Listas enlazadas e iteradores

Algoritmos y Estructuras de Datos

Departamento de Computación, FCEyN, UBA

3 de octubre de 2024

Repaso: clases

Nos interesa la abstracción (énfasis en el *qué*, no en el *cómo*) y el encapsulamiento (ocultar detalles de implementación).

Repaso: clases

Nos interesa la abstracción (énfasis en el *qué*, no en el *cómo*) y el encapsulamiento (ocultar detalles de implementación).
Para ello, utilizamos **clases**:

- *Atributos privados*: representan el estado de un objeto, y son accesibles sólo desde la propia clase (encapsulamiento).
- *Métodos públicos*: representan el comportamiento de un objeto; definen su **interfaz** (abstracción).

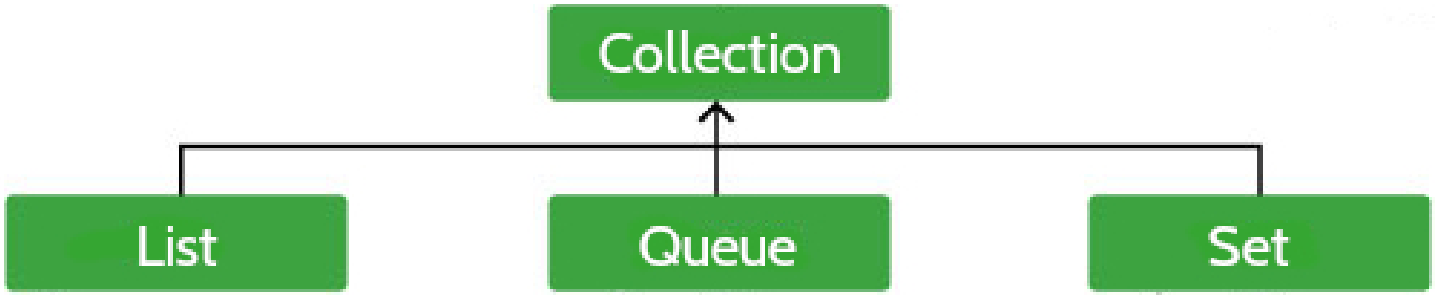
Interfaces

- Una **interfaz** es un contrato que define un conjunto de métodos que una clase debe implementar.
- No contiene implementaciones de los métodos.
- Permite definir un comportamiento común a distintas clases.

Interfaz colección

Preguntas típicas sobre colecciones (grupo de objetos):

- Dado un elemento, ¿está en la colección?
- Agregar un elemento a la colección
- Obtener el tamaño de la colección
- *etc.*



Interfaz colección

Preguntas típicas sobre colecciones (grupos de objetos):

- Dado un elemento, ¿está en la colección?
- Agregar un elemento a la colección
- Obtener el tamaño de la colección
- *etc.*

```
public interface Collection<T> {
    boolean contains(T elem);
    boolean add(T elem);
    int size();
    ...
};
```

Paréntesis: tipos paramétricos

```
public interface Collection<T> {...}
```

¿Qué tipo de dato es `T`?

Paréntesis: tipos paramétricos

```
public interface Collection<T> {...}
```

¿Qué tipo de dato es `T`?
Se los llama **tipo paramétrico:** constituye una *variables de tipo*.

- Es decir, `T` puede tomar como valor cualquier tipo.
- Nos permiten definir una interfaz *genérica*, que puede ser implementada por distintos tipos de datos.

Paréntesis: tipos paramétricos

```
public interface Collection<T> {...}
```

¿Qué tipo de dato es `T`?
Se los llama **tipo paramétrico:** constituye una *variables de tipo*.

- Es decir, `T` puede tomar como valor cualquier tipo.
- Nos permiten definir una interfaz *genérica*, que puede ser implementada por distintos tipos de datos.
- No obstante, si nuestra implementación requiere de un orden, entonces `T` debe ser *comparable* (i.e., definir una relación de orden total).
  - Esto se logra a través de la interfaz `Comparable<T>`, *sobrecargando* el método `CompareTo(T otro)`.

Volvemos: interfaz lista

Similar al TAD secuencia, hay una interfaz para lista que *extiende* a la interfaz de colección:

```
public interface List<T> extends Collection<T> {
    void addFirst(T elem); // agregar adelante
    void addLast(T elem); // agregar atrás
    int indexOf(T elem); // obtener índice de elemento
    ...
};
```

Implementaciones de lista

Las interfaces permiten definir comportamiento común a distintas clases.
Es decir, si tenemos dos posibles implementaciones de una lista, ambas deben tener los mismos métodos.

```
public interface List<T> extends Collection<T> {...}
public class ArrayList<T> implements List<T> {...} //
↪  arreglo redimensionable
public class LinkedList<T> implements List<T> {...} //
↪  lista enlazada
```

Implementaciones de lista

Las interfaces permiten definir comportamiento común a distintas clases.
Es decir, si tenemos dos posibles implementaciones de una lista, ambas deben tener los mismos métodos.

```
public interface List<T> extends Collection<T> {...}
public class ArrayList<T> implements List<T> {...} //
↪  arreglo redimensionable
public class LinkedList<T> implements List<T> {...} //
↪  lista enlazada
```

¿Por qué querríamos más de una implementación para lo mismo?

Implementaciones de lista

Distintas implementaciones proveen distintas **garantías:**
"[...] La operación *add* se ejecuta en tiempo constante amortizado, es decir, agregar n elementos requiere tiempo $O(n)$. Todas las otras operaciones se ejecutan en tiempo lineal [...]."

–ArrayList, Java 8

Implementaciones de lista

Distintas implementaciones proveen distintas **garantías:**
"[...] La operación *add* se ejecuta en tiempo constante amortizado, es decir, agregar n elementos requiere tiempo $O(n)$. Todas las otras operaciones se ejecutan en tiempo lineal [...]."

–ArrayList, Java 8

¿Cómo sé qué garantías me provee una clase? Voy a su documentación:
`https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`

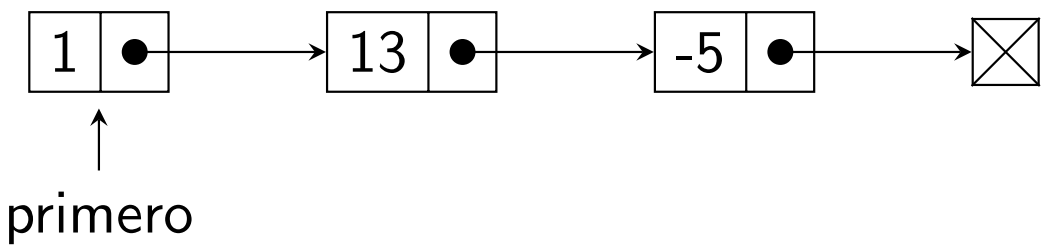
Listas simplemente enlazadas

Listas simplemente enlazadas

Una *lista simplemente enlazada* es una estructura que sirve para representar una secuencia de elementos, distinta del arreglo redimensionable.

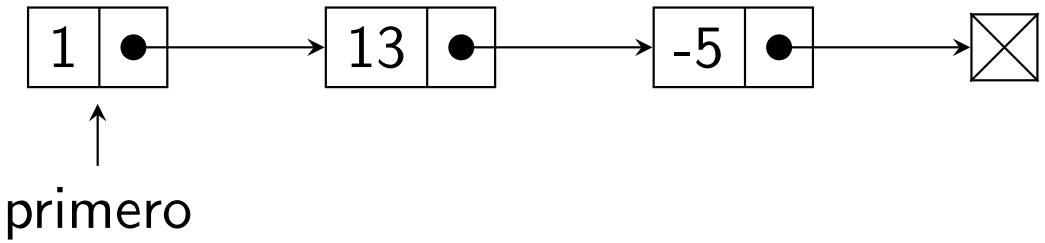
```
public interface List<T> extends Collection<T> {...}
public class ArrayList<T> implements List<T> {...}
public class LinkedList<T> implements List<T> {...}
```

Gráficamente



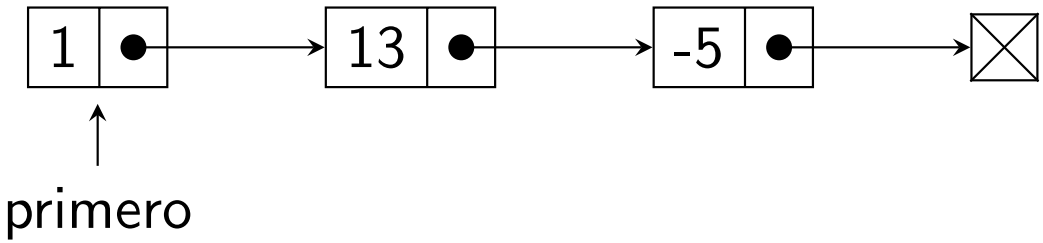
Cada elemento de la secuencia se representa mediante un *nodo*, que contiene un elemento y una referencia al siguiente nodo.

Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Qué propiedades deben cumplir estas variables para que la estructura sea válida?

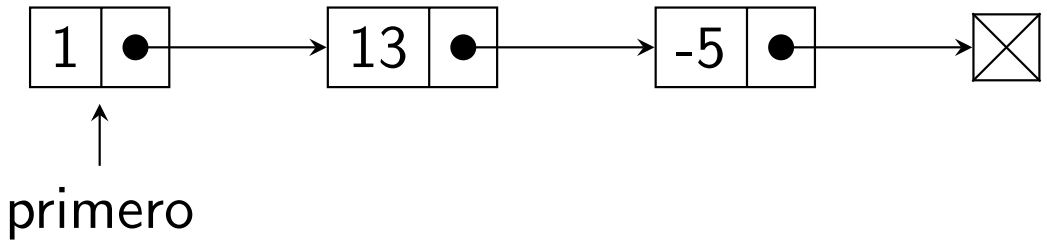
Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Qué propiedades deben cumplir estas variables para que la estructura sea válida?

1. Si la lista está vacía, entonces `primero` es `null` y `size` vale 0.

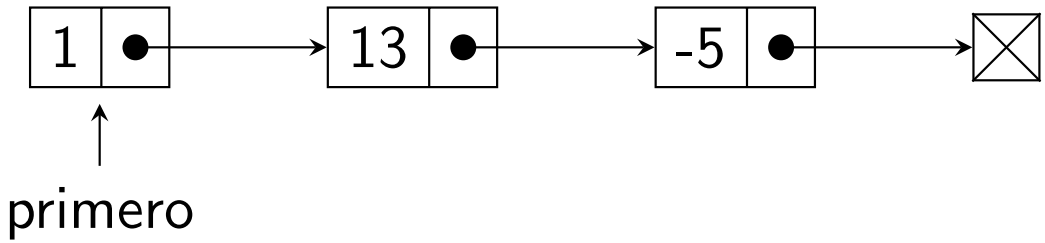
Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Qué propiedades deben cumplir estas variables para que la estructura sea válida?

1. Si la lista está vacía, entonces `primero` es `null` y `size` vale 0.
2. Si la lista no está vacía, entonces `primero` apunta al primer nodo de la lista y `size` es la cantidad de nodos.

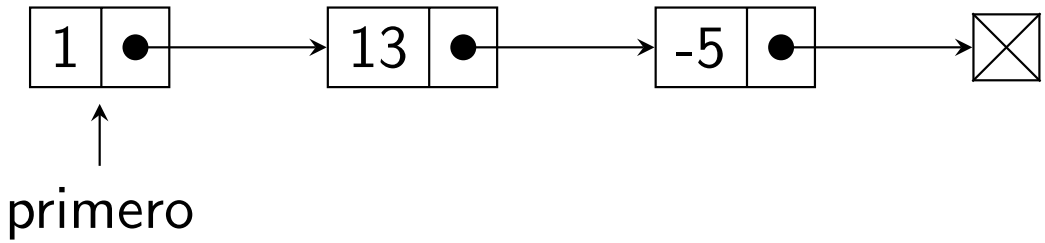
Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Qué propiedades deben cumplir estas variables para que la estructura sea válida?

1. Si la lista está vacía, entonces `primero` es `null` y `size` vale 0.
2. Si la lista no está vacía, entonces `primero` apunta al primer nodo de la lista y `size` es la cantidad de nodos.
3. Todos los nodos de la lista apuntan al siguiente, excepto el último.

Lista simplemente enlazadas



Asumiendo que tenemos una referencia al primer elemento (`primero`) y una variable `size`.
¿Qué propiedades deben cumplir estas variables para que la estructura sea válida?

1. Si la lista está vacía, entonces `primero` es `null` y `size` vale 0.
2. Si la lista no está vacía, entonces `primero` apunta al primer nodo de la lista y `size` es la cantidad de nodos.
3. Todos los nodos de la lista apuntan al siguiente, excepto el último.
4. El último nodo apunta a `null`.

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

Listas simplemente enlazadas

Sus principales características son:

- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

¿Cuál es su desventaja?

Listas simplemente enlazadas

Sus principales características son:

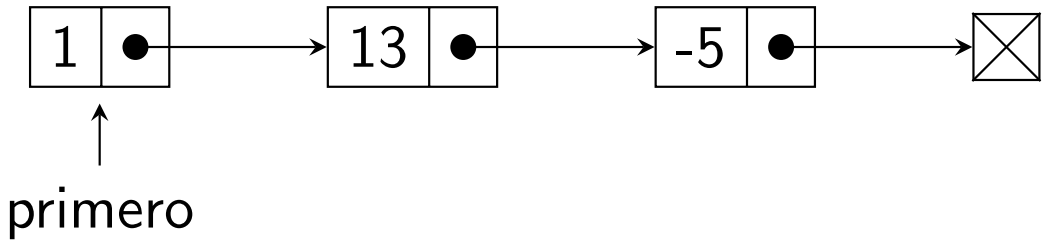
- Permiten un manejo más fino del uso de memoria (no es necesario reservar memoria por adelantado).
- Permiten insertar al principio (y potencialmente al final) de forma eficiente.
- Son eficientes para reacomodar elementos (útil para ordenar).

¿Cuál es su desventaja?
Perdemos el *acceso aleatorio* a los elementos.

Secuencia de Enteros

```
interface SecuenciaDeInts {
    /** Devuelve el largo de la secuencia. */
    public int longitud();

    /** Agrega un elemento al principio de la secuencia. */
    public void agregarAdelante(int elem);

    /** Agrega un elemento al final de la secuencia. */
    public void agregarAtras(int elem);

    /** Retorna el elemento en la i-esima posicion. */
    public T obtener(int indice);

    /** Elimina el elemento en la i-esima posicion de la
    ↪  secuencia. */
    public void eliminar(int indice);
}
```

Lista de Enteros



Implementemos la clase `ListaDeInts`, sobre una lista simplemente enlazada, con los siguientes métodos:

```
class ListaDeInts implements SecuenciaDeInts {
    private ...

    ListaDeInts();
    ListaDeInts(ListaDeInts otro);
    void agregarAtras(int elem);
    void agregarAdelante(int elem);
    void eliminar(int indice);
    ...
}
```

Lista de Enteros: estructura y constructores

```
class ListaDeInts implements SecuenciaDeInts {
    private Nodo primero;

    private class Nodo {
        int valor;
        Nodo sig;

        Nodo(int v) { valor = v; }
    }

    public ListaDeInts() {
        primero = null;
    }

    public ListaDeInts(ListaDeInts otra) {
        Nodo actual = otra.primero;
        while (actual != null) {
            agregarAtras(actual.valor);
            actual = actual.sig;
        }
    }
```

Lista de Enteros: agregando elementos

```
public void agregarAdelante(int elem) {
    Nodo nuevo = new Nodo(elem);
    nuevo.sig = primero;
    primero = nuevo;
}

public void agregarAtras(int elem) {
    Nodo nuevo = new Nodo(elem);
    if (primero == null) {
        primero = nuevo;
    } else {
        Nodo actual = primero;
        while (actual.sig != null) {
            actual = actual.sig;
        }
        actual.sig = nuevo;
    }
}
```

Lista de Enteros: eliminando un elemento

```
public void eliminar(int i) {
    Nodo actual = primero;
    Nodo prev = primero;
    for (int j = 0; j < i; j++) {
        prev = actual;
        actual = actual.sig;
    }
    if (i == 0) {
        primero = actual.sig;
    } else {
        prev.sig = actual.sig;
    }
}
```

Recorriendo colecciones

¿Cómo recorremos una colección?

Recorriendo colecciones

¿Cómo recorremos una colección?

```
import java.util.*;
class RecorriendoColecciones
{
  public static void main(String[] arg)
  {
    List<String> seleccion = new ArrayList<String>();
    seleccion.add("Messi");
    seleccion.add("Martínez");
    for (String jugador : seleccion)
      System.out.println(jugador);
  }
}
```

Recorriendo colecciones

No obstante, la **estructura subyacente** a una colección puede estar implementada de muchas maneras distintas. No es lo mismo iterar sobre un arreglo que sobre una lista enlazada...

- Esta estructura es **privada** y, por lo tanto, invisible para el usuario.

Recorriendo colecciones

No obstante, la **estructura subyacente** a una colección puede estar implementada de muchas maneras distintas. No es lo mismo iterar sobre un arreglo que sobre una lista enlazada...

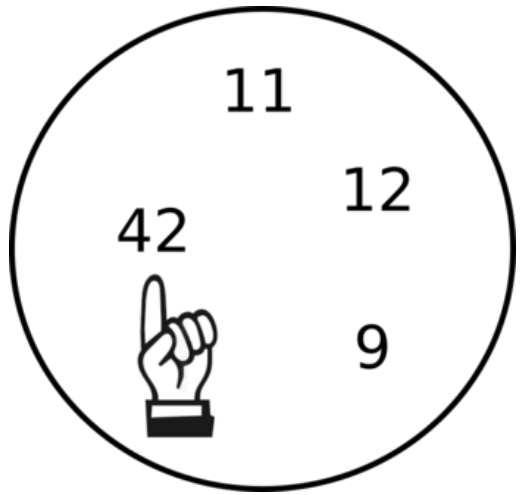
- Esta estructura es **privada** y, por lo tanto, invisible para el usuario.
- Entonces... ¿cómo podemos recorrer una colección sin conocer su estructura?

Iteradores

Un **iterador** es una manera abstracta de recorrer colecciones, independientemente de su estructura.

Informalmente

$$\text{iterador} = \text{colección} + \text{dedo}$$



Iteradores

Operaciones con iteradores:

- ¿Está posicionado sobre un elemento?
- Obtener el elemento actual.
- Avanzar al siguiente elemento.
- Retroceder al elemento anterior. *(Bidireccional)*

Iteradores en Java

Como corresponde, Java provee de una interfaz para iteradores:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
}
```

- *Obtener* y *avanzar* se combinan en el método `next()`.
- Nosotros vamos a ser los responsables de implementarlo sobre nuestra estructura de datos.

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext(); → true
it.next();

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next(); $\rightarrow$ 1

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next(); $\rightarrow$ 2

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next();
it.next(); $\rightarrow$ 0

Interpretando al iterador en una secuencia

1	2	0	3	0

```
Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next();
it.next();
it.next(); → 3
```

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next();
it.next();
it.next();
it.next(); $\rightarrow$ 0

Interpretando al iterador en una secuencia

1	2	0	3	0

Iterator it = secuencia.iterator();
it.hasNext();
it.next();
it.next();
it.next();
it.next();
it.next();
it.hasNext(); → false

Implementando el iterador de ArrayList

Supongamos la siguiente implementación de la clase `ArrayList`:

```
public class ArrayList<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  public Iterator<T> iterator(){
    return new Iterador();
  }
  ...
}
```

Implementando el iterador de ArrayList

Supongamos la siguiente implementación de la clase `ArrayList`:

```
public class ArrayList<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  public Iterator<T> iterator(){
    return new Iterador();
  }
  ...
}
```

¿Dónde va a estar implementado `Iterador`?

Implementando el iterador de ArrayList

Supongamos la siguiente implementación de la clase `ArrayList`:

```
public class ArrayList<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  public Iterator<T> iterator(){
    return new Iterador();
  }
  ...
}
```

¿Dónde va a estar implementado `Iterador`?

¡Dentro de la clase! ¿Por qué?

Implementando el iterador de ArrayList

Supongamos la siguiente implementación de la clase `ArrayList`:

```
public class ArrayList<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  public Iterator<T> iterator(){
    return new Iterador();
  }
  ...
}
```

¿Dónde va a estar implementado `Iterador`?
¡Dentro de la clase! ¿Por qué? Porque necesitamos acceder a la estructura interna de la clase (i.e., sus atributos privados) para poder recorrerla.

Diferenciando los tantos

Disponemos de la *interfaz* Iterador, la *clase* Iterador, y el *método* que devuelve un iterador...

- La interfaz `Iterator<T>` define los métodos que deben implementar los iteradores de *cualquier* clase.
- La clase `Iterador` implementa los métodos de la interfaz `Iterator<T>` dentro de la clase `ArrayList<T>`, definiendo cómo se recorre.
- El método *público* `iterator()` devuelve una instancia de la clase `Iterador` (noten que llama a su constructor), permitiendo al usuario de la clase acceder a un iterador.

Implementando el iterador de ArrayList

```
public class ArrayList<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  private class Iterador implements Iterator<T>{
    ...
  }
}
```

Implementación del iterador de ArrayList

```
public class ArrayList<T> implements List<T>{
  private T[] elementos;
  private int size;
  ...
  private class Iterador implements Iterator<T>{
      int dedito;
      Iterador(){
        dedito = 0;
      }
      public boolean hasNext(){
        return dedito != size;
      }
      public T next(){
        int i = dedito;
        dedito = dedito + 1;
        return elementos[i];
      }
    }
  }
```

Usando nuestro iterador

```
import java.util.ArrayList;
import java.util.Iterator;
public class ArrayListIteratorExample {
  public static void main(String[] args) {
      ArrayList<String> frutas = new ArrayList<String>();
      frutas.add("Manzana");
      frutas.add("Naranja");
      frutas.add("Durazno");

      // Ahora podemos usar for-each!
      for (String fruta : frutas) {
        System.out.println(fruta);
      }

      Iterator it = frutas.iterator();
      while(it.hasNext()){
        System.out.println(it.next());
      }
  }
}
```

Iterando listas enlazadas

Distintas estructuras de datos requieren de distintos métodos para recorrerlas.
Ya comentamos que las listas enlazadas no permiten acceso aleatorio, por lo que no podemos implementar un iterador de la misma forma que en un arreglo.
Es decir, no podemos usar índices para recorrer una lista enlazada...

Jerarquía de colecciones en Java