

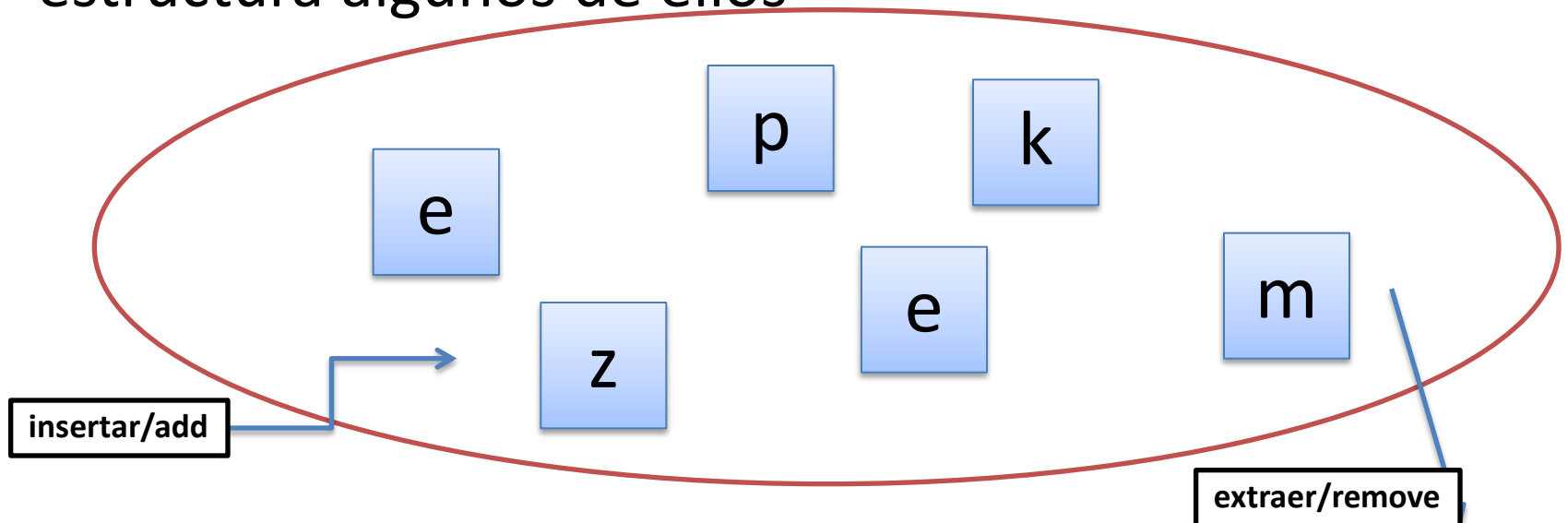
# Implementación de Estructuras de Datos Lineales

# Contenido

- Estructuras de datos lineales: pilas, colas y listas
- Implementación de la interfaz Lista
  - Implementación con Arrays (ListArray)
    - Lista de enteros y acotada
    - La lista nunca se llena
    - Implementación genérica
  - Implementación con nodos enlazados (ListLinked)
    - Lista de enteros
    - La clase nodo es una clase interna estática
    - Implementación genérica
  - Iteradores sobre listas
    - La clase Iterator es una clase anidada interna

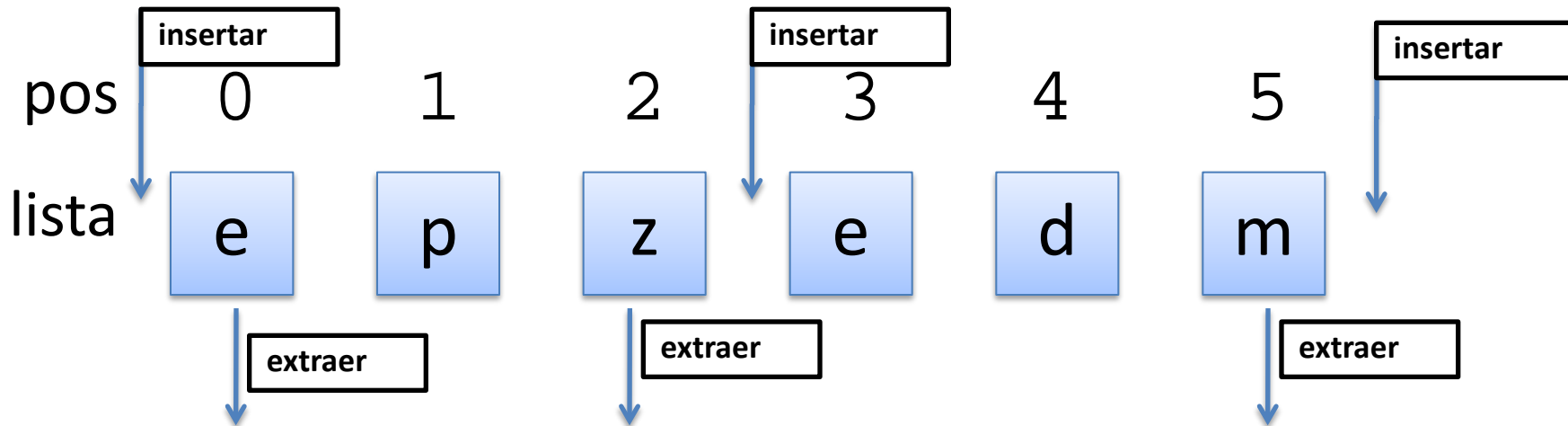
# Estructuras de datos lineales

- Una *colección* (*Collection*) es un objeto que contiene otros objetos, que son sus elementos
- Puede haber elementos repetidos, se pueden añadir y extraer elemento, o comprobar si está en la estructura algunos de ellos



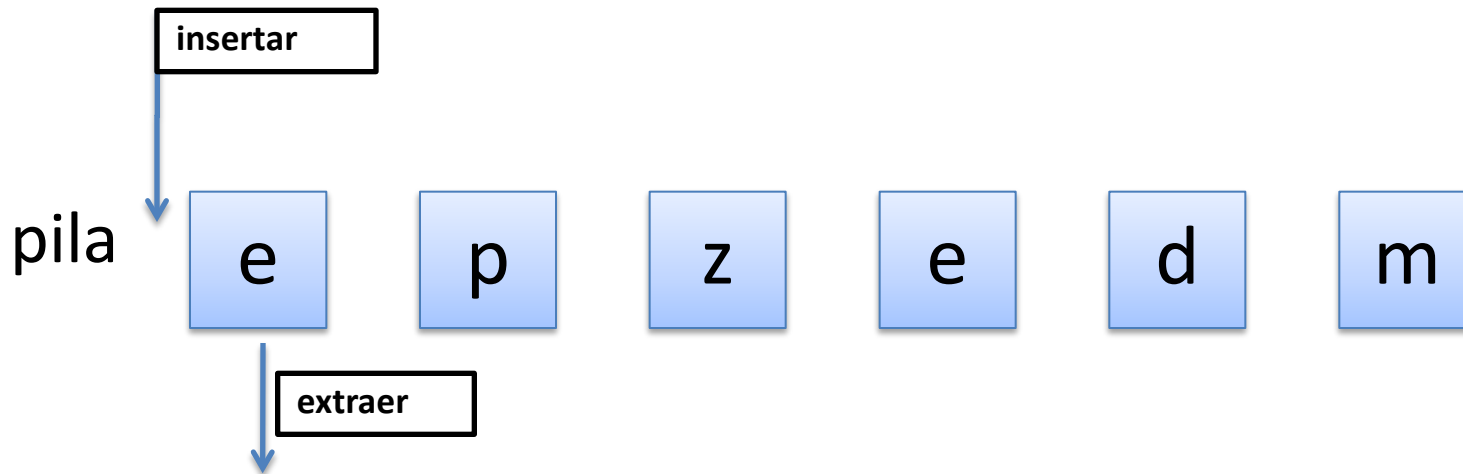
# Estructuras de datos lineales

- Una *lista* (*List*) es una colección de elementos ordenados según su posición en la estructura. Se puede extraer e insertar elementos en cualquier posición de la lista.



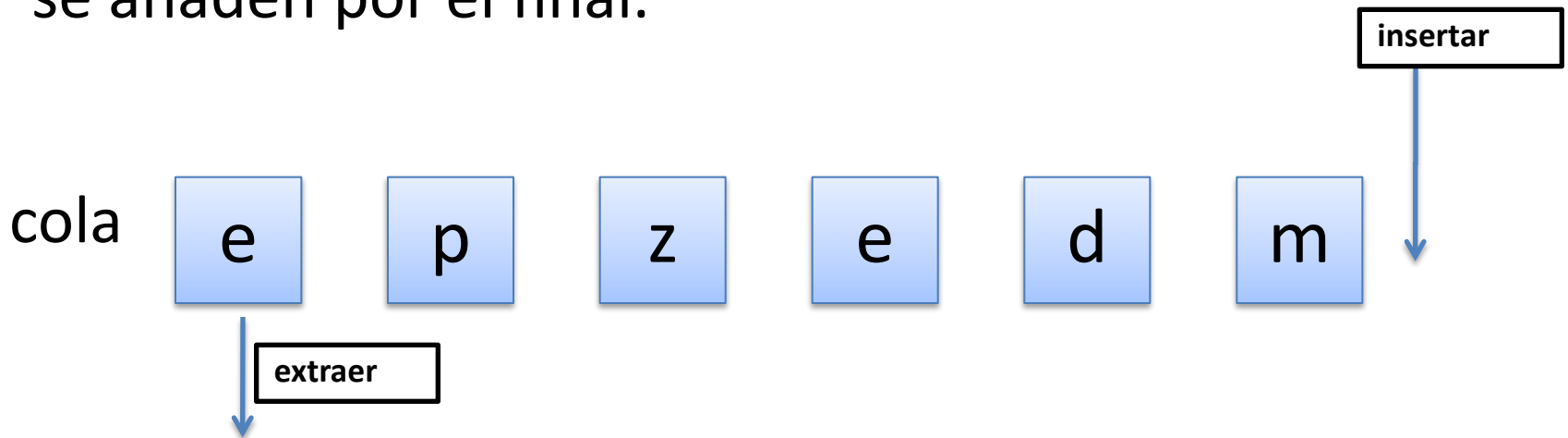
# Estructuras de datos lineales

- Una *pila* (*Stack*) es una colección de elementos ordenados según el tiempo que llevan en la estructura. Los elementos se extraen e insertan siempre por el mismo sitio (la cima de la pila)

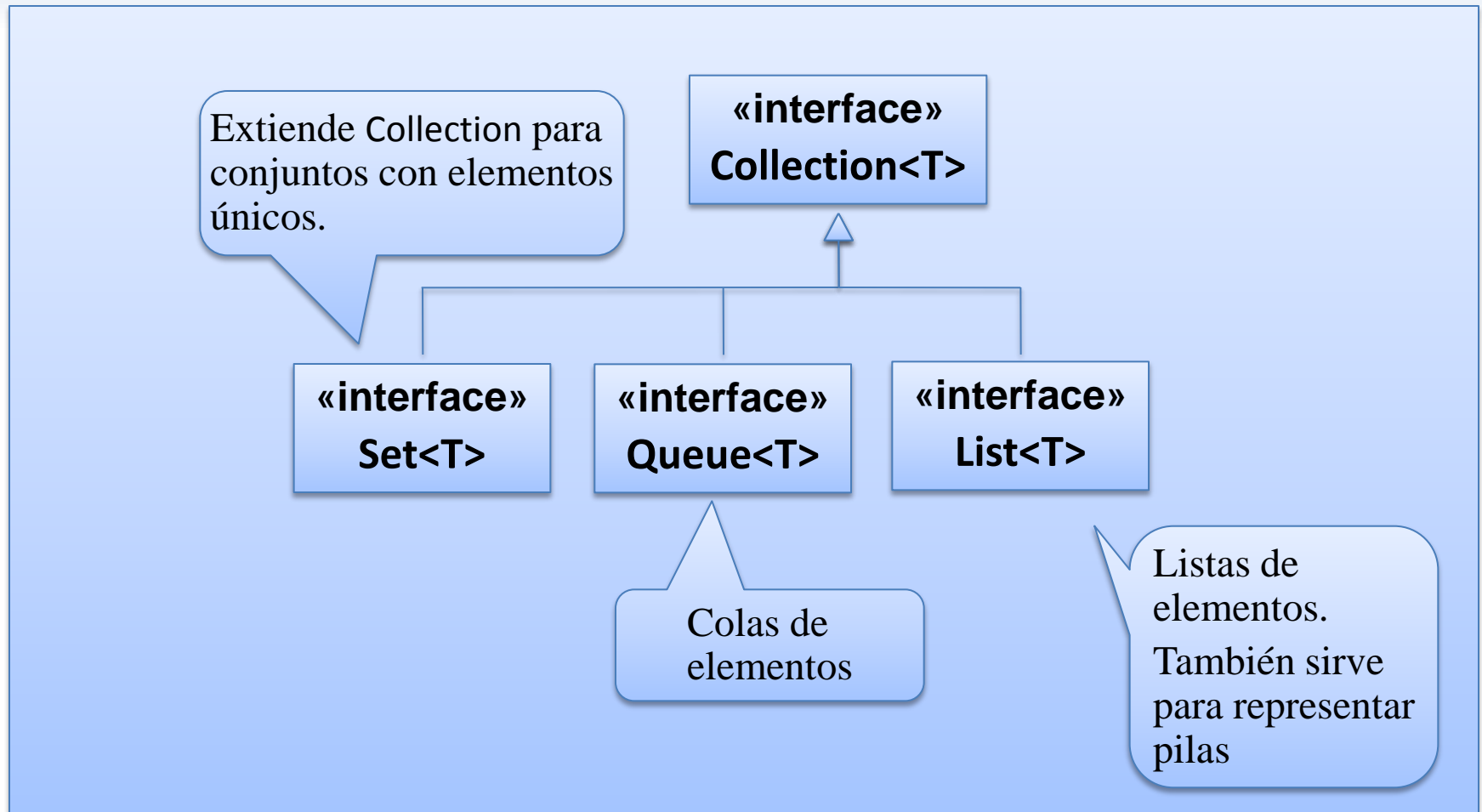


# Estructuras de datos lineales

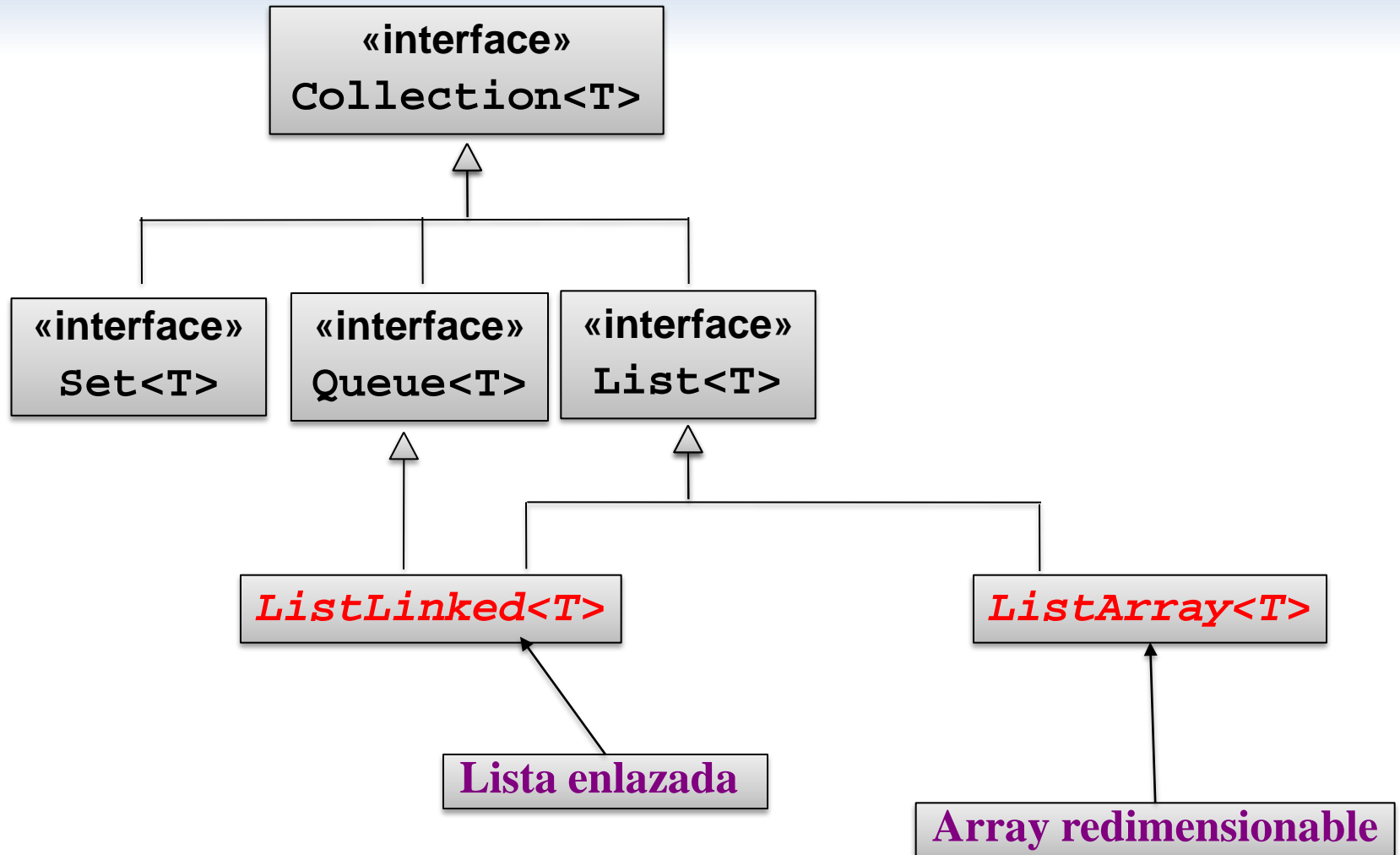
- Una *cola* (*Queue*) es una colección de elementos ordenados según el tiempo que llevan en la estructura. Se extrae siempre el elemento que lleva más tiempo en la estructura, y los nuevos elementos se añaden por el final.



# Implementación en el marco de colecciones



# Interfaces básicas y sus implementaciones





# Interfaz Collection (simplificada)

```
public interface Collection<T> extends Iterable<T> {
    // Operaciones básicas
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(T element);           // Opcional
    boolean remove(Object element);   // Opcional

    // Operaciones con grupos de elementos
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends T> c);
    boolean removeAll(Collection<?> c); // Opcional
    boolean retainAll(Collection<?> c); // Opcional

    void clear();                     // Opcional

    // Operaciones con arrays
    Object[] toArray();
    <S> S[] toArray(S a[]);
}
```

# Interfaz List<T> simplificada

```
public interface List<T> extends Collection<T> {  
    // Acceso posicional  
    T get(int index);  
    T set(int index, T element); // Opcional  
    void add(int index, T element); // Opcional  
    T remove(int index); // Opcional  
    boolean addAll(int index,  
                   Collection<? extends T> c); // Opcional  
  
    // Búsqueda  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    // Iteración  
    ListIterator<T> listIterator();  
    ListIterator<T> listIterator(int index);  
  
    // Vista de subrango  
    List<T> subList(int from, int to);  
}
```

# Implementación de Listas con Arrays

- Inicialmente consideraremos sólo listas de enteros
- Utilizaremos un array en el que algunas posiciones estarán ocupadas y otras no
- Usaremos una variable *size*, para distinguir entre celdas ocupadas y vacías
- Inicialmente suponemos que las listas son acotadas
- Los métodos de la interfaz que no implementamos lanzarán la excepción **UnsupportedOperationException**
  - Sólo serán relevantes los valores dentro del rango  $[0, \text{size} - 1]$ .

<i>indice</i>	0	1	2	3	4	5	6	...	98	99
<i>data</i>	17	932085	-32053278	100	3	0	0	...	0	0
<i>size</i>	5									

# Implementación de Listas con Arrays

```
public class ListArray{  
    private static final int DEFAULT_CAPACITY = 10;  
  
    //lista de datos  
    private int[] data;  
  
    // número actual de datos en la lista  
    private int size;
```

# Añadir Elementos a la Lista

- ¿Cómo añadir al final de la lista?
  - Almacena el elemento e incrementa el tamaño.
  - Inicialmente, suponemos que el array no está lleno

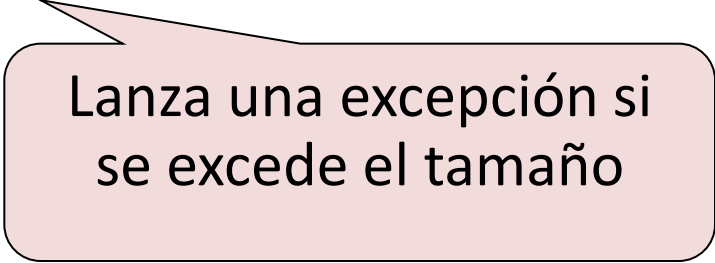
<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

– **`list.add(42);`**

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	42	0	0	0
<i>size</i>	7									

# Añadir Elementos a la lista

```
public boolean add(int value){  
    if (size + 1 > data.length){  
        throw new IndexOutOfBoundsException  
            ("Tamaño excedido");  
    }  
    data[size] = value;  
    size++;  
    return true;  
}
```



Lanza una excepción si se excede el tamaño

# Añadir Elementos a la Lista

- ¿Cómo añadir a la mitad o al final de la lista?
  - Desplaza los elementos hacia la derecha para dejar sitio para el nuevo valor

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

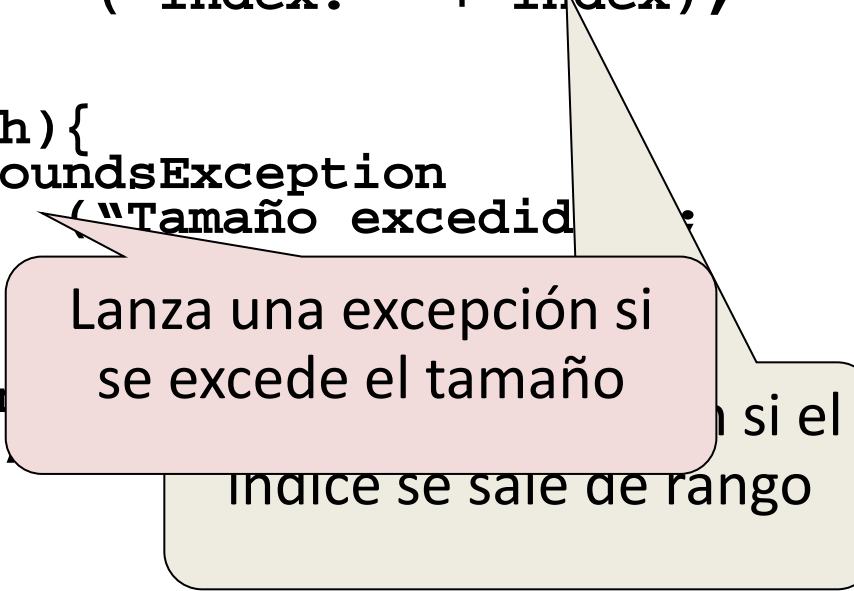
– **list.add(3, 42);**

**// inserta 42 en índice 3**

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	42	7	5	12	0	0	0
<i>size</i>	7									

# Añadir Elementos a la Lista

```
public void add(int index, int value) {  
    if (index < 0 || index > size) {  
        throw new IndexOutOfBoundsException  
            ("index: " + index);  
    }  
    if (size + 1 > data.length){  
        throw new IndexOutOfBoundsException  
            ("Tamaño excedido");  
    }  
    // Todo va bien  
    for (int i = size; i >= index; i--)  
        data[i] = data[i - 1];  
    data[index] = value;  
    size++;  
}
```



The diagram consists of two callout boxes. The first is a pink box with a pointer to the `throw new IndexOutOfBoundsException("index: " + index);` line, containing the text "Lanza una excepción si se excede el tamaño". The second is a light yellow box with a pointer to the `throw new IndexOutOfBoundsException("Tamaño excedido");` line, containing the text "Lanza una excepción si el índice se sale de rango".



# Eliminar Elementos de la Lista

- ¿Cómo podemos eliminar un elemento de la lista?
  - De nuevo debemos desplazar los elementos del array
  - Esta vez es un desplazamiento a la izquierda

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	6									

– **list.remove(2);**

// **borra 9 de índice 2**

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	7	5	12	0	0	0	0	0
<i>size</i>	<b>5</b>									

# Eliminar Elementos de la Lista

```
public void remove(int index) {  
    if (index < 0 || index > size)  
        throw new IndexOutOfBoundsException  
            ("index: " + index);
```

Lanza una excepción si el índice se sale de rango

```
    for (int i = index; i < size - 1; i++) {  
        data[i] = data[i + 1];  
    }  
    size--;  
}
```

# Implementación ListArray<T>

- Hasta aquí hemos discutido algunas operaciones básicas acerca de la manipulación de listas con arrays (de números enteros)
- Vamos ahora a implementar una clase genérica (ListArray<T>) que implemente el interfaz List<T> haciendo uso de arrays
- Seguimos realizando una implementación acotada

# Implementando ListArray<T>

```
public class ListArray<T> implements List<T>{  
  
    private static final int DEFAULT_CAPACITY=10;  
  
    //lista de datos  
    private T[] data;  
  
    // número actual de datos en la lista  
    private int size;  
  
    ...  
}
```

# Implementando ListArray<T> Constructores

```
public ListArray() {  
    data = (T[]) new Object[DEFAULT_CAPACITY];  
    size = 0;  
}
```

# Implementando ListArray<T>

## Constructores

- Es posible añadir un nuevo constructor que tenga un parámetro con la capacidad

```
public ListArray(int capacity) {  
    if (capacity < 0)  
        throw new IllegalArgumentException  
            ("capacity: " + capacity);  
    data = (T[]) new Object[capacity];  
    size = 0;  
}
```

- Ambos constructores son muy similares ¿podemos evitar la redundancia?

# Implementando ListArray<T>

## Constructores

```
public ListArray(int capacity) {  
    if (capacity < 0)  
        throw new IllegalArgumentException  
            ("capacity: " + capacity);  
    data = (T[]) new Object[capacity];  
    size = 0;  
}
```

```
public ListArray() {  
  
    // Llamar al constructor desde otro constructor  
    this(DEFAULT_CAPACITY);  
}
```

# Implementando ListArray<T>

boolean add(T value)

```
public boolean add(T value) {  
    if (size + 1 > data.length) {  
        throw new IndexOutOfBoundsException  
            ("tamaño excedido");  
    }  
  
    data[size] = value;  
    size++;  
    return true;  
}
```

Lanza una excepción si  
el nuevo dato no cabe  
en el array



# Implementando ListArray<T>

```
void add (int index, T value)
```

```
public void add(int index, T value) {  
    if (index < 0 || index > size) {  
        throw new IndexOutOfBoundsException  
            ("index: " + index);  
    }  
    if (size + 1 > data.length) {  
        throw new IndexOutOfBoundsException  
    }  
  
    // Todo va bien  
    for (int i = size; i >= index; i--)  
        data[i] = data[i - 1];  
    data[index] = value;  
    size++;  
}
```

Lanza una excepción si  
el nuevo dato no cabe  
en el array

si el  
índice se sale de rango

# Capacidad del Array

- Una lista del tipo `ArrayList<T>` de Java no se llena nunca
- En `ArrayList<T>` cuando el tamaño del array es insuficiente para almacenar todos los datos, se crea un array mayor, en lugar de lanzar una excepción
- Para que `ListArray<T>` tenga este mismo comportamiento tenemos que modificar la implementación anterior, para que la capacidad del array data crezca cuando no pueda almacenar más datos
  - Cuando el array se llena se **DOBLA** su capacidad

# Capacidad del Array

void ensureCapacity(int capacity)

```
public void ensureCapacity(int capacity){  
    if (capacity > data.length) {  
        int newCapacity = data.length * 2;  
        if (capacity > newCapacity) {  
            newCapacity = capacity;  
        }  
        data=Arrays.copyOf(data,newCapacity);  
    }  
}
```

# Implementando ListArray<T>

## boolean add(T value) (2)

```
public boolean add(T value) {  
    ensureCapacity(size + 1);  
    data[size] = value;  
    size++;  
    return true;  
}
```

# Implementando ListArray<T>

void add (int index, T value) (2)

```
public void add(int index, T value) {  
    if (index < 0 || index > size) {  
        throw new IndexOutOfBoundsException  
            ("index: " + index);  
    }  
    ensureCapacity(size + 1);  
    // Todo va bien  
    for (int i = size; i >= index + 1; i--){  
        data[i] = data[i - 1];  
    }  
    data[index] = value;  
    size++;  
}
```

# Implementando ListArray<T>

void remove (int index)

```
public T remove(int index) {  
    if (index < 0 || index > size) {  
        throw new IndexOutOfBoundsException  
            ("index: " + index);  
    }  
    T elem = data[index];  
    for (int i = index; i < size - 1; i++) {  
        data[i] = data[i + 1];  
    }  
    size--;  
    return elem;  
}
```

# Implementación ListArray<T>

boolean contains(Object element)

```
public boolean contains(Object element){  
    int i=0;  
    while ((i < size)&&(!(data[i].equals(element)))) {  
        i++;  
    }  
    return (i < size);  
}
```

Atención, la  
comparación de  
igualdad no se puede  
hacer con ==

# Implementación ListArray<T>

- Con el fin de completar la implementación de ListArray<T>, el alumno deberá implementar los siguientes métodos:

```
int size();  
boolean isEmpty();  
void clear();  
boolean remove(Object o);  
  
T get(int index);  
T set(int index, T element);  
int indexOf(Object value);  
int lastIndexOf(Object o);
```



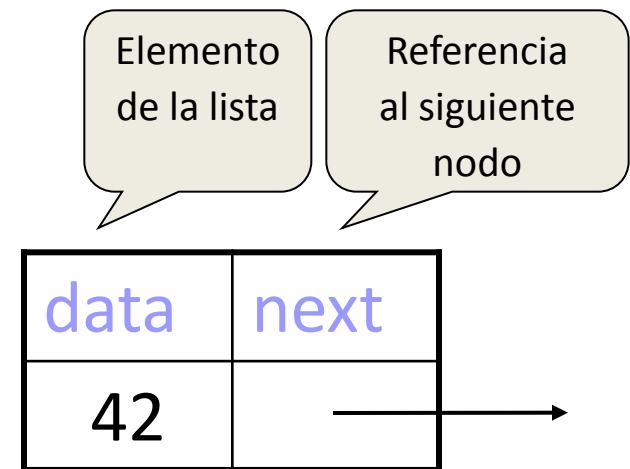
# Implementación con nodos enlazados

## Concepto de Nodo

- Las Listas Enlazadas están compuestas por bloques individuales denominados *Nodos*
- Cada nodo contiene un elemento de la lista
- Un nodo básico:

```
public class ListNode {  
    int data;  
    ListNode next;  
}
```

Estructura de  
datos  
recursiva

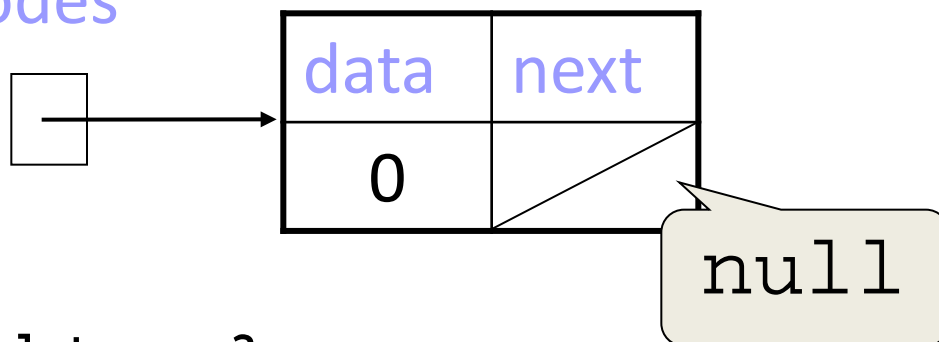


# Implementación con nodos enlazados

## Construyendo una Lista

```
ListNode lNodes = new ListNode();
```

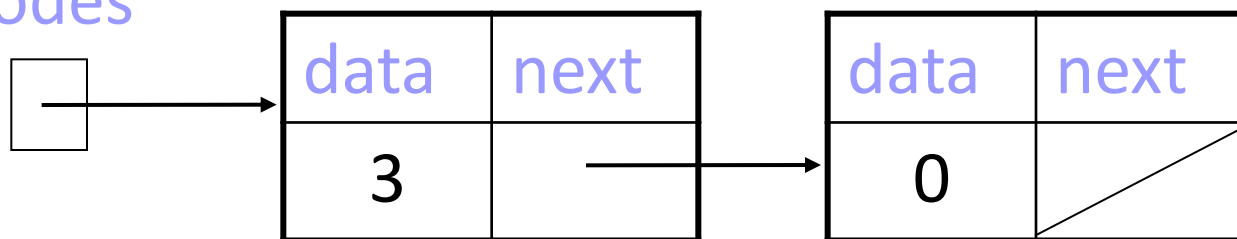
lNodes



```
lNodes.data = 3;
```

```
lNodes.next = new ListNode();
```

lNodes



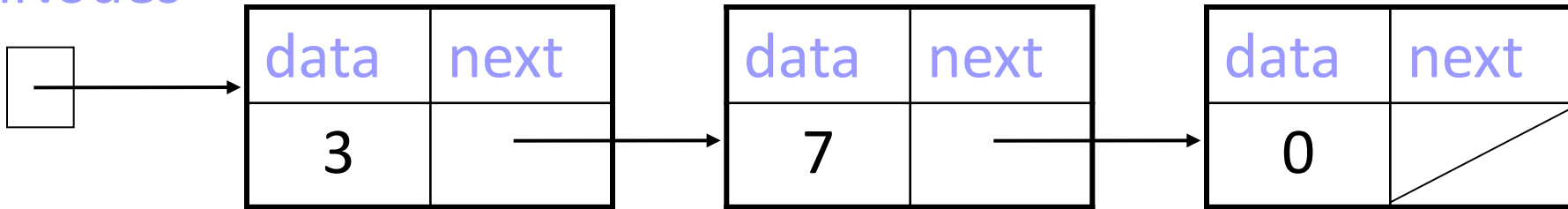
# Implementación con nodos enlazados

## Construyendo una Lista

```
lNodes.next.data = 7;
```

```
lNodes.next.next = new ListNode();
```

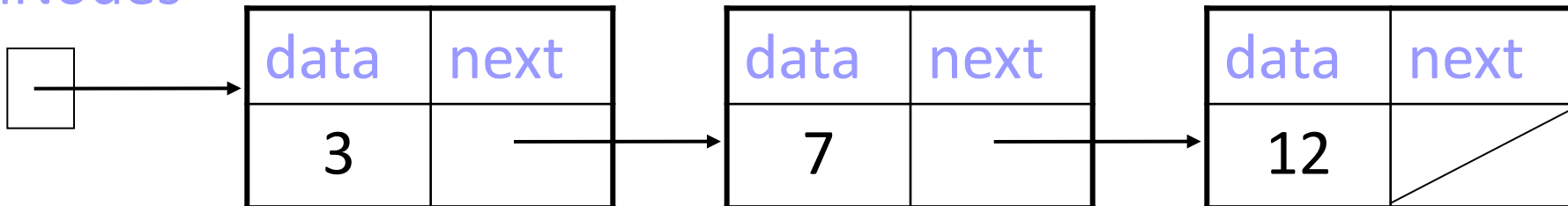
lNodes



```
lNodes.next.next.data = 12;
```

```
lNodes.next.next.next = null; // no es necesario
```

lNodes



# Implementación con nodos enlazados

## Construyendo una Lista

```
public class ConstructList1 {  
    public static void main(String[] args) {  
        ListNode lNodes = new ListNode();  
        lNodes.data = 3;  
        lNodes.next = new ListNode();  
        lNodes.next.data = 7;  
        lNodes.next.next = new ListNode();  
        lNodes.next.next.data = 12;  
        lNodes.next.next.next = null;  
        System.out.println(lNodes.data + " "  
                           + lNodes.next.data + " "  
                           + lNodes.next.next.data);  
    }  
}
```

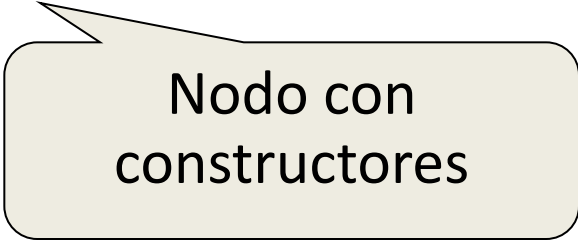
Tedioso e  
impracticable

3 7 12

# Implementación con nodos enlazados

## Construyendo una Lista

```
public class ListNode {  
    int data;  
    ListNode next;  
  
    public ListNode(int data) {  
        this(data,null);  
    }  
  
    public ListNode(int data, ListNode next) {  
        this.data = data;  
        this.next = next;  
    }  
}
```



Nodo con constructores

# Implementación con nodos enlazados

## Construyendo una Lista

```
public class ConstructList2 {  
    public static void main(String[] args) {  
        ListNode lNodes = new ListNode(3,  
            new ListNode(7,  
                new ListNode(12)));  
  
        System.out.println(lNodes.data + " "  
            + lNodes.next.data + " "  
            + lNodes.next.next.data);  
    }  
}
```

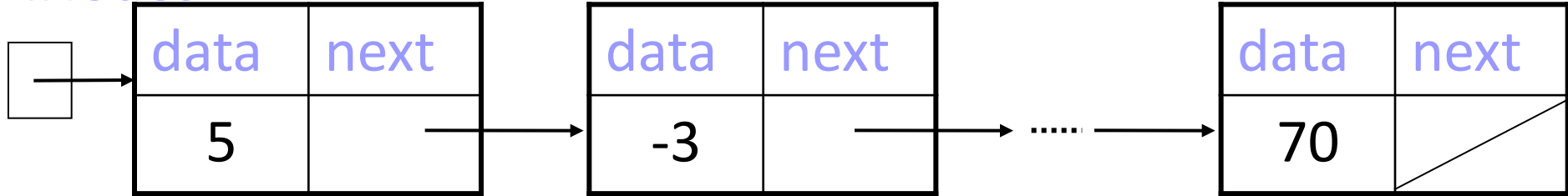
Menos tedioso  
pero  
impracticable

Necesidad de usar Bucles  
(o Recursividad) para  
manipular Listas Enlazadas

# Implementación con nodos enlazados

## Recorrido de una Lista (por ej. para imprimir)

lNodes



```
while (lNodes != null) {  
    System.out.println(lNodes.data);  
    lNodes = lNodes.next;  
    // avanzar al siguiente nodo  
}
```



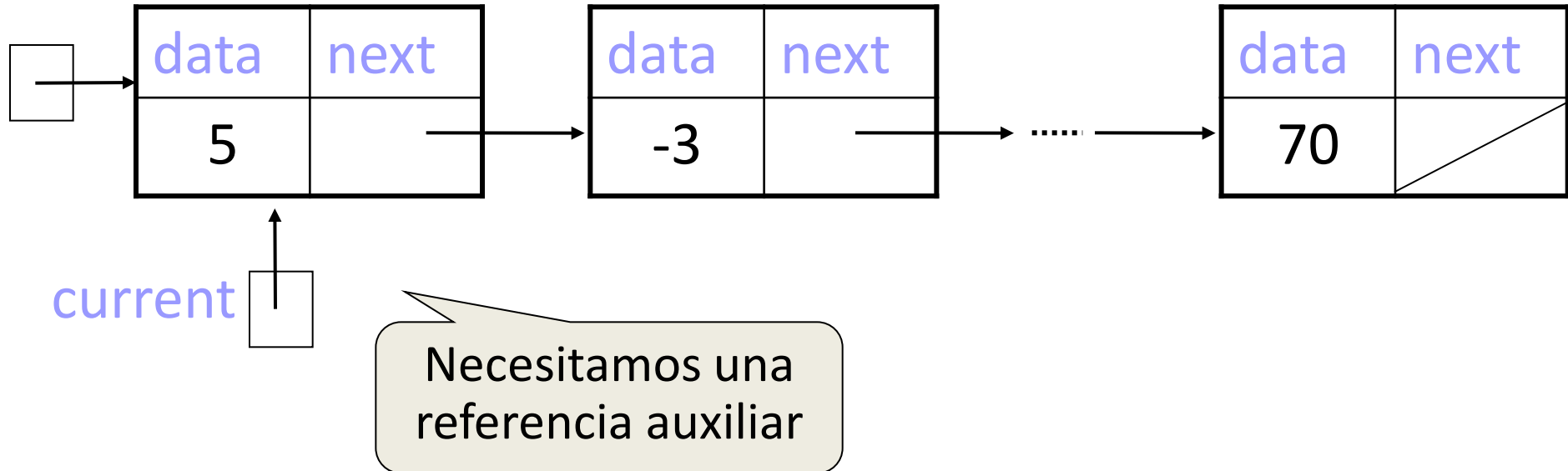
“Perdemos” la lista tras el recorrido

`lNodes == null`

# Implementación con nodos enlazados

Recorrido de una Lista (por ej. para imprimir)

lNodes



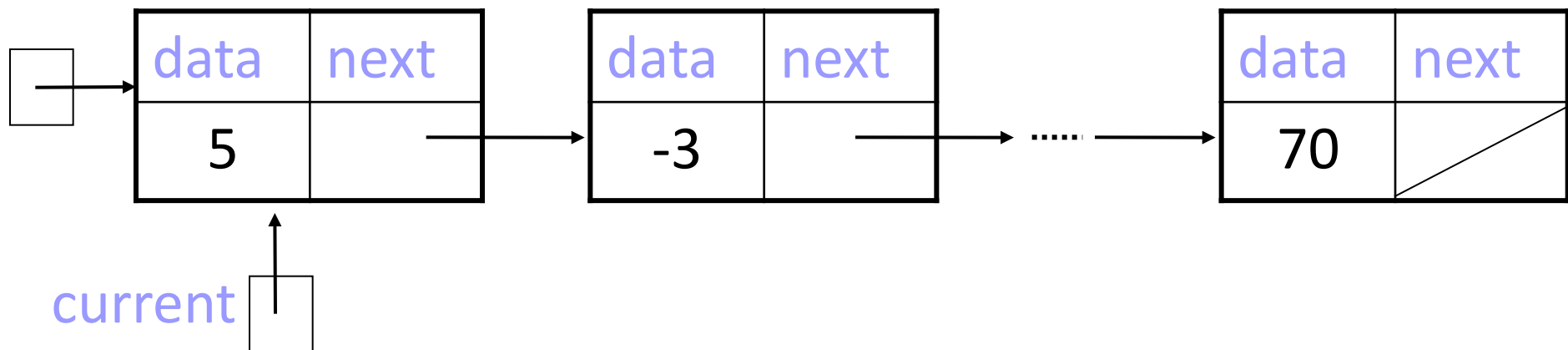
```
ListNode current = lNodes;  
while (current != null) {  
    System.out.println(current.data);  
    current = current.next; // avanzar  
}
```



# Implementación con nodos enlazados

## Recorrido Condicional de una Lista (por ej. para buscar un elemento)

lNodes

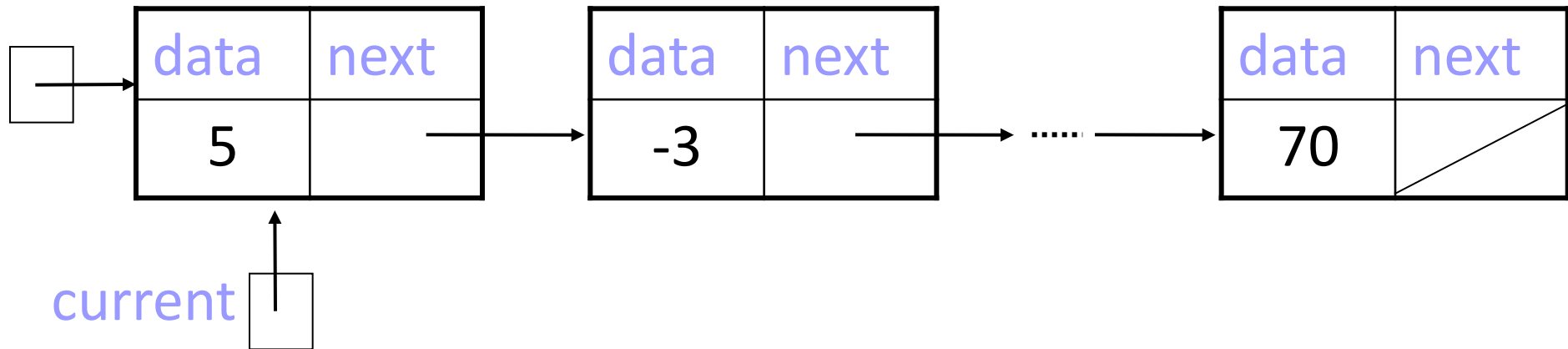


```
ListNode current = lNodes;  
while ((current != null) && (current.data != elem)){  
    current = current.next; // avanzar  
}  
// PD: si current!=null apunta al nodo con elem
```

# Implementación con nodos enlazados

## Recorrido por Posición de una Lista

listNodes



```
ListNode current = lNodes;  
for (int i = 0; i < pos; i++) {  
    current = current.next; // avanzar  
}  
// PD: current apunta al nodo deseado
```

# Implementación con nodos enlazados

## Lista Vacía

```
ListNode lNodes = null;
```

lNodes

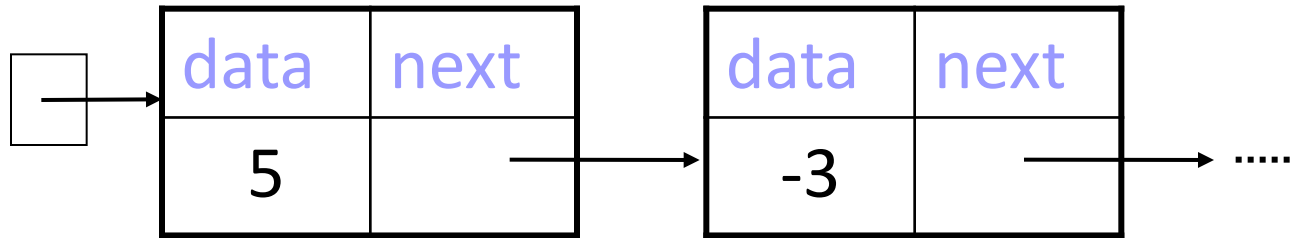


# Implementación con nodos enlazados

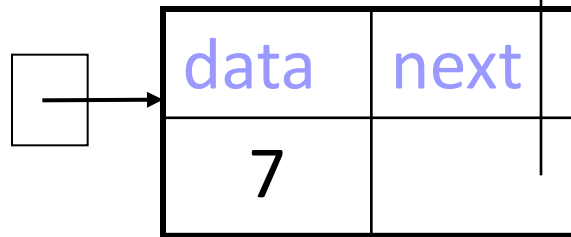
Añadir un elemento a una Lista

Añadir al principio

lNodes



newNode



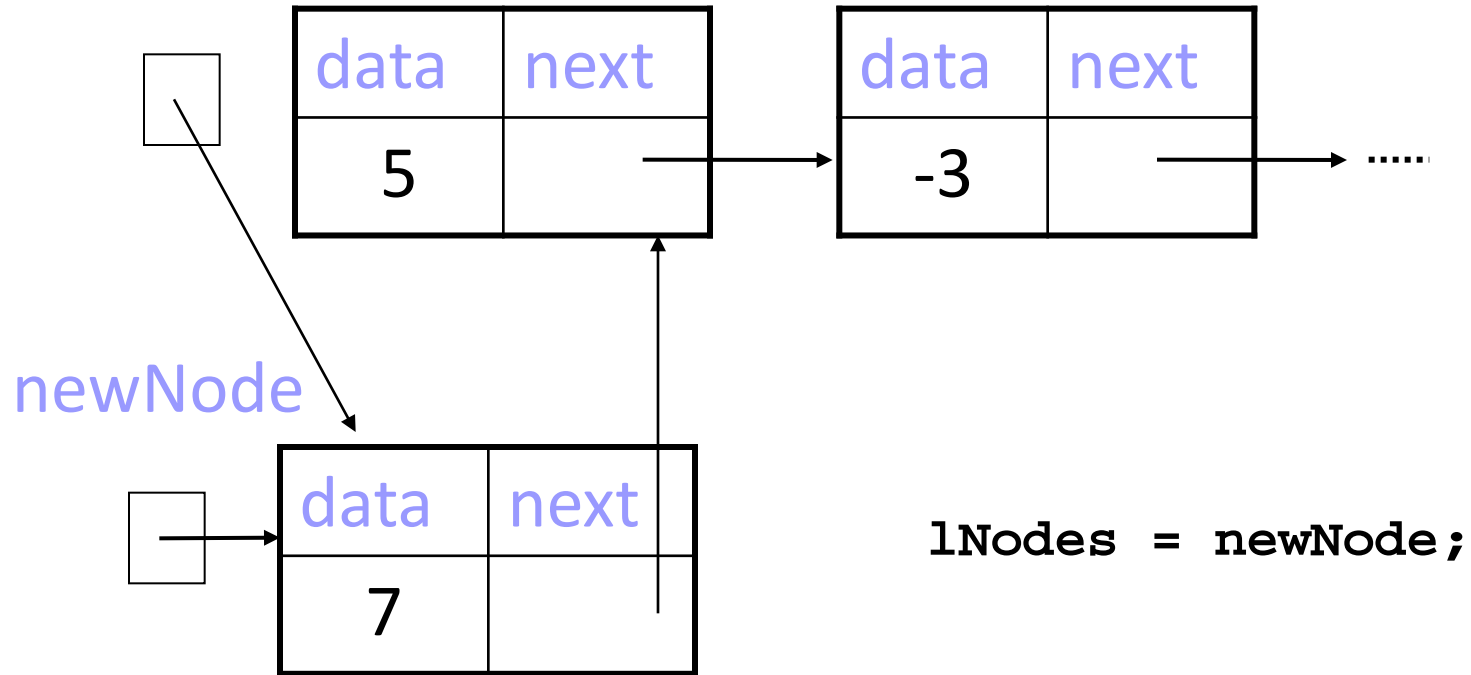
```
ListNode newNode =  
    new ListNode(7, lNodes);
```

# Implementación con nodos enlazados

Añadir un elemento a una Lista

Añadir al principio

lNodes



`lNodes = newNode;`

# Implementación con nodos enlazados

Añadir un elemento a una Lista

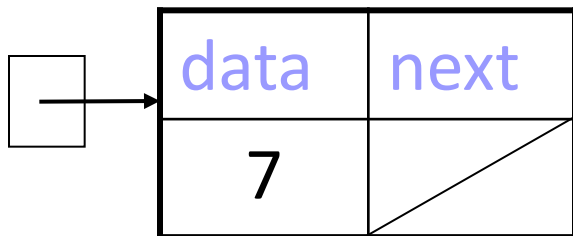
Añadir al principio  
(igual si la lista está vacía)

lNodes



```
ListNode newNode =  
    new ListNode(7, lNodes);
```

newNode



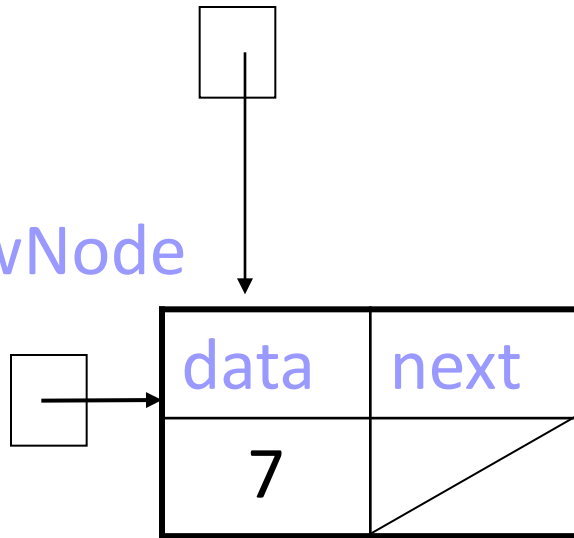
# Implementación con nodos enlazados

Añadir un elemento a una Lista

Añadir al principio  
(igual si la lista está vacía)

lNodes

newNode



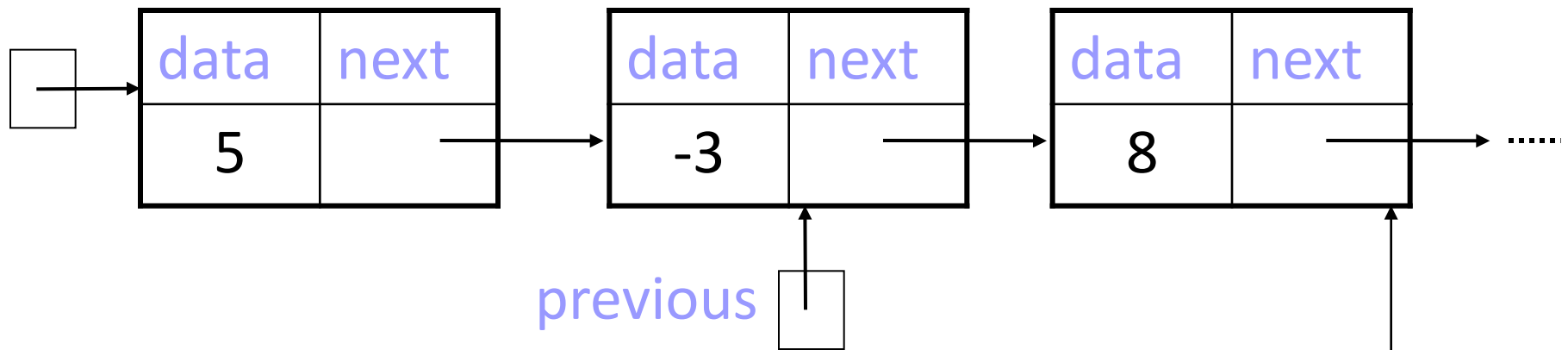
```
lNodes = newNode;
```

# Implementación con nodos enlazados

Añadir un elemento a una Lista

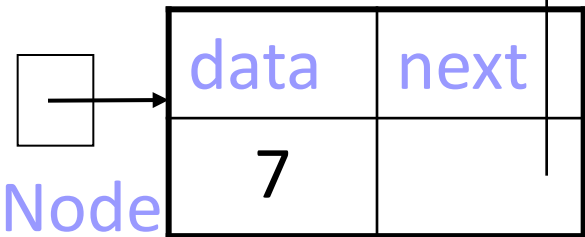
Añadir en medio (o al final)

INodes



```
ListNode newNode =  
    new ListNode(7,  
        previous.next);
```

newNode



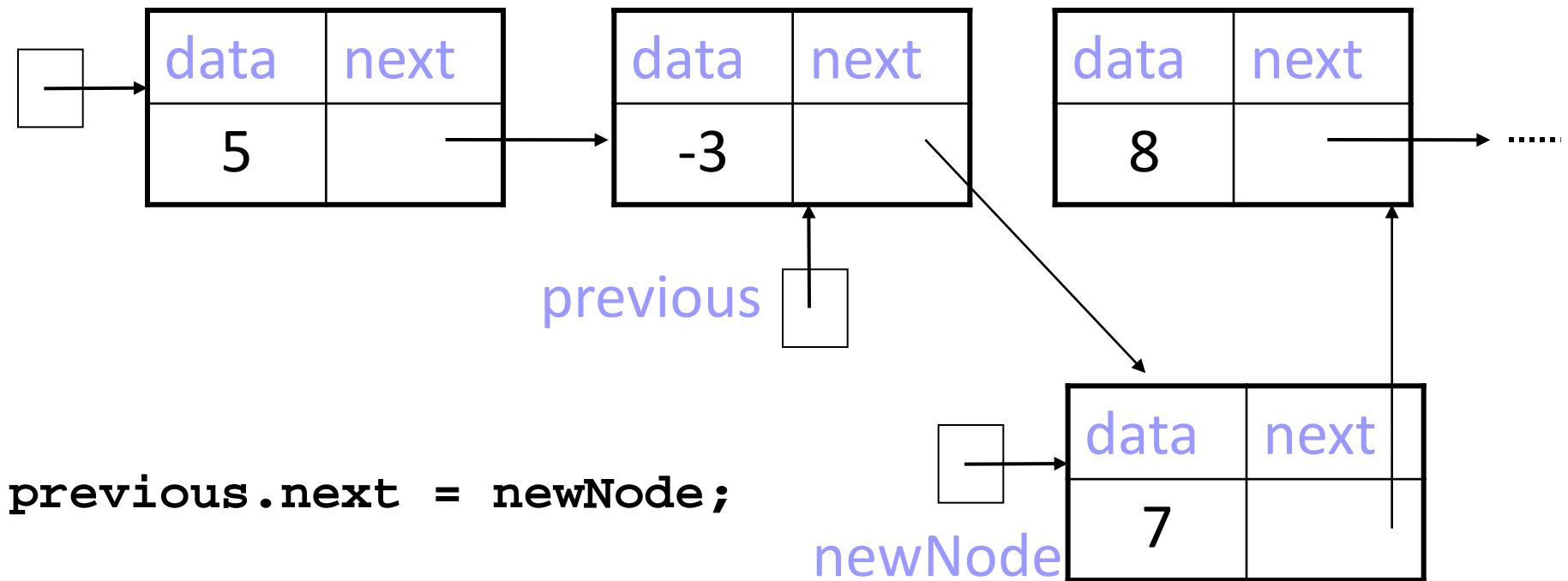


# Implementación con nodos enlazados

Añadir un elemento a una Lista

Añadir en medio (o al final)

INodes

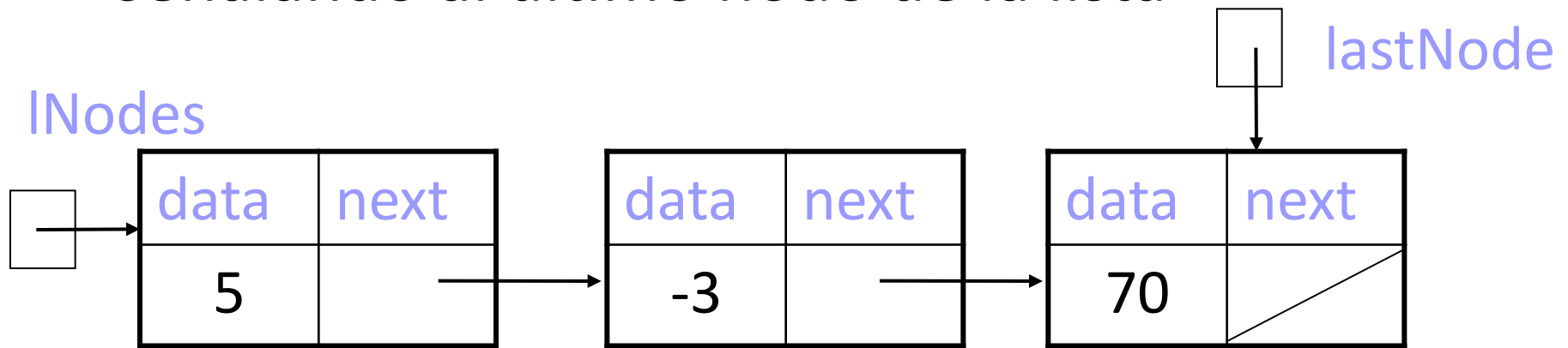


# Implementación con nodos enlazados

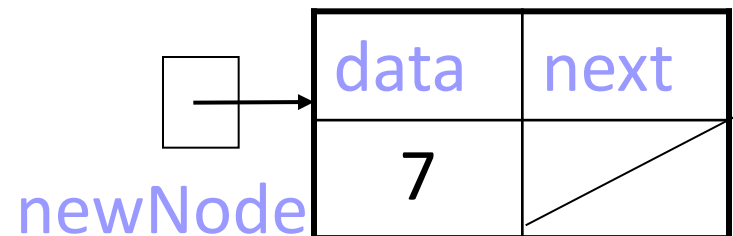
## Añadir un elemento a una Lista

### Añadir al final (de forma más eficiente)

- Utilizar una referencia auxiliar que siempre esté señalando al último nodo de la lista



```
ListNode newNode =  
    new ListNode(7, null);
```



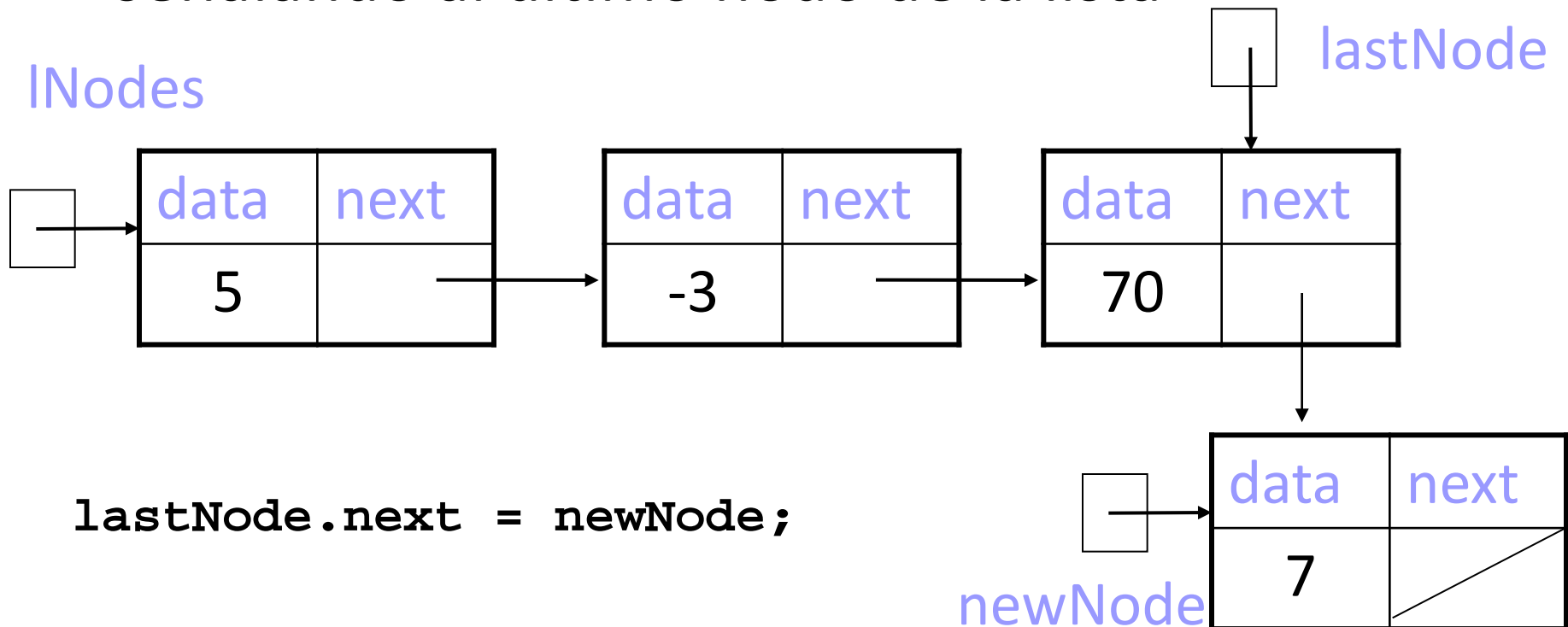
# Implementación con nodos enlazados

## Añadir un elemento a una Lista

### Añadir al final (de forma más eficiente)

- Utilizar una referencia auxiliar que siempre esté señalando al último nodo de la lista

INodes

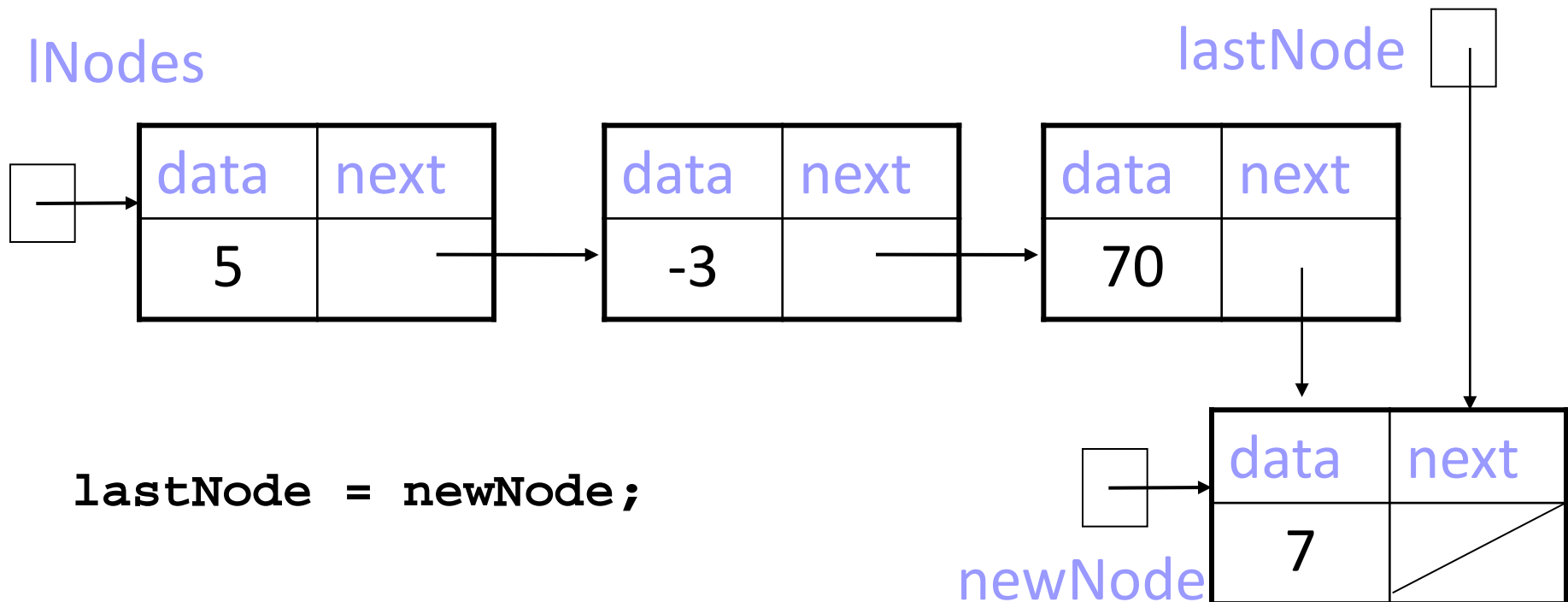


# Implementación con nodos enlazados

## Añadir un elemento a una Lista

### Añadir al final (de forma más eficiente)

- Utilizar una referencia auxiliar que siempre esté señalando al último nodo de la lista

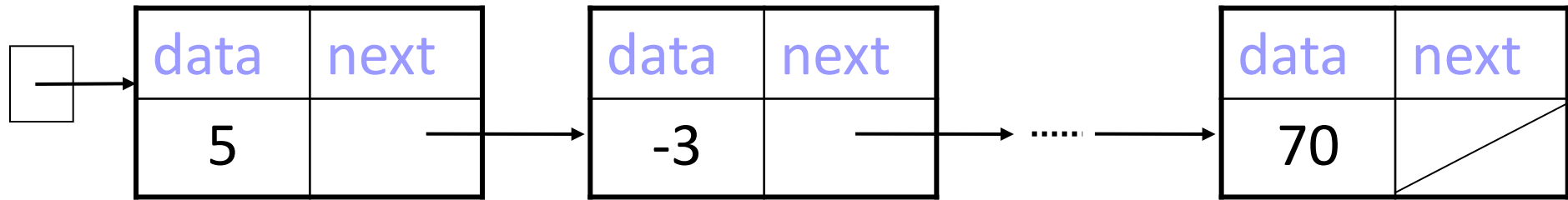


# Implementación con nodos enlazados

Eliminar un elemento de una Lista (no vacía)

Eliminar el primero

INodes

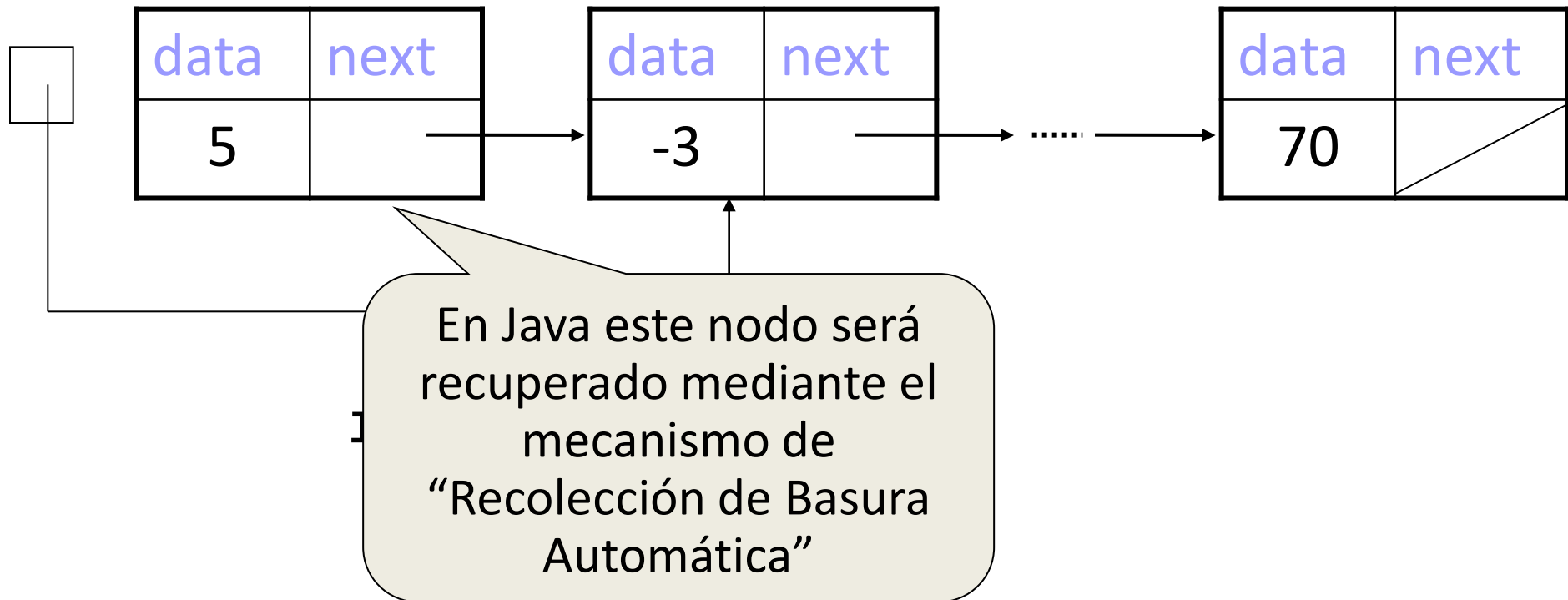


# Implementación con nodos enlazados

Eliminar un elemento de una Lista (no vacía)

Eliminar el primero

INodes

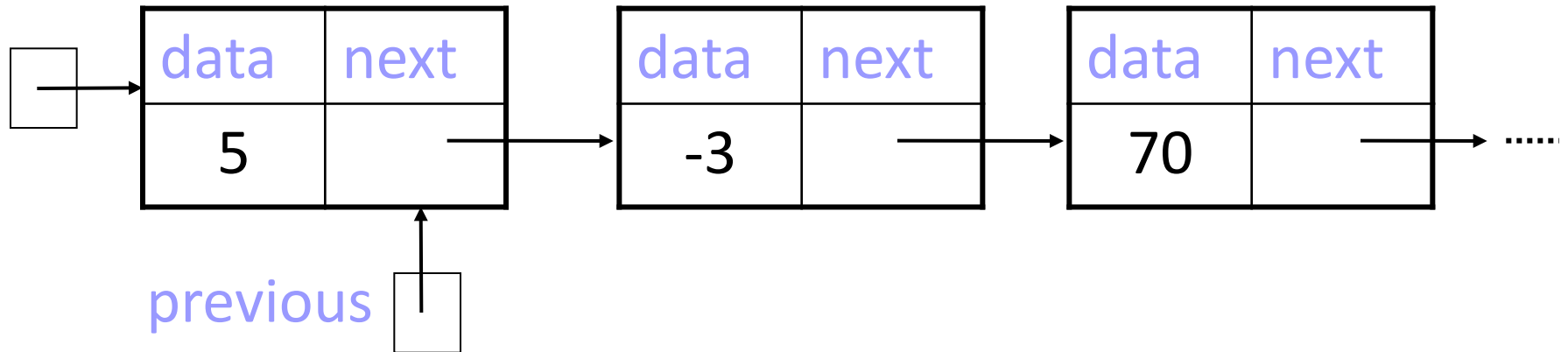


# Implementación con nodos enlazados

Eliminar un elemento de una Lista (no vacía)

Eliminar en medio (o al final)

INodes

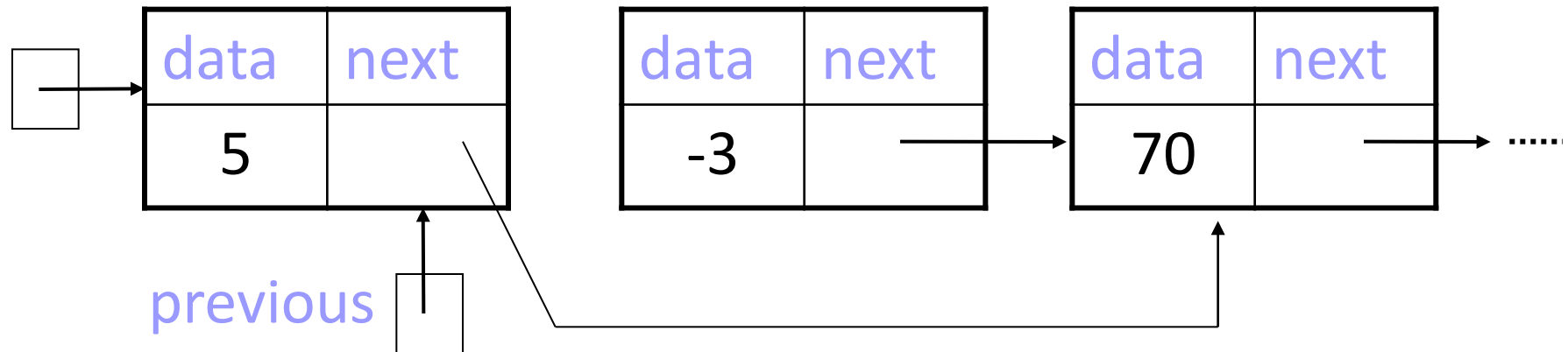


# Implementación con nodos enlazados

Eliminar un elemento de una Lista (no vacía)

Eliminar en medio (o al final)

listNodes



```
previous.next = previous.next.next;
```



# Implementación con nodos enlazados

## Listas Enlazadas vs. Arrays

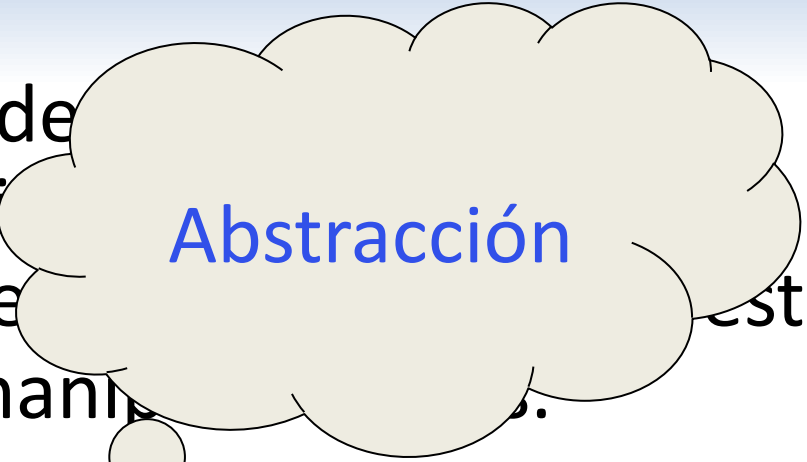
- Arrays:
  - + Acceso Aleatorio: acceso rápido a los elementos
  - Desplazamiento de datos al insertar o extraer
  - No puede crecer ni menguar su tamaño
- Listas Enlazadas:
  - Acceso Secuencial: acceso lento a los elementos
  - + No desplazamiento de datos al insertar o extraer
  - + Su tamaño puede crecer y menguar fácilmente

# Implementación ListLinked<T>

- Hasta aquí hemos discutido los conceptos básicos sobre la manipulación de listas enlazadas (de números enteros)
- Vamos ahora a analizar cómo se podría diseñar una clase genérica (ListLinked<T>) que implemente el interfaz List<T> mediante el uso de listas enlazadas

# Implementación ListLinked<T>

- La clase ListArray<T> de ofrecía al potencial cliente manipular una lista de este tenga necesidad de manipular.
- De igual forma, la clase ListLinked<T> ofrecerá al cliente una clase para trabajar con listas sin que éste tenga que conocer el manejo (detalles de “bajo nivel”) de listas enlazadas



Abstracción

# Implementación ListLinked<T>

```
public class ListLinked<T> implements List<T>{  
  
    // número actual de datos en la lista  
    private int size;  
  
    // primer nodo de la lista  
    private ListNode<T> front;  
  
    // último nodo de la lista  
    private ListNode<T> back;
```

# Implementación ListLinked<T>

```
// clase interna estática para los nodos
```

```
static class ListNode<D> {  
    D data;  
    ListNode<D> next;  
  
    public ListNode(D data) {  
        this(data,null);  
    }  
  
    public ListNode(D data, ListNode<D> next) {  
        this.data = data;  
        this.next = next;  
    }  
}
```

# Implementación ListLinked<T>

```
// constructores

// lista vacía
public ListLinked() {
    size = 0;
    front = null;
    back = null;
}

// construir lista con los elementos de otra
// (puede ser tanto ListArray como ListLinked
public ListLinked(List<T> list) {
    this();
    // se necesita un iterador para este for
    for (T elem : list) {
        add(elem);
    }
}
```

# Implementación ListLinked<T>

- Con el fin de completar la implementación de ListLinked<T>, el alumno deberá implementar todos los métodos necesarios, al igual que se hizo con ListArray<T>

# La clase Iterador en ListArray

- En el marco colecciones es útil usar un iterador para recorrer una colección.
- Por ejemplo, es necesario para utilizar la instrucción **foreach**

```
for (T elem : list){...}
```

- Para que una clase sea iterable debe implementar la interfaz genérica `Iterable<T>`. Como la interfaz `List<T>` extiende `Collection<T>`, y `Collection<T>` extiende `Iterable<T>`, `ListArray<T>` también es `Iterable<T>`.

```
public interface Collection<T> extends Iterable<T> {  
    ...  
}  
public interface List<T> extends Collection<T>{  
    ...  
}  
  
public class ListArray<T> implements List<T>{  
    ...  
}
```

- La interfaz `Iterable` obliga a implementar el método  
`Iterator<T> iterator()`



# La clase Iterador en ListArray

- Una clase `Iterator<T>` debe proporcionar tres operaciones básicas:
  - `hasNext()`, devuelve cierto si hay más elementos en la lista
  - `next()`, devuelve el siguiente elemento de la lista y avanza la posición del iterador en uno
  - `remove()`, elimina el elemento más recientemente devuelto por `next()`

# La clase `Iterador` `IterListArray`

- Desarrollaremos una clase llamada `IterListArray` que implemente esta funcionalidad para `ListArray`
- Esta clase implementará la interfaz `iterator<T>`
- La clase `IterListArray`, es una clase interna de `ListArray` ya que necesita acceder a algunos de sus métodos y atributos

# La clase Iterator

## IterListArray

- La función principal del iterador es mantener una traza de una posición particular en una lista, por lo tanto será necesario un atributo, `position` de tipo `int`, para almacenar esta información
- Un iterador es un objeto de la clase `IterListArray`, que es una clase interna dentro de `ListArray<T>`
- `Iterator<T>` es una interfaz genérica que se encuentra en el paquete `java.util`

```
private class IterListArray implements Iterator<T>{  
    private int position;  
    ...  
}
```

# La clase Iterador

## IterListArray

- Constructor
  - Inicialmente `position` se inicializa a 0

```
public IterListArray() {  
    position = 0;  
}
```

# La clase Iterador

## IterListArray

- El método `hasNext()` ha de determinar si todavía quedan en la lista valores sobre los que iterar
- Preguntamos si existe algún elemento más en la lista

```
public boolean hasNext() {  
    return position < size()-1;  
}
```

Usa el metodo `size()`  
de `ListArray<T>`

# La clase Iterador

## IterListArray

- El método `next ( )` devuelve el siguiente valor de la lista, e incrementa el valor de `position` una unidad.
- El método lanza una excepción en caso de que el iterador este fuera del rango de valores a devolver

```
public T next() {  
    if (!hasNext()) {  
        throw new NoSuchElementException();  
    }  
    T result = data[position];  
    position++;  
    return result;  
}
```

Usa el array `data` de  
`ListArray<T>`

# La clase Iterador

- Para poder usar un iterador desde la clase `ListArray<T>`, necesitamos un método llamado `iterator` que construya dicho iterador

```
public Iterator<T> iterator() {  
    return new IterListArray();  
}
```

# La clase Iterador en ListArray

```
public class ListArray<T> implements List<T>{
    .....
    private T[] data;
    private int size;

    ....
    public Iterator<T> iterator(){
        return new IterListArray();
    }

    private class IterListArray implements Iterator<T>{
        private int position;

        public boolean hasNext(){
            return position < size()-1;
        }
        public T next(){
            if (!hasNext()) {
                throw new NoSuchElementException();
            }

            T result = data[position];
            position++;
            return result;
        }
        public void remove{
            throw new UnsupportedOperationException();
        }
    }
}
```



# La clase Iterador en ListArray

- Una vez que una clase tiene implementada la interfaz iterable, es posible utilizar foreach para recorrer sus elementos:

```
List<Integer> l = new ListArray<Integer>();  
for (int i = 0; i<l.size(); i++)  
    l.add(i);
```

```
for (int i:l)  
    System.out.println(i);
```

```
Iterator<Integer> it = l.iterator();  
while (it.hasNext())  
    System.out.println(it.next());
```



Códigos equivalentes

# La clase Iterador en ListLinked

- Para que la clase ListLinked<T> sea iterable debe implementar la interfaz genérica Iterable<T>, que obliga a implementar el método **iterator()**. Esto es así porque
  - ListLinked<T> implements List<T>
  - List<T> extends Collection<T>
  - Collection<T> extends Iterable<T>

```
import java.util.*;
public class ListLinked<T> implements List<T>{
    ...
    // primer nodo de la lista
    private ListNode<T> front;
    ...
    public Iterator<T> iterator(){
        return new IterListLinked();
    }
    ...
}
```

# La clase Iterador en ListLinked

```
public class ListLinked<T> implements List<T>{
    ...
    // primer nodo de la lista
    private ListNode<T> front;
    ...
    public Iterator<T> iterator(){
        return new IterListLinked();
    }

    private class IterListLinked implements Iterator<T>{
        ListNode<T> iter = front;

        public boolean hasNext(){
            return iter!=null;
        }
        public T next(){
            if (!hasNext()) throw new NoSuchElementException();
            T elem = iter.data;
            iter = iter.next;
            return elem;
        }
        public void remove{
            throw new UnsupportedOperationException();
        }
    }
}
```

# La clase Iterador en ListLinked

- Como en el caso de ListArray, se puede utilizar la instrucción foreach. Observa que el ejemplo es igual que el que se usó para ListArray, salvo que ahora se crea un objeto de la clase ListLinked.

```
List<Integer> l = new ListLinked<Integer>();  
for (int i = 0; i<l.size(); i++)  
    l.add(i);
```

```
for (int i:l)  
    System.out.println(i);
```

```
Iterator<Integer> it = l.iterator();  
while (it.hasNext())  
    System.out.println(it.next());
```



Códigos equivalentes