# Estructuras de Datos

Tema 1. Programación Imperativa de Computadores

Tema 2. Fundamentos de Complejidad Algorítmica

Tema 3. Técnicas de Implementación

Tema 4. concepto y especificación de Tipos Abstractos de Datos (TADs)

Tema 5. Estructuras de datos lineales

Tema 6. Estructuras de datos jerárquicas



# Bibliografía

Sahni, Sartaj, "Data structures, algorithms, and applications in Java". McGraw Hill, 2000

Weiss, Mark Allen, "Estructuras de datos y algoritmos". Addison-Wesley Iberoamericana, 1995.

Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, "Data structures and algorithms". Addison-Wesley, 1983.

Weiss, Mark Allen, "Estructuras de datos en Java : compatible con Java 2". Addison-Wesley, 2000

Michael T. Goodrich, Roberto Tamassia, "Data structures and algorithms in Java". John Wiley & Sons, 2006.



# Objetivos

- Conocer las principales técnicas de implementación de los principales tipos abstractos de datos
- Conocer las ventajas e inconvenientes de las distintas técnicas de acuerdo a su complejidad espacial y temporal

# Tema 1. Técnicas de Implementación

1. *Introducción*

2. *Estructuras de datos lineales*
   - Implementación mediante arrays, enlazado mediante punteros y cursores

3. *Tablas hash*
   - Funciones de *hashing*. Técnicas de resolución de colisiones: abierto y cerrado. Implementación de tablas *hash*

4. *Estructuras arbóreas*
   - Árboles binarios y de búsqueda. Árboles binarios equilibrados, Árboles AVL y Árboles rojinegros. Implementación de árboles





# 1 Introducción

*Estructura de Datos*:

- Forma sistemática de organizar y almacenar datos
- Su elección estará guiada por conceptos como: eficiencia temporal o espacial, facilidad de uso, escalabilidad, ...

Constituyen una *pieza fundamental de cualquier programa*

Cualquier programa (no trivial) requiere el uso de estructuras de datos para la gestión de los datos que maneja:

- contactos de una agenda
- diccionario con palabras y sus significados
- facturas, clientes y proveedores de una empresa
- características y relaciones entre los participantes en un juego
- ...





# Estructuras de datos más utilizadas

En este tema veremos las estructuras de datos más habituales:

- Estructuras de datos lineales
- Tablas *hash* (mapas)
- Árboles

# Concepto de Tipo Abstracto de Datos

**Tipo Abstracto de Datos (TAD)**

- **Un tipo, es decir un conjunto de individuos**
- **Junto con una serie de operaciones (básicas) aplicables a dicho conjunto de individuos**
- **Que *se define de forma independiente a su representación***

**Ejemplo: TAD "lista"**

- **Tipo: secuencia de un número finito de elementos, en la que la posición de dichos elementos en la secuencia es relevante**
- **Operaciones: añade, elimina, busca, obtiene, ...**

`(En el tema 4 veremos una manera más formal de especificar un TAD)`

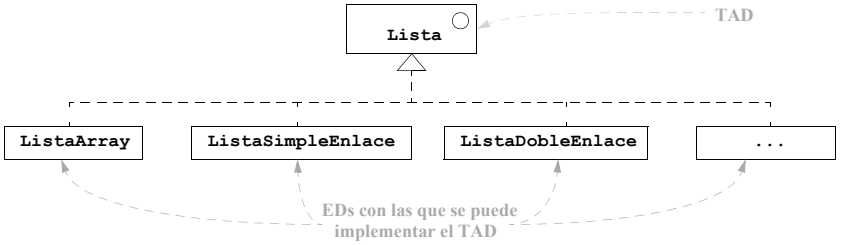




# Estructura de Datos vs. TAD

**Los TADs se implementan utilizando estructuras de datos**

- **algunas EDs serán más eficientes que otras para implementar un TAD determinado**




**En este tema veremos las estructuras de datos básicas,**

- **en el tema 4 la especificación formal de TADs**
- **y en los temas 5 y 6 los TADs más importantes**





# 2 Estructuras de datos lineales

**Estructuras de datos en las que sus elementos se organizan como una secuencia:**

$\{e_1, e_2, \ldots, e_n\}$ $n \geq 0$

- **Todos sus elementos ($e_i$) son del mismo tipo**
- **cuando $n=0$ se trata de una secuencia vacía**

**Operaciones típicas a realizar:**

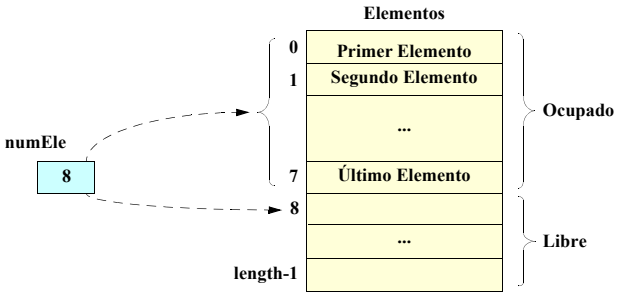
- ***añadir* un elemento (primero, intermedio, último)**
- ***eliminar* un elemento (primero, intermedio, último)**
- ***acceder* a un elemento**
- ***recorrer* todos los elemento**
- **...**

# 2.1 Implementación mediante array

**Array en el que cada casilla almacena un elemento**

**Para conocer la parte ocupada se utiliza un contador**

- **su valor indica el número de elementos de la secuencia y, a la vez, el índice del comienzo de la zona libre**








# Eficiencia de las operaciones

**El acceso posicional es inmediato *O*(1)**

- **en el array tenemos acceso posicional de forma natural**

**La inserción al final es eficiente *O*(1)**

- **meter el nuevo elemento en la casilla `numEle`**
- **incrementar `numEle`**

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

- **hacer hueco**
  - **mover las casillas en el rango `[i,numEle-1]` una casilla hacia adelante, yendo desde el final al principio**
- **meter el nuevo elemento en la casilla i**
- **incrementar el cursor `numEle`**





- **La eliminación del elemento final es eficiente *O*(1)**
  - **almacenar el elemento en la casilla `numEle-1`**
  - **decrementar el cursor `numEle`**
  - **retornar el elemento almacenado**
- **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,numEle-1]` una casilla hacia atrás, yendo del principio hacia el final**
  - **decrementar el cursor `numEle`**
  - **retornar la variable con el elemento almacenado**

***Ver pseudocódigo e implementación en Java en la web de la asignatura***

**Resumen de la eficiencia de las operaciones:**

| Operación | Complejidad temporal |
|---|---|
| añade intermedio | O(n) |
| añade último | O(1) |
| elimina intermedio | O(n) |
| elimina último | O(1) |
| obtiene elemento i-ésimo | O(1) |
| obtiene tamaño lista | O(1) |





# Cambio dinámico de tamaño

**Es posible implementar una *lista ilimitada* mediante arrays**

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

1. **se crea un nuevo array (por ejemplo del doble de tamaño)**
2. **se copian todos los elementos del array viejo en el nuevo**
3. **se libera la memoria del array viejo**

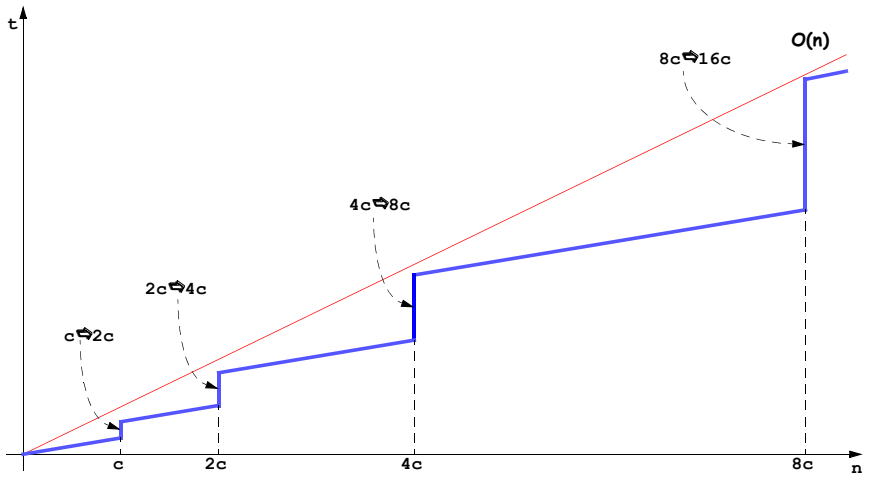
**¡Añadir al final sigue tardando un tiempo *amortizado* constante!**

**A medida que se van añadiendo elementos al final de la lista**

- **algunas adiciones tardan mucho (las de las copias entre arrays)**
- **pero ese tiempo se *amortiza* entre las demás adiciones**
- **de forma que el *tiempo promedio por adición es constante* *O*(1)**
- **o, dicho de otra forma, el tiempo de `n` adiciones es *O*(n)**








`c: capacidad inicial del array`
`n: número de adiciones al final el array`

# 2.2 Recorrido de una secuencia: Iterador

**Un *iterador* nos permite recorrer los elementos de una estructura de datos lineal**

- **pasando una única vez por cada uno de los elementos**

**Operaciones básicas de un iterador:**

- **`constructor`: posiciona el iterador al principio de la secuencia**
- **`hasNext`: indica si hay más elementos (no se ha llegado al final)**
- **`next`: retorna el siguiente elemento y avanza el iterador**

**Otras operaciones habituales:**

- **`hasPrevious`: indica si no se está al comienzo de la secuencia**
- **`previous`: retorna el elemento anterior y retrocede el iterador**
- **`add`: inserta un elemento en la posición del iterador**
- **`remove`: elimina el último elemento retornado por el iterador**





# Recorrido de una secuencia

**Sin usar iterador:**

```
tamaño : Entero := lista.size()
para i desde 0 hasta tamaño-1 hacer
   hace_algo(lista.get(i))
finhacer
```

**Usando iterador:**

```
Iterador iter := lista.iterator()
mientras iter.hasNext() hacer
   hace_algo(iter.next())
finhacer
```

**En la implementación con array ambos recorridos son igual de eficientes *O*(n) (puesto que tanto `get` como `next` son *O*(1))**

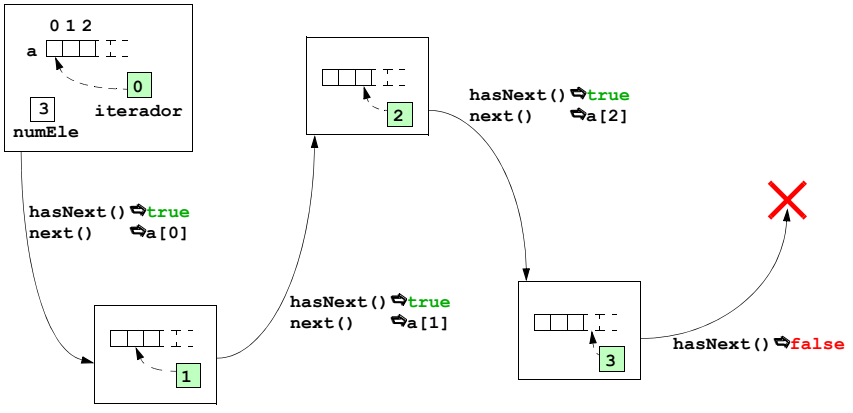
- **un poco más rápido con iterador (menos comprobaciones)**
- **en otras implementaciones el iterador es mucho más eficiente**





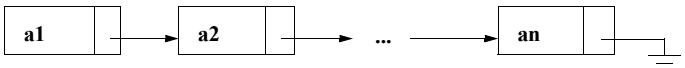
# Iterador en la implementación mediante array

- **Muy sencillo de implementar: cursor con el valor del índice del próximo elemento a retornar**



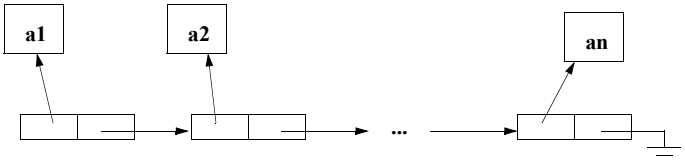
# 2.3 Implementación mediante listas simplemente enlazadas

**Basadas en una "cadena" de celdas enlazadas mediante punteros**




- cada celda tiene un puntero a la siguiente
- la última celda tiene un puntero a `null`

**Siendo más exactos, la mayoría de las implementaciones tienen esta estructura:**

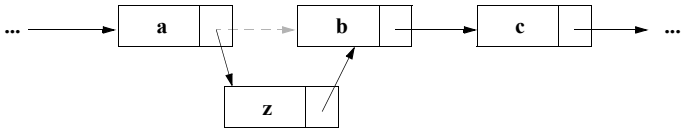




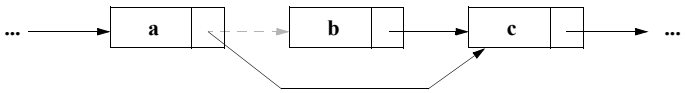



# Operaciones básicas de inserción y extracción

**Inserción de un nuevo elemento en la cadena**




**Eliminación de un elemento de la cadena**

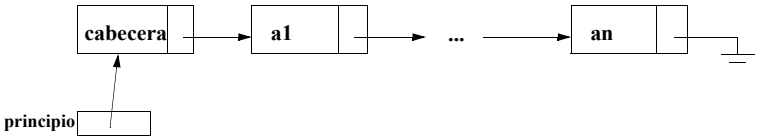







# Uso de nodos de cabecera

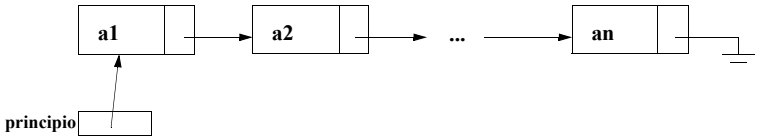
**El primer nodo de la lista es un nodo especial de cabecera**




- \+ La inserción y eliminación del primer elemento *no* es un caso especial
- \- Es un pequeño "truco"
- \- Utiliza algo de memoria extra

**En las trasparencias y en el ejemplo de implementación `ListaSimpleEnlace.java` se utiliza el nodo de cabecera**

**Sin nodo cabecera**




+ **No hay "trucos"**
- **La inserción y eliminación del primer elemento es un caso especial**

**El pseudocódigo `lista_simple_enlace` no utiliza nodo de cabecera**





# Iterador en listas simplemente enlazadas

**El acceso posicional es muy costoso**

- **acceder al elemento n-ésimo es *O*(n)**

**Para recorrer la lista es preferible utilizar el iterador**

- **la operación `next` es *O*(1)**

**Por eficiencia las operaciones `add` y `remove` se añaden al iterador**

- **`add`: inserta el elemento justo antes del que será devuelto por la siguiente llamada a `next` (definición en el API de Java)**
- **`remove`: elimina el elemento retornado por la última llamada a `next` o `previous` (definición en el API de Java)**
- **pueden utilizarse otras definiciones alternativas para `add` y `remove`**

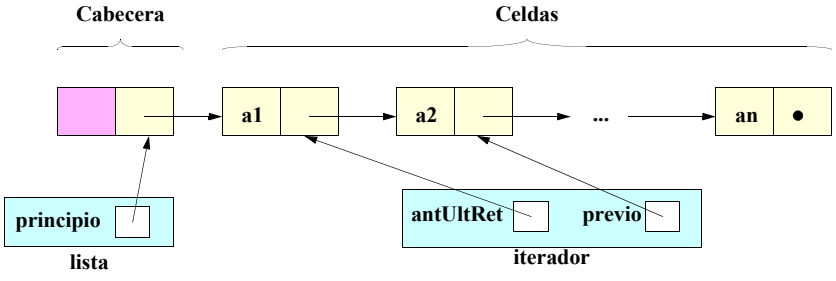




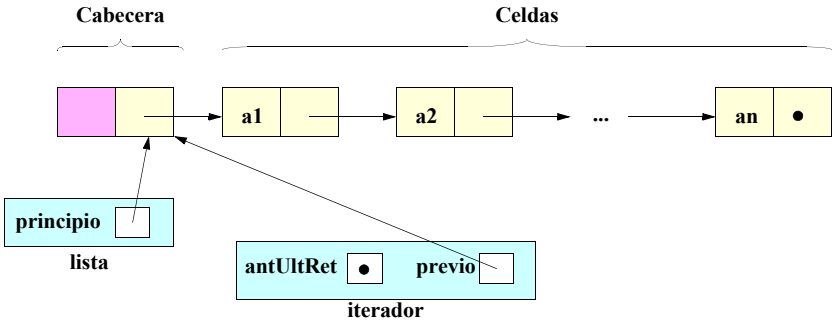
# Implementación del iterador

**El iterador de la lista contiene**

- **`previo`: puntero al elemento a retornar por `previous` (o dicho de otra forma, al anterior al que debe retornar `next`)**
- **`antUltRet`: puntero al elemento anterior al último retornado por `next` o `previous`**
  - **utilizado por `remove`**



# Situación inicial del iterador

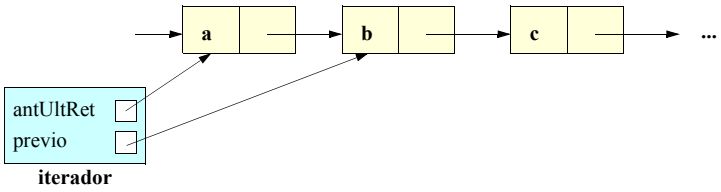




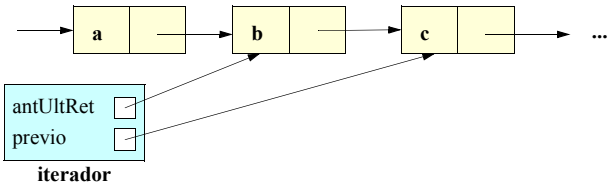



# Avance del iterador: `next`

1) situación inicial




2) después de `next`



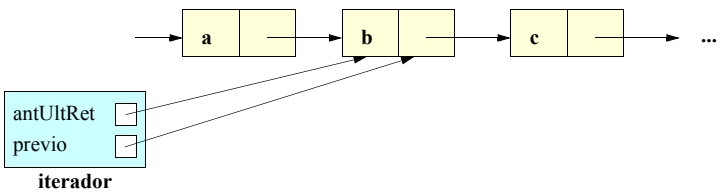

3) `next` retorna `c`



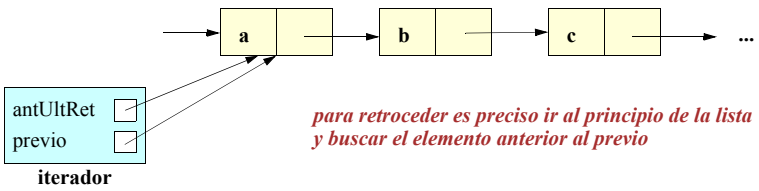


# Avance del iterador: `previous`

1) situación inicial



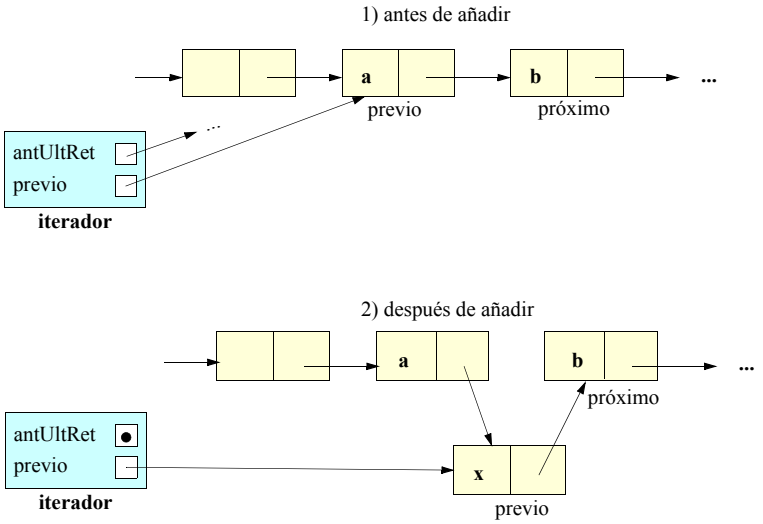

2) después de `previous`




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

3) `previous` retorna `b`

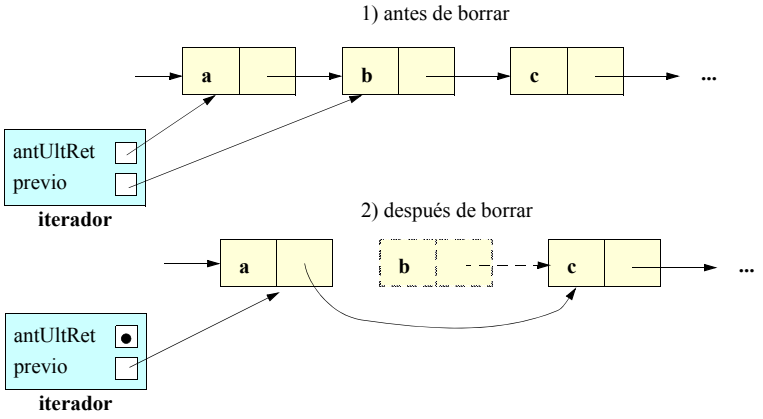
# Diagrama de añadir







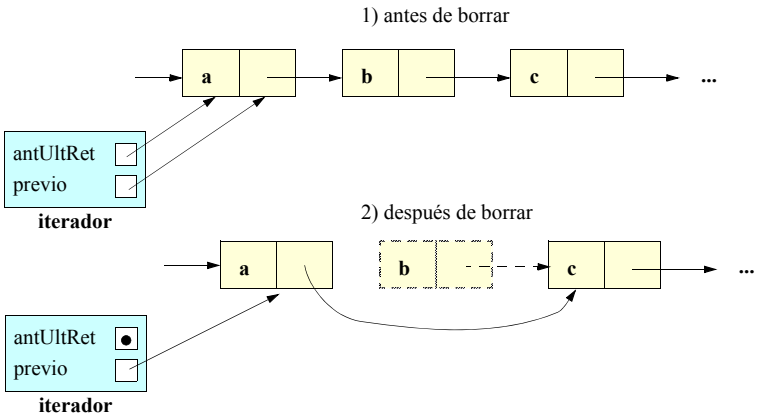

# Diagrama de borra (después de próximo)








# Diagrama de borra (después de previo)

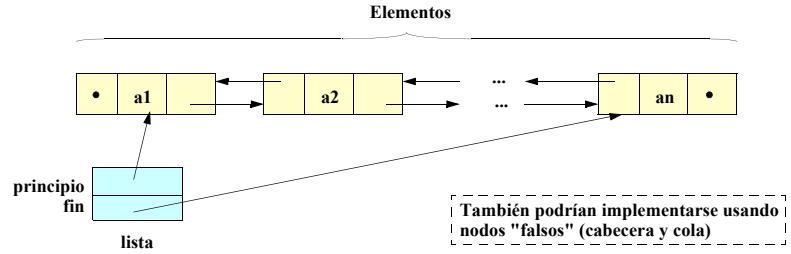


# 2.4 Implementación mediante listas doblemente enlazadas

**En una lista simplemente enlazada, las operaciones para acceder a la posición última y retroceder el iterador son costosas ($O(n)$)**

**Las *listas doblemente enlazadas* evitan este problema**

- **"cadena" de celdas con punteros al anterior y al siguiente**

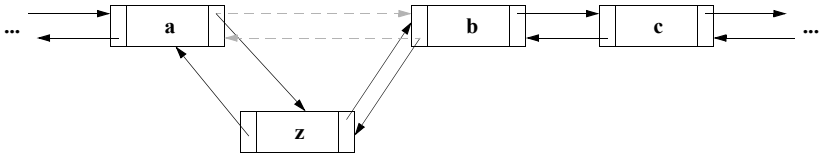




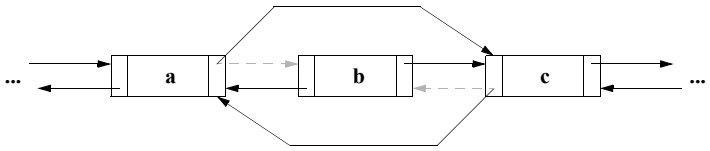



# Operaciones básicas de inserción y extracción

**Inserción de un nuevo elemento en la cadena**




**Eliminación de un elemento de la cadena**



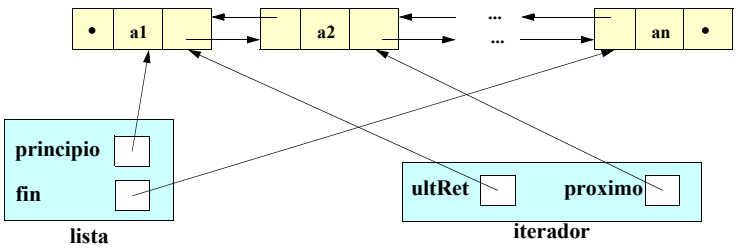




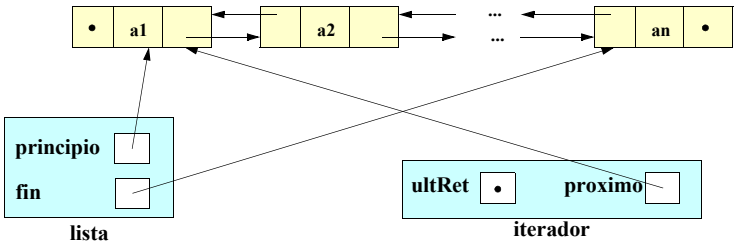

# Iterador de la lista

**El iterador de la lista contiene**

- **`proximo`: puntero al elemento a retornar por `next` (o dicho de otra forma, al posterior al que debe retornar `previous`)**
- **`ultRet`: puntero al último elemento retornado por `next` o `previous`**
  - **utilizado por `remove`**



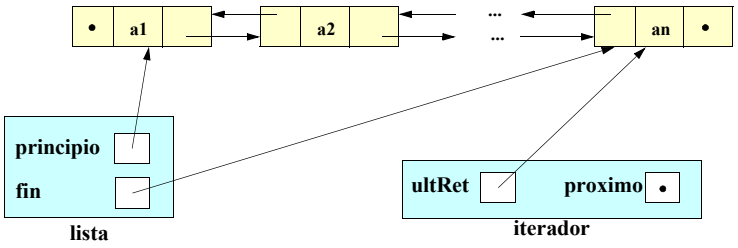
# Situación inicial del iterador







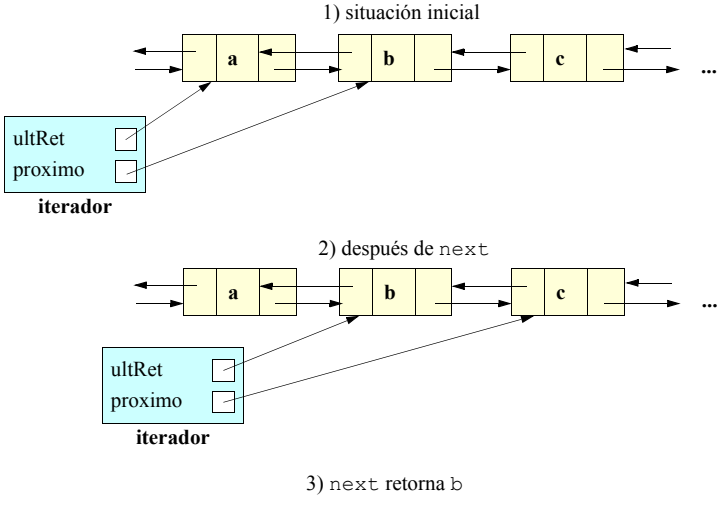

# Iterador al final de la lista



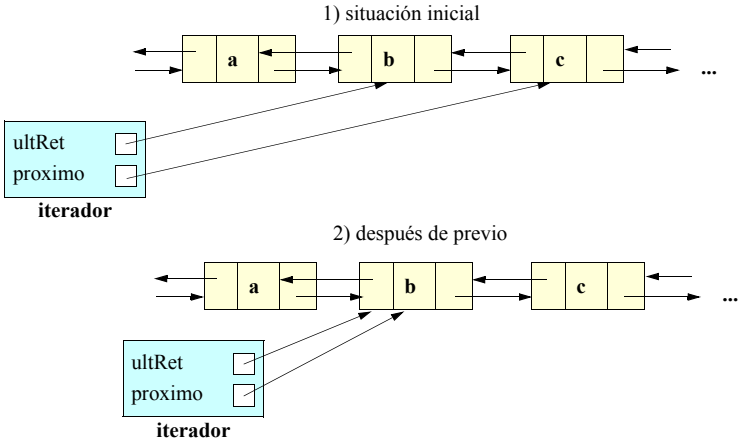





# Avance del iterador: `next`



# Avance del iterador: `previous`

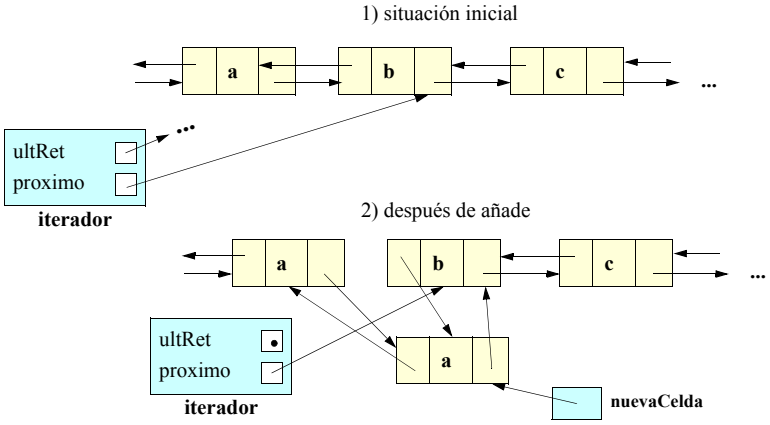



2) `previous` retorna `b`





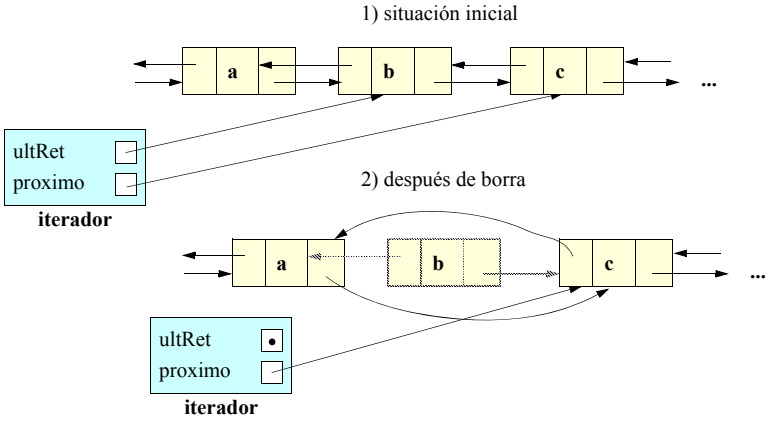
# Diagrama de añade



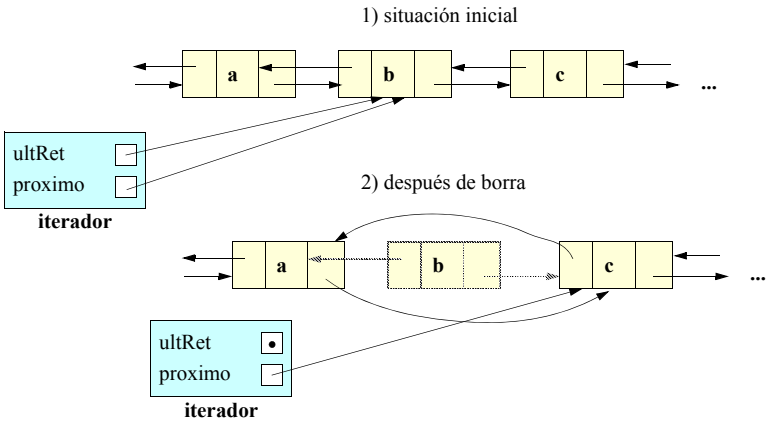





# Diagrama de borra (después de `next`)



# Diagrama de borra (después de `previous`)








# Comparación de las implementaciones

**Eficiencia temporal:**

| Operación | Array | Simple enlace | Doble enlace |
|---|---|---|---|
| añade primero/intermedio/último | O(n) O(n) O(1) | O(1) O(n)[1] O(n)[2] | |
| elimina primero/intermedio/último | O(n) O(n) O(1) | O(1) O(n)[1] O(n)[2] | |
| obtiene elemento i-ésimo | O(1) | O(n) | |
| iterador.next | O(1) | | |
| iterador.previous | O(1) | O(n) | O(1) |
| iterador.add | O(n) | O(1) | |
| iterador.remove | O(n) | O(1) | |

[1] `Hay que encontrar el elemento (O(n)) antes de añadirle/eliminarle`
[2] `Se puede hacer en O(1) añadiendo a la lista un puntero al final`





**Lista enlazada vs. implementación con array:**

- **+ No acotada, fácil crecimiento**
- **+ Sólo se utiliza la memoria que realmente se necesita**
- **+ Desplazar subestructuras es *O*(1)**
- **- Acceso posicional más costoso**
- **- Puede que tengamos que gestionar la liberación de memoria**