estructuras de datos propias para colecciones Java

- `AbstractList` — una lista en la que los elementos se guardan en un array o similar
    - como mínimo hay que proporcionar los métodos de acceso posicional (`get` y opcionalmente `set`, `remove` y `add`) y el método `size`
    - la clase ya proporciona el iterador de listas
- `AbstractSequentialList` — una lista en la que los elementos se guardan en una estructura de datos secuencial (por ejemplo una lista enlazada)
    - como mínimo hay que proporcionar el iterador de lista y el método `size`
    - la clase abstracta proporciona los métodos posicionales

Implementación de estructuras de datos propias para colecciones Java

- `AbstractQueue` — una cola
  - como mínimo hay que proporcionar `offer`, `peek`, `poll`, y `size` y un iterador que soporte `remove`
- `AbstractMap` — un Mapa
  - como mínimo hay que proporcionar la vista `entrySet`
  - habitualmente esto se hace con la clase `AbstractSet`
  - si el mapa es modificable, como mínimo hay que proporcionar el método `put`

Proceso para escribir una implementación propia

UNIVERSIDAD DE CANTABRIA

1. Elegir la clase abstracta apropiada
2. Proporcionar implementaciones para los métodos abstractos
3. Si la colección es modificable, será necesario redefinir también algunos métodos concretos
   - El manual nos describe lo que hace cada uno
4. Codificar y probar la implementación

Relaciones entre datos

UNIVERSIDAD DE CANTABRIA

En muchas estructuras de datos es preciso establecer *relaciones* o *referencias* entre diferentes datos

- ahorran espacio al no repetir datos
- evitan inconsistencias

Si los datos están en un array, se pueden utilizar *cursores*

- el cursor es un entero que indica el número de la casilla del array donde está el dato

Si la lista de alumnos no es un array deben utilizarse *referencias* o *punteros* (si los soporta el lenguaje de programación)

- son datos especiales que sirven para apuntar o referirse a otros datos

Ejemplo: listas de alumnos de asignaturas

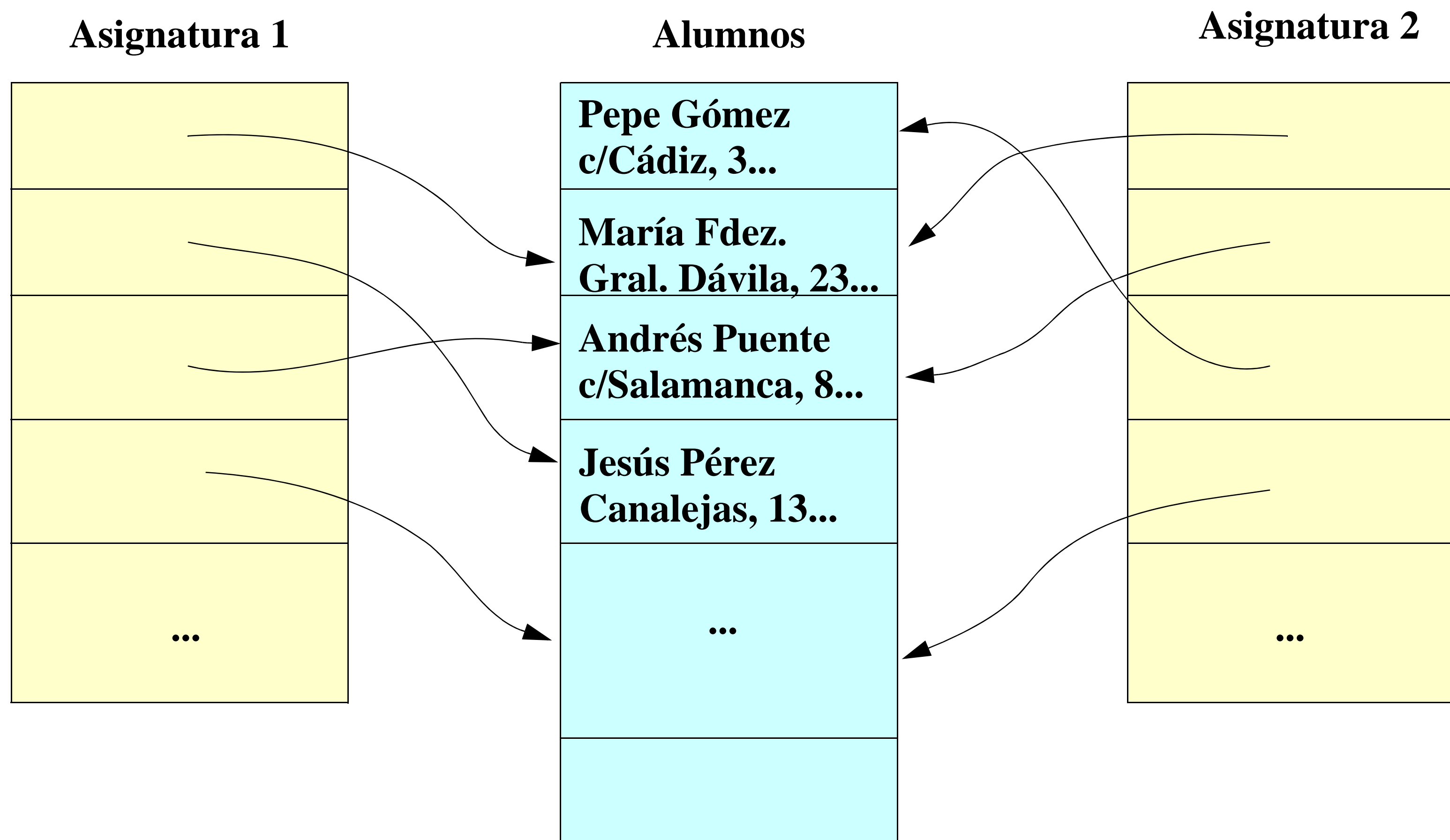
UNIVERSIDAD DE CANTABRIA

Asignatura 1
María Fdez. Gral. Dávila, 23...
Jesús Pérez Canalejas, 13...
Andrés Puente c/Salamanca, 8...

Asignatura 2
María Fdez. Gral. Dávila, 23...
Andrés Puente c/Salamanca, 8...
Pepe Gómez c/Cádiz, 3...

¡Hay datos repetidos!

Alternativa: Referencias entre datos



Punteros

UNIVERSIDAD DE CANTABRIA

Las referencias, también llamadas punteros o tipos acceso, proporcionan acceso a otros objetos de datos

En Java, todos los objetos y arrays se usan a través de referencias

- las variables de los tipos primitivos no
  - enteros, reales, caracteres, booleanos
- pero tienen clases asociadas que permiten usar estos datos como objetos
  - ej., `Integer`, `Double`, `Boolean`, `Character`

Hay un valor predefinido, `null`, que es el valor por omisión, y no se refiere a ningún dato

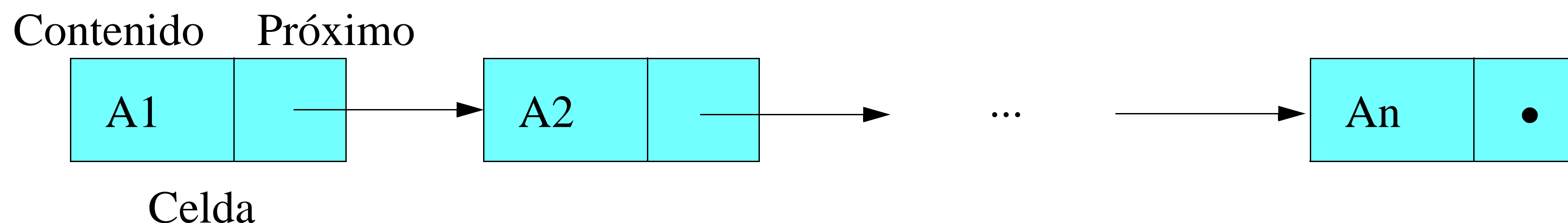
© Michael González Harbour
17/nov/09

Estructuras de datos dinámicas

UNIVERSIDAD DE CANTABRIA

Las referencias son útiles cuando se usan para crear estructuras de datos con relaciones entre objetos

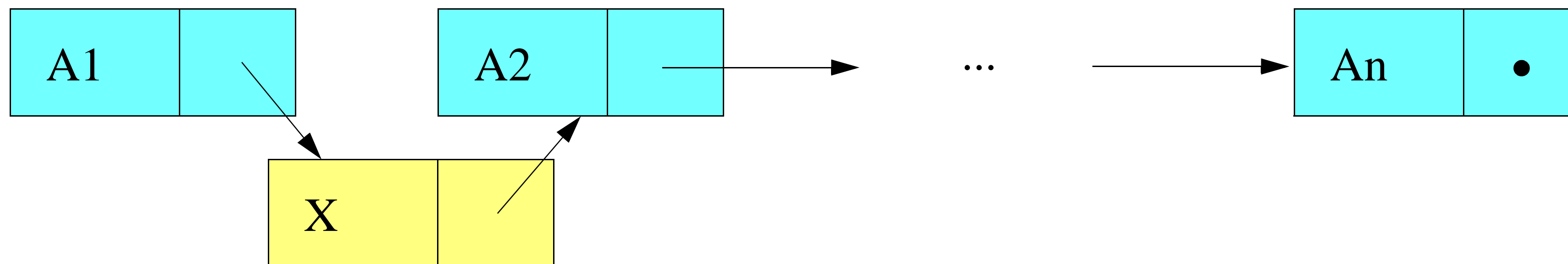
Por ejemplo, una lista enlazada



- **cada elemento tiene un contenido y un puntero al *próximo***
- **el último tiene un puntero próximo *nulo***

Flexibilidad de la estructura de datos dinámica

Se pueden insertar nuevos elementos en la posición deseada, eficientemente:



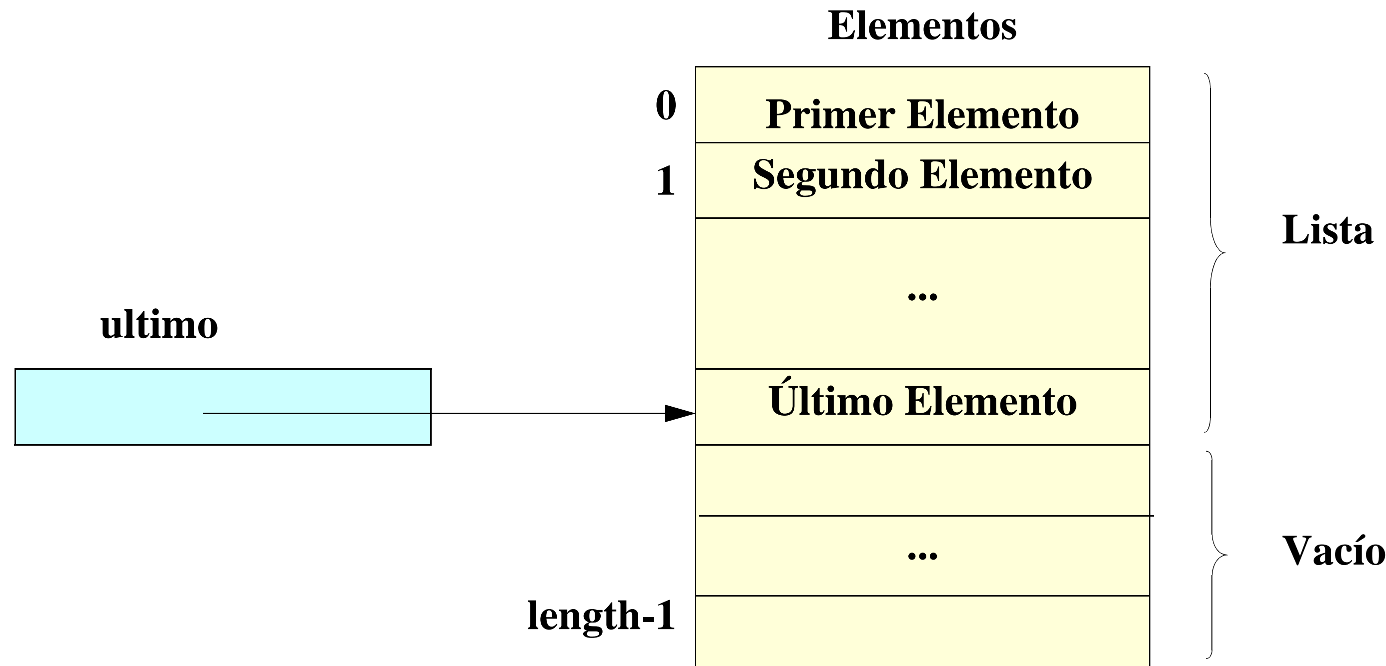
Diferencias con el array:

- los arrays tienen tamaño fijo: ocupan lo mismo, incluso medio vacíos
- con estructuras de datos dinámicas se gasta sólo lo preciso
- pero se necesita espacio para guardar los punteros

5.2. Pilas, colas y vectores implementados mediante arrays

UNIVERSIDAD DE CANTABRIA

La lista se representa mediante un array en el que cada casilla almacena un elemento, y los elementos se ordenan según el índice de la casilla



Cambio dinámico de tamaño

UNIVERSIDAD DE CANTABRIA

Cuando la lista está llena y se intenta añadir un nuevo elemento

- lista *acotada*: indicar un error
- lista *ilimitada*: cambiar el tamaño del array

El cambio de tamaño directo no es posible en Java ni en la mayoría de los lenguajes

- crear un nuevo array (por ejemplo del doble de tamaño)
- copiar todos los elementos en el nuevo array
  - en el momento del cambio
  - o poco a poco, en sucesivas operaciones (*amortización*)
- eliminar el array viejo (en Java esto es automático)

5.2.1. Implementación de las operaciones de las listas

UNIVERSIDAD DE CANTABRIA

El acceso posicional es inmediato

- en el array tenemos acceso posicional de forma natural

La inserción al final es eficiente: O(1)

- incrementar el cursor `ultimo`
- meter el nuevo elemento en la casilla `ultimo`

La inserción en la posición `i` es O(n)

- hacer hueco
  - mover las casillas en el rango `[i,ultimo]` una casilla hacia adelante, yendo desde el final al principio
- meter el nuevo elemento en la casilla `i`
- incrementar el cursor `ultimo`

Implementación de las operaciones de las listas (cont.)

La eliminación de la posición `i` es O(n)

- almacenar el elemento de la casilla `i` en una variable
- cerrar el hueco
  - mover las casillas en el rango `[i+1,ultimo]` una casilla hacia atrás, yendo del principio hacia el final
- decrementar el cursor `ultimo`
- retornar la variable con el elemento borrado

Iterador

- se implementa con un cursor que apunta al elemento *próximo*

Ejemplo: implementación de una lista acotada en Java

Queremos implementar algo similar al `ArrayList`, pero que no cambie de tamaño si no cabe un elemento

- puede ser útil en sistemas con listas de tamaño limitado
- o en sistemas de tiempo de respuesta predecible
  - sistemas de tiempo real
- si se excede el tamaño es por un error: conviene indicarlo

En las colecciones Java disponemos de clases abstractas que facilitan la implementación de las diferentes interfaces

Código Java de la lista limitada

```java
import java.util.*;

/**
 * Clase con una lista implementada en un array
 * Se limita el numero de elementos que pueden
 * meterse
 */
public class ListaArrayAcotada<E>
    extends AbstractList<E>
{
  public static int maxPorOmision=50;

  // El array y el cursor ultimo
  private E[] lista;
  private int ultimo;
```

Código Java de la lista limitada (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Constructor.Se le pasa tamaño máximo de la lista
 */
public ListaArrayAcotada(int max) {
    lista=(E[]) new Object[max];
    ultimo=-1;
}
/**
 * Constructor que crea la lista con un tamaño igual
 * a maxPorOmision
 */
public ListaArrayAcotada() {
    this(maxPorOmision);
}
```

Código Java de la lista limitada (cont.)

```
/**
 * Constructor. Se le pasa colección para copiarla
 */
public ListaArrayAcotada(Collection<E> c) {
   // crea la lista del tamaño de la colección
   this(c.size());
   // anade todos los elementos a la lista
   for (E e:c) {
      add(ultimo+1,e);
   }
}
```

Código Java de la lista limitada (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Lectura posicional: get
 * Lanza IndexOutOfBoundsException si
 * el indice es incorrecto
 */
public E get(int indice) {
   // comprueba errores
   if (indice>ultimo) {
      throw new IndexOutOfBoundsException();
   }
   // retorna el elemento
   return lista[indice];
}
```

Código Java de la lista limitada (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Escritura posicional: set
 * Lanza IndexOutOfBoundsException si
 * el indice es incorrecto
 */
public E set(int indice, E elemento) {
  // comprueba errores
  if (indice>ultimo) {
     throw new IndexOutOfBoundsException();
  }
  // escribe el elemento y retorna el antiguo
  E antiguo=lista[indice];
  lista[indice]=elemento;
  return antiguo;
}
```

Código Java de la lista limitada (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Añadir posicional: add
 * Lanza IndexOutOfBoundsException si el indice es
 * incorrecto; Lanza IllegalStateException si el
 * elemento no cabe
 */
public void add(int indice, E elemento) {
  // comprueba errores
  if (indice>ultimo+1) {
     throw new IndexOutOfBoundsException();
  }
  if (ultimo==lista.length-1) {
      throw new IllegalStateException();
  }
```

Código Java de la lista limitada (cont.)

UNIVERSIDAD DE CANTABRIA

```
    // desplaza elementos hacia adelante
    for (int i=ultimo; i>=indice; i--) {
        lista[i+1]=lista[i];
    }
    // añade el elemento
    lista[indice]=elemento;
    ultimo++;
}
```

Código Java de la lista limitada (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Borrar posicional. Si el indice es incorrecto
 * Lanza IndexOutOfBoundsException */
public E remove(int indice) {
  if (indice>ultimo) {
     throw new IndexOutOfBoundsException();
  }
  E borrado=lista[indice];
  // desplaza elementos hacia atrás
  for (int i=indice+1; i<=ultimo; i++) {
     lista[i-1]=lista[i];
  }
  ultimo--;
  return borrado;
}
```

Código Java de la lista limitada (cont.)

UNIVERSIDAD DE CANTABRIA

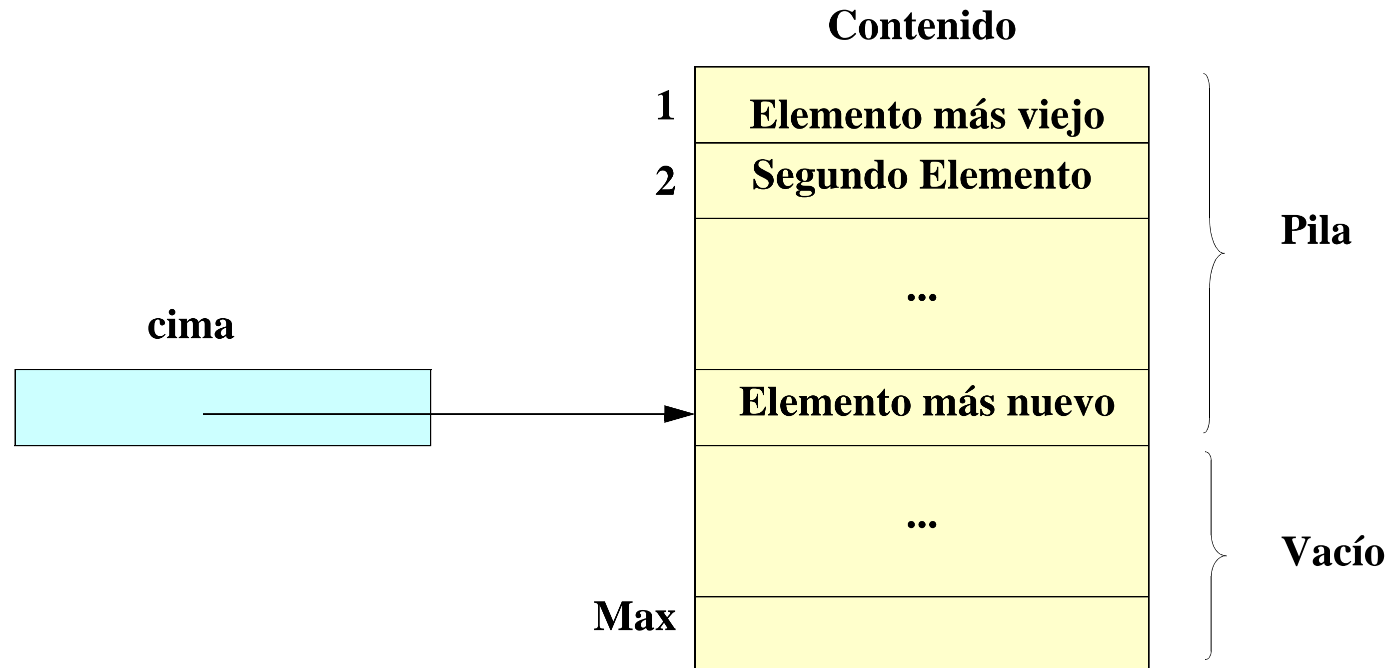
```
  /**
   * Retorna el tamaño: size
   */
  public int size() {
     return ultimo+1;
  }
}
```

5.2.2. Implementación de Pilas con array simple

UNIVERSIDAD DE CANTABRIA

Los elementos se guardan en un array

El extremo activo se guarda en el atributo `cima`



Implementación de las operaciones

UNIVERSIDAD DE CANTABRIA

Constructor que crea la pila vacía (y hazNula)

- hace la `cima=-1`

Apila

- si la pila está llena lanza `NoCabe` o agranda el array
- incrementa el cursor `cima`
- mete el nuevo elemento en la casilla `cima`

Desapila

- si la pila está vacía lanza `noExiste`
- decrementa el cursor `cima`
- retorna el elemento de la casilla `cima+1`

Implementación de las operaciones (cont.)

UNIVERSIDAD DE CANTABRIA

Cima

- si la pila está vacía lanza `noExiste`
- retorna el elemento de la casilla `cima`

EstaVacia

- retorna `cima==-1`

Tamaño

- retorna `cima+1`

Implementación en Java de la pila limitada de array simple

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;
/**
 * Pila implementada mediante un array simple
 */
public class PilaArraySimple<E> implements Pila<E>
{
    public static int maxPorOmision=50;

    // atributos de la pila
    private E[] contenido;
    private int cima;
    /**
     * Constructor que crea una pila vacía cuyo
     * tamano es maxPorOmision
     */
```

Implementación en Java de la pila limitada de array simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
public PilaArraySimple()
{
    this(maxPorOmision);
}

/**
 * Constructor que crea una pila vacía cuyo
 * tamano es max
 */
public PilaArraySimple(int max)
{
    contenido = (E[]) new  Object[max];
    cima=-1;
}
```

Implementación en Java de la pila limitada de array simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Apilar un elemento
 * Lanza IllegalStateException si está llena
 */
public void apila(E e) {
    if (estaLlena()) {
        throw new IllegalStateException();
    }
    cima++;
    contenido[cima]=e;
}
```

Implementación en Java de la pila limitada de array simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Desapilar un elemento
 */
public E desapila() throws NoSuchElementException
{
    if (estaVacia()) {
        throw new NoSuchElementException();
    }
    cima--;
    return contenido[cima+1];
}
```

Implementación en Java de la pila limitada de array simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 *  Retornar el elem. de la cima, sin desapilarlo
 */
public E cima() throws NoSuchElementException {
    if (cima==-1) {
        throw new NoSuchElementException();
    } else {
        return contenido[cima];
    }
}
```

Implementación en Java de la pila limitada de array simple (cont.)

UNIVERSIDAD DE CANTABRIA

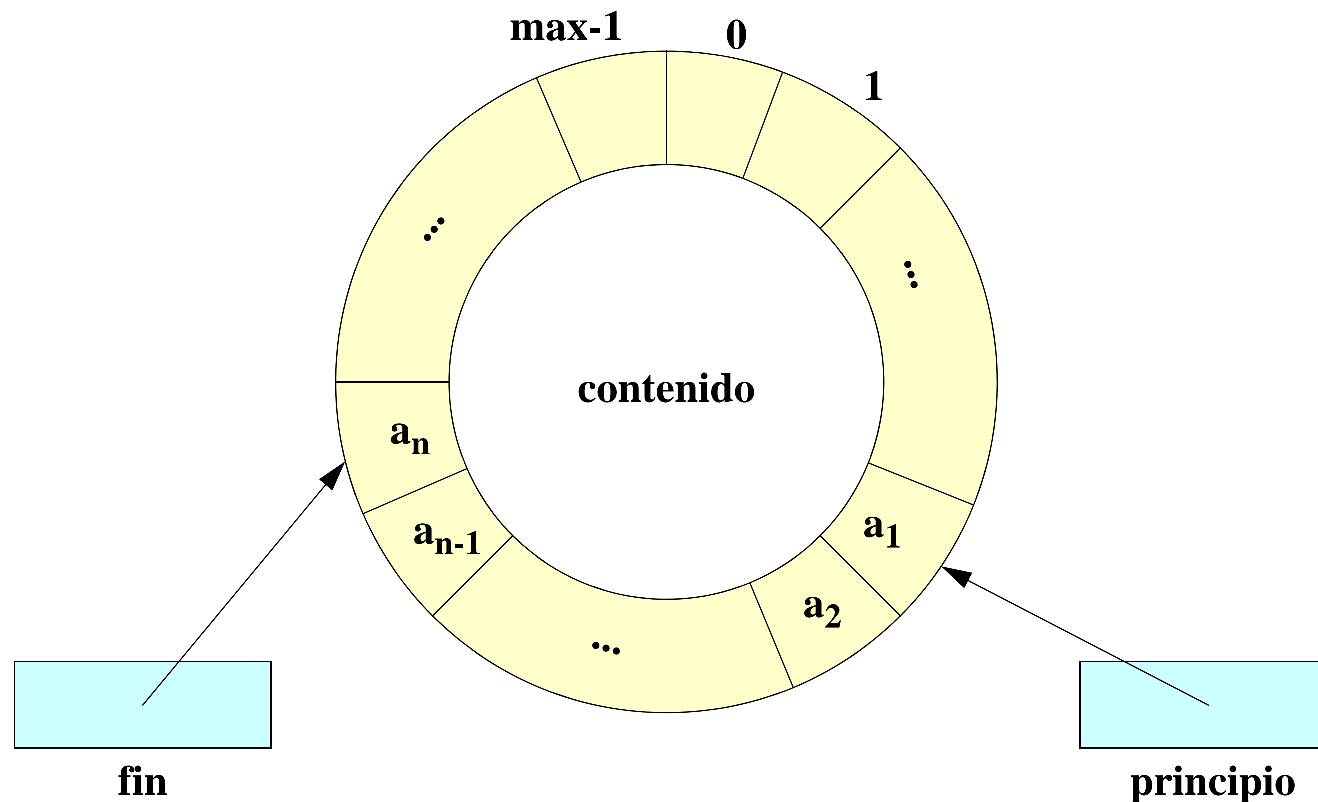
```java
/**
 * Saber si la pila esta vacía
 */
public boolean estaVacia() {
     return cima==-1;
}

/**
 * Saber si la pila esta llena
 */
public boolean estaLlena() {
     return cima==contenido.length-1;
}
```

Implementación en Java de la pila limitada de array simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
    /**
     * Hacer nula la pila
     */
    public void hazNula() {
        cima=-1;
    }

    /**
     * Retornar el numero de elementos en la pila
     */
    public int tamano() {
        return cima+1;
    }

}
```

5.2.3. Implementación de colas mediante arrays circulares



Elementos de la implementación mediante cola circular

UNIVERSIDAD DE CANTABRIA

Array circular

- `contenido`: es un array lineal en el que se considera que el elemento siguiente al último es el primero

Cursores

- `principio`: primer elemento
- `fin`: último elemento

Almacenamiento de la situación de *cola vacía*

- dejando siempre al menos un hueco vacío, para distinguir la cola llena de la cola vacía
- o almacenando el número de elementos, o un booleano que diga si la cola está vacía o no

Operaciones con colas circulares (dejando un hueco vacío)

UNIVERSIDAD DE CANTABRIA

Definimos la operación de incrementar un cursor como:

- `x⊕n= (x+n) mod max`
- sirve para tratar el array como circular

Constructor, hazNula

- `principio=0`
- `fin=max-1`

Está vacía

- `fin⊕1==principio`

Está Llena

- `fin⊕2==principio`

Operaciones con colas circulares (dejando un hueco vacío)

UNIVERSIDAD DE CANTABRIA

Tamaño

- `(fin-principio+max+1) mod max`

Encola

- si la cola está llena lanza `NoCabe`
- `fin=fin⊕1`
- meter el elemento nuevo en la casilla `fin`

Desencola

- si la cola está vacía lanza `NoExiste`
- el elemento a retornar es el de la casilla `principio`
- `principio=principio⊕1`

Iterador de la cola

UNIVERSIDAD DE CANTABRIA

El iterador es un cursor al elemento siguiente

siguiente:

- avanzar (circularmente) el cursor siguiente

hay siguiente:

- es falso si el cursor está después del final de la cola

borrar:

- lo dejaremos sin implementar, ya que el borrado en mitad de la cola sería muy poco eficiente

Implementación en Java de la cola limitada con array circular

```java
import java.util.*;

/**
 * Clase que representa una cola implementada mediante
 * un array circular
 */
public class ColaCircular<E> extends AbstractQueue<E>
{
  public static int maxPorOmision=50;

  // atributos de la cola
  private E[] contenido;
  private int principio,fin;
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Constructor que crea una cola vacía cuyo
 * tamano es maxPorOmision
 */
public ColaCircular()
{
   this(maxPorOmision);
}
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Constructor que crea una cola vacía cuyo
 * tamano es max
 */
public ColaCircular(int max)
{
   contenido = (E[]) new Object[max+1];
   principio=0;
   fin=contenido.length-1;
}

/**
 * Constructor que crea una cola cuyos elementos
 * obtiene de la colección que se le pasa
 */
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
public ColaCircular(Collection<E> c) {
    // crea la cola del tamano de la colección
    this(c.size());
    // añade todos los elementos a la cola
    for (E e:c) {
        offer(e);
    }
}

// incrementar un cursor
private int incr(int x, int n) {
     return (x+n)%contenido.length;
}
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Añade un elem. a la cola; retorna false si no cabe
  * y true si cabe; no admite elementos nulos
  */
public boolean offer(E e) {
    if (isFull() || e==null) {
        return false;
    } else {
        fin=incr(fin,1);
        contenido[fin]=e;
        return true;
    }
}
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Elimina y retorna un elemento de la cola
 * retorna null si no hay elementos
 */
public E poll() {
    if (isEmpty()) {
        return null;
    } else {
        E borrado=contenido[principio];
        principio=incr(principio,1);
        return borrado;
    }
}
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Retorna el primer elem. de la cola sin eliminarlo
 * retorna null si no hay elementos
 */
public E peek() {
    if (isEmpty()) {
        return null;
    } else {
        return contenido[principio];
    }
}
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Retorna el número de elementos
 */
public int size() {
   return (fin-principio+contenido.length+1)%
          contenido.length;
}

/**
 *  Indica si la cola está vacía
 */
public boolean isEmpty() {
   return incr(fin,1)==principio;
}
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 *  Indica si la cola esta llena
 */
public boolean isFull() {
   return incr(fin,2)==principio;
}

/**
 * Clase iterador de la cola circular
 */
public static class ColaCircularIterator<E>
implements Iterator<E>
{
   private int siguiente;
   private ColaCircular<E> cola;
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Constructor al que se le pasa la cola
 * No es público, para que se use el metodo
 * iterator() al crearlo
 */

ColaCircularIterator(ColaCircular<E> cola) {
    this.cola=cola;
    siguiente=cola.principio;
}
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtener el proximo elemento y
 * avanzar el iterador
 */
public E next() {
    if (hasNext()) {
        E sig=cola.contenido[siguiente];
        siguiente=cola.incr(siguiente,1);
        return sig;
    } else {
        throw new NoSuchElementException();
    }
}
```

Implementación en Java de la cola limitada con array circular (cont.)

UNIVERSIDAD DE CANTABRIA

```
    /**
     * Indicar si hay elemento siguiente
     */
    public boolean hasNext() {
        return ! (cola.incr(cola.fin,1)==siguiente);
    }

    /**
     * Borra no se implementa por poco eficiente
     */
     public void remove() {
      throw new  UnsupportedOperationException();
     }
} // fin de la clase iterador
```

Implementación en Java de la cola limitada con array circular (cont.)

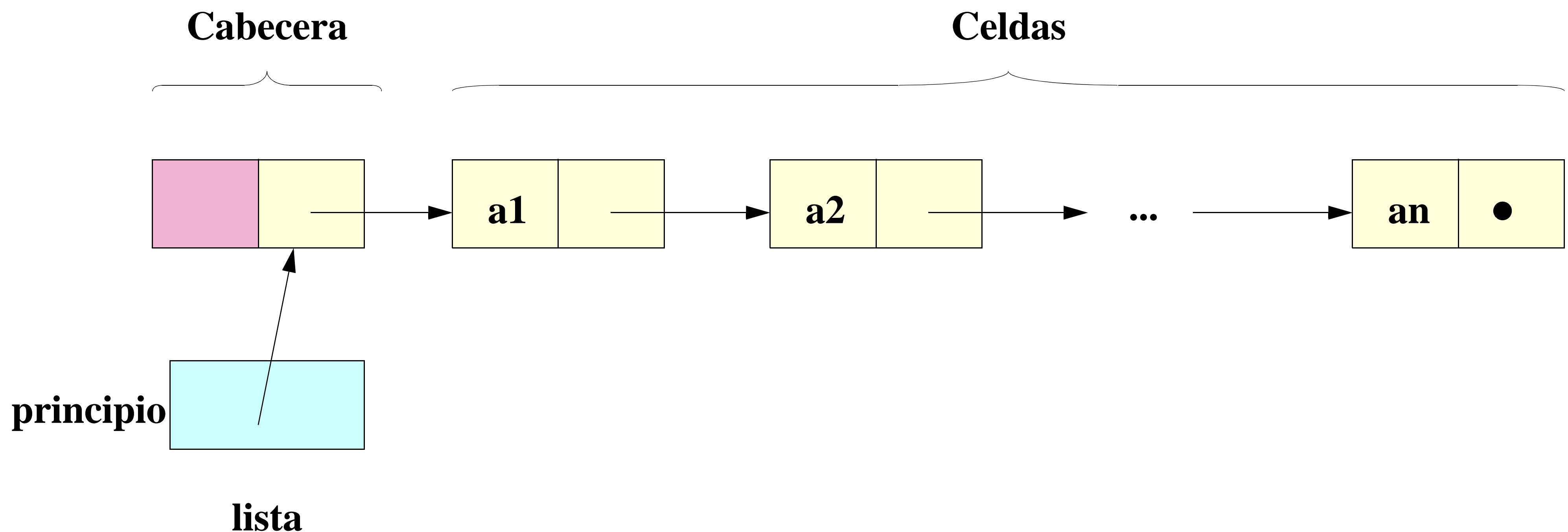
UNIVERSIDAD DE CANTABRIA

```
/**
 * Iterador de la cola
 */
public Iterator<E> iterator() {
   return new ColaCircularIterator<E>(this);
}

} // fin de la clase ColaCircular
```

5.3. Listas y colas implementadas con listas enlazadas simples

UNIVERSIDAD DE CANTABRIA

La lista de celdas simplemente enlazadas con punteros

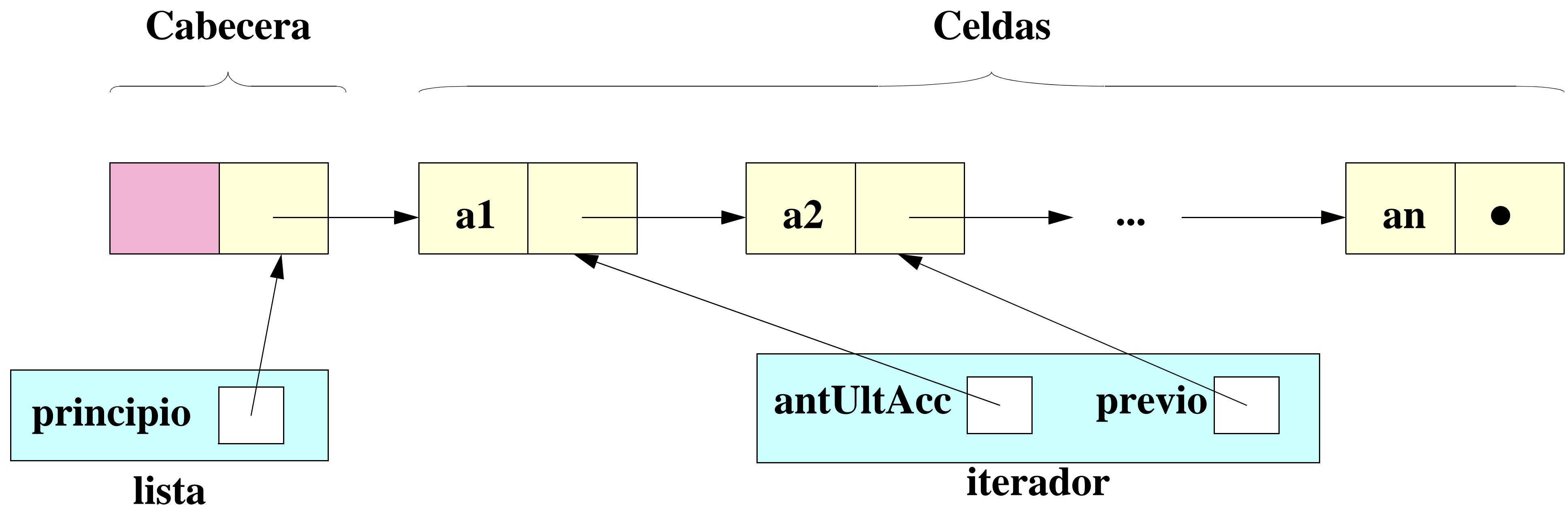


- el primer elemento (*cabecera*) está vacío
- la lista contiene un puntero (`principio`) a la cabecera

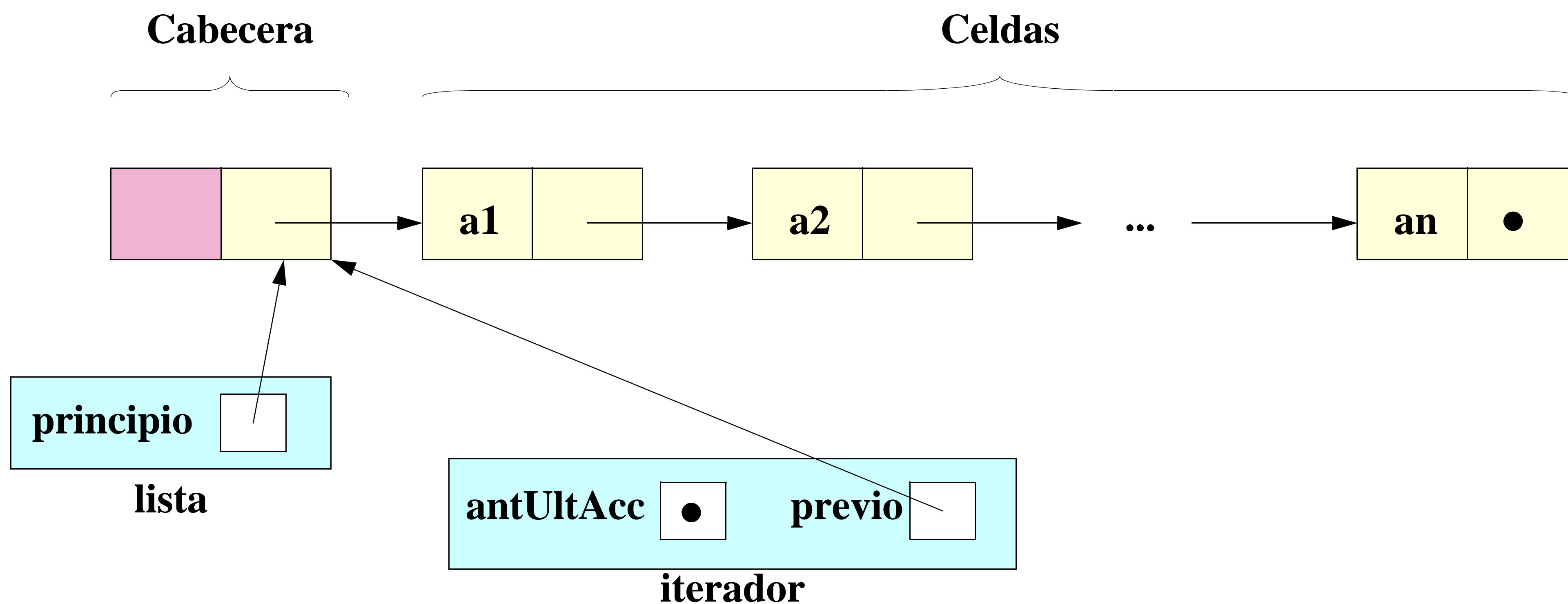
5.3.1. Implementación de listas: Iterador de la lista

El iterador de la lista contiene

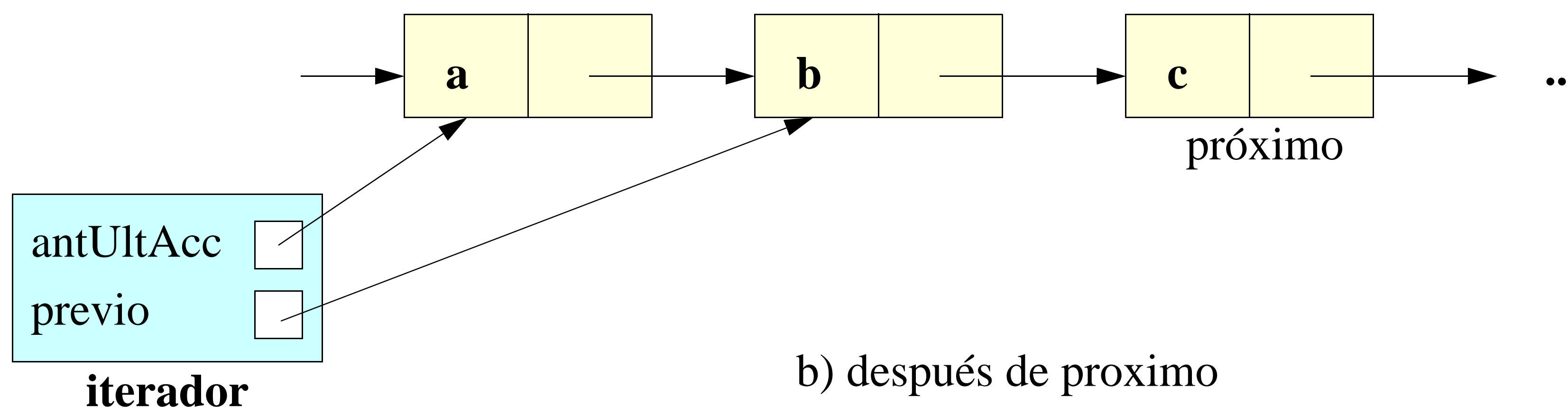
- `previo`: un puntero al elemento previo
- `antUltAcc`: un puntero al elemento anterior al último elemento accedido a través del iterador (con `proximo` o `previo`)

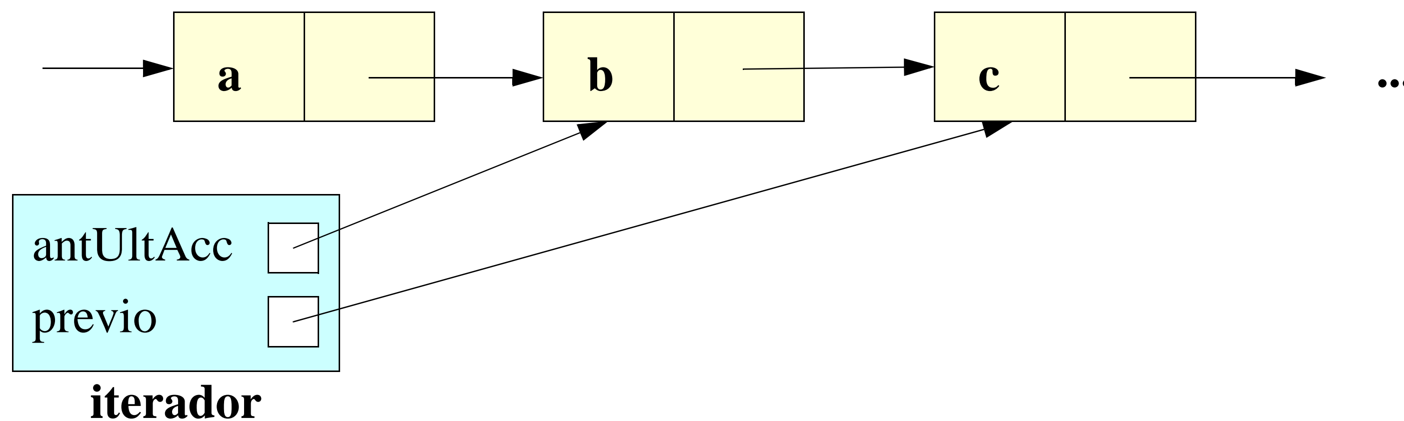


Situación inicial del iterador



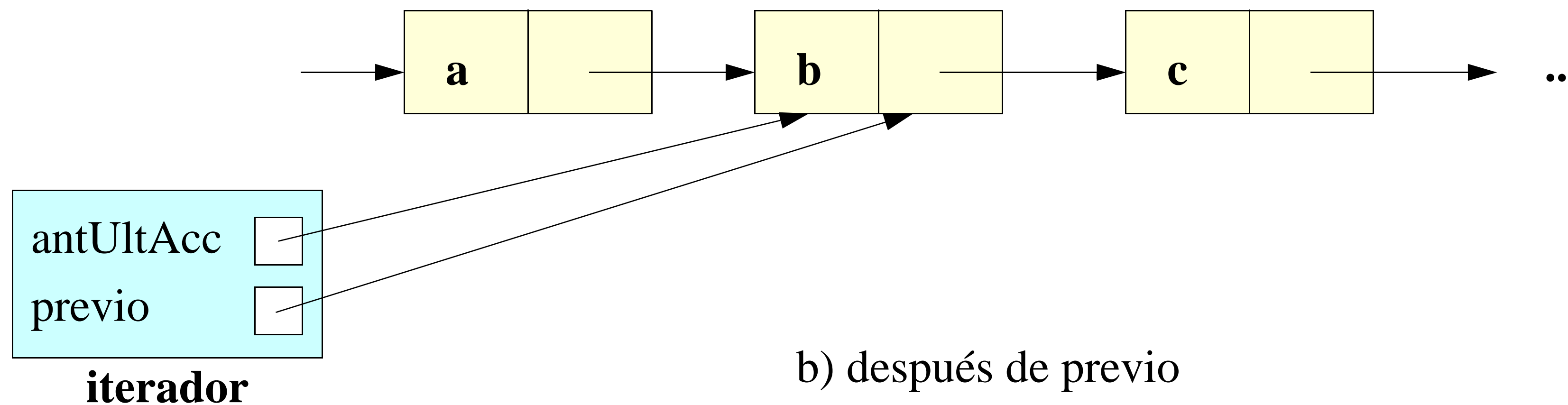
Avance del iterador: próximo

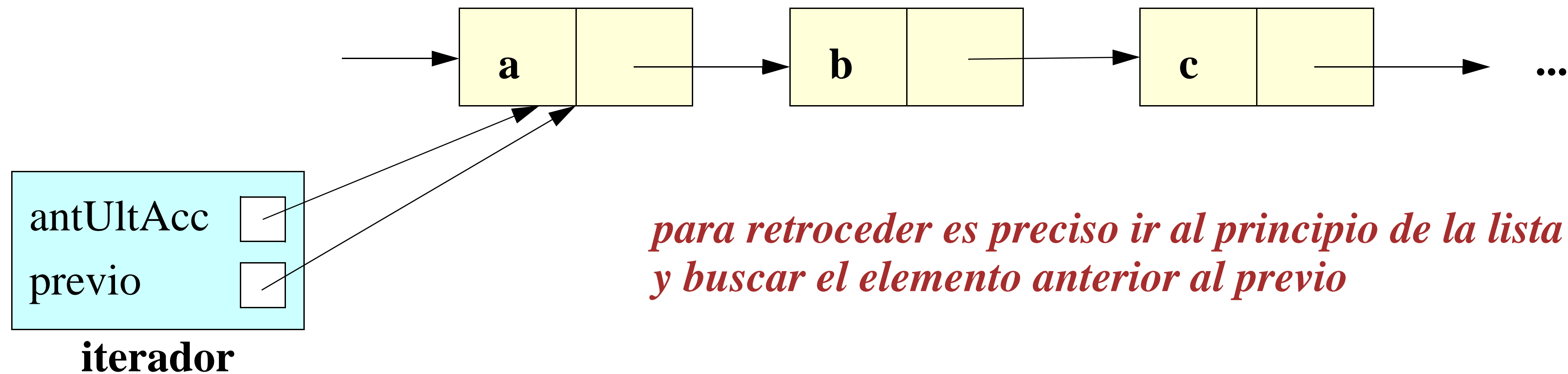




Avance del iterador: previo

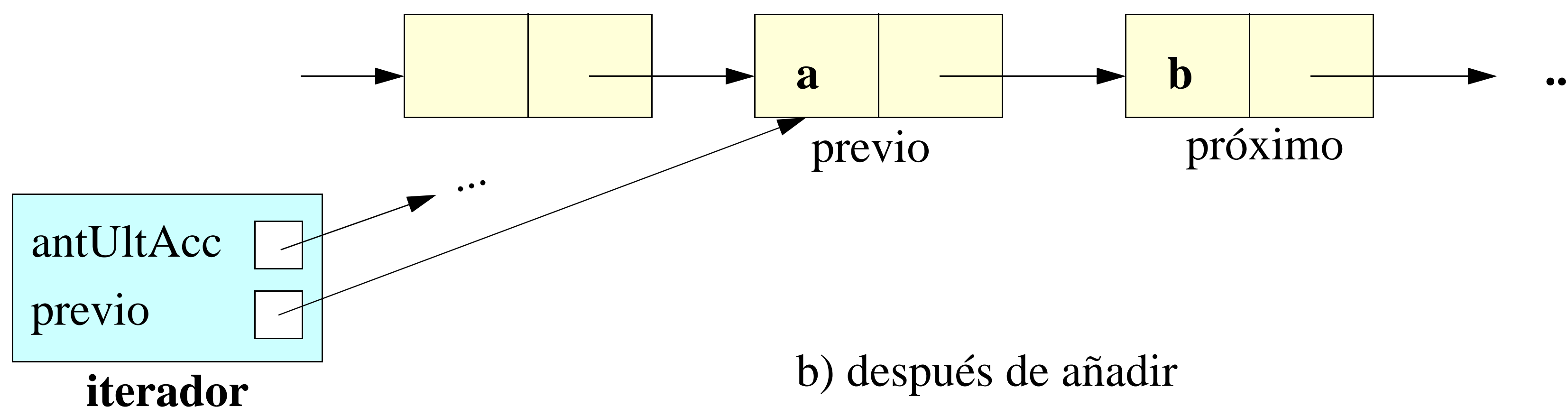
UNIVERSIDAD DE CANTABRIA





para retroceder es preciso ir al principio de la lista
y buscar el elemento anterior al previo

Diagrama de añade



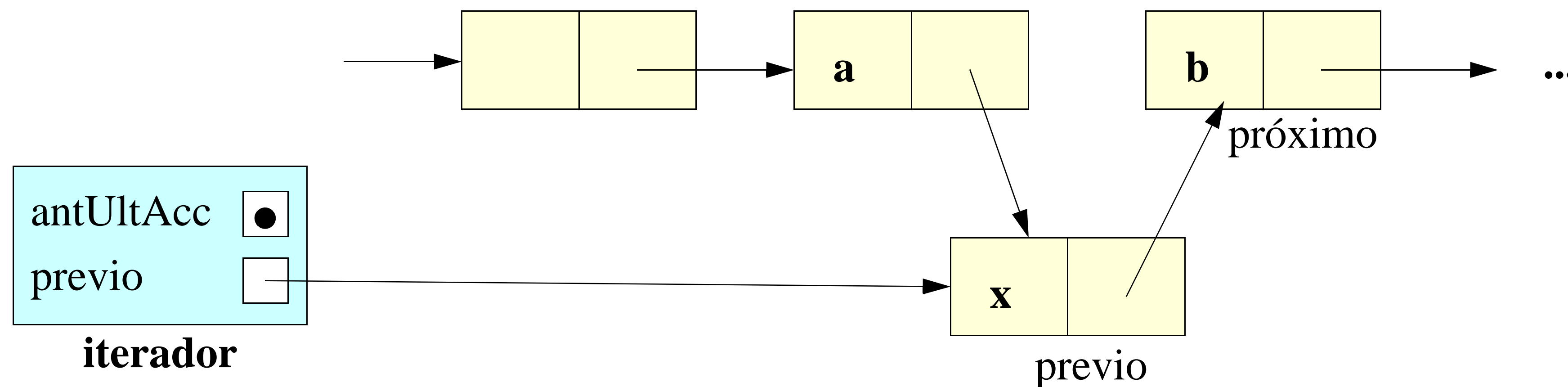
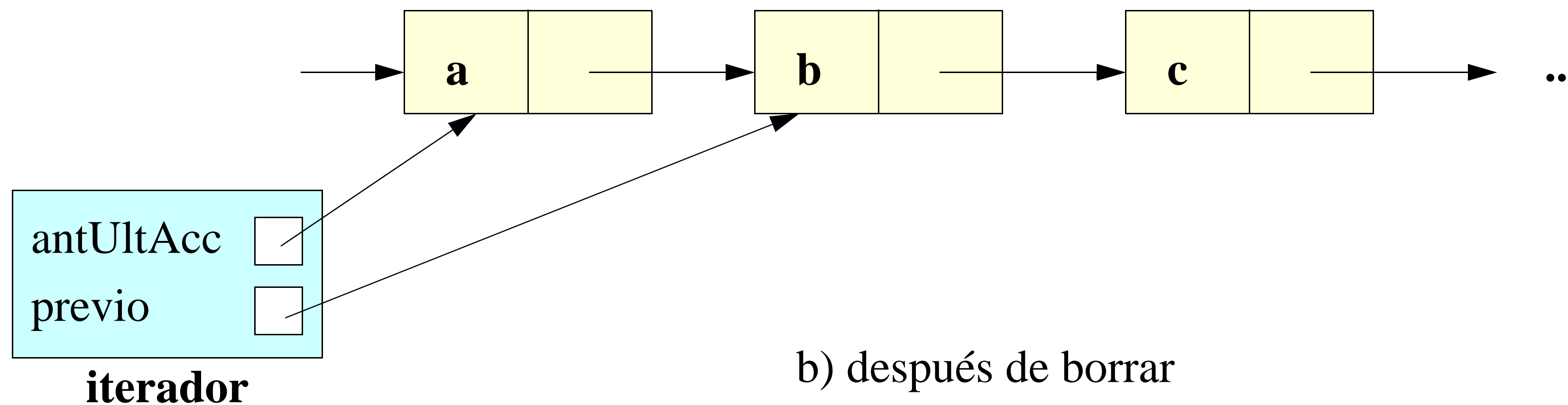


Diagrama de borra (después de próximo)

UNIVERSIDAD DE CANTABRIA



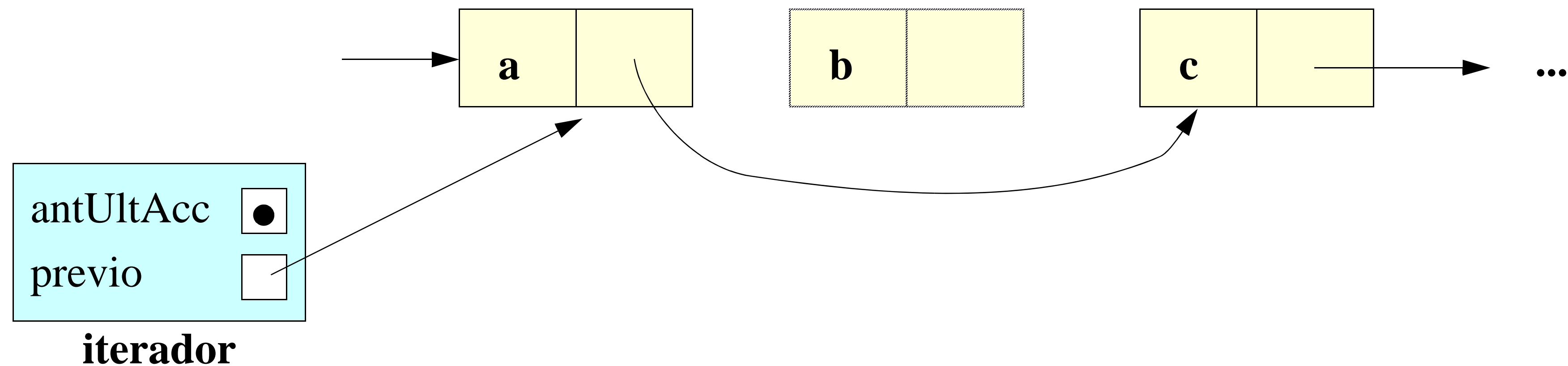
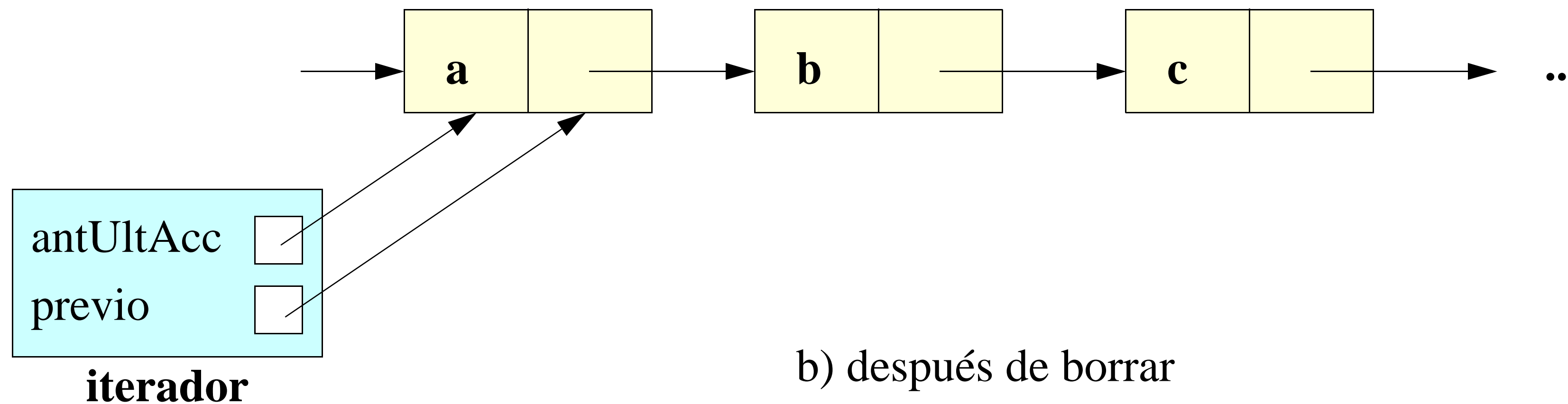
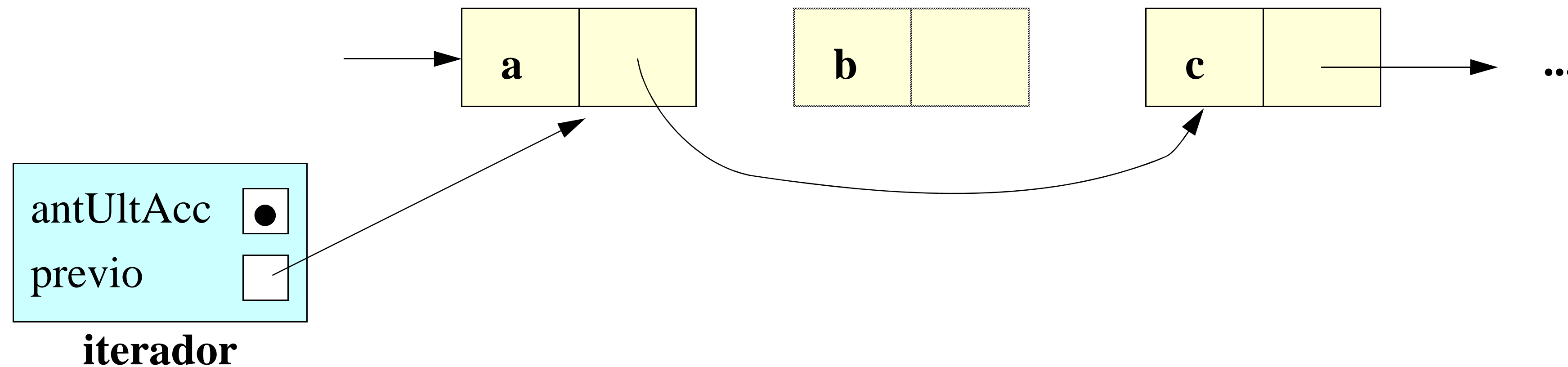


Diagrama de borra (después de previo)

UNIVERSIDAD DE CANTABRIA





Comparación de las implementaciones

UNIVERSIDAD DE CANTABRIA

	array simple	lista enlazada con punteros
Tamaño máximo lista	fijo	dinámico
añade	O(n)	O(1)
borra	O(n)	O(1)
busca	O(n)	O(n)
obten elemento i	O(1)	O(n)
cambia elemento i	O(1)	O(n)
obtener iterador al principio	O(1)	O(1)
obtener iterador en posición i	O(1)	O(n)
Tiempo de Siguiente	O(1)	O(1)
Tiempo de Anterior	O(1)	O(n)

Implementación Java de la lista enlazada simple

UNIVERSIDAD DE CANTABRIA

```java
import java.util.*;

/**
 * Clase con una lista implementada mediante
 * una lista enlazada simple
 */
public class ListaEnlazadaSimple<E>
    extends AbstractSequentialList<E>
{
   // atributos privados
   private Celda<E> principio;
   private int num;
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
// clase privada que define la celda
private static class Celda<E> {
    E contenido;
    Celda<E> siguiente;

    Celda(E cont) {
        contenido=cont;
    }
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Constructor que crea la lista vacía
 */
public ListaEnlazadaSimple()
{
    // crea la cabecera
    principio=new Celda<E>(null);
    num=0;
}
```

Implementación Java de la lista enlazada simple

```
/**
 * Constructor que crea la lista vacía con los
 * elementos de la colección c
 */
public ListaEnlazadaSimple(Collection<E> c)
{
   this();
   Celda<E> actual=principio;
   for (E e:c) {
      actual.siguiente=new Celda<E>(e);
      actual=actual.siguiente;
      num++;
   }
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Retorna el tamano de la lista
 */
public int size() {
    return num;
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Clase iteradora de lista
 */
public static class IteradorListaEnlazada<E>
    implements ListIterator<E>
{
    // atributos del iterador
    private Celda<E> previo;
    private Celda<E> antUltAcc;
    private ListaEnlazadaSimple<E> lista;
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/*
 * Constructor del iterador; no es publico
 */
IteradorListaEnlazada
    (ListaEnlazadaSimple<E> lista)
{
    this.lista=lista;
    previo=lista.principio;
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/*
 * Constructor del iterador; no es publico
 */
IteradorListaEnlazada
     (ListaEnlazadaSimple<E> lista, int i)
{
    this(lista);
    for (int j=0; j<i; j++) {
        next();
    }
    antUltAcc==null;
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Indica si hay elemento siguiente
 */
public  boolean hasNext() {
    return previo.siguiente!=null;
}

/**
 * Indica si hay elemento previo
 */
public boolean hasPrevious() {
   return previo!=lista.principio;
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtiene el siguiente y avanza el iterador
 */
public E next() {
    if (hasNext()) {
        antUltAcc=previo;
        previo=previo.siguiente;
        return previo.contenido;
    } else {
        throw new NoSuchElementException();
    }
}
/**
 * Obtiene elem. previo y retrocede el iterador
 */
```

Implementación Java de la lista enlazada simple (cont.)

```
public E previous() {
   if (hasPrevious()) {
      E cont=previo.contenido;
      // buscar el elemento anterior al previo
      Celda<E> anterior=lista.principio;
      while (anterior.siguiente!=previo) {
          anterior=anterior.siguiente;
      }
      previo=anterior;
      antUltAcc=previo;
      return cont;
   } else {
       throw new NoSuchElementException();
   }
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtiene el indice del elemento proximo
 */
public int nextIndex() {
   return previousIndex()+1;
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtiene el indice del elemento previo
 */
public int previousIndex() {
   int ind=-1;
   Celda<E> actual=lista.principio;
   while (actual!=previo) {
      actual=actual.siguiente;
      ind++;
   }
   return ind;
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Borra el ultimo elemento accedido */
public void remove() {
   if (antUltAcc!=null) {
       Celda<E> borrar=antUltAcc.siguiente;
       antUltAcc.siguiente=borrar.siguiente;
       if (previo==borrar) {
           previo=antUltAcc;
       }
       antUltAcc=null;
       lista.num--;
   } else {
       throw new IllegalStateException();
   }
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Cambia el ultimo elemento accedido
 */
public void set(E e) {
    if (antUltAcc!=null) {
        antUltAcc.siguiente.contenido=e;
    } else {
        throw new IllegalStateException();
    }
}
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
    /**
     * Anade un elem. entre el previo y el proximo
     */
    public void add(E e) {
       Celda<E> nueva=new Celda<E>(e);
       nueva.siguiente=previo.siguiente;
       previo.siguiente=nueva;
       previo=nueva;
       antUltAcc=null;
       lista.num++;
    }
 } // fin de la clase iteradora
```

Implementación Java de la lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Retorna el iterador
 */
public ListIterator<E> listIterator() {
    return new IteradorListaEnlazada<E>(this);
}
```

Implementación Java de la lista enlazada simple (cont.)

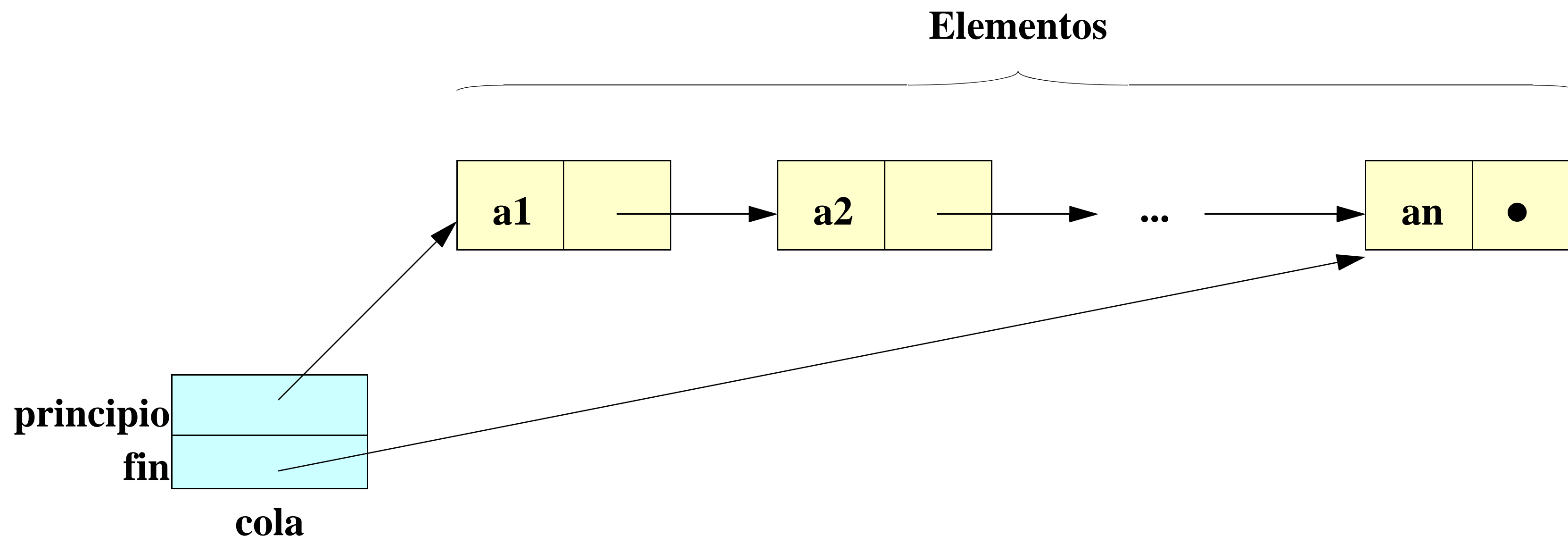
UNIVERSIDAD DE CANTABRIA

```
  /**
   * Retorna el iterador colocado
   * en la posicion i
   */
  public ListIterator<E> listIterator(int i) {
     if (i>=0 && i<=size()) {
        return new IteradorListaEnlazada<E>(this,i);
     } else {
        throw new IndexOutOfBoundsException();
     }
  }
}
```

5.3.2. Implementación de colas mediante una lista enlazada

Los elementos se guardan en una lista enlazada

La cola tiene un puntero al principio y otro al final; los elementos se insertan por el final y se extraen por el principio

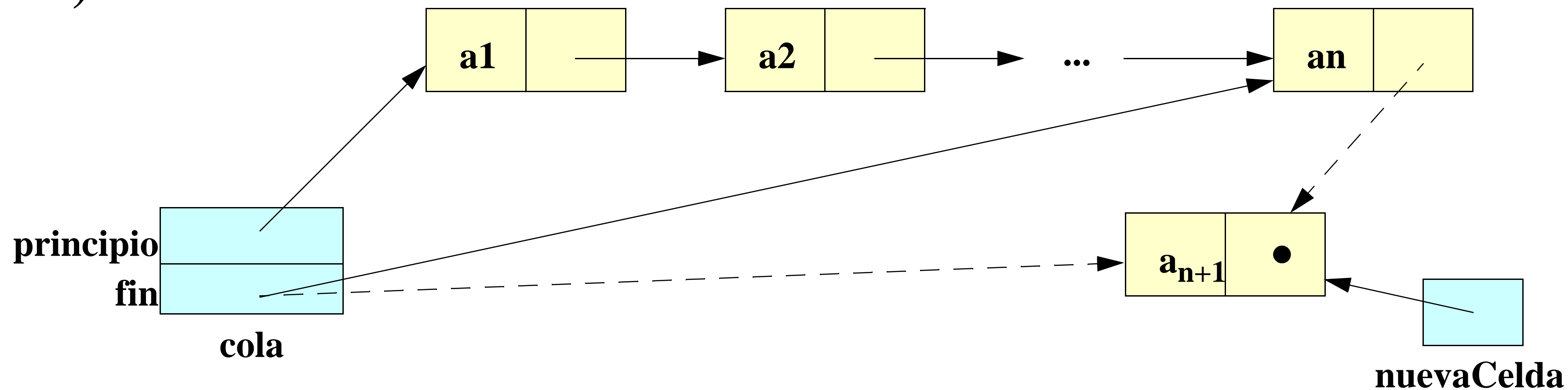


Las pilas con listas enlazadas son similares (cima=principio)

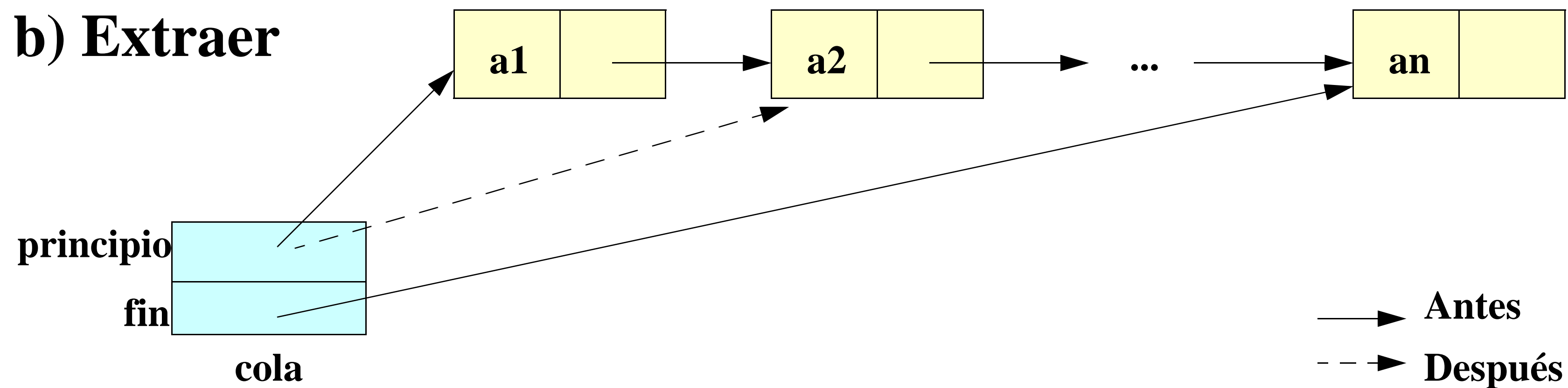
Operaciones de colas con punteros: insertar

UNIVERSIDAD DE CANTABRIA

a) Insertar



Operaciones de colas con punteros: extraer



Implementación en Java de las colas con lista enlazada simple

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;

/**
 * Clase que representa una cola implementada mediante
 * una lista enlazada
 */
public class ColaEnlazada<E> extends AbstractQueue<E>
{

  // atributos de la cola
  private Celda<E> principio,fin;
  private int num;

  // clase privada que define la celda
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
private static class Celda<E> {
    E contenido;
    Celda<E> siguiente;

    Celda(E cont) {
        contenido=cont;
    }
}

/**
 * Constructor que crea una cola vacía
 */
public ColaEnlazada() {
     num=0;
}
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Constructor que crea una cola con los elementos
 * de la colección que se le pasa
 */
public ColaEnlazada(Collection<E> c) {
    // crea la cola vacía
    this();
    // anade todos los elementos a la cola
    for (E e:c) {
        offer(e);
    }
}
/**
 * Anade elemento a la cola; retorna false si no cabe
 * y true en otro caso; no admite elementos nulos */
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
public boolean offer(E e) {
    if (e==null) {
        return false;
    } else {
        Celda<E> nuevaCelda=new Celda<E> (e);
        if (principio==null) {
            principio=nuevaCelda;
        } else {
            fin.siguiente=nuevaCelda;
        }
        fin=nuevaCelda;
        num++;
        return true;
    }
}
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/** Elimina y retorna un elemento de la cola
 * retorna null si no hay elementos */
public E poll() {
    if (isEmpty()) {
        return null;
    } else {
        E borrado=principio.contenido;
        principio=principio.siguiente;
        if (principio==null) {
            fin=null; // la cola esta vacía
        }
        num--;
        return borrado;
    }
}
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Retorna el 1º elemento de la cola sin eliminarlo
 * retorna null si no hay elementos
 */
public E peek() {
    if (isEmpty()) {
        return null;
    } else {
        return principio.contenido;
    }
}
```


© Michael González Harbour
17/nov/09

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Retorna el numero de elementos
 */
public int size() {
    return num;
}

/**
 *  Indica si la cola esta vacía
 */
public boolean isEmpty() {
   return principio==null;
}
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```java
/**
 * Clase iterador de la cola enlazada
 */
public static class ColaEnlazadaIterator<E>
    implements Iterator<E>
{
    private Celda<E> siguiente;
    private ColaEnlazada<E> cola;

    ColaEnlazadaIterator(ColaEnlazada<E> cola) {
        this.cola=cola;
        siguiente=cola.principio;
    }
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtener el proximo elem. y avanzar el iterador
 */
public E next() {
    if (hasNext()) {
        E sig=siguiente.contenido;
        siguiente=siguiente.siguiente;
        return sig;
    } else {
        throw new NoSuchElementException();
    }
}
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Indicar si hay elemento siguiente
 */
public boolean hasNext() {
   return siguiente!=null;
}

/**
 * Borra no se implementa, por poco eficiente
 */
 public void remove() {
  throw new  UnsupportedOperationException();
 }
}
```

Implementación en Java de las colas con lista enlazada simple (cont.)

UNIVERSIDAD DE CANTABRIA

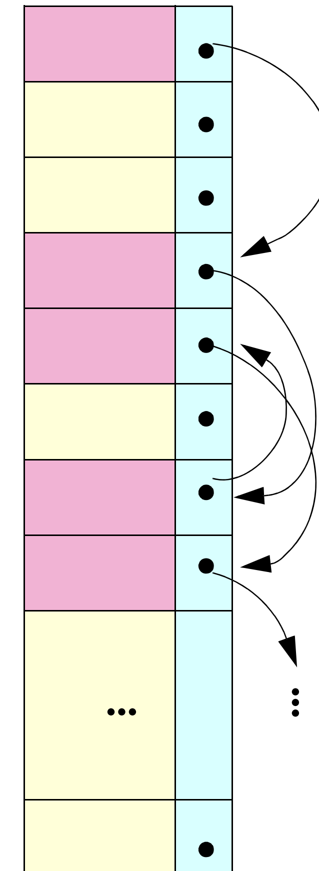
```
   /**
    * Retorna el iterador de la cola enlazada
    */
   public Iterator<E> iterator() {
      return new ColaEnlazadaIterator<E>(this);
   }

}
```

5.4. Implementación con listas enlazadas mediante cursores

UNIVERSIDAD DE CANTABRIA

Cualquier estructura de datos realizada mediante punteros se puede implementar también mediante cursores

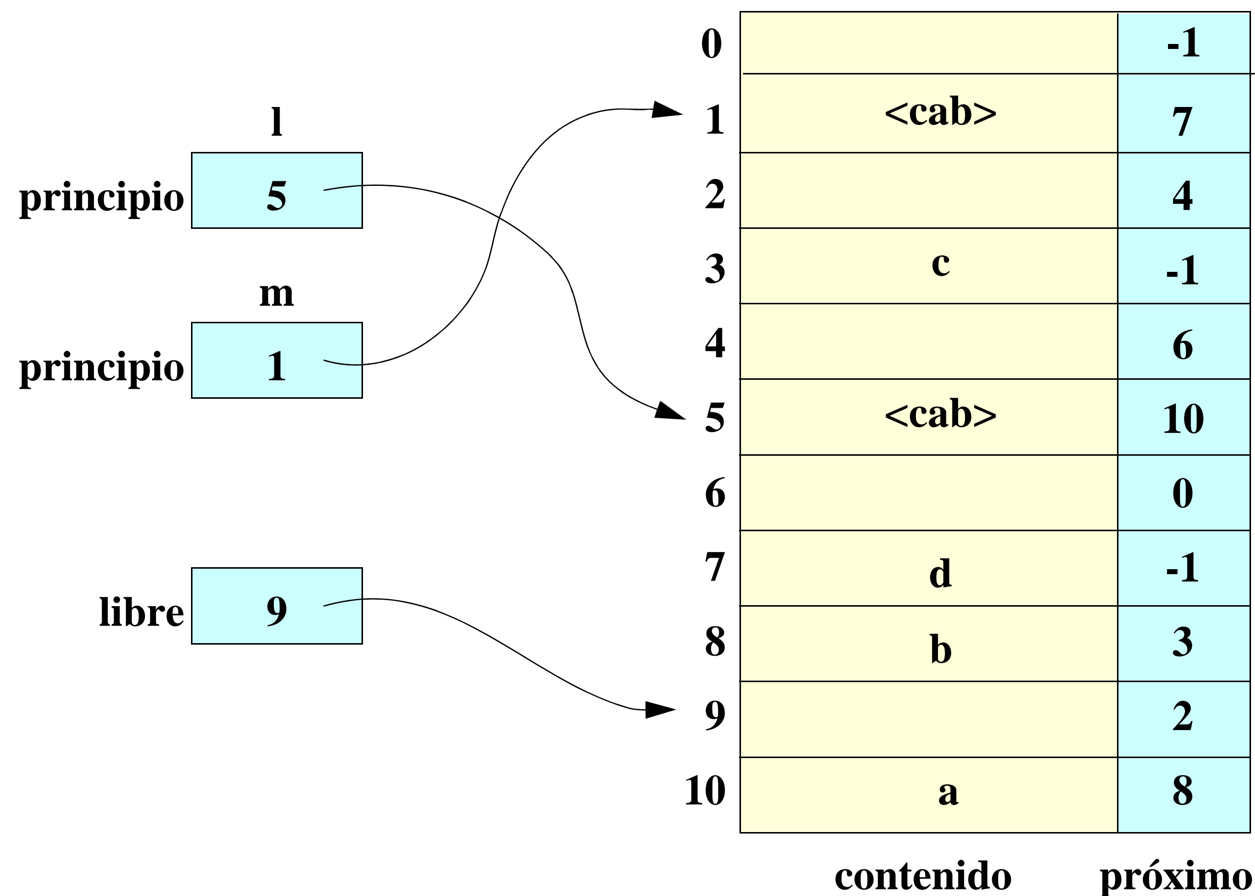
- los elementos se almacenan en un array
- los punteros se sustituyen por cursores
- pueden coexistir varias estructuras de datos si los elementos son de la misma jerarquía de clases
- los elementos vacíos se organizan en forma de lista encadenada



Se desaprovecha memoria si la estructura no está llena

© Michael González Harbour
17/nov/09

Almacenamiento de varias listas

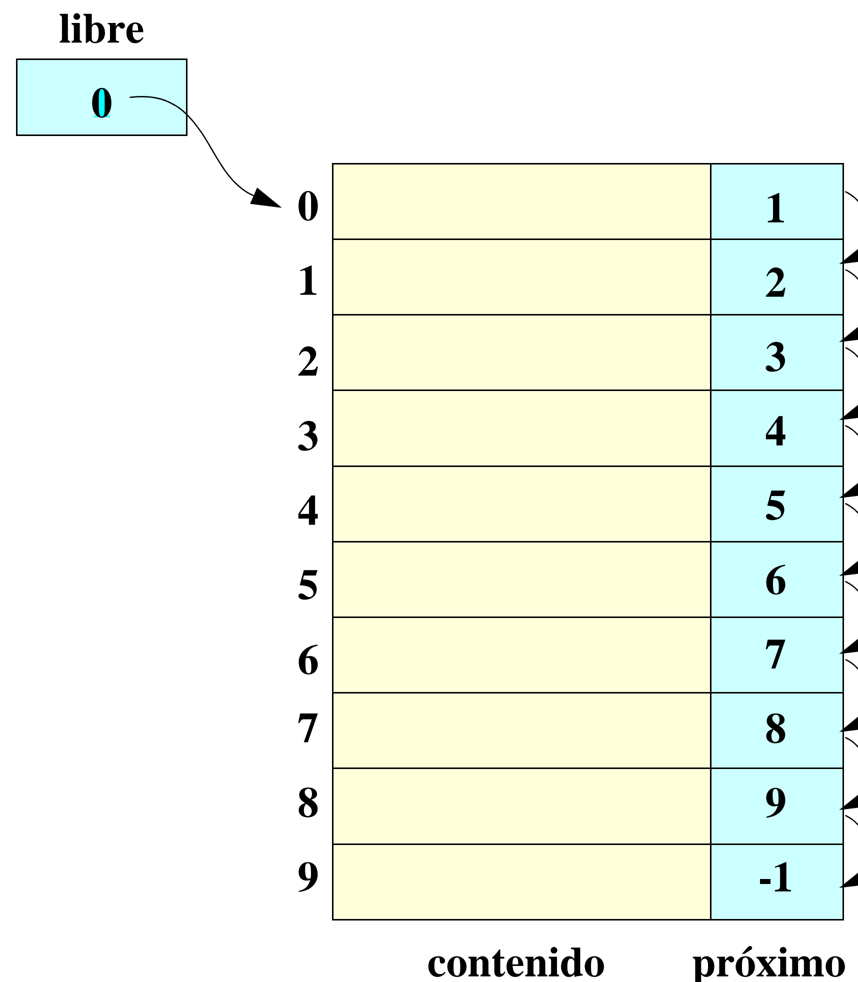


Inicialización de las listas

UNIVERSIDAD DE CANTABRIA

Antes de poder usar las listas es preciso poner todas las celdas en la lista de celdas libres

Para inicializar una lista se reserva una celda cabecera vacía



Inicialización

UNIVERSIDAD DE CANTABRIA

```
método inicializa
   para i desde 0 hasta maxElem-2, paso -1
   hacer
      proximo[i]:=i+1;
   fpara;
   libre:=0;
   proximo[maxElem-1]:=-1; // final de lista
fmétodo
```

Si las tablas son estáticas, hay que asegurarse de que se inicializan antes de que otros objetos puedan usarlas

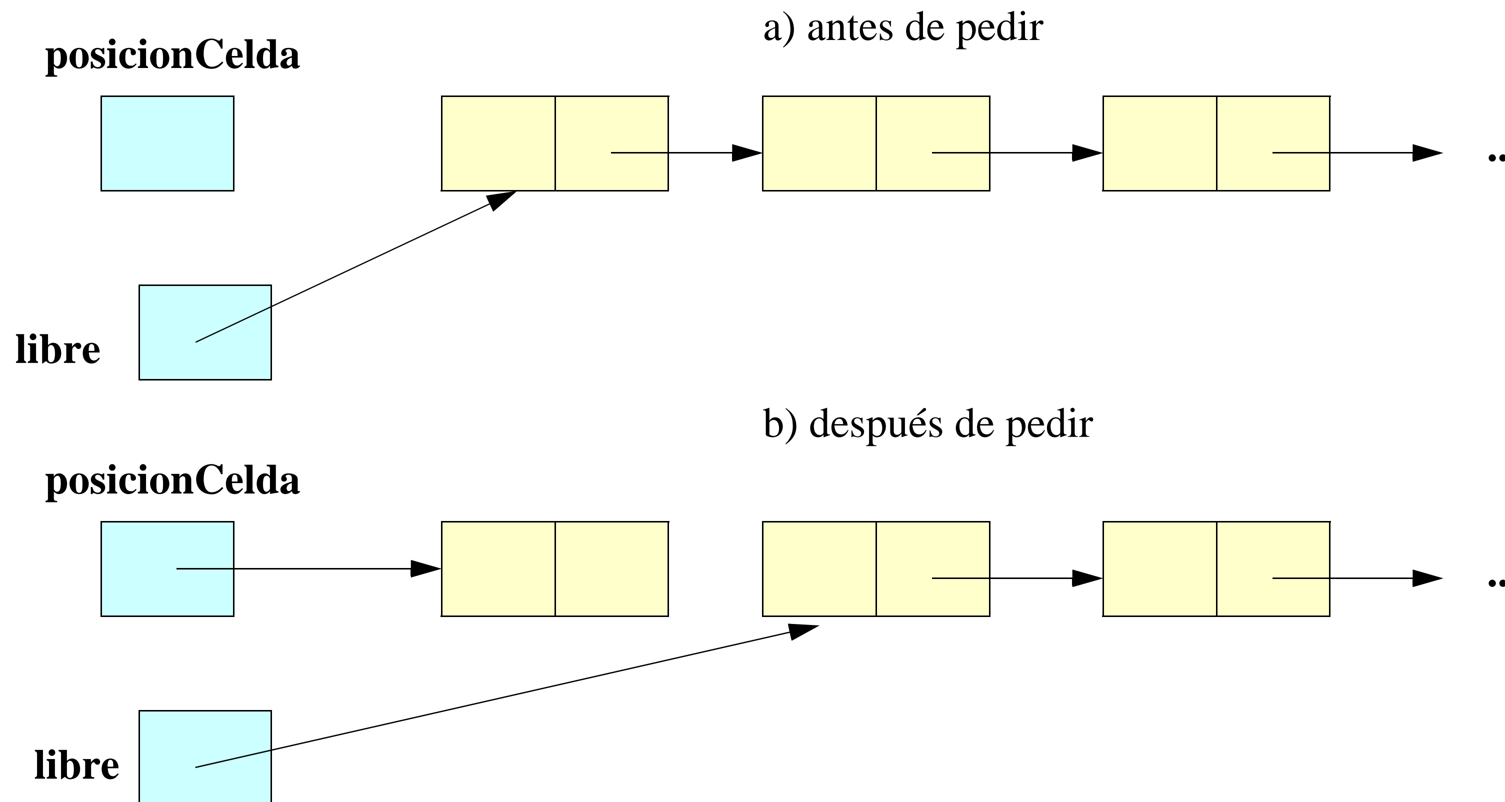
- En java se puede usar un inicializador estático

```
static {
    inicializa();
}
```

Operaciones básicas con las celdas

UNIVERSIDAD DE CANTABRIA

Pedir una celda nueva:



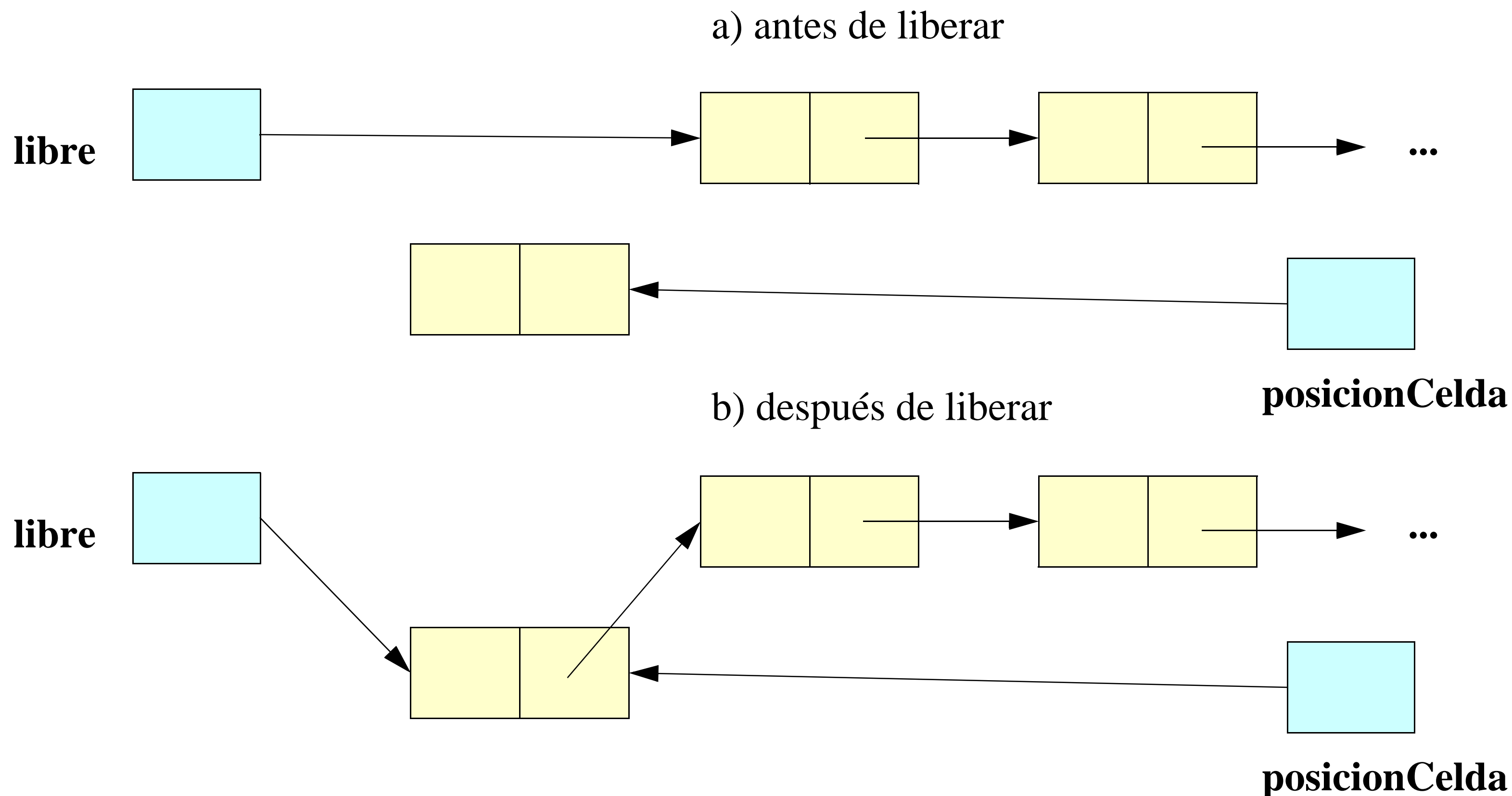
Pedir Celda

UNIVERSIDAD DE CANTABRIA

```
método pedirCelda(Elemento e) retorna entero
begin
   var entero posicionCelda; fvar;
   si libre==-1 then
      lanza NoCabe;
   si no
      posicionCelda:=libre;
      libre:=proximo[libre];
      proximo[posicionCelda]:=-1;
      contenido[posicionCelda]:=e;
   fsi;
   retorna posicionCelda;
fmétodo;
```

Operaciones básicas con las celdas

Liberar una celda:



Liberar Celda

UNIVERSIDAD DE CANTABRIA

```
método liberarCelda (entero posicionCelda)
   proximo[posicionCelda]:=libre;
   libre:=posicionCelda;
fmétodo
```

Crear una lista vacía

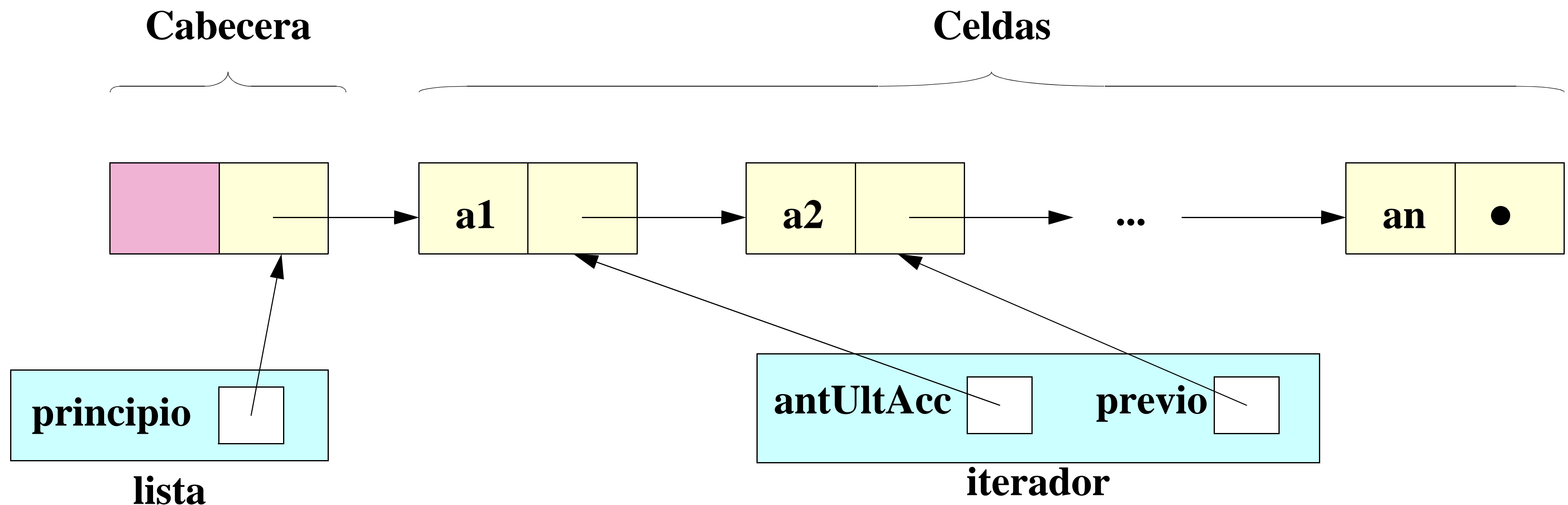

```
método Constructor()
   principio=pedirCelda(null);
fmétodo;
```

Iterador de la lista

UNIVERSIDAD DE CANTABRIA

El iterador de la lista es como en la lista enlazada con punteros:

- `previo`: un cursor al elemento previo
- `antUltAcc`: un cursor al elemento anterior al último elemento accedido a través del iterador (con `proximo` o `previo`)



Implementación Java de la lista enlazada con cursores

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;
/**
 * Clase que representa una lista implementada con
 * una lista enlazada simple
 */
public class ListaCursores<E>
   extends AbstractSequentialList<E>
{
  // Espacio para colocar las listas
  // se define con atributos estáticos
  public static int maxElem=1000;
  private static int[] proximo=new int[maxElem];
  private static Object[] contenido=
     new Object[maxElem];
  private static int libre;
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
private static void inicializa() {
    for(int i=maxElem-2; i>=0; i--) {
        proximo[i]=i+1;
    }
    libre=0;
    proximo[maxElem-1]=-1; // final de lista
}

static {
    inicializa();
}
```

Implementación Java de la lista enlazada con cursores (cont.)

```
private static int pedirCelda(Object elem) {
    int posicionCelda;
    if (libre==-1) {
        throw new IllegalStateException();
    } else {
        posicionCelda=libre;
        libre=proximo[libre];
        proximo[posicionCelda]=-1;
        contenido[posicionCelda]=elem;
    }
    return posicionCelda;
}
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
private static void liberarCelda(int posicionCelda)
{
    proximo[posicionCelda]=libre;
    libre=posicionCelda;
}

// atributos privados
private int principio;
private int num;
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Constructor que crea la lista vacía
 */
public ListaCursores()
{
    // crea la cabecera
    principio=pedirCelda(null);
    num=0;
}

/**
 * Constructor que crea la lista vacía con los
 * elementos de la colección c
 */
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
public ListaCursores(Collection<E> c) {
    this();
    int actual=principio;
    for (E e:c) {
      proximo[actual]=pedirCelda(e);
      actual=proximo[actual];
      num++;
    }
}
/**
 * Retorna el tamano de la lista
 */
public int size() {
    return num;
}
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```java
/**
 * Clase iteradora de lista
 */
public static class IteradorListaCursores<E>
    implements ListIterator<E>
{
    // atributos del iterador
    private int previo;
    private int antUltAcc;
    private ListaCursores<E> lista;

    /*
     * Constructor del iterador; no es publico
     */
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
IteradorListaCursores(ListaCursores<E> lista) {
    this.lista=lista;
    previo=lista.principio;
    antUltAcc=-1;
}
/** Constructor del iterador; no es publico */
IteradorListaCursores(ListaCursores<E> lista,
    int i)
{
     this(lista);
     for (int j=0; j<i; j++) {
        next();
     }
     antUltAcc==-1;
}
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Indica si hay elemento siguiente
 */
public  boolean hasNext() {
        return proximo[previo]!=-1;
}

/**
 * Indica si hay elemento previo
 */
public boolean hasPrevious() {
      return previo!=lista.principio;
}
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtiene el siguiente y avanza el iterador
 */
public E next() {
   if (hasNext()) {
      antUltAcc=previo;
      previo=proximo[previo];
      return (E) contenido[previo];
   } else {
      throw new NoSuchElementException();
   }
}
/**
 * Obtiene el elemento previo y retrocede
 */
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
public E previous() {
   if (hasPrevious()) {
      E cont=(E) contenido[previo];
      // buscar el elemento anterior al previo
      int anterior=lista.principio;
      while (proximo[anterior]!=previo) {
         anterior=proximo[anterior];
      }
      previo=anterior;
      antUltAcc=previo;
      return cont;
   } else {
      throw new NoSuchElementException();
   }
}
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtiene el indice del elemento previo
 */
public int previousIndex() {
   int ind=-1;
   int actual=lista.principio;
   while (actual!=previo) {
      actual=proximo[actual];
      ind++;
   }
   return ind;
}
/**
 * Borra el ultimo elemento accedido
 */
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
public void remove() {
    if (antUltAcc!=-1) {
        int borrar=proximo[antUltAcc];
        proximo[antUltAcc]=proximo[borrar];
        if (previo==borrar) {
            previo=antUltAcc;
        }
        antUltAcc=-1;
        lista.num--;
        liberarCelda(borrar);
    } else {
        throw new IllegalStateException();
    }
}
```

Implementación Java de la lista enlazada con cursores (cont.)

```java
/**
 * Obtiene el indice del elemento proximo
 */
public int nextIndex() {
   return previousIndex()+1;
}
/**
 * Cambia el ultimo elemento accedido */
public void set(E e) {
  if (antUltAcc!=-1) {
     contenido[proximo[antUltAcc]]=e;
  } else {
     throw new IllegalStateException();
  }
}
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
    /**
     * Anade un nuevo e entre el previo y el proximo
     */
    public void add(E e) {
        int nueva=pedirCelda(e);
        proximo[nueva]=proximo[previo];
        proximo[previo]=nueva;
        previo=nueva;
        antUltAcc=-1;
        lista.num++;
    }
}
```

Implementación Java de la lista enlazada con cursores (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Metodo que retorna el iterador
 */
public ListIterator<E> listIterator() {
    return new IteradorListaCursores<E>(this);
}
```

Implementación Java de la lista enlazada con cursores (cont.)

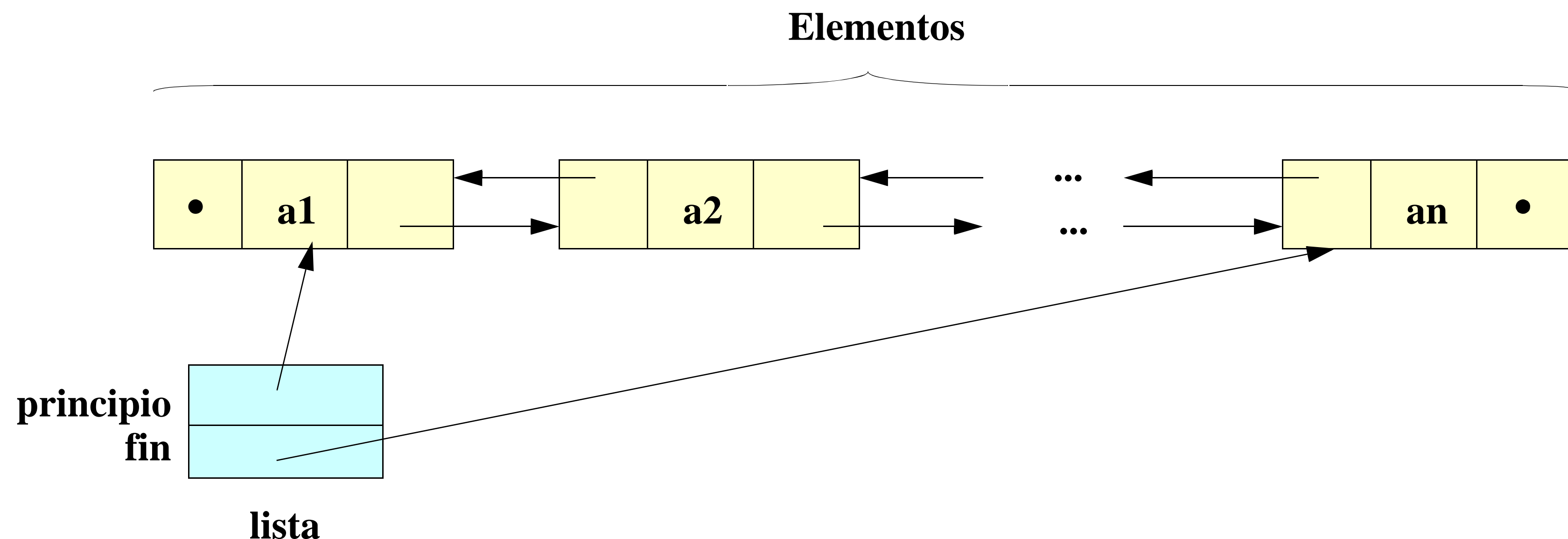
UNIVERSIDAD DE CANTABRIA

```
  /**
   * Metodo que retorna el iterador colocado
   * en la posicion i
   */
  public ListIterator<E> listIterator(int i) {
     if (i>=0 && i<=size()) {
        return new IteradorListaCursores<E>(this,i);
     } else {
        throw new IndexOutOfBoundsException();
     }
  }
}
```

5.5. Listas doblemente enlazadas

En las listas enlazadas simples, las operaciones para acceder a la posición última y hacer retroceder al iterador son costosas (O(*n*))

Para evitar este problema se pueden hacer listas doblemente enlazadas:

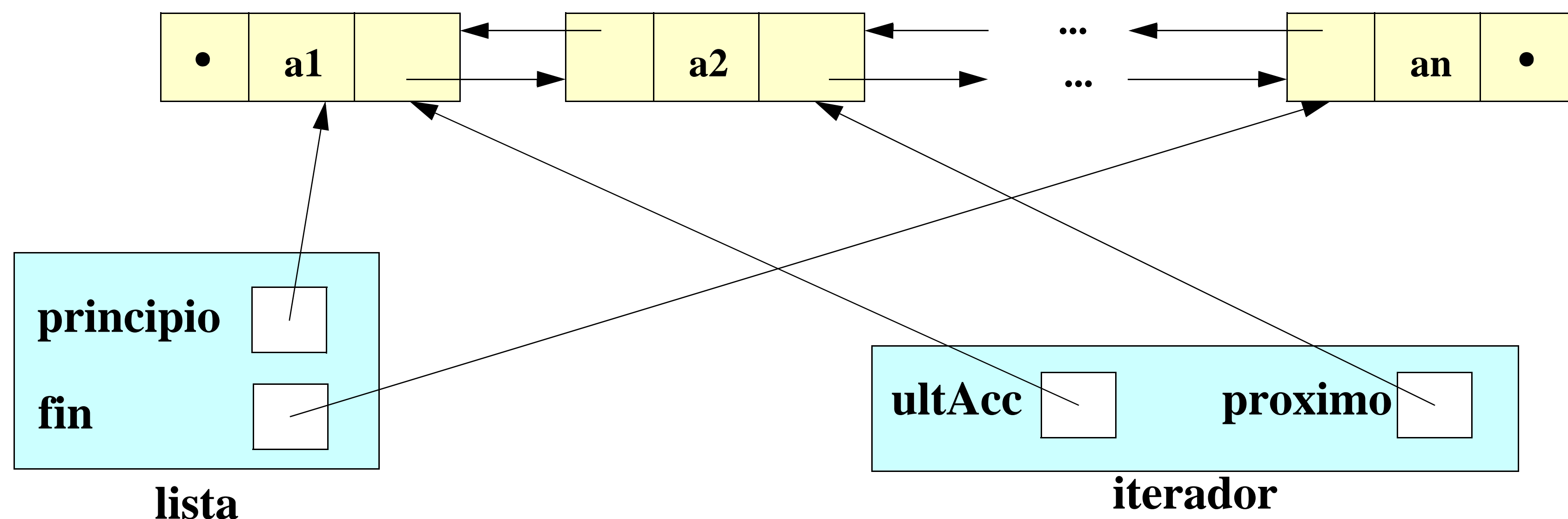


© Michael González Harbour
17/nov/09

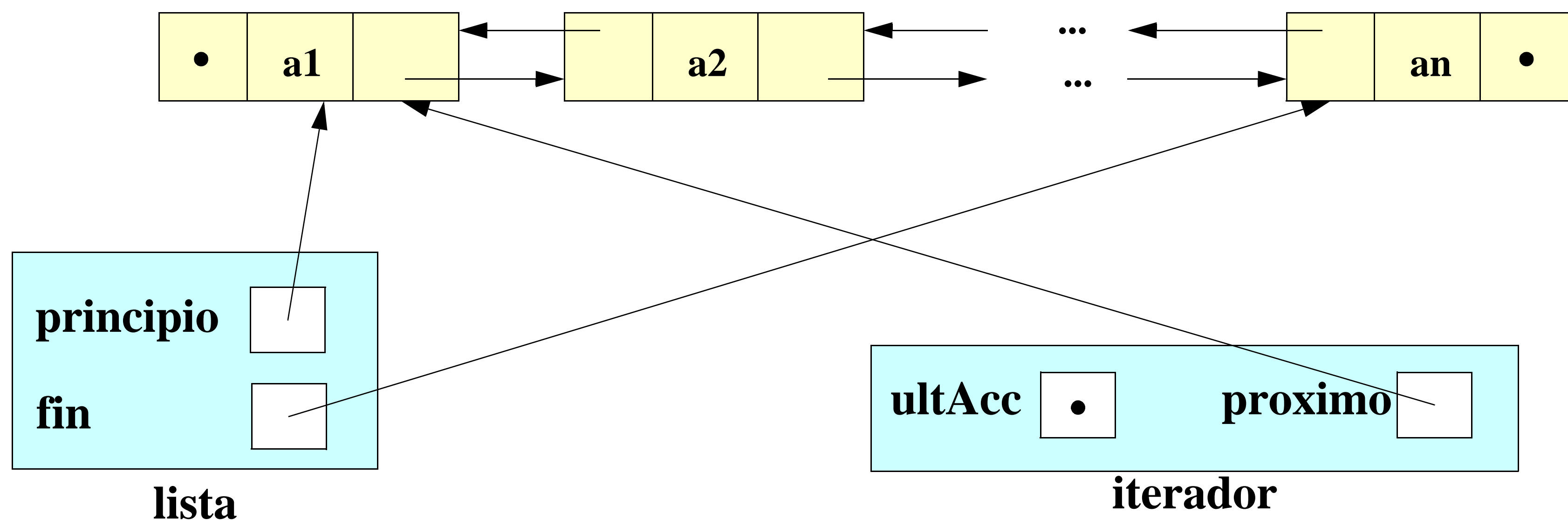
Iterador de la lista

El iterador de la lista contiene

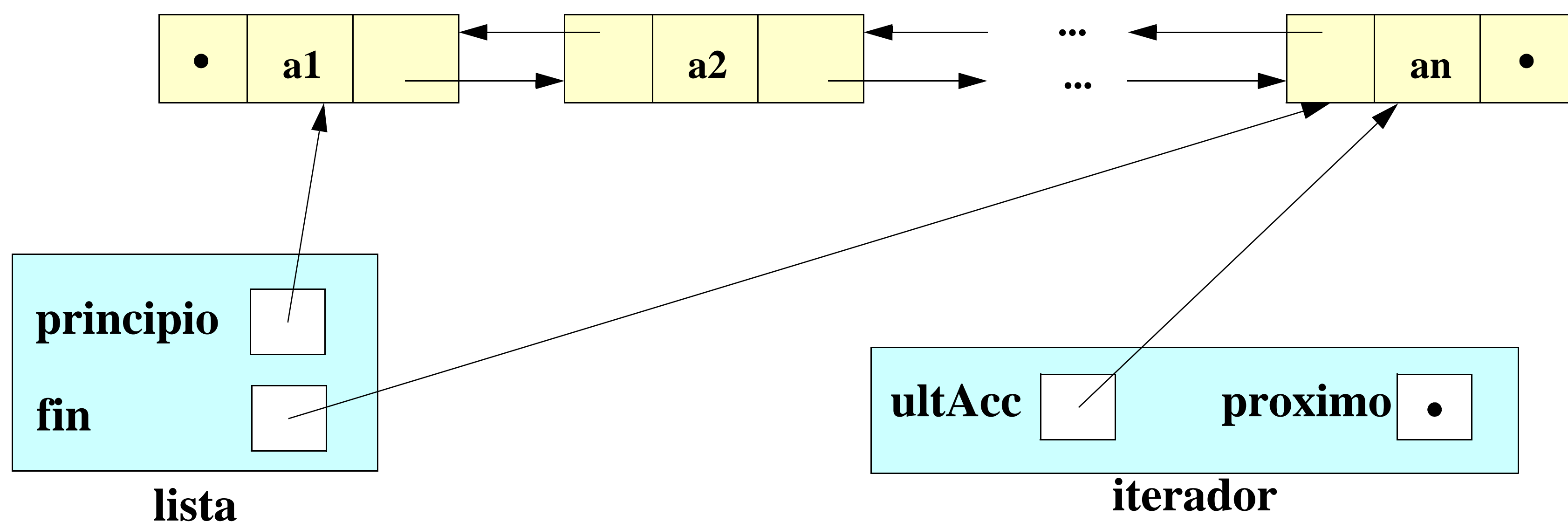
- `proximo`: un puntero al elemento proximo
- `ultAcc`: un puntero al último elemento accedido a través del iterador (con `proximo` o `previo`)



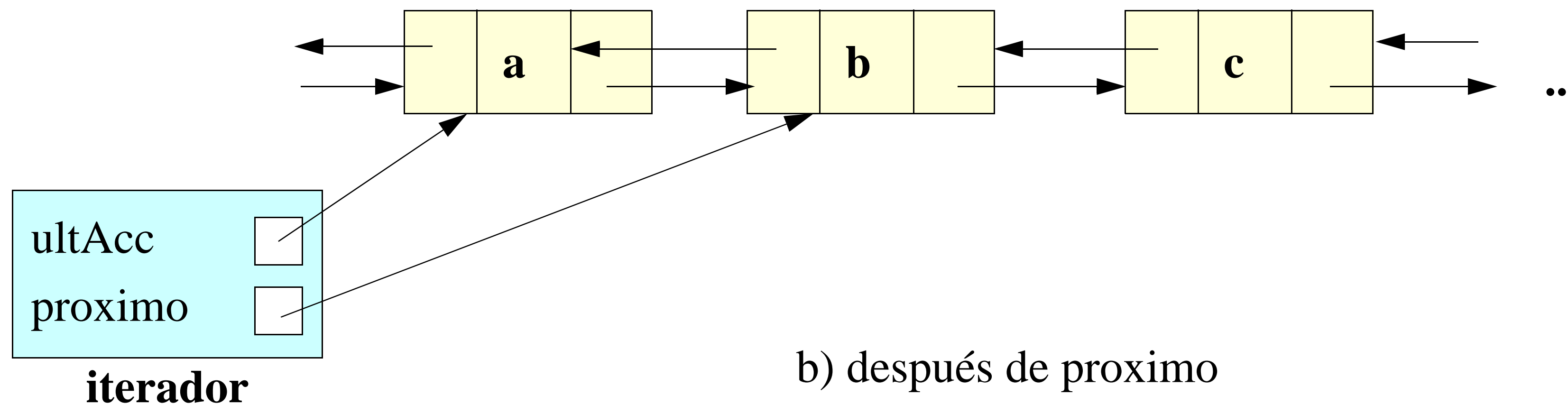
Situación inicial del iterador

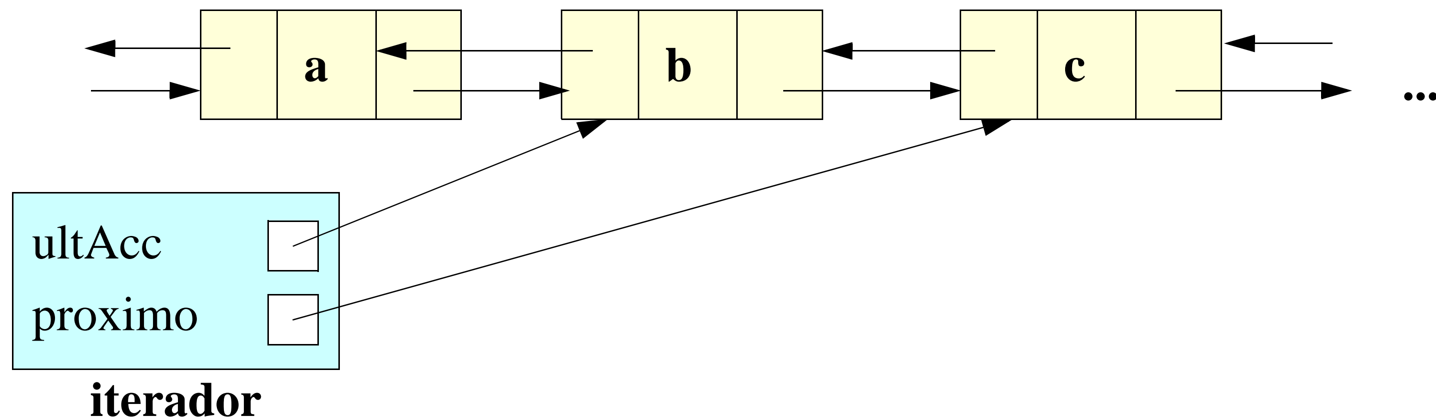


Iterador al final de la lista



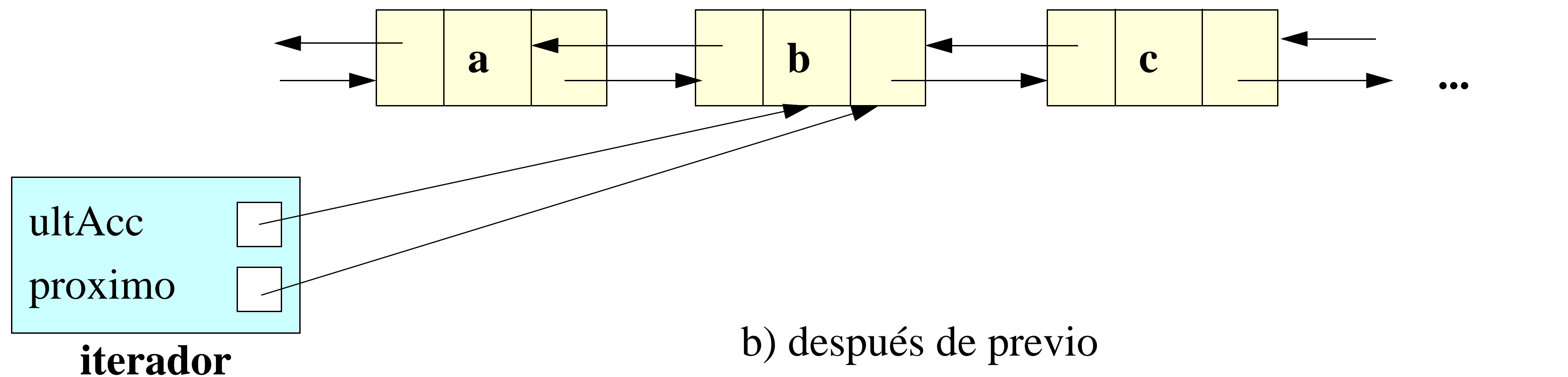
Avance del iterador: próximo





Avance del iterador: previo

UNIVERSIDAD DE CANTABRIA



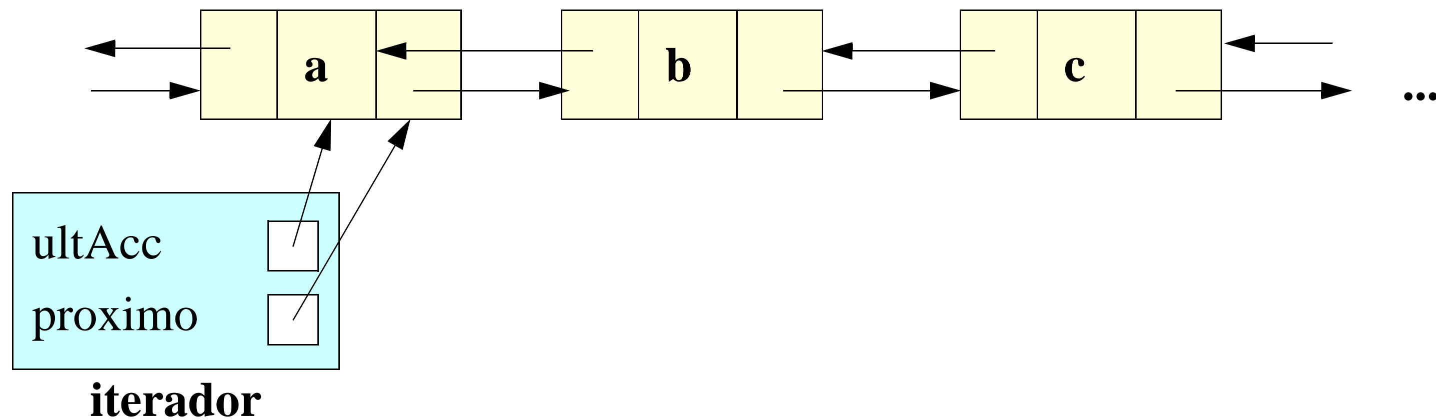
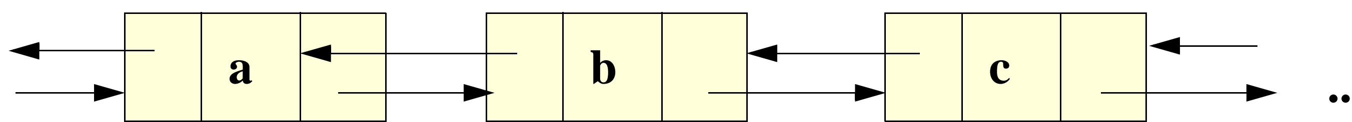
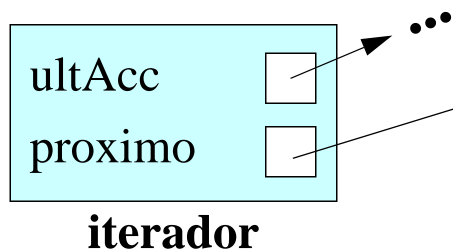


Diagrama de añade

a) situación inicial





b) después de añade

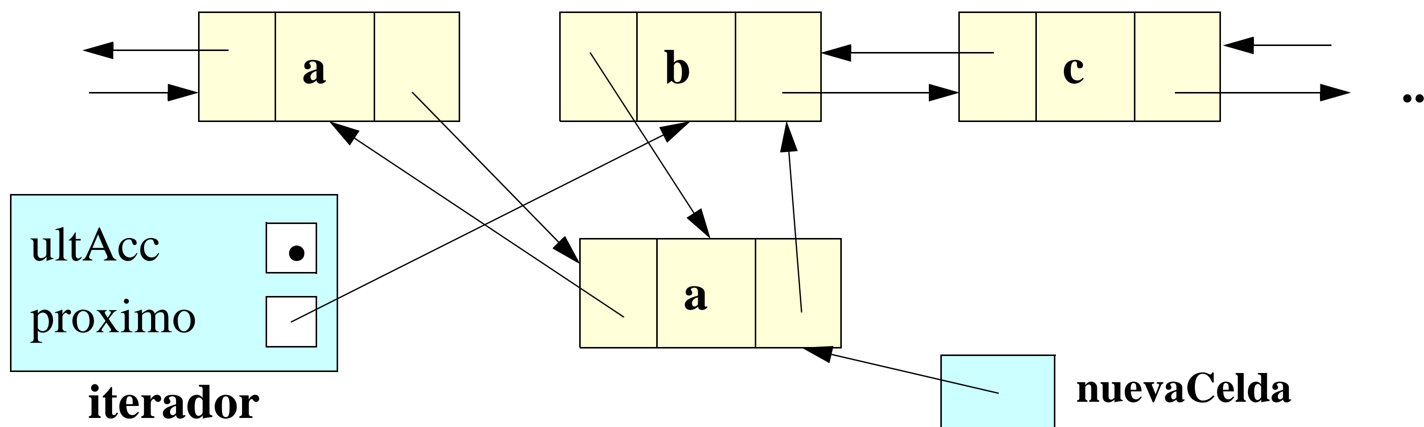
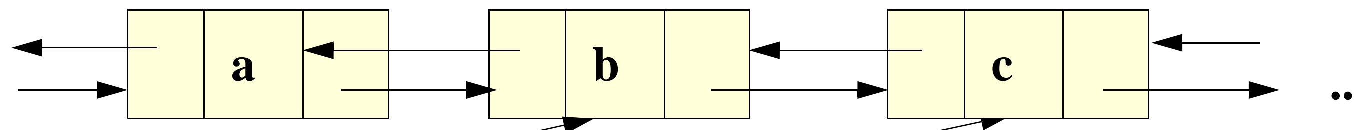
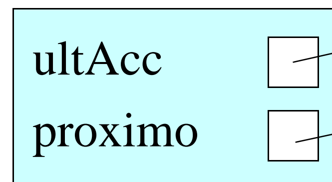


Diagrama de borra (después de próximo)

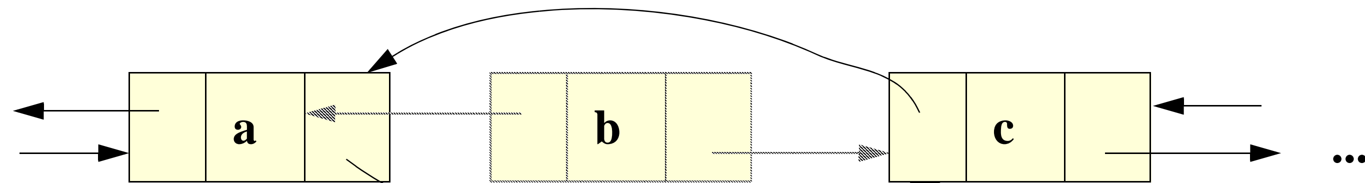
UC UNIVERSIDAD DE CANTABRIA

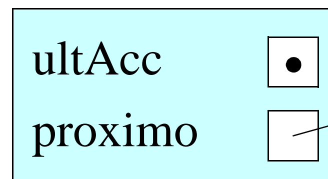
a) situación inicial





b) después de borra

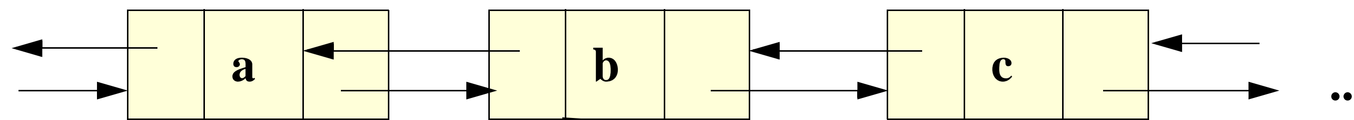


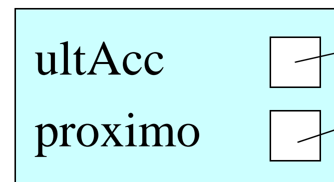


© Michael González Harbour
17/nov/09

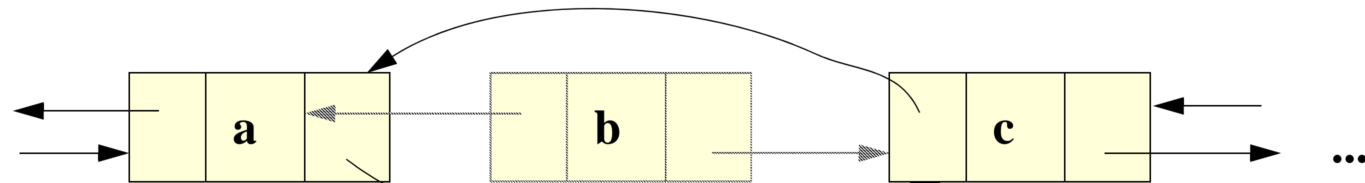
Diagrama de borra (después de previo)

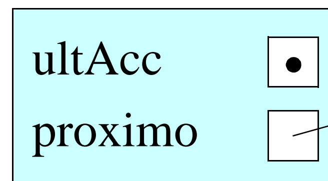
a) situación inicial





b) después de borra





© Michael González Harbour
17/nov/09

Implementación en Java de las listas doblemente enlazadas

UNIVERSIDAD DE CANTABRIA

```
import java.util.*;

/**
 * Clase que representa una lista implementada con
 * una lista doblemente enlazada
 */
public class ListaDoblementeEnlazada<E>
   extends AbstractSequentialList<E>
{
  // atributos privados
  private Celda<E> principio;
  private Celda<E> fin;
  private int num;
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
// clase privada que define la celda
private static class Celda<E> {
    E contenido;
    Celda<E> siguiente;
    Celda<E> anterior;

    Celda(E cont) {
        contenido=cont;
    }
}

/**
 * Constructor que crea la lista vacía
 */
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
public ListaDoblementeEnlazada()
{
   num=0;
}
/**
 * Constructor que crea la lista vacía con los
 * elementos de la colección c
 */
public ListaDoblementeEnlazada(Collection<E> c) {
   this();
   for (E e:c) {
     add(e); // inserta al final
   }
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Retorna el tamano de la lista
 */
public int size() {
    return num;
}

/**
 * Anade al final
 */
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
public boolean add(E e) {
    Celda<E> nuevaCelda=new Celda<E> (e);
    if (isEmpty()) {
        principio=nuevaCelda;
        fin=nuevaCelda;
    } else {
        fin.siguiente=nuevaCelda;
        nuevaCelda.anterior=fin;
        fin=nuevaCelda;
    }
    num++;
    return true;
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

```
/**
 * Clase iteradora de lista
 */
public static class
    IteradorListaDoblementeEnlazada<E>
    implements ListIterator<E>
{
    // atributos del iterador
    private Celda<E> proximo;
    private Celda<E> ultAcc;
    private ListaDoblementeEnlazada<E> lista;
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
/*
 * Constructores del iterador; no son públicos
 */
IteradorListaDoblementeEnlazada
    (ListaDoblementeEnlazada<E> lista)
{
   this.lista=lista;
   proximo=lista.principio;
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
IteradorListaDoblementeEnlazada
    (ListaDoblementeEnlazada<E> lista, int i)
{
    this(lista);
    if (i==lista.num) {
        proximo=null; // nos ponemos al final
    } else {
      // este bucle se podría optimizar empezando
       // por el final si i>num/2
        for (int j=0; j<i; j++) {
            next();
        }
        ultAcc=null;
    }
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

```java
/**
 * Indica si hay elemento siguiente
 */
public  boolean hasNext() {
   return proximo!=null;
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

```java
/**
 * Obtiene el siguiente y avanza el iterador
 */
public E next() {
    if (hasNext()) {
        ultAcc=proximo;
        proximo=proximo.siguiente;
        return ultAcc.contenido;
    } else {
        throw new NoSuchElementException();
    }
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Indica si hay elemento previo
 */
public boolean hasPrevious() {
   return proximo!=lista.principio;
}

/**
 * Obtiene el elemento previo y
 * retrocede el iterador
 */
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
public E previous() {
    if (hasPrevious()) {
        if (proximo==null) {
            // estamos después del final
            proximo=lista.fin;
        } else {
            proximo=proximo.anterior;
        }
        ultAcc=proximo;
        return proximo.contenido;
    } else {
        throw new NoSuchElementException();
    }
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtiene el indice del elemento proximo
 */
public int nextIndex() {
    return previousIndex()+1;
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Obtiene el indice del elemento previo
 */
public int previousIndex() {
   int ind=-1;
   Celda<E> actual=lista.principio;
   while (actual!=proximo) {
     actual=actual.siguiente;
     ind++;
   }
   return ind;
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```java
/**
 * Borra el ultimo elemento accedido
 */
public void remove() {
   if (ultAcc!=null) {
      if (ultAcc.anterior==null) {
         // borrando el primero
         lista.principio=ultAcc.siguiente;
      } else {
         ultAcc.anterior.siguiente=
            ultAcc.siguiente;
      }
      if (ultAcc.siguiente==null) {
         // borrando el ultimo
         lista.fin=ultAcc.anterior;
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
            } else {
               ultAcc.siguiente.anterior=
                  ultAcc.anterior;
            }
            if (proximo==ultAcc) {
               proximo=ultAcc.siguiente;
            }
            ultAcc=null;
            lista.num--;
         } else {
            throw new IllegalStateException();
         }
      }
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Cambia el ultimo elemento accedido
 */
public void set(E e) {
    if (ultAcc!=null) {
        ultAcc.contenido=e;
    } else {
        throw new IllegalStateException();
    }
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Anade un nuevo elemento entre el previo
 * y el proximo
 */
public void add(E e) {
    if (proximo==null) {
        //anade al final
        lista.add(e);
    } else {
        Celda<E> nueva=new Celda<E>(e);
        Celda<E> previo=proximo.anterior;
        nueva.siguiente=proximo;
        proximo.anterior=nueva;
        nueva.anterior=previo;
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
            if (previo==null) {
               // insertar al principio
               lista.principio=nueva;
            } else {
               previo.siguiente=nueva;
            }
            lista.num++;
         }
         ultAcc=null;
      }
   }
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
/**
 * Metodo que retorna el iterador
 */
public ListIterator<E> listIterator() {
   return new
      IteradorListaDoblementeEnlazada<E>(this);
}
```

Implementación en Java de las listas doblemente enlazadas (cont.)

UNIVERSIDAD DE CANTABRIA

```
   /**
    * Metodo que retorna el iterador colocado
    * en la posicion i
    */
   public ListIterator<E> listIterator(int i) {
      if (i>=0 && i<=size()) {
         return new
          IteradorListaDoblementeEnlazada<E>(this,i);
      } else {
         throw new IndexOutOfBoundsException();
      }
   }
}
```

UNIVERSIDAD
DE CANTABRIA

© Michael González Harbour
17/nov/09