Estructuras de Datos. ATD y estructuras lineales

Profesores Estructuras de Datos, 2024.

Dpto. Lenguajes y Ciencias de la Computación.

University of Málaga

El TAD Lista

- Una **lista** es una colección que almacena elementos en un orden lineal.
- Cada elemento tiene una **posición** (o **índice**) en la lista (0 para el primer elemento, 1 para el segundo, etc.).

Operaciones

- `append` : Inserta un elemento al final de la lista.
- `prepend` : Inserta un elemento al principio de la lista.
- `insert` : inserta un elemento **en una posición dada** en la lista.
- `delete` : elimina el elemento **en una posición dada** en la lista.
- `get` : Devuelve (sin eliminar) el elemento **en una posición determinada** en la lista.
- `set` : Reemplaza el elemento **en una posición dada** posición en la lista.
- `isEmpty` : Comprueba si la lista está vacía.
- `size` : Devuelve el número de elementos en la lista.
- `clear` : Elimina todos los elementos de la lista.
- `contains` : Comprueba si la lista contiene un elemento determinado.
- `iterator` : Devuelve un `Iterador` para recorrer los elementos de la lista.

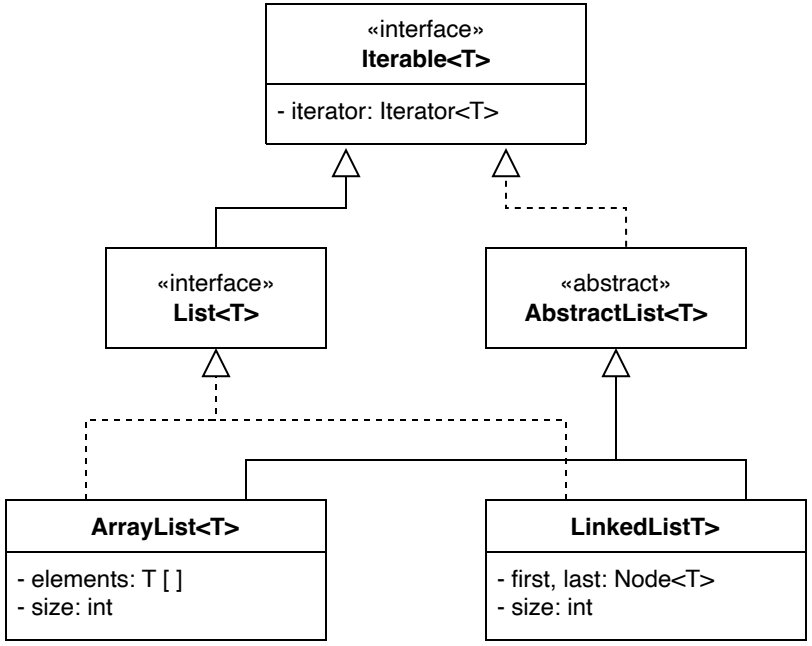
El ADT de lista en Java

- La interfaz `List<T>` define una lista con elementos de tipo `T`. A partir de Java 8 algunos métodos se pueden implementar en la interfaz (`default`).

```
public interface List<T> extends Iterable<T> {
  boolean isEmpty();
  int size();
  void insert(int index, T element);
  default boolean contains(T element) {
        for (T elem : this) {
            if (Objects.equals(elem, element)) {
                return true;
                //Código simple se puede hacer, pero dificulta la lectura en bloques grandes
            }
        }
        return false;
  }
  void delete(int index);
  void clear();
  T get(int index);
  void set(int index, T element);
  void append(T element);
  void prepend(T element);
  default void append(T... elements) {
        for (T element : elements) {
            append(element);
        }
    }
}
```

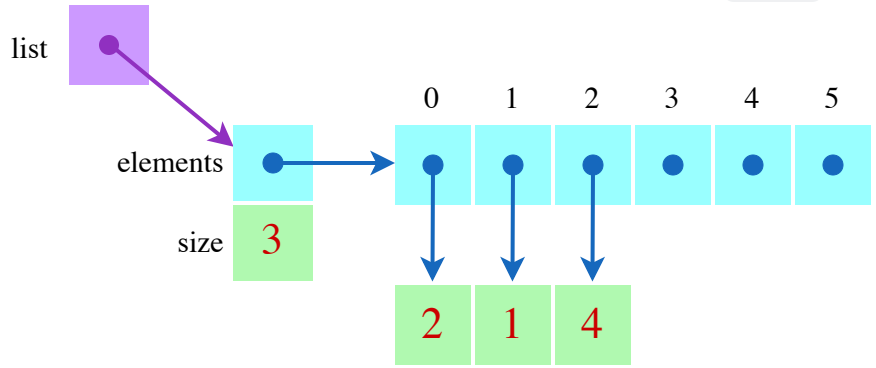
Lista de implementaciones de ADT

- Diferentes *clases* pueden implementar la interfaz `List<T>`:
  - `ArrayList<T>`: utiliza una array para almacenar elementos.
  - `LinkedList<T>`: utiliza una estructura vinculada para almacenar elementos.
- La clase abstracta base `AbstractList<T>` proporciona implementación para los métodos `equals`, `hashCode` y `toString`, **los cuales son independiente de la implementación**.



La clase `ArrayList`

- `ArrayList<T>` implementa la interfaz `List<T>` utilizando una array para almacenar elementos.
- Inicialmente, el array tiene un tamaño fijo (la *capacidad* de la lista), pero puede crecer dinámicamente cuando sea necesario.
- La posición de cada elemento en la lista corresponde directamente a su índice en el array, y el elemento en el índice de la lista `i` se ubica en el índice de el array `i`.
- Insertar nuevos elementos en la lista provoca un **desplazamiento** hacia la derecha de los elementos subsiguientes de el array, y se debe asegurar que hay espacio.
- Por el contrario, cuando se elimina un elemento, los elementos subsiguientes se **desplazan** hacia la izquierda, eliminando cualquier espacio libre.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en la lista.

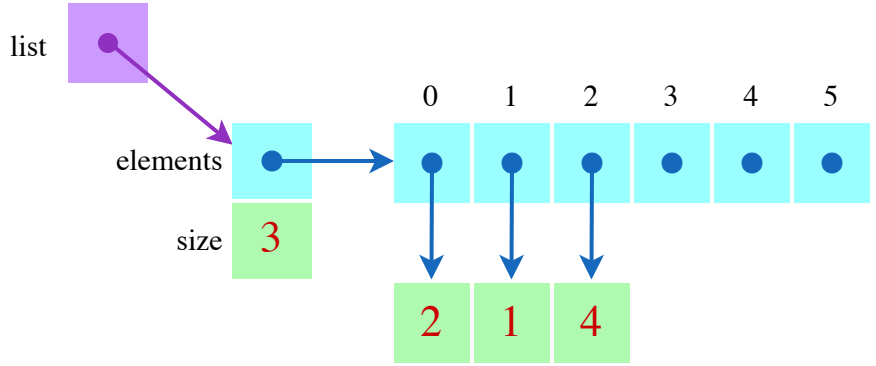


Implementación de `ArrayList`

```
public class ArrayList<T> extends AbstractList<T> implements List<T> {
  private static final int DEFAULT_INITIAL_CAPACITY = 16;

  private T[] elements;
  private int size;

  public ArrayList(int initialCapacity) {  // ArrayList constructor
    if (initialCapacity <= 0) {
      throw new IllegalArgumentException("initial capacity must be greater than 0");
    }
    elements = (T[]) new Object[initialCapacity];
    size = 0;
  }

  public ArrayList() {  // ArrayList constructor
    this(DEFAULT_INITIAL_CAPACITY);
  }

  ...
}
```

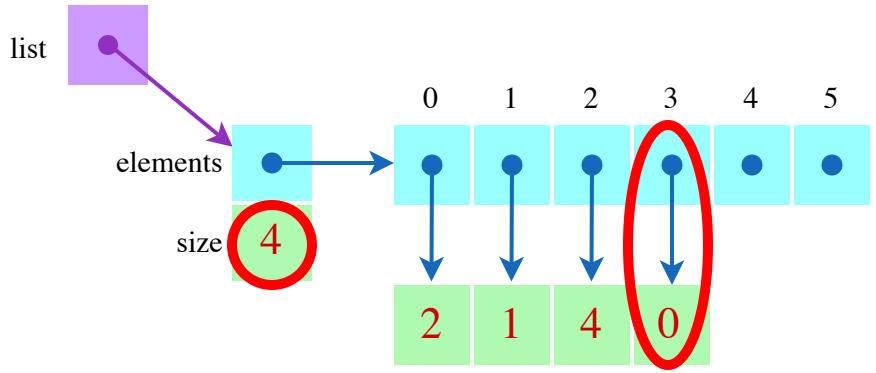
Agregar un elemento al final en `ArrayList`

- `append` agrega un elemento después del último elemento de la lista
- Se debe **asegurar la capacidad del array** para alojar el nuevo elemento.
- el array almacena el nuevo elemento en el índice designado por `size`.
- `size` se incrementa para realizar un seguimiento de la cantidad de elementos en la lista.
- A partir de esta configuración, vamos a 'añadir' el elemento 0 a la lista:



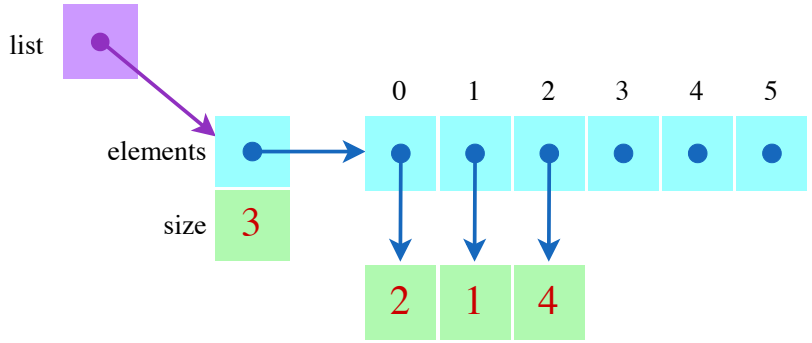
Agregar un elemento al final en `ArrayList` (II)

- Después de agregar 0:

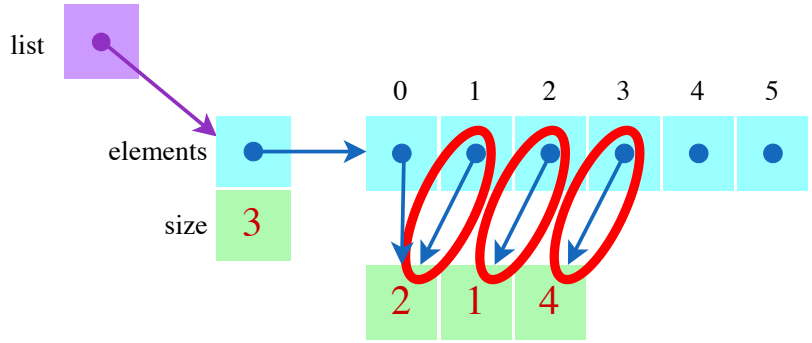


Insertar un elemento al principio en `ArrayList`

- A partir de esta configuración, vamos a anteponer el elemento 7 al inicio:

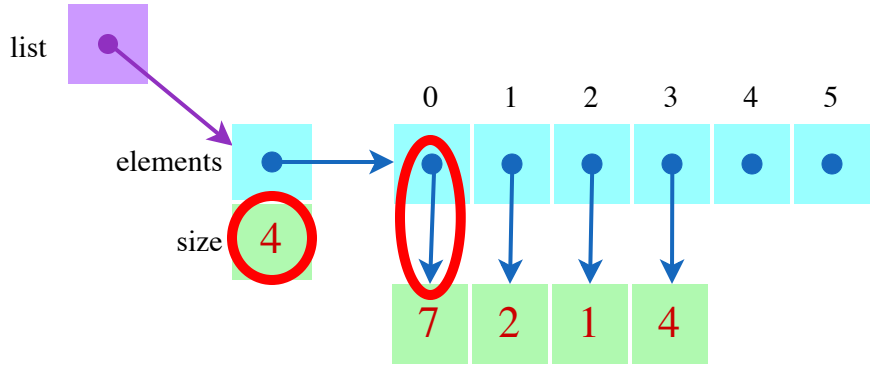


- Las referencias en las posiciones 2, 1 y 0 se desplazan hacia la derecha para hacer espacio para el nuevo elemento:



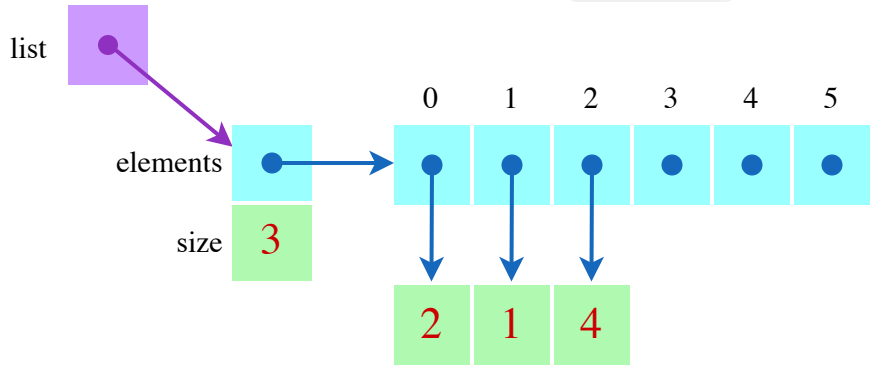
Insertar un elemento al principio en `ArrayList`

- El nuevo elemento (7) se almacena en la posición 0 y se incrementa el tamaño:

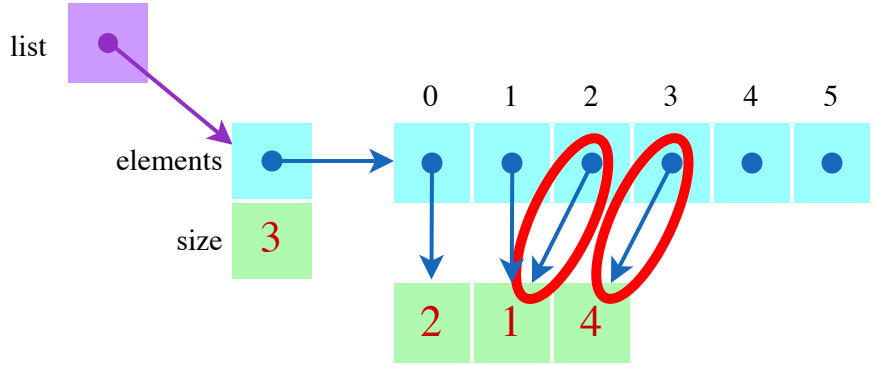


Insertar un elemento en una posición arbitraria en `ArrayList`

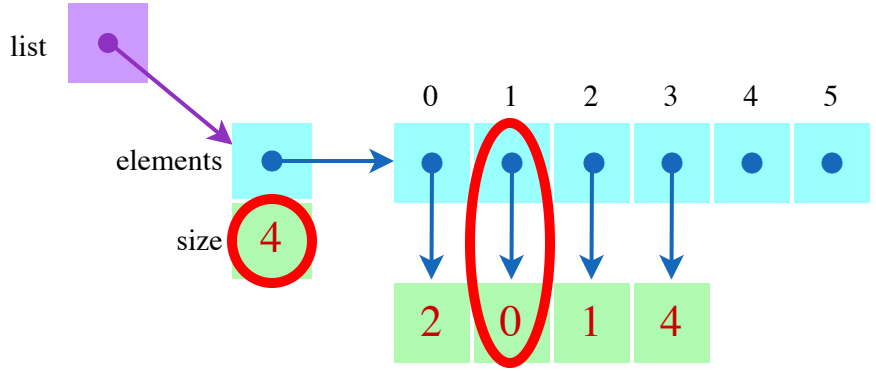
- A partir de esta configuración, vamos a `insertar` el elemento 0 en la posición 1:



- Las referencias en las posiciones 2 y 1 se desplazan *hacia la derecha* para hacer espacio para el nuevo elemento:

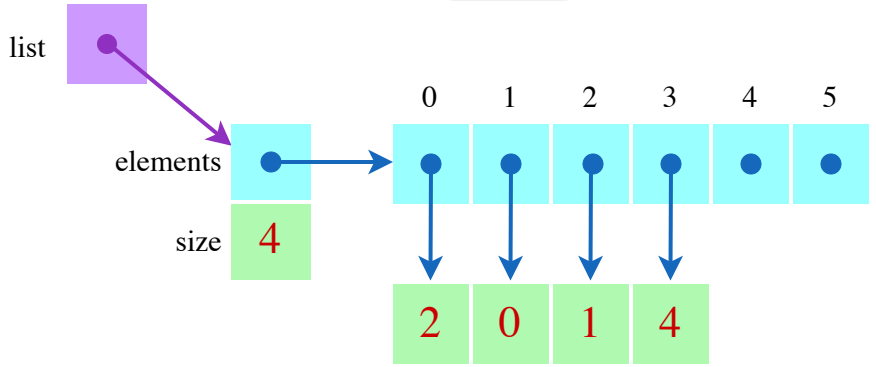


- El nuevo elemento (0) se almacena en la posición 1 y se incrementa el tamaño:

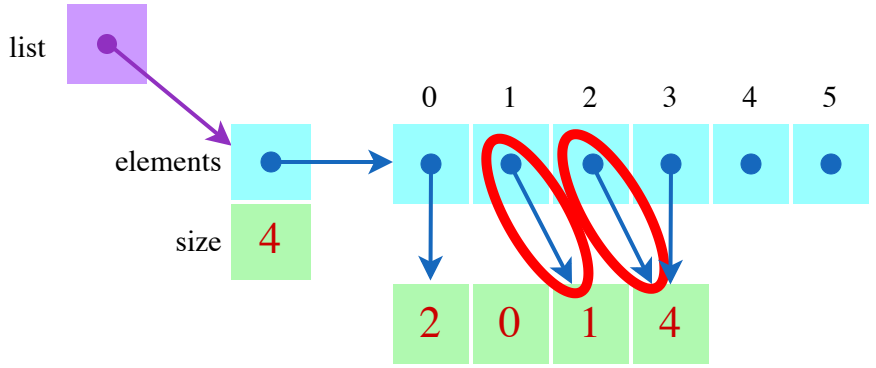


Eliminar un elemento en una posición arbitraria en `ArrayList`

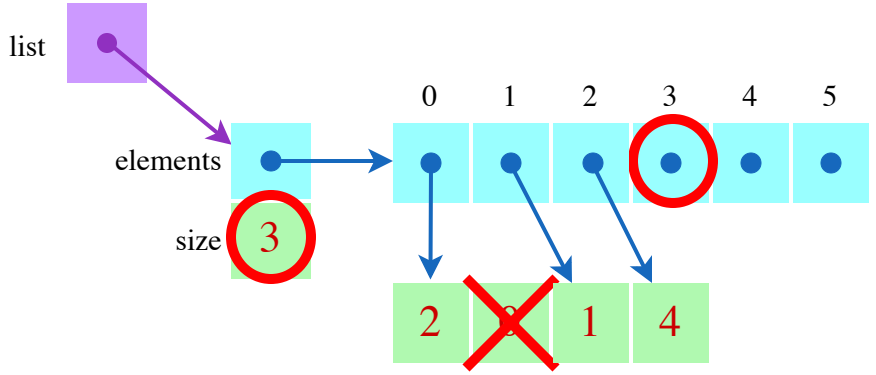
- A partir de esta configuración, vamos a `eliminar` el elemento en la posición 1:



- Las referencias en las posiciones 2 y 3 se desplazan *hacia la izquierda* para llenar el espacio dejado por el elemento eliminado:



- `size` se reduce y se establece la referencia en el índice `size` a `null`. El *recolector de basura* recuperará la memoria utilizada por el elemento eliminado:



Métodos de fábrica para `ArrayList`

Los métodos de fábrica ofrecen una forma conveniente de crear instancias de objetos `ArrayList<T>` sin invocar directamente constructores.
Estos métodos incluyen:

- `empty()`: construye una *lista vacía* con una capacidad inicial *predeterminada*, adecuada para cuando se desconoce el número esperado de elementos.
- `withCapacity(int initialCapacity)`: construye una *lista vacia* con una capacidad inicial especificada, optimizando la asignación de memoria para un número conocido de elementos.
- `of(T... elementos)`: construye una lista *previamente rellenada* con los elementos proporcionados, lo que permite una configuración de lista rápida y sencilla.
- `copyOf(List<T> list)`: construye una nueva lista que es un *duplicado* de la `lista` dada, preservando el orden de los elementos.
- `from(Iterable<T> iterable)`: construye una nueva lista que contiene todos los elementos del *iterable* especificado, manteniendo su orden de iteración.

```
List<Integer> list1 = ArrayList.empty(); // Create an empty list with default initial capacity

List<Integer> list2 = ArrayList.of(1, 2, 3); // Create a list containing the elements 1, 2 and 3

List<Integer> list3 = ArrayList.copyOf(list2); // Create a copy of list2 and append the element 4
list3.append(4);

// Create a list from a queue of elements and calculate their sum
List<Integer> list4 = ArrayList.from(ArrayQueue.of(5, 6, 7));
int sum = 0;
for (Integer element : list4) {
  sum += element;
}
```

La interfaz `Iterator` en Java

- La interfaz `Iterator<T>` proporciona una manera de *recorrer* los elementos de una colección.

```
package java.util;
public interface Iterator<T> {
  boolean hasNext();
  T next();
}
```

- Un `Iterador` es un objeto *con estado* que realiza un seguimiento de su posición actual en la colección.
- Tiene dos métodos:
  - `hasNext()`: Devuelve `true` si hay más elementos.
  - `next()`: devuelve el siguiente elemento de la colección y avanza el iterador, garantizando que la llamada posterior produzca un elemento diferente.
- Utilizando un 'Iterador':

```
List<Integer> list = ArrayList.of(1, 2, 3);
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {  // this loop iterates over all elements of the list
  Integer element = it.next();
  System.out.println(element);
}
```

La interfaz «Iterable» en Java

- Quien implemente la interfaz `Iterable<T>` debe definir un método para devolver un `Iterator<T>`

```
package java.lang;

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

- Un clase que implemente `Iterable`, tendrá objetos que podrán ser recorridos en un bucle *foreach* para iterar sobre sus elementos:

```
List<Integer> list = ArrayList.of(1, 2, 3);
for (Integer element : list) { // this loop iterates over all elements of the list
  System.out.println(element);
}
```

Mejorar `ArrayList` con iterabilidad

- Para dotar a la clase `ArrayList` de iterabilidad, debe implementar la interfaz `Iterable` y proporcionar un método `iterator`. Este método es responsable de producir una instancia `Iterator` para recorrer los elementos de la lista.
- Se crea una *clase interna* llamada `ArrayListIterator` para implementar la interfaz `Iterator`, proporcionando la funcionalidad necesaria.
- Al invocar el método `iterator` se crea una instancia y se devuelve un nuevo objeto `ArrayListIterator`.
- El `ArrayListIterator` está diseñado para implementar el recorrido de los elementos de la lista:
  - Determina si la iteración ya ha cubierto todo elementos de la lista.
  - Identifica el índice del próximo elemento a recorrer.
  - Esto se logra manteniendo una variable `int` `current` que almacena el índice del próximo elemento que se devolverá.

Mejora de `ArrayList` con iterabilidad. Código

```
import java.util.Iterator;

public class ArrayList<T> extends AbstractList<T> implements List<T> {  // List extends Iterable
  private T[] elements;
  private int size;

  ...

  public Iterator<T> iterator() {  // implements Iterable interface method
    return new ArrayListIterator();  // return a new instance of ArrayListIterator
  }

  // Inner class to implement the Iterator interface
  private final class ArrayListIterator implements Iterator<T> {
    int current;  // index of the next element to be returned

    public ArrayListIterator() {  // ArrayListIterator constructor
      current = 0;  // start at the first element
    }

    public boolean hasNext() {
      return current < size;  // there are more elements if current is less than size
    }

    public T next() {
      if (!hasNext()) {
        throw new NoSuchElementException();  // all elements have been traversed
      }
      T element = elements[current];  // get the next element
      current++;  // advance iterator's state for subsequent invocations
      return element;  // return the element
    }
  }
}
```

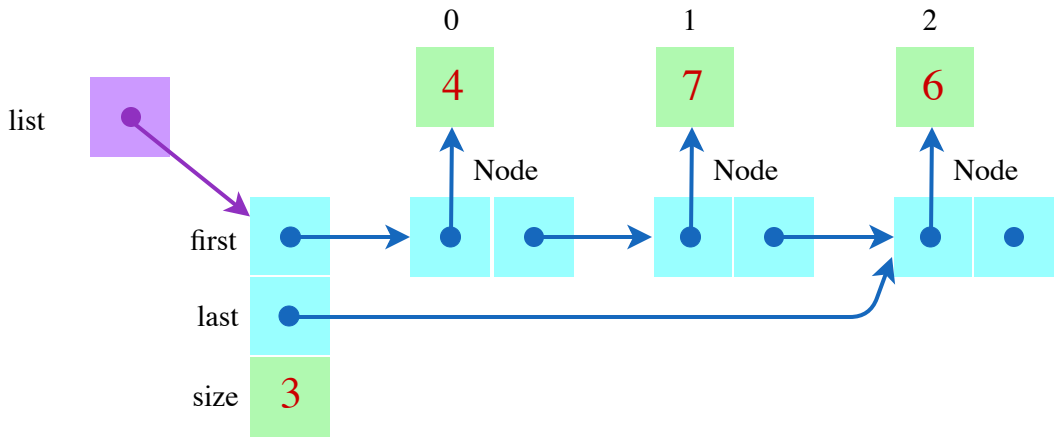
Complejidad computacional de las operaciones de `ArrayList`

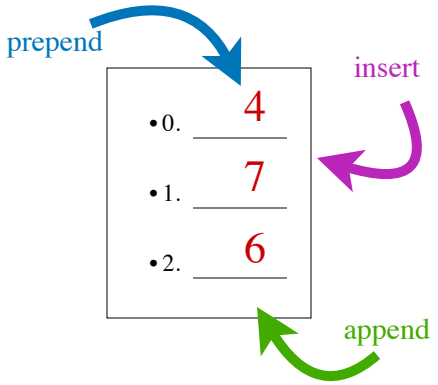
Operation	Cost
`empty`	O(1) †
`append`	O(1), O(n) §
`prepend`	O(n)
`insert`	O(n)
`delete`	O(n)
`get`	O(n)
`set`	O(n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(n) *
`contains`	O(n)

† In `empty` the size of the created array is a constant.
§ `enqueue` will need to copy n elements when the array has to be enlarged.
* `clear` will need to set n references to `null`.

La clase `LinkedList`

- `LinkedList<T>` implementa la interfaz `List<T>` utilizando una estructura vinculada de nodos.
- La posición de cada elemento en la lista corresponde directamente a su posición en la estructura vinculada, y el elemento en el índice de lista `i` se ubica en el nodo en la posición `i`.
- La clase mantiene una referencia `first` al primer nodo en la estructura vinculada que corresponde al primer elemento de la lista (el que está en el índice 0).
- La clase también mantiene una referencia `last` al último nodo en la estructura vinculada que corresponde al último elemento de la lista (el que está en el índice `size - 1`).
- Cada nodo contiene un elemento y una referencia (`next`) al nodo que contiene el elemento después de él en la lista, excepto el último nodo que tiene su referencia `next` establecida en `null`.
- La clase también mantiene una variable entera `size` para realizar un seguimiento de la cantidad de elementos en la lista.



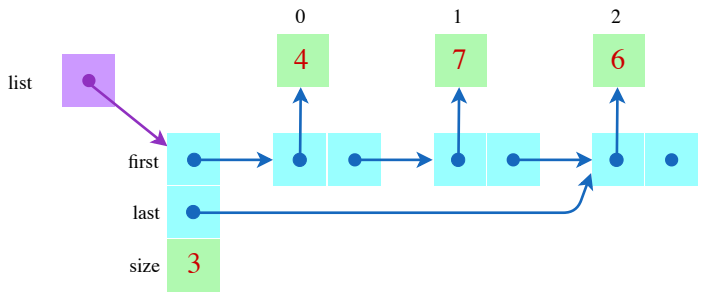


Implementación de `LinkedList`

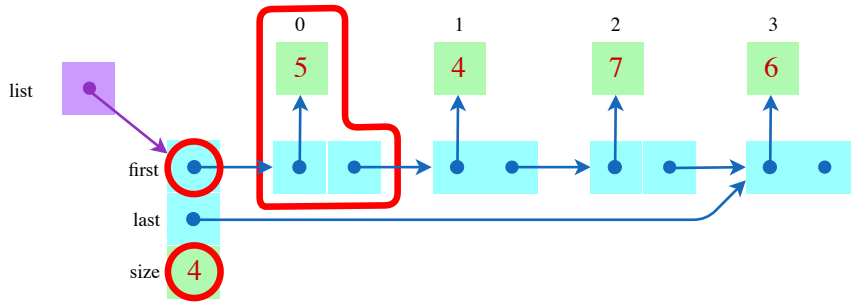
```
public class LinkedList<T> extends AbstractList<T> implements List<T> {
  private static final class Node<E> {  // Node inner class
    E element;
    Node<E> next;

    Node(E element, Node<E> next) {  // Node constructor
      this.element = element;
      this.next = next;
    }
  }

  private Node<T> first, last;
  private int size;

  public LinkedList() {  // LinkedList constructor
    first = null;
    last = null;
    size = 0;
  }
  ...
```

Insertar un elemento al principio en `LinkedList`

- Partiendo de esta configuración, vamos a anteponer el elemento 5:



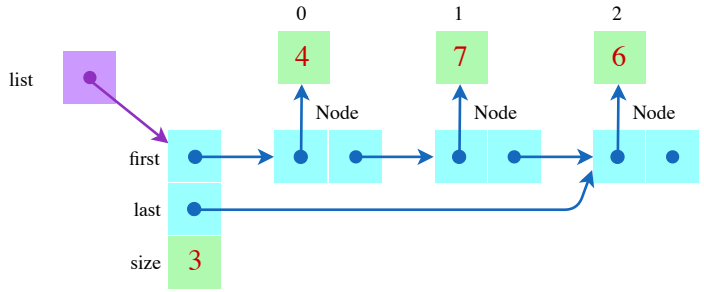
- Después de anteponer 5:



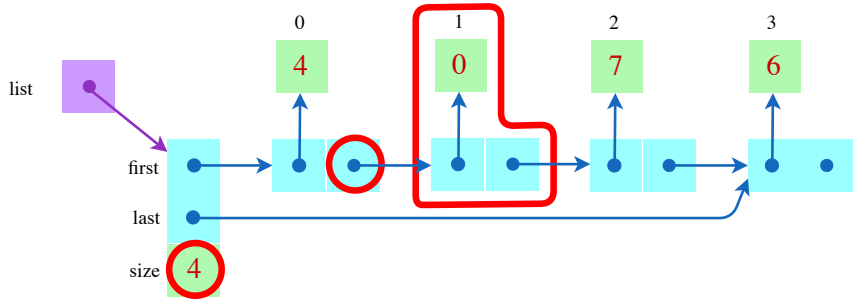
- Tener en cuenta que, si la lista estaba vacía, `last` también debe actualizarse para apuntar al nuevo `nodo`.

Insertar un elemento en una posición arbitraria en `LinkedList`

- Casos sencillos:
  - Insertar en la posición `0` es lo mismo que anteponer.
  - Insertar en la posición `tamaño` es lo mismo que añadir.
- Insertar dentro de la lista:
  - Partiendo de esta configuración, vamos a `insertar` el 0 en la posición 1:

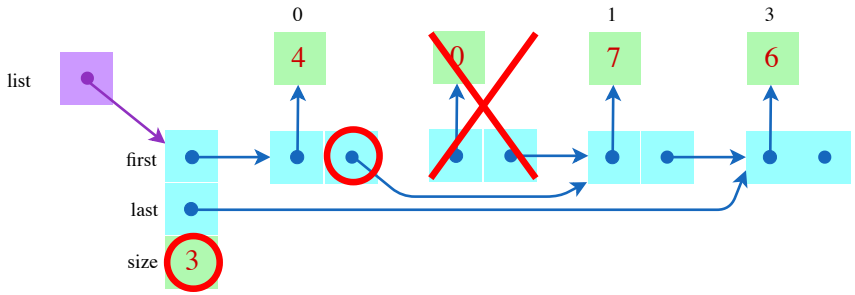


  - Después de insertar 0 en la posición 1:



Eliminar un elemento en una posición arbitraria en `LinkedList`

- Casos sencillos:
  - Eliminar en la posición `0` implica actualizar `first`. Si la lista se vuelve vacía, `last` también se establece en `null`.
  - Eliminar en la posición `size - 1` implica actualizar `last`. Si la lista se vuelve vacía, `first` también se establece en `null`.
- Eliminar dentro de la lista en la posición `i`:
  - La referencia `siguiente` del nodo en la posición `i - 1` es actualizado con la referencia 'siguiente' del nodo que se está eliminando.
  - Se reduce el tamaño.
  - Lista después de eliminar el elemento que estaba en la posición 1:



Métodos de fábrica para `LinkedList`

Los métodos de fábrica ofrecen una forma conveniente de crear instancias de objetos `LinkedList<T>` sin invocar directamente constructores.
Estos métodos incluyen:

- `empty()`: construye una *lista vacía* con una capacidad inicial *predeterminada*, adecuada para cuando se desconoce el número esperado de elementos.
- `of(T... elementos)`: construye una lista *previamente rellenada* con los elementos proporcionados, lo que permite una configuración de lista rápida y sencilla.
- `copyOf(List<T> list)`: construye una nueva lista que es una *duplicado* de la `lista` dada, preservando el orden de los elementos.
- `from(Iterable<T> iterable)`: construye una nueva lista que contiene todos los elementos del *iterable* especificado, manteniendo su orden de iteración.

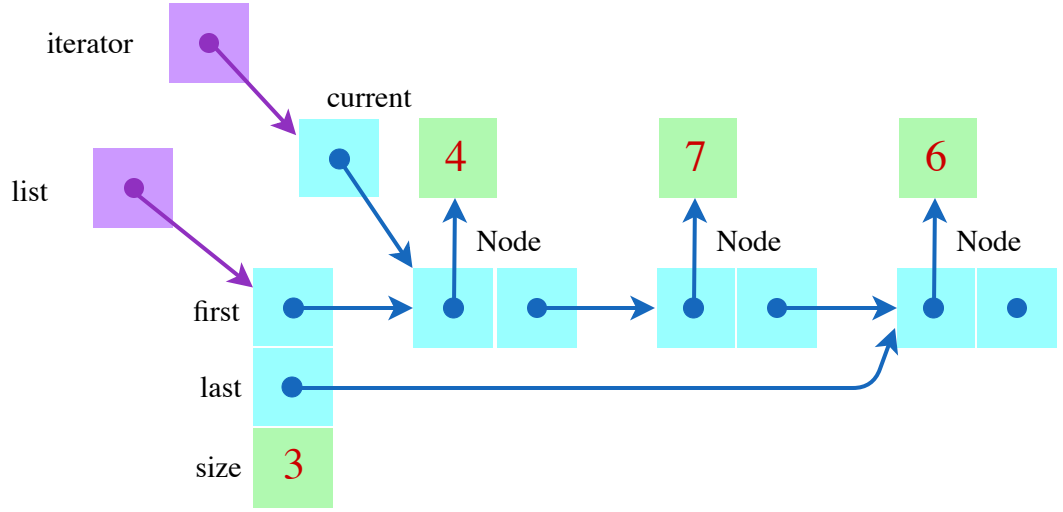
```
List<Integer> list1 = LinkedList.empty(); // Create an empty list with default initial capacity

List<Integer> list2 = LinkedList.of(1, 2, 3); // Create a list containing the elements 1, 2 and 3

List<Integer> list3 = LinkedList.copyOf(list2); // Create a copy of list2 and append the element 4
list3.append(4);

// Create a list from a queue of elements and calculate their sum
List<Integer> list4 = LinkedList.from(ArrayQueue.of(5, 6, 7));
int sum = 0;
for (Integer element : list4) {
  sum += element;
}
```

Mejorar `LinkedList` con iterabilidad

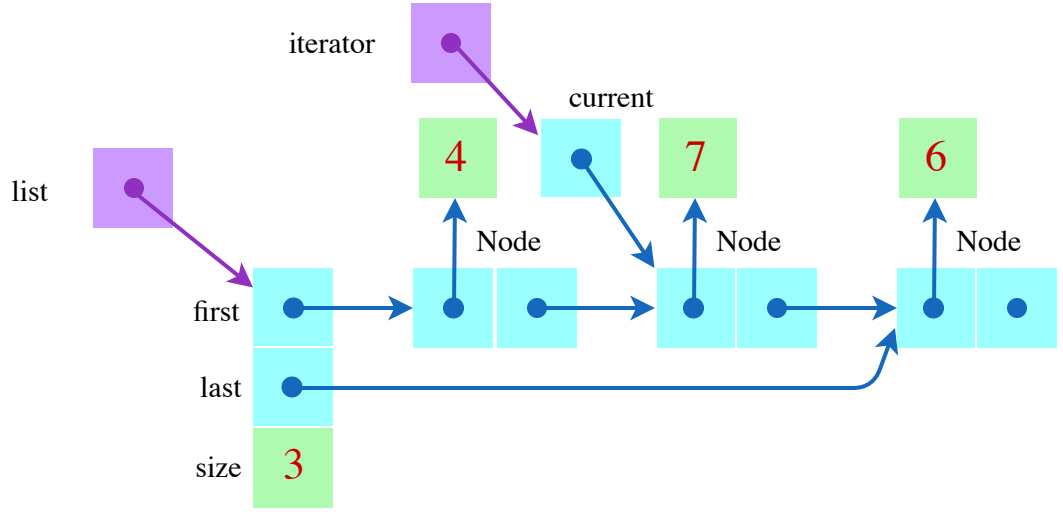
- Para dotar a la clase `LinkedList` de iterabilidad, debe implementar la interfaz `Iterable` y proporcionar un método `iterator`. Este método es responsable de producir una instancia `Iterator` para recorrer los elementos de la lista.
- Se crea una *clase interna* llamada `LinkedListIterator` para implementar la interfaz `Iterator`, proporcionando la funcionalidad necesaria.
- Al invocar el método `iterator` se crea una instancia y se devuelve un nuevo objeto `LinkedListIterator`.
- El `LinkedListIterator` está diseñado con conciencia de estado para implementar el recorrido de los elementos de la lista:
- Determina si la iteración ya ha cubierto todo
  elementos de la lista.
- Identifica el nodo del próximo elemento a recorrer.
- Esto se logra manteniendo una referencia al nodo que Será devuelto a continuación.

Mejora de `LinkedList` con iterabilidad. Ejemplo

- `LinkedListIterator` después de la inicialización:

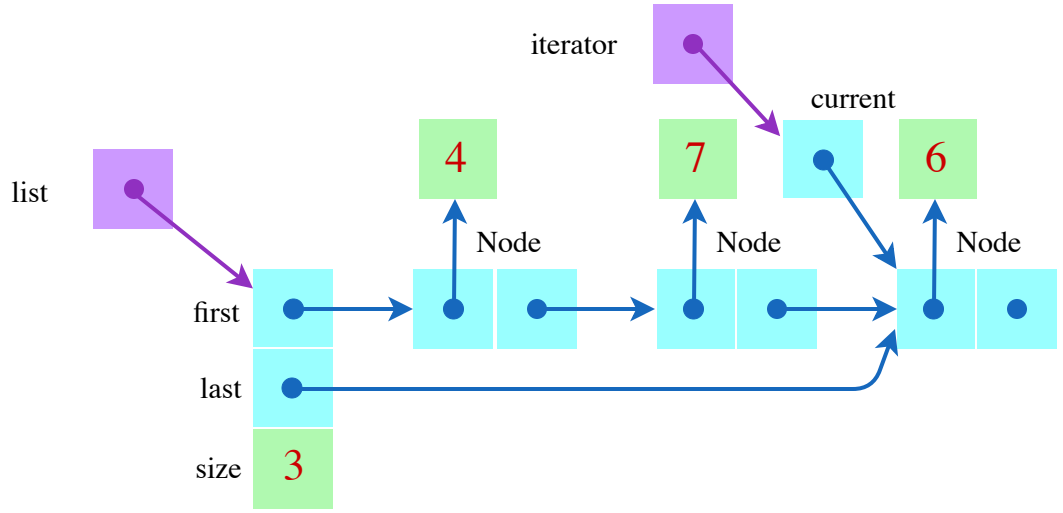


- Después de devolver el primer elemento:

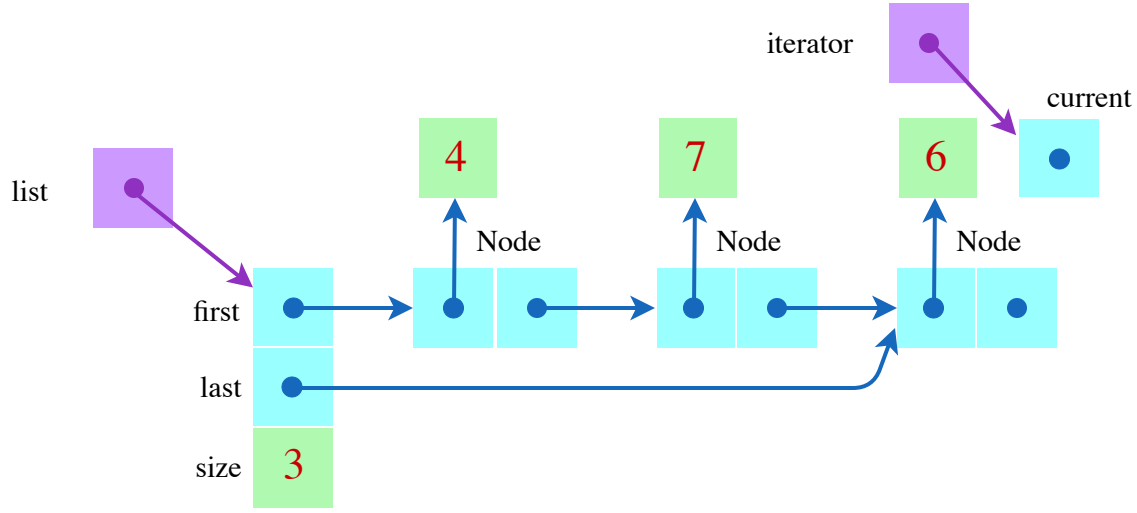


Mejora de `LinkedList` con iterabilidad. Ejemplo

- Después de devolver el segundo elemento:



- Después de devolver el tercer elemento:



Mejorar `LinkedList` con iterabilidad. Código

```
import java.util.Iterator;

public class LinkedList<T> extends AbstractList<T> implements List<T> {  // List extends Iterable
  ...
  private Node<T> first, last;
  private int size;
  ...

  public Iterator<T> iterator() {  // implements Iterable interface method
    return new ArrayListIterator();  // return a new instance of ArrayListIterator
  }

  // Inner class to implement the Iterator interface
  private final class LinkedListIterator implements Iterator<T> {
    Node<T> current;  // reference to the node with the next element to be returned

    public LinkedListIterator() {  // ArrayListIterator constructor
      current = ???;  // start at the first element
    }

    public boolean hasNext() {
      return ???;  // check if there are more elements
    }

    public T next() {
      if (!hasNext()) {
        throw new NoSuchElementException();  // all elements have been traversed
      }
      T element = ???;  // get the next element
      ???;  // advance iterator's state for subsequent invocations
      return element;  // return the element
    }
  }
}
```

Complejidad computacional de las operaciones de `LinkedList`

Operation	Cost
`empty`	O(1)
`append`	O(1)
`prepend`	O(1)
`insert`	O(n)
`delete`	O(n)
`get`	O(n)
`set`	O(n)
`isEmpty`	O(1)
`size`	O(1)
`clear`	O(1)
`contains`	O(n)

Comparación experimental entre «ArrayList» y «LinkedList»

- Medimos el tiempo de ejecución al realizar 100000 operaciones aleatorias (`insertar`, `eliminar` o `obtener`) en una lista inicialmente vacía.
- Usando una CPU Intel i7 860 y JDK 22:
  - `ArrayList` fue aproximadamente 4,30 veces más rápido que `LinkedList`.