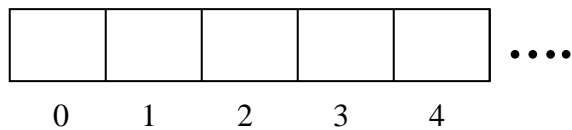


## CAPÍTULO III: COLECCIONES

### 3.1 Listas con nexos

#### Arreglos:

- El arreglo es la estructura de datos más comúnmente utilizada.
- Sea  $A(i)$   $i = 0, n$



Los elementos  $a_i$  y  $a_{i+1}$  se almacenan en las localizaciones  $i$  e  $i+1$  del arreglo.

- Esto permite recuperar o modificar los valores asociados a cualquier posición, en una cantidad constante de tiempo, debido a que la memoria del computador, permite acceso directo a cualquier posición.

#### Desventajas de los arreglos:

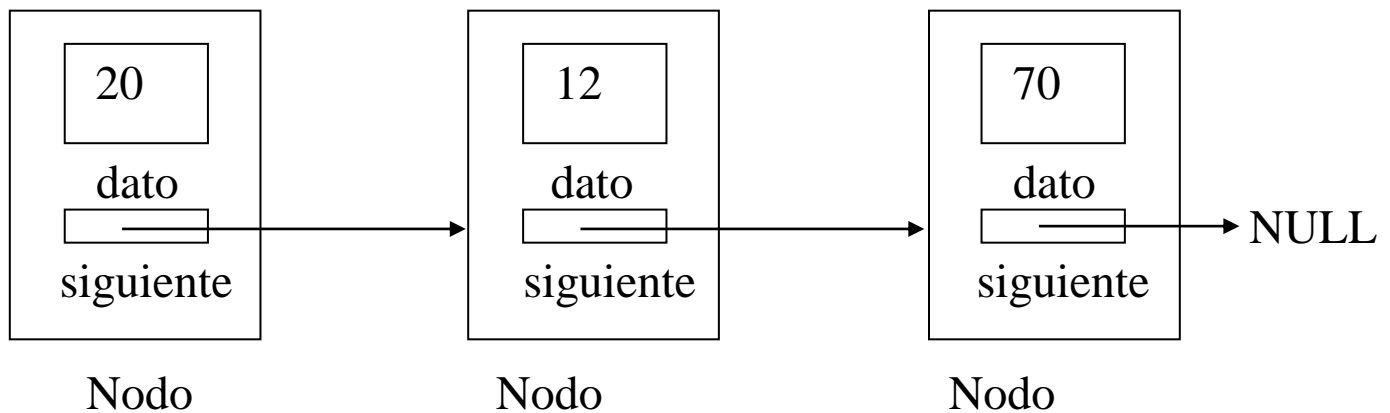
- En un arreglo no ordenado: la búsqueda es lenta.
- En un arreglo ordenado: la inserción es lenta.
- En ambos casos:
  - √ Eliminación es lenta.
  - √ El tamaño de un arreglo no puede ser cambiado, después que es creado.

## Listas con Nexos:

- La lista con nexos (lista enlazada) es una estructura de datos, que consta de un **número variable** de ítems.
- Después de los arreglos, son la estructura de datos más popular.
- **Ventajas:**      Inserción es rápida.  
                         Eliminación es rápida.
- **Desventajas:** Búsqueda es lenta.

## Definición de una lista con nexos en Java

### Ejemplo de lista con nexo



```

public class Nodo {
    private int dato;
    private Nodo siguiente;
    .....
    .....
}
  
```

¡siguiente, contiene una referencia a un Nodo!

```

public class Nodo {
    private int dato;
    private Nodo siguiente;

    public Nodo (int nuevoDato, Nodo sig){
        dato = nuevoDato;
        siguiente = sig;
    }

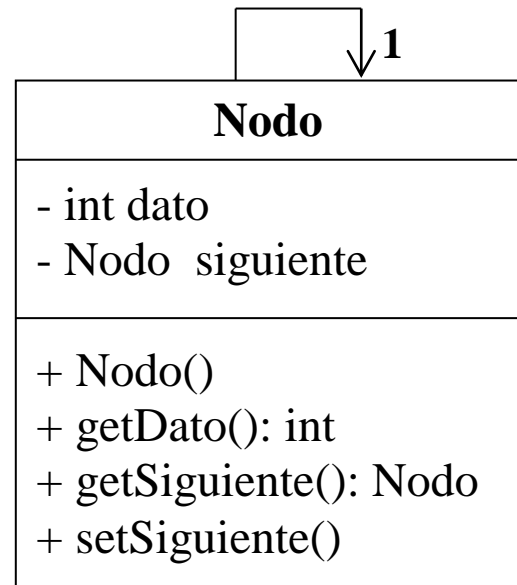
    public int getDato() {
        return dato;
    }

    public Nodo getSiguiente() {
        return siguiente;
    }

    public void setSiguiente(Nodo n){
        Siguiente = n;
    }

}

```

**Diagrama de la clase Nodo**

En las listas con nexos:

- a) Se debe contar con una referencia al primer elemento de la lista.
- b) Cada elemento de la lista (nodo), contiene una referencia al siguiente elemento de la lista.
- c) El último elemento de la lista tiene una referencia nula, la que indica que no existe un siguiente elemento.
- d) Cada elemento de la lista debe contener al menos dos campos:
  - información (datos)
  - siguiente (referencia)

Debe destacarse que, en un **arreglo**, cada ítem ocupa una determinada posición. Esta posición puede ser directamente accesada, utilizando un valor para el índice.

En una **lista con nexos**, la única manera de encontrar un elemento en particular, es buscándolo a través de la cadena de elementos.

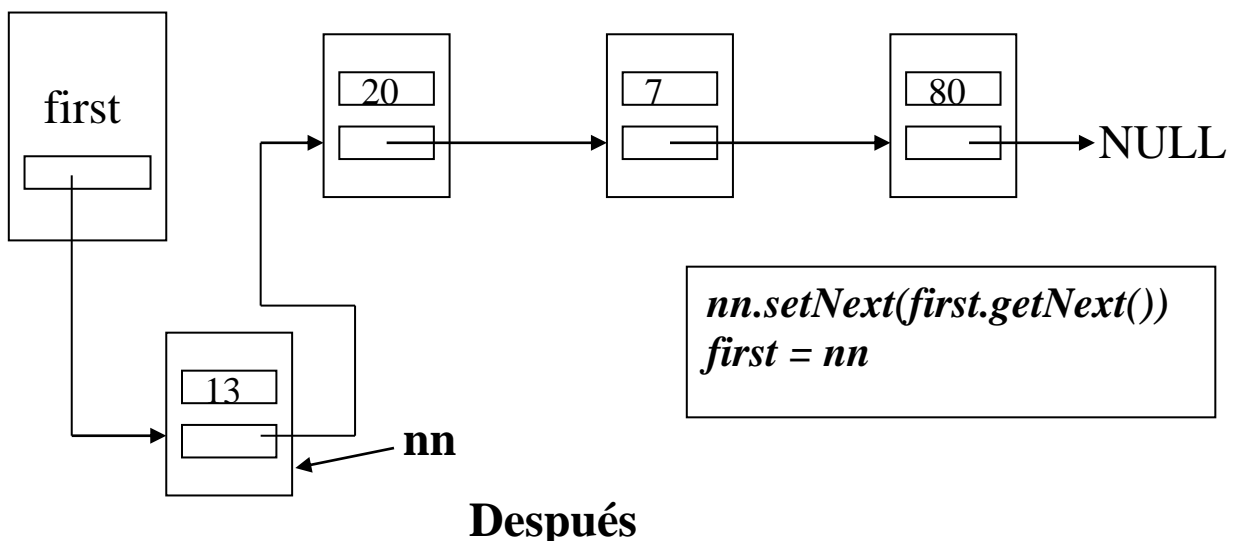
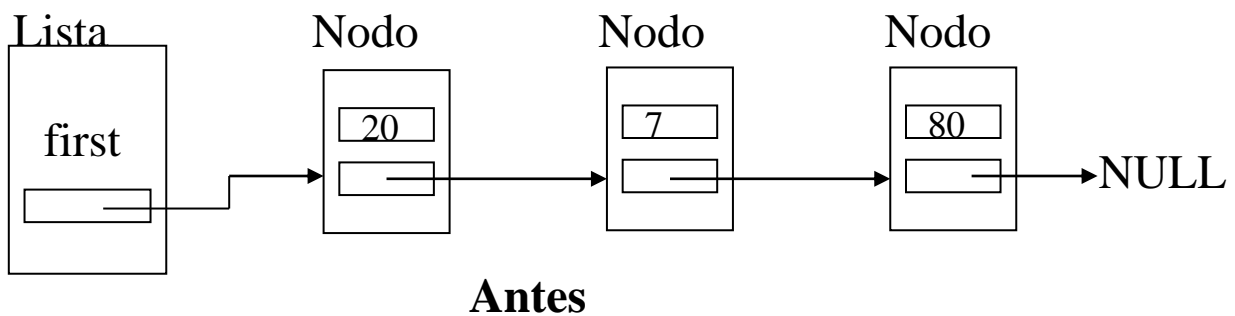
Arreglo  $\equiv$  Almacenamiento Secuencial.  
**Lista con Nexos**  $\equiv$  **Almacenamiento No Secuencial.**

## Lista con Nexos Simple

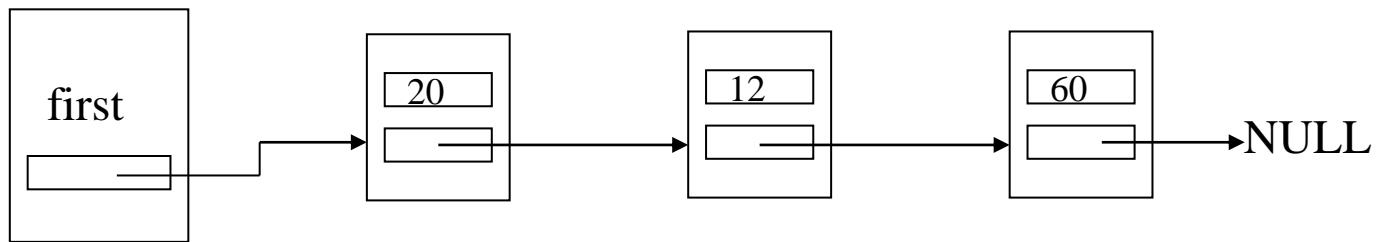
### Operaciones de interés en la lista

- Insertar un ítem al comienzo de la lista.
- Iterar a través de la lista para desplegar su contenido.
- Buscar en la lista, el ítem con una clave dada.
- Eliminar un ítem, con una clave dada.

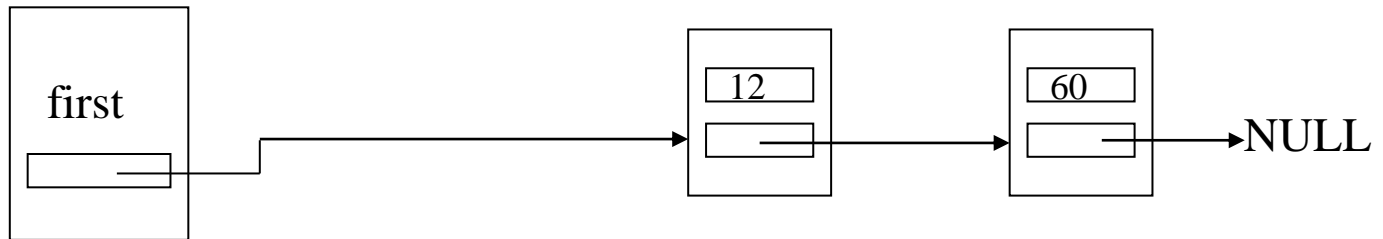
### Inserción en la Lista



## Eliminación en la Lista

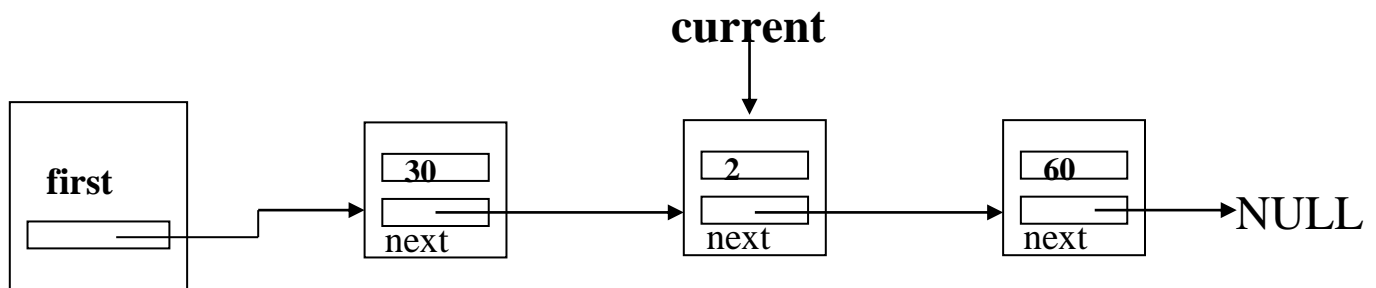


**Antes**

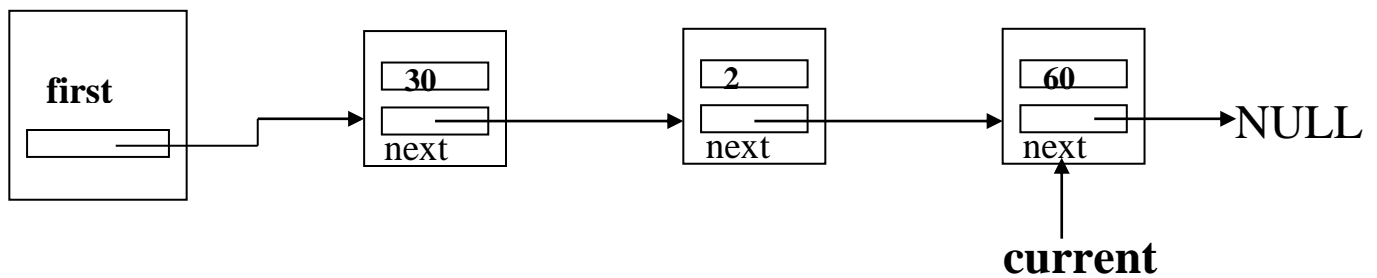


**Después de:**  
*first = first.getNext()*

## Avanzar en la Lista

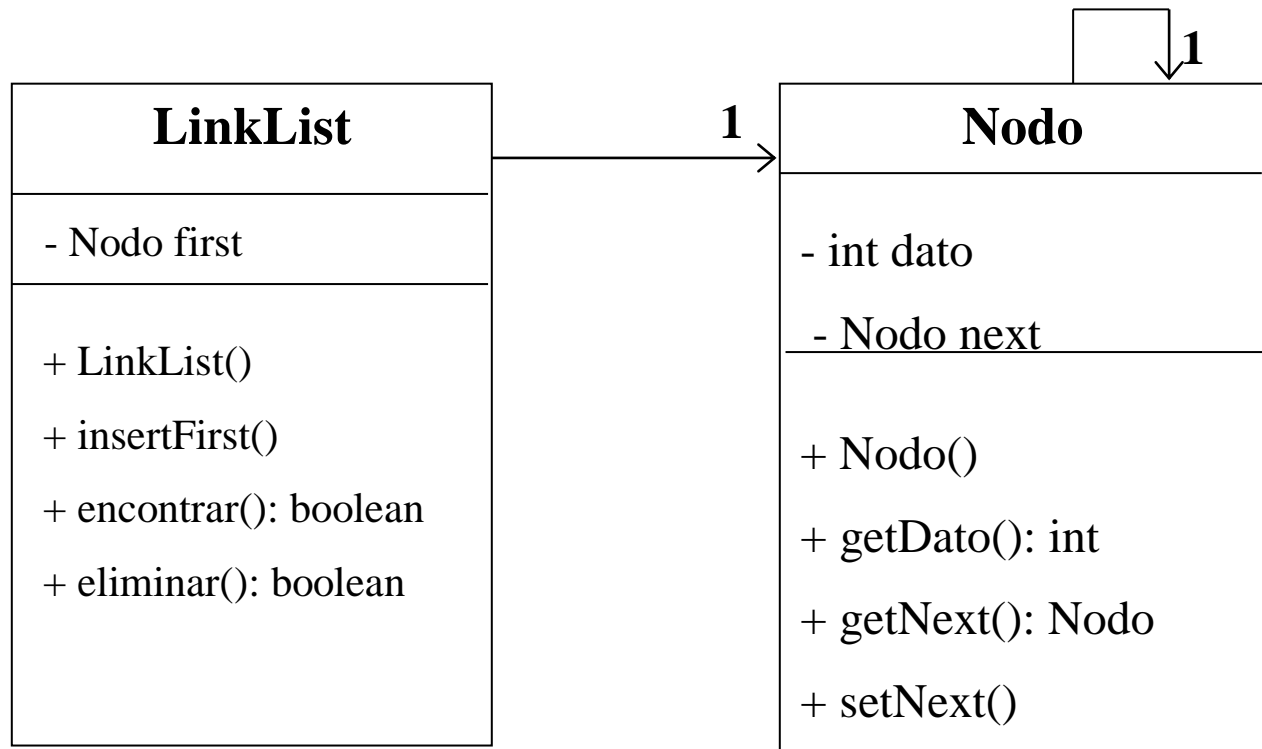


**Antes de** *current = current.getNext()*



**Después de** *current = current.getNext()*

## Ejemplo de una lista con nexos en Java



**// muestra una lista con nexos**

```

public class Nodo{
    private int dato; //item de dato (clave)
    private Nodo next; //próximo nodo en la lista
  
```

**// -----**

```

public Nodo(int id) {// constructor
    dato = id;
    next = null;
}
  
```

```
// -----  
public int getDato() {  
    return dato;  
}  
  
// -----  
public Nodo getNext() {  
    return next;  
}  
  
// -----  
public void setNext(Nodo n){  
    this.next = n;  
}  
  
} // fin de clase Nodo
```

```
class LinkedList {
```

```
// first referencia al primer nodo de la lista
private Nodo first;
```

```
// -----  
public LinkedList(){ // constructor  
    first = null; //no hay nodos en la lista todavía  
}
```

```
// -----  
public void insertFirst(int id) {  
    Nodo newLink = new Nodo(id); // crea un nuevo nodo  
    newLink.setNext(first); //apunta al antiguo primer nodo  
    first = newLink; // ahora first apunta a este  
    // nuevo primer nodo  
}
```

```

public boolean encontrar(int key) { //public Nodo encontrar..
    // encuentra el nodo con la clave dada.
    Nodo current=first; //comienza en 'first'
    while(current != null && current.getDato() != key){
        current = current.getNext();
    }
    if (current != null){
        return true;
    }
    else{
        return false;
    }
}

// -----
public boolean eliminar(int key){//elimina nodo con clave dada
    Nodo current=first;
    Nodo previo = first;
    while(current != null && current.getDato() != key) {
        previo = current;
        current = current.getNext();
    }
    if (current != null) //lo encontré
        if(current == first){ // si el nodo es el primero
            first = first.getNext(); //cambia first
        }
        else { //en caso contrario
            previous.setNext(current.getNext());//lo "bypasea"
        }
        return true;
    }
    else{
        return false;
    }
}
// fin clase LinkedList

```

*Se podría retornar el  
nodo con el elemento  
encontrado*



```
public class LinkList2App {
```

```
public static void desplegarLista(LinkList theList){
```

```
    Nodo current = theList.first(); //comienza del principio
```

```
    while(current != null) { //hasta que sea fin de lista
```

```
        //imprime el dato
```

```
        StdOut.println(current.getDato());
```

```
        current = current.getNext();//se mueve al próximo nodo
```

```
    }
```

```
    StdOut.println("");
```

```
}
```

*Recuerde que el desplegar la lista daría origen a:*

- *Un contrato para obtener los datos de la lista si es que se trabaja con el toString() o*
- *Un contrato para obtener la lista*

```
public static void main(String[] args){
```

```
    LinkList theList = new LinkList();//crea la lista
```

```
    // inserta 4 items
```

```
    theList.insertFirst(22);
```

```
    theList.insertFirst(44);
```

```
    theList.insertFirst(66);
```

```
    theList.insertFirst(88);
```

```
    //Despliega la lista
```

```
    desplegarLista(theList);
```

```
    boolean encontrado = theList.encontrar(44);//encontrar item
```

```
    if( encontrado){
```

```
        StdOut.println("Nodo encontrado");
```

```
    }
```

```
    else{
```

```
        StdOut.println("No encontrado");
```

```
    }
```

```

    boolean eliminado = theList.eliminar(66);//elimina ítem

    if( eliminado ){
        StdOut.println("Nodo eliminado");
    }
    else{
        StdOut.println("No se puede eliminar");
    }

//Despliega la lista nuevamente
    desplegarLista(theList);

} // fin main

} // fin clase LinkList2App

```

## Ejercicio

Un médico tiene asociado un rut (string) y nombre. Se desea crear una lista con un único nexa que contenga todos los médicos. Los datos se leen desde pantalla, donde viene el rut y nombre del médico. Fin de datos, rut = 111. Una vez ingresados los datos, se pide desplegar el nombre de los médicos.

### Se pide:

- Modelo del dominio
- Contratos
- Diagrama de clases del dominio de la aplicación
- Diagrama de clases
- Código

### Modelo del dominio

| <b>Medico</b> |
|---------------|
| Rut           |
| Nombre        |

## Contratos

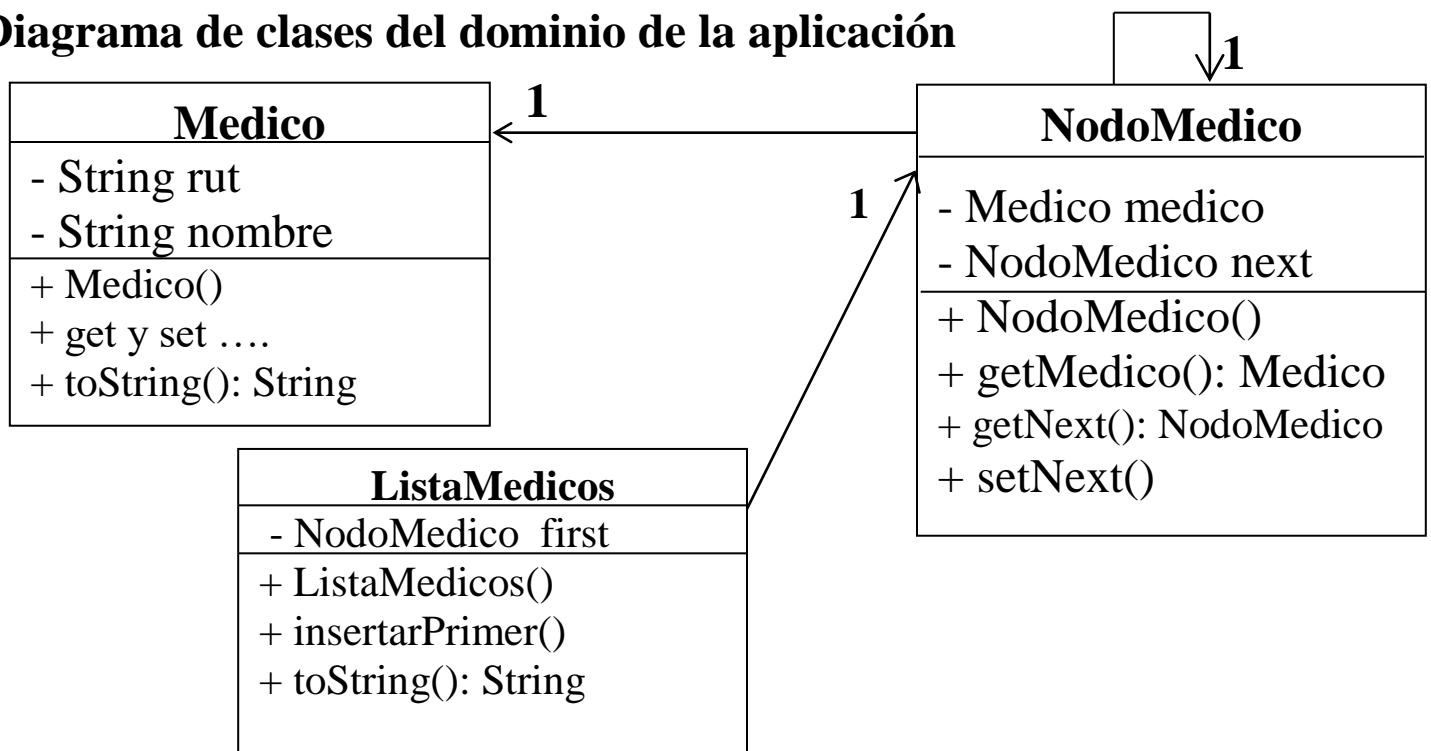
|                        |                                      |
|------------------------|--------------------------------------|
| <b>Operación</b>       | <b>Ingresar medico</b> (rut, nombre) |
| <b>Descripción</b>     | Se ingresa un médico                 |
| <b>Precondiciones</b>  |                                      |
| <b>Postcondiciones</b> | Médico ingresado                     |

|                        |                               |
|------------------------|-------------------------------|
| <b>Operación</b>       | <b>Obtener medicos</b>        |
| <b>Descripción</b>     | Se obtienen todos los médicos |
| <b>Precondiciones</b>  |                               |
| <b>Postcondiciones</b> |                               |

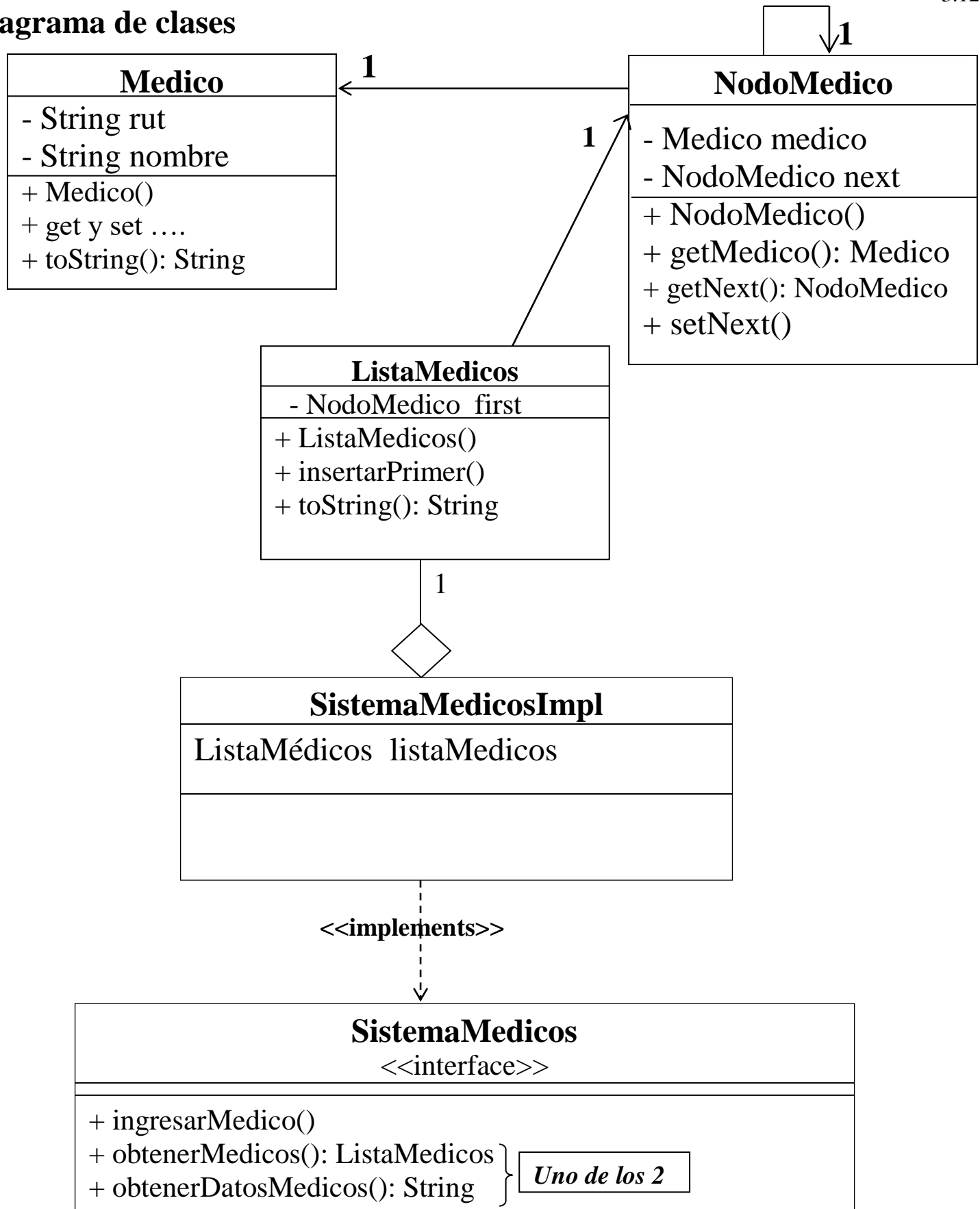
*Uno de los dos,  
dependiendo si se usará  
el toString o no*

|                        |  |
|------------------------|--|
| <b>Operación</b>       | <b>Obtener datos de todos los médicos</b>  |
| <b>Descripción</b>     | Se obtienen los datos de todos los médicos |
| <b>Precondiciones</b>  |  |
| <b>Postcondiciones</b> |  |

## Diagrama de clases del dominio de la aplicación



## Diagrama de clases



```

package cl.ucn.ei.pa.sistemaMedicos.dominio;

public class Medico {
    private String rut;
    private String nombre;

    public Medico(String rut, String nombre) {
        this.rut = rut;
        this.nombre = nombre;
    }

    public String getRut() {
        return rut;
    }

    public void setRut(String rut) {
        this.rut = rut;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public String toString() {
        return "Medico [" + (rut != null ? "rut=" + rut +
            ", " : "") + (nombre != null ? "nombre=" +
            nombre : "") + "]";
    }
}

```

```
package cl.ucn.ei.pa.sistemaMedicos.logica;

import cl.ucn.ei.pa.sistemaMedicos.dominio.*;

public class NodoMedico {
    private Medico medico;
    private NodoMedico next; //próximo nodo en la lista

    public NodoMedico(Medico m) { //Constructor
        this.medico = m;
        next = null;
    }

    public Medico getMedico() {
        return medico;
    }

    public NodoMedico getNext() {
        return next;
    }

    public void setNext(NodoMedico n) {
        next= n;
    }
}
```

```

package cl.ucn.ei.pa.sistemaMedicos.logica;
import cl.ucn.ei.pa.sistemaMedicos.dominio.*;

public class ListaMedicos {
    private NodoMedico first;
    // first referencia al primer nodo de la lista

    public ListaMedicos() { // constructor
        first = null; //no hay nodos en la lista todavía
    }

    public void insertarPrimer(Medico m) {
        NodoMedico nuevoNodo=new NodoMedico(m);
        // crea un nuevo nodo
        nuevoNodo.setNext(first); // apunta al antiguo primer nodo
        first = nuevoNodo;
        // ahora first apunta a este nuevo primer nodo
    }

    public NodoMedico getFirst() {
        return first;
    }

    @Override
    public String toString() { // Se implementa el toString
        String salida = "";
        NodoMedico actual = first;
        // comienza del principio de la lista
        while(actual != null) {
            // mientras no sea fin de lista
            Medico medico= actual.getMedico();
            // Obtiene el médico
            // Se concatena el nombre y rut
            salida = salida + "nombre: " +
                medico.getNombre() + ", rut: " +
                medico.getRut() + "\n";
            actual = actual.getNext();
            // se mueve al próximo nodo
        }
        return salida;
    }
}

```

salida = salida +  
medico.toString();

```

package cl.ucn.ei.pa.sistemaMedicos.logica;

public interface SistemaMedicos {
    public void ingresarMedico(String rut, String nombre);

    public ListaMedicos obtenerMedicos();
    public String obtenerDatosMedicos();
}

```

Uno de los 2

```

package cl.ucn.ei.pa.sistemaMedicos.logica;

import cl.ucn.ei.pa.sistemaMedicos.dominio.Medico;

public class SistemaMedicosImpl implements SistemaMedicos{

    private ListaMedicos listaMedicos;

    public SistemaMedicosImpl(){
        listaMedicos = new ListaMedicos();
    }

    public void ingresarMedico(String rut, String nombre){
        Medico medico = new Medico(rut, nombre);
        listaMedicos.insertarPrimer(medico);
    }

    public ListaMedicos obtenerMedicos(){
        return listaMedicos;
    }

    public String obtenerDatosMedicos(){
        return listaMedicos.toString();
    }
}

```

Uno de los 2



```

package cl.ucn.ei.pa.sistemaMedicos.logica;
import cl.ucn.ei.pa.sistemaMedicos.dominio.*;
import ucn.StdIn;
import ucn.StdOut;

public class App {

    public static void leerMedicos(SistemaMedicos sistema) {
        //Lectura de los datos
        StdOut.print("Ingrese rut. Fin medicos rut=111: ");
        String rut = StdIn.readString();
        while (! rut.equals("111")){
            StdOut.print("Ingrese nombre medico: ");
            String nombre =StdIn.readString();
            sistema.ingresarMedico(rut, nombre);
            StdOut.print("Ingrese rut. Fin medicos rut=111: ");
            rut = StdIn.readString();
        }
    }

    public static void desplegarMedicos(ListaMedicos
                                       listaMedicos) {
        StdOut.println("Despliega la lista desde el
                       principio al final: ");
        NodoMedico actual = listaMedicos.getFirst();
        //comienza del principio de la lista
        while(actual != null) {
            //mientras no sea fin de lista
            Medico medico= actual.getMedico();
            //Obtiene al médico
            //imprime su nombre y rut
            StdOut.println ("nombre: " +
                           medico.getNombre()+", rut: " +
                           medico.getRut());
            salida = salida +
                       medico.toString();

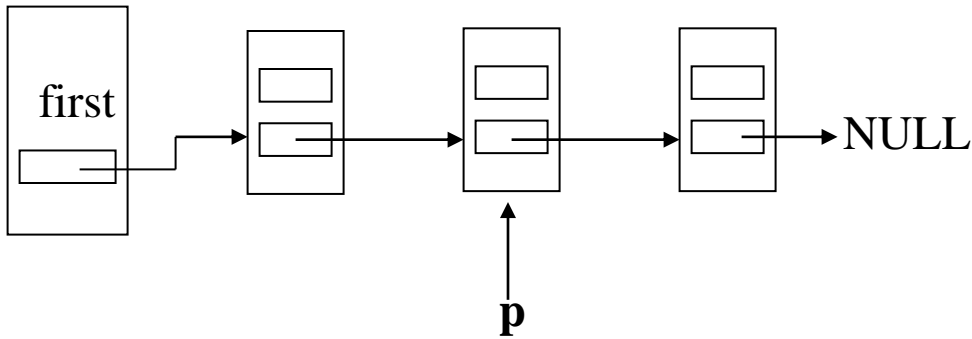
            actual = actual.getNext();
            //se mueve al próximo nodo
        }
        StdOut.println("");
    }
}

```

```
public static void main(String[] args) {  
  
    SistemaMedicos sistema = new SistemaMedicosImpl();  
  
    leerMedicos(sistema);  
  
    // (1) o (2)  
  
    StdOut.println(sistema.obtenerDatosMedicos()); (1)  
  
    ListaMedicos listaMedicos =  
        sistema.obtenerMedicos(); } (2)  
    desplegarMedicos(listaMedicos);  
}  
}
```

## Listas con Doble Nexo

En las listas con un único nexo existen inconvenientes si se desea ubicar el nodo que precede al que se encuentra referenciado por **p**.

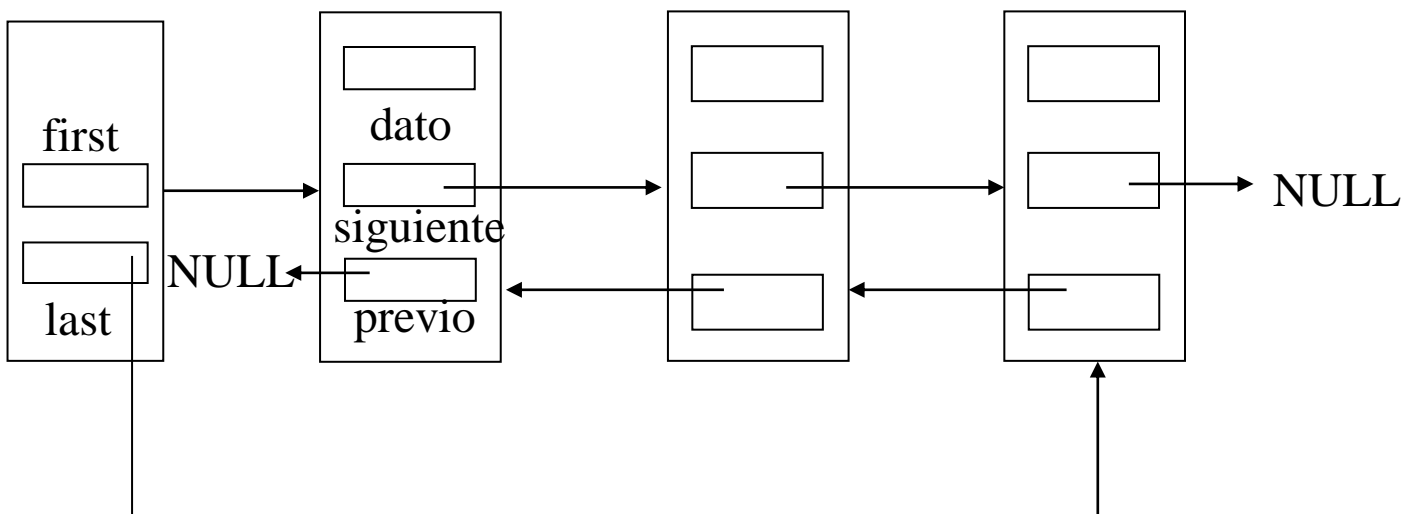


Situación similar se produce en el caso que se requiera eliminar un nodo arbitrario referenciado por **p**.

Las listas con doble nexo son útiles en el caso de problemas en que se requiera avanzar en ambas direcciones de la lista.

En este caso, cada nodo no sólo tiene un puntero al siguiente nodo, sino que también tiene un puntero al nodo anterior.

## Lista con Doble Nexo

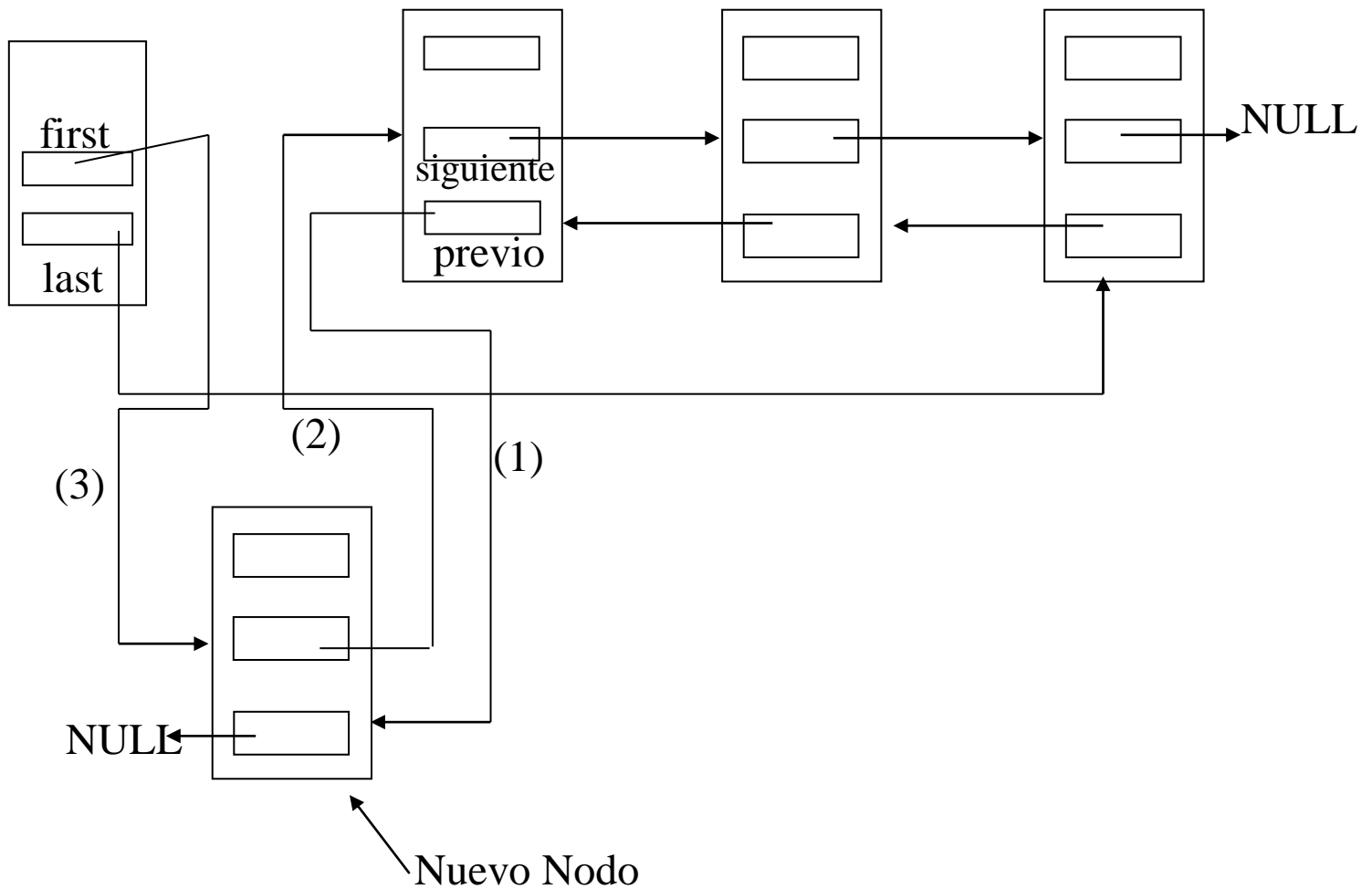


```

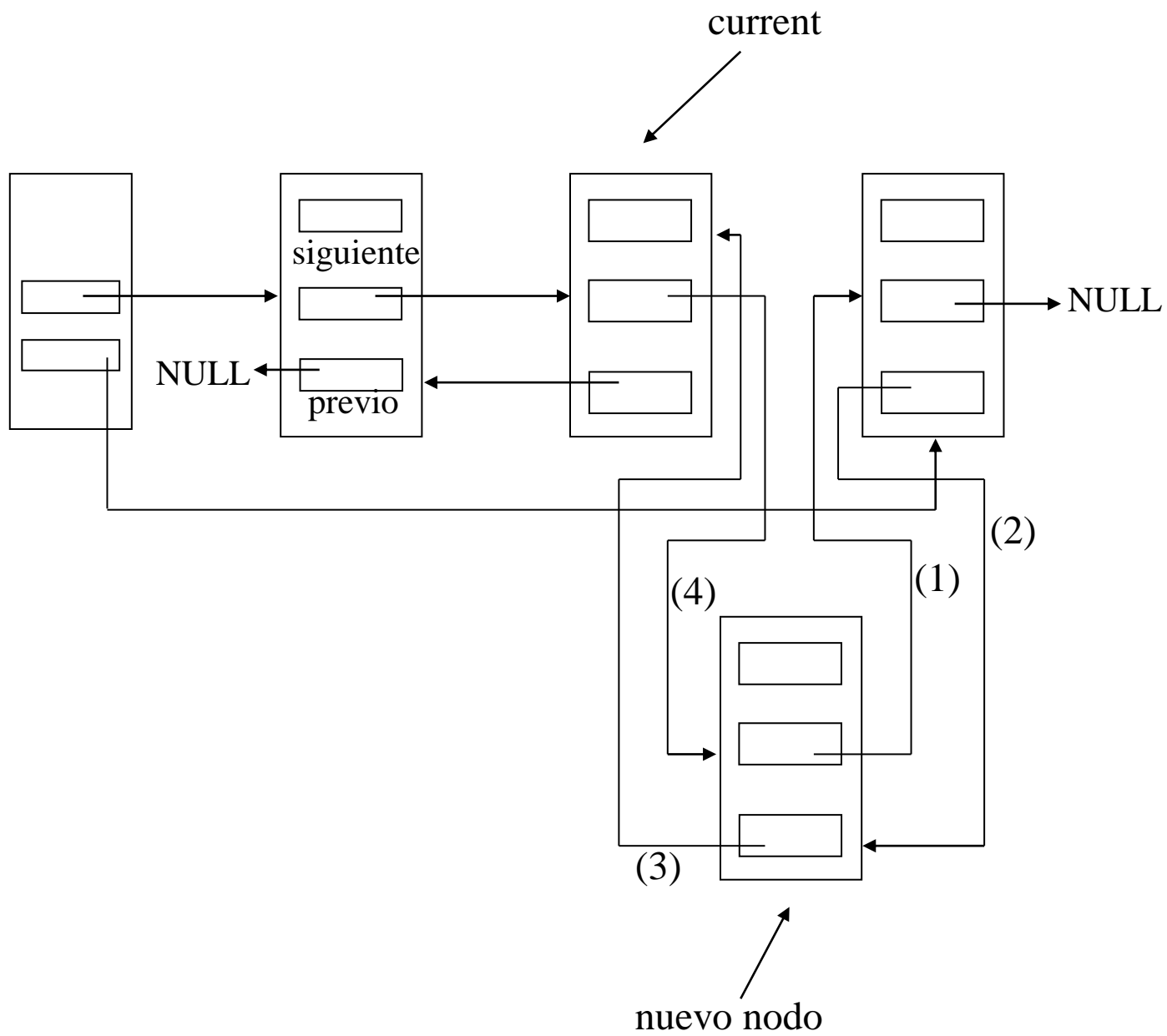
public class Nodo {
    private double dData;
    private Nodo    next;
    private Nodo    previo;
    .....
    .....
}

```

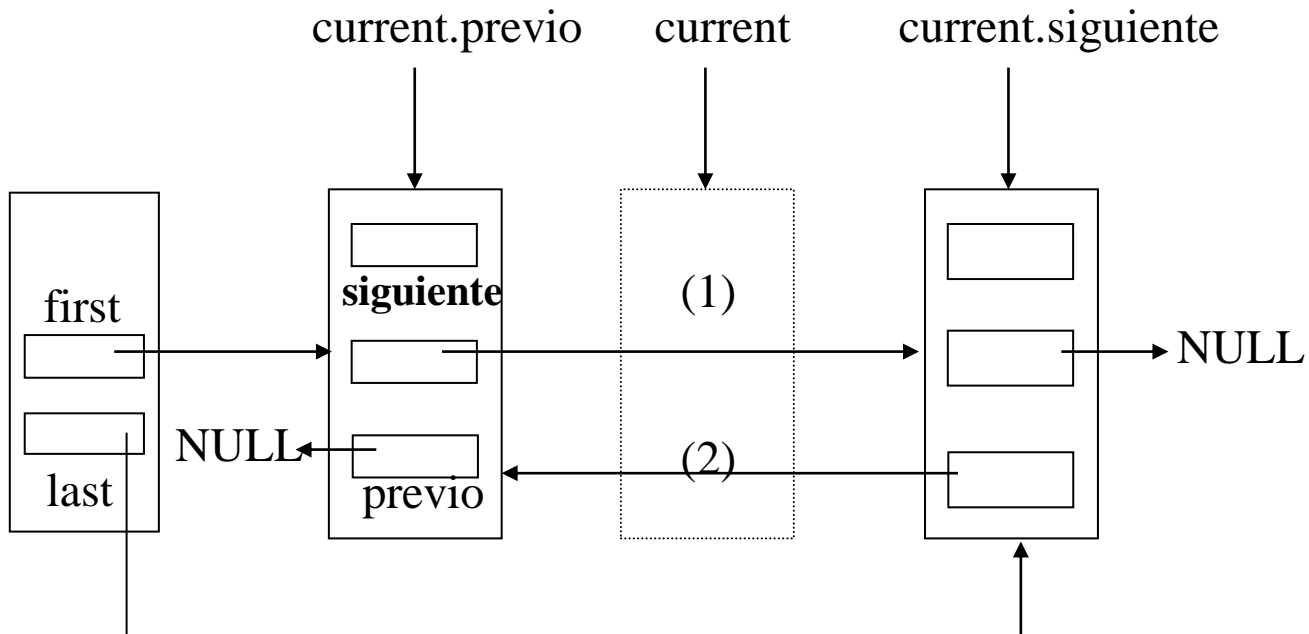
### Inserción al comienzo de la lista



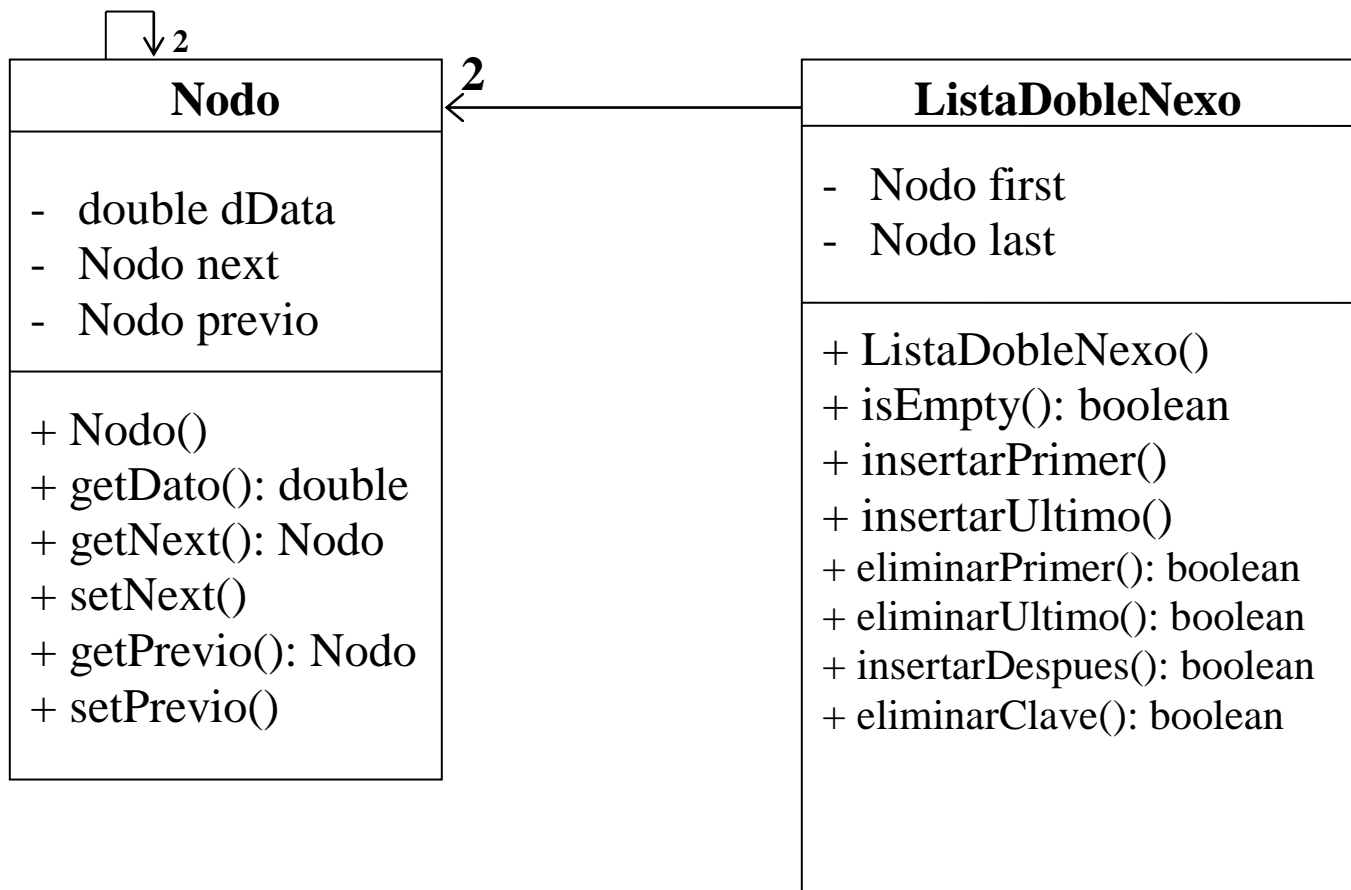
## Inserción en una localización arbitraria



## Eliminación de un nodo arbitrario



## Ejemplo de lista con doble nexos



```
public class Nodo {  
    private dData; // item de dato  
    private Nodo next; // referencia al siguiente  
    private Nodo previo; // referencia al anterior  
  
    public Nodo(double d) { // constructor  
        dData = d;  
        next = null;  
        previo = null;  
    }  
  
    public double getDato() {  
        return dData;  
    }  
  
    public Nodo getNext() {  
        return next;  
    }  
  
    public Nodo getPrevio() {  
        return previo;  
    }  
  
    public void setNext(Nodo nodo) {  
        next = nodo;  
    }  
  
    public void setPrevio(Nodo nodo) {  
        previo = nodo;  
    }  
}
```

```

public class ListaDobleNexo {
    private Nodo first; //referencia al primer item
    private Nodo last; // referencia al último item

    public ListaDobleNexo(){ // constructor
        first = null;
        last = null;
    }

    public boolean isEmpty(){ //verdadero si no hay nodos
        return first==null;
    }

    public void insertarPrimer(double dd) { //inserta al frente de la lista
        Nodo nuevoNodo = new Nodo(dd);
        if( isEmpty() ){
            last = nuevoNodo;
        }
        else{
            first.setPrevio(nuevoNodo);
        }
        nuevoNodo.setNext(first);
        first = nuevoNodo;
    }

    public void insertarUltimo(double dd) { //inserta al final de la lista
        Nodo nuevoNodo = new Nodo(dd);
        if( isEmpty() ) {
            first = nuevoNodo;
        }
        else {
            last.setNext(nuevoNodo);
            nuevoNodo.setPrevio(last);
        }
        last = nuevoNodo;
    }
}

```



```

public boolean eliminarPrimer() { //elimina el primer nodo
    if (!this.isEmpty()) {
        if(first.getNext()== null) { //Tiene un solo elemento
            last = null;
        }
        else{
            first.getNext().setPrevio(null);
        }
        first = first.getNext();
        return true;
    }
    else{
        return false;
    }
}

```

```

public boolean eliminarUltimo () { //elimina el último nodo
    if (!this.isEmpty()) {
        if(first.getNext() == null) { //Tiene un solo elemento
            first = null;
        }
        else{
            last.getPrevio().setNext(null);
        }
        last= last.getPrevio();
        return true;
    }
    else {
        return false;
    }
}

```

```
public boolean insertarDespues (double key, double dd) {  
// inserta dd justo después de key
```

```
    Nodo current = first;  
    while(current != null && current.getDato() != key) {  
        current = current.getNext();  
    }  
    if (current != null) {  
        Nodo nuevoNodo = new Nodo(dd);  
        if(current == last) {  
            nuevoNodo.setNext(null);  
            last = nuevoNodo;  
        }  
        else {  
            nuevoNodo.setNext(current.getNext());  
            current.getNext().setPrevio(nuevoNodo);  
        }  
        nuevoNodo.setPrevio(current);  
        current.setNext(nuevoNodo);  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

```

public boolean eliminarClave(double key) {
//elimina nodo con key

    Nodo current = first;
    while(current != null && current.getDato() != key){
        current = current.getNext();
    }
    if (current != null) { //La encontré
        if(current==first) {
            first = current.getNext();
        }
        else {
            current.getPrevio().setNext(current.getNext());
        }
        if(current==last) {
            last = current.getPrevio();
        }
        else {
            current.getNext().setPrevio(current.getPrev());
        }
        return true;
    }
    else {
        return false;
    }
}

// fin clase ListaDobleNexo

```

```
public class App {
```

```
    public static void desplegarAdelante(ListaDobleNexo listaDobleNexo) {
        StdOut.print("Lista (Principio→Final): ");
        Nodo current = listaDobleNexo.first();
        while(current != null) {
            StdOut.print (current.getDato() + " ");
            current = current.getNext();
        }
        StdOut.println("");
    }
}
```

```
    public static void desplegarAtras(ListaDobleNexo listaDobleNexo) {
        StdOut.print("Lista (Final→Principio): ");
        Nodo current = listaDobleNexo.last();
        while(current != null) {
            StdOut.print (current.getDato() + " ");
            current = current.getPrev();
        }
        StdOut.println("");
    }
}
```

```
    public static void main(String[] args) {

        ListaDobleNexo listaDobleNexo = new ListaDobleNexo();

        // inserta al frente de la lista
        listaDobleNexo.insertarPrimer(22);
        listaDobleNexo.insertarPrimer(44);
        listaDobleNexo.insertarPrimer(66);

        // inserta al final de la lista
        listaDobleNexo.insertarUltimo(11);
        listaDobleNexo.insertarUltimo(33);
        listaDobleNexo.insertarUltimo(55);
    }
}
```

```
App.desplegarAdelante(listaDobleNexo);  
// despliega hacia adelante
```

```
App.desplegarAtras(listaDobleNexo);  
//despliega la lista hacia atrás
```

```
boolean elimina = listaDobleNexo.eliminarPrimer();  
//elimina el primer item
```

```
elimina = listaDobleNexo.eliminarUltimo();  
//elimina el último item
```

```
elimina = listaDobleNexo.eliminarClave(11);  
//elimina el item con clave 11
```

```
App.desplegarAdelante(listaDobleNexo);  
//despliega hacia adelante
```

```
boolean inserta = listaDobleNexo.insertarDespues (22, 77);  
//inserta al 77 después del 22
```

```
inserta = listaDobleNexo.insertarDespues(33, 88);  
//inserta al 88 después del 33
```

```
App.desplegarAdelante(listaDobleNexo);  
//despliega hacia adelante
```

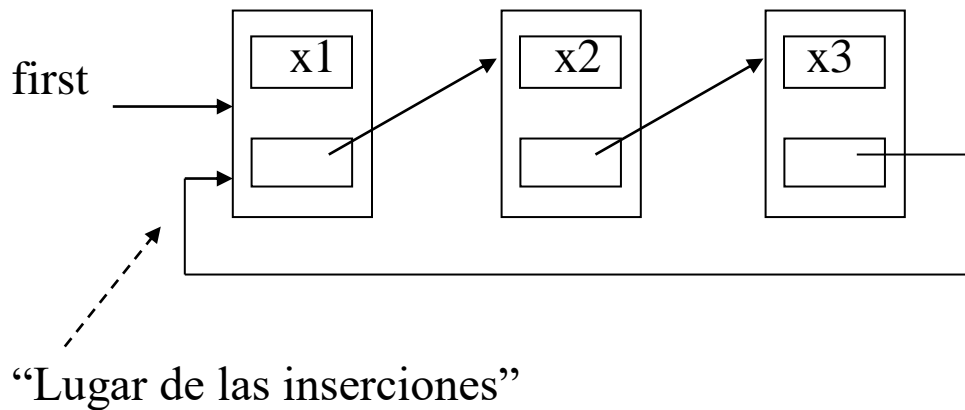
```
} // fin main()
```

```
} // fin App
```

## Alternativas en el diseño de listas

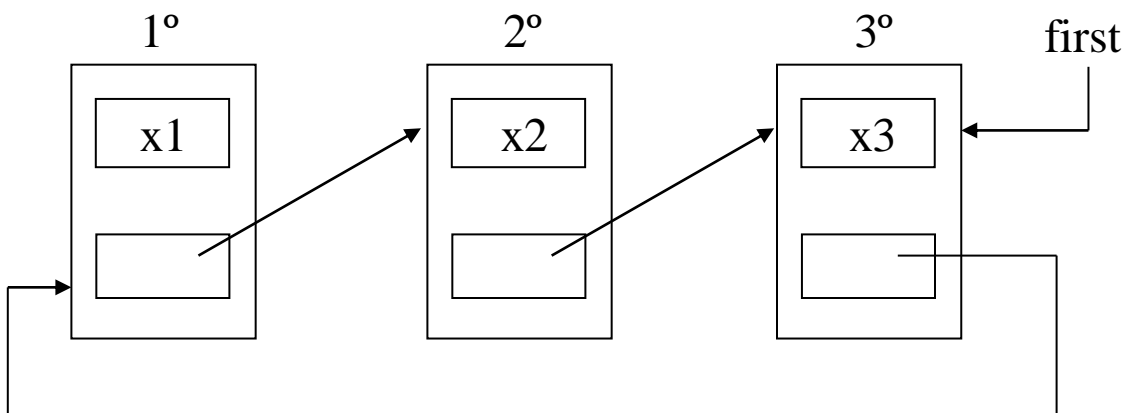
### a) Lista circular

En una lista circular (lista con nexos circular), el campo **next** del último nodo apunta al primer nodo.



Se necesita cambiar el campo **next** del último nodo. Por lo tanto se debe recorrer la lista completa hasta ubicar ese nodo.

Una representación más conveniente de la lista, es que **first** apunte al último nodo.



Si first = null → lista circular vacía

## Despliegue de una lista circular

```
if (l.first != null) { //Lista no vacía
    Nodo current = l.first.getNext();
    while (current != l.first) {
        StdOut.println("dato: " + current.getDato());
        current = current.getNext();
    }
    //Falta el último nodo
    StdOut.println("dato: " + current.getDato());
}
```

## Otra alternativa

```
if (l.first != null) { //Lista no vacía
    Nodo current = l.first.getNext();
    do {
        StdOut.println("dato: " + current.getDato());
        current = current.getNext();
    } while(current != l.first);
    //Falta el último nodo
    StdOut.println("dato: " + current.getDato());
}
```

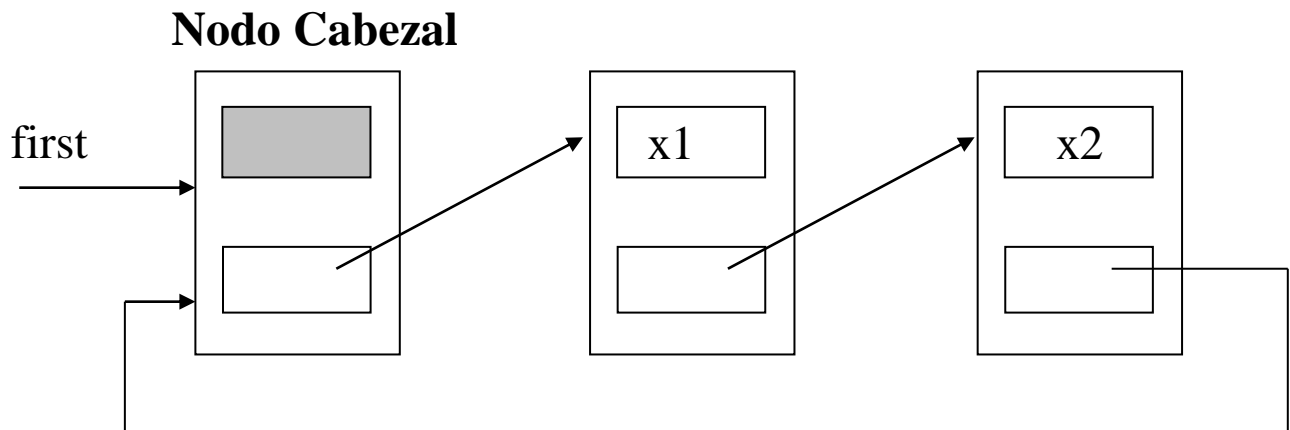
## Otra alternativa

```
if (l.first != null) { //Lista no vacía
    Nodo current = l.getFirst().getNext();
    do {
        StdOut.println("dato: " + current.getDato());
        current = current.getNext();
    } while(current != l.first.getNext());
}
```

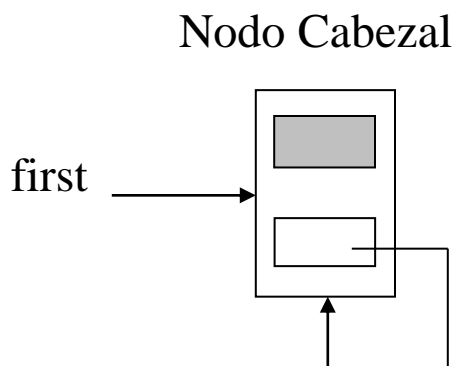
## b) Lista circular con nodo header

Se puede utilizar un nodo cabecal (nodo de encabezamiento, header node), al comienzo o al final de la lista.

Por ejemplo



### Caso de lista vacía



**Se cumple:**

`first.getNext() == first`

```
public class ListaCircularHeader {
    private Nodo first;

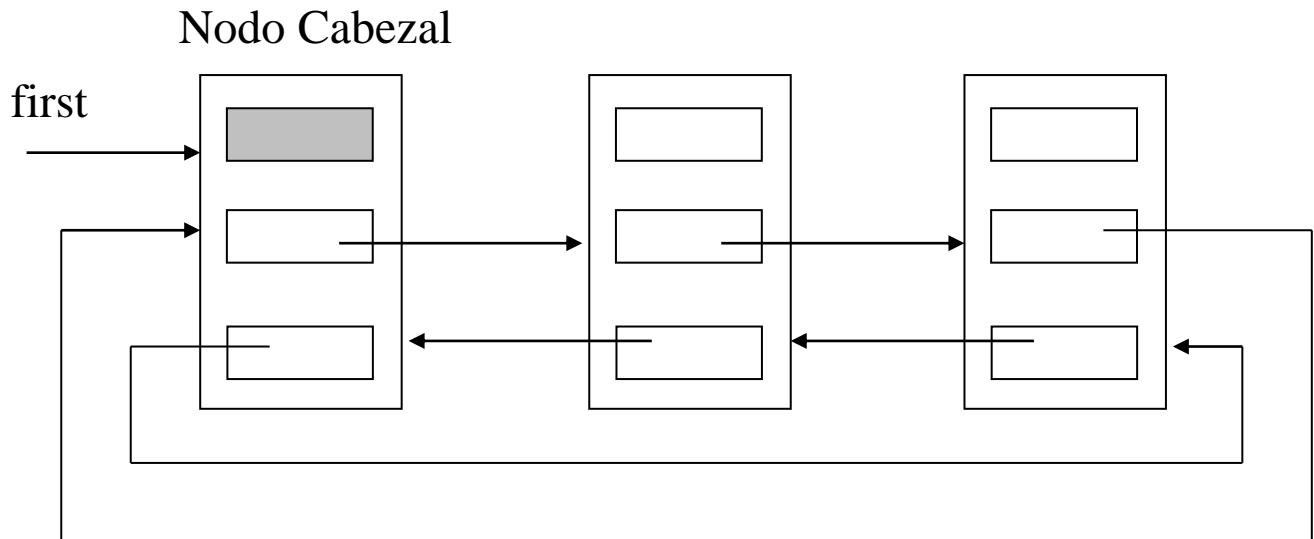
    public ListaCircularHeader () {
        Nodo n = new Nodo();
        first = n;
        first.setNext(n);
    }
    ...
}

public class Nodo {
    private int dato;
    private Nodo next;

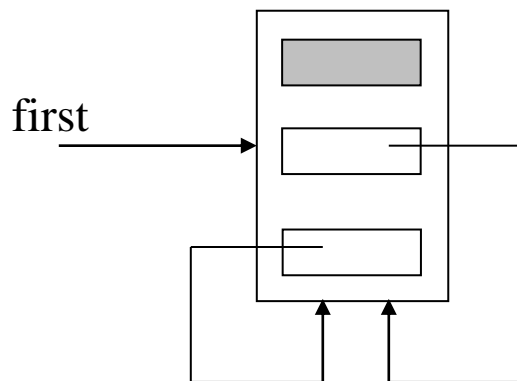
    public Nodo() {
        dato = null;
        next = null;
    }
    public Nodo( int dato){
        this.dato = dato;
        next = null;
    }
    ...
}
```



### c) Caso de lista con doble nexo circular y con nodo de encabezamiento



### Caso de lista con doble nexo vacía con nodo header



Se cumple:

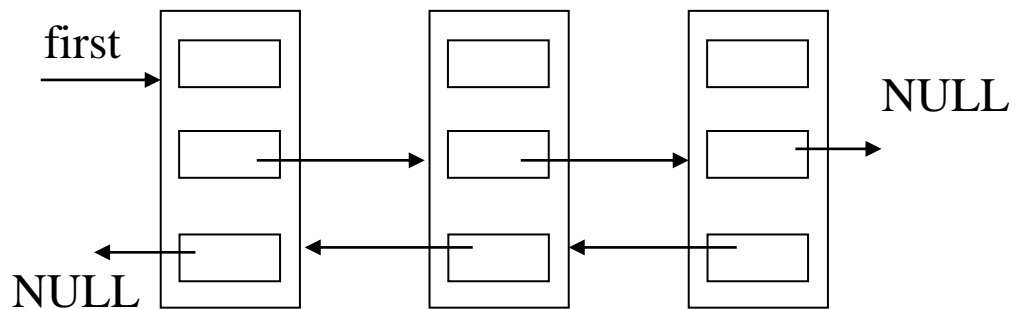
`first.getNext() == first`

`first.getPrevio() == first`

```
public class ListaDobleNexoHeader {
    private Nodo first;
```

```
    public ListaDobleNexoHeader() {
        Nodo n = new Nodo();
        first = n;
        first.setNext(n);
        first.setPrevio(n);
    }
    ...
}
```

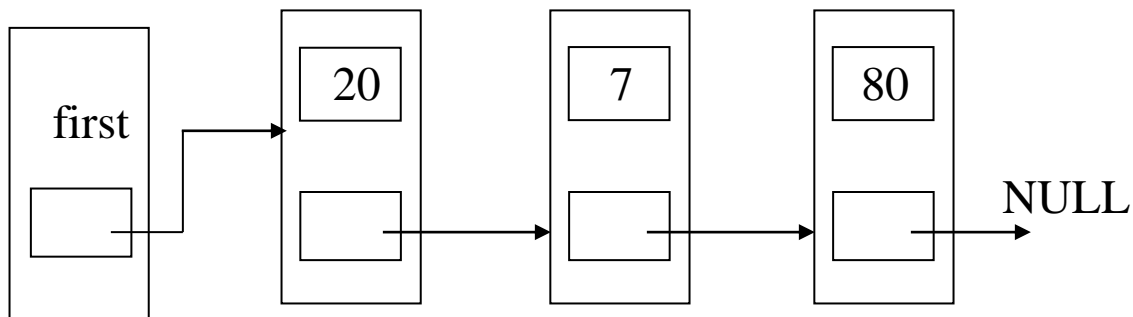
**d) Variación al esquema anterior:** Lista con doble nexo no circular, con o sin nodo de encabezamiento.



**Ejercicio:** Invertir una cadena.

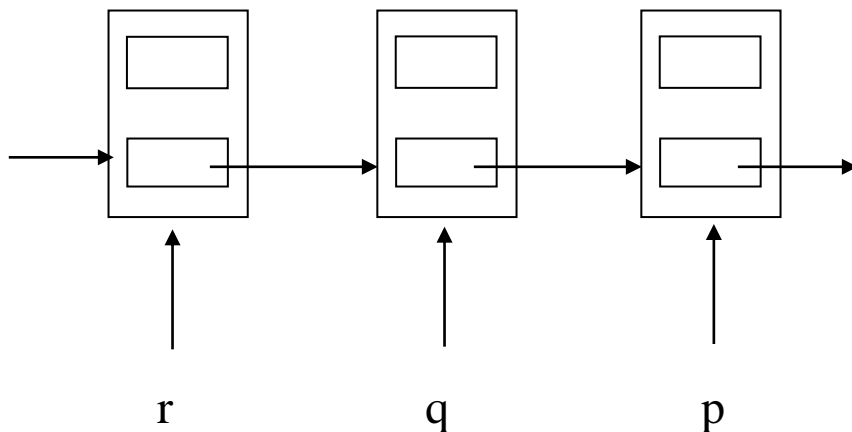
**Cadena:** Lista con nexos en que el último nodo apunta a NULL.

**Datos:** Lista  $X = (a_1, a_2, a_3, \dots, a_m)$



**Resultado:** Lista  $X = (a_m, a_{m-1}, \dots, a_2, a_1)$

Considerar, que para invertir, se puede hacer que  $q$  apunte a  $r$ , es decir,  $q.setNext(r)$



```

public class Nodo{
    private int dato;
    private Nodo next;

    public Nodo(int d){
        dato = d;
        next = null;
    }
    //Métodos get y set
}

```

```

public class Lista {
    ...
    public void invertir(){
        Nodo p, q, r;
        p = first;
        q = null;
        while ( p != null ) {
            r = q;
            q = p;
            p = p.getNext();
            q.setNext(r);
        }
        first = q;
    } //fin invertir
    ...
}

```

```

Lista x;
.....
.....
x.invertir();
.....
.....

```

} **Programa principal**

## Ejercicios propuestos:

- 1) Método, que permita insertar el nodo referenciado por x, al frente de la lista circular. first, referencia al último nodo de la lista.
- 2) Método, que permita calcular la longitud de una lista circular.
- 3) Método que permita concatenar la cadena y, a la cadena actual.

Es decir:

Programa Principal

```

Lista x, y;
.....
.....
x.concatenar(y);
.....
.....

```

Si:

$$x = (a_1, a_2, a_3, \dots, a_m)$$

$$y = (b_1, b_2, b_3, \dots, b_n)$$

$$m, n \geq 0$$

**Resultado es:**

$$X = (a_1, a_2, a_3, \dots, a_m, b_1, b_2, b_3, \dots, b_n)$$

## Ejemplo de aplicación

### Suma de polinomios

Representación de polinomios:

$$A(x) = a_m x^{e_m} + \dots + a_1 x^{e_1}$$

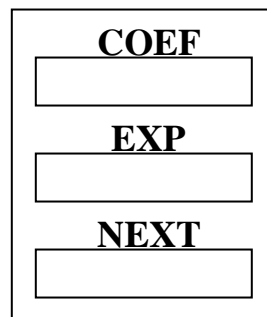
donde:

$a_i$  , coeficientes distintos de cero.

$e_i$  , exponentes

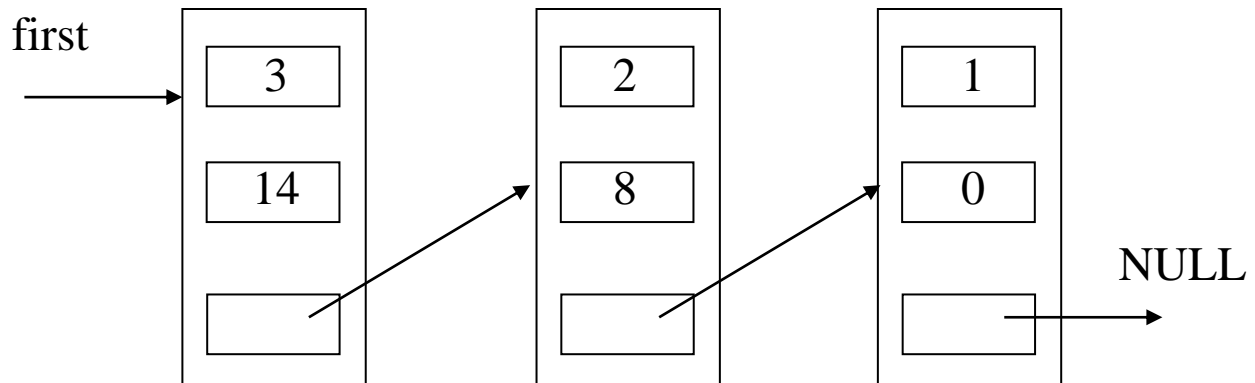
$$e_m > e_{m-1} > \dots > e_2 > e_1 \geq 0$$

Cada término se representa

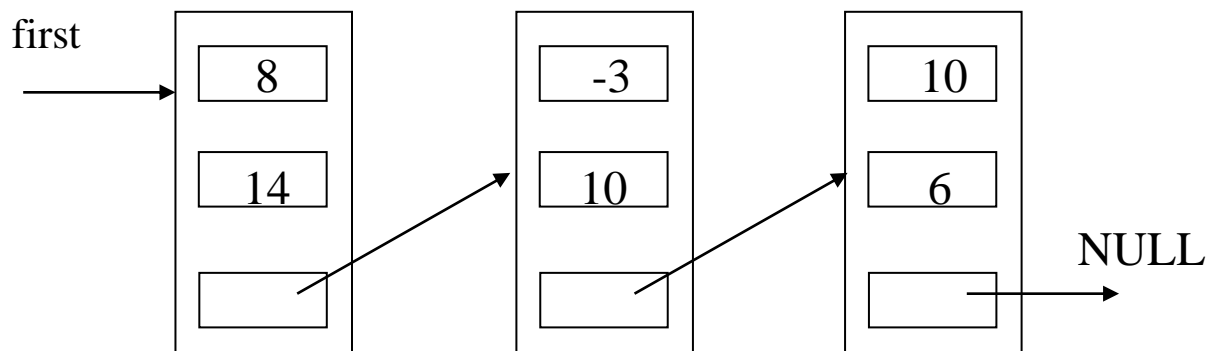


Por ejemplo:

$$A(x) = 3x^{14} + 2x^8 + 1$$



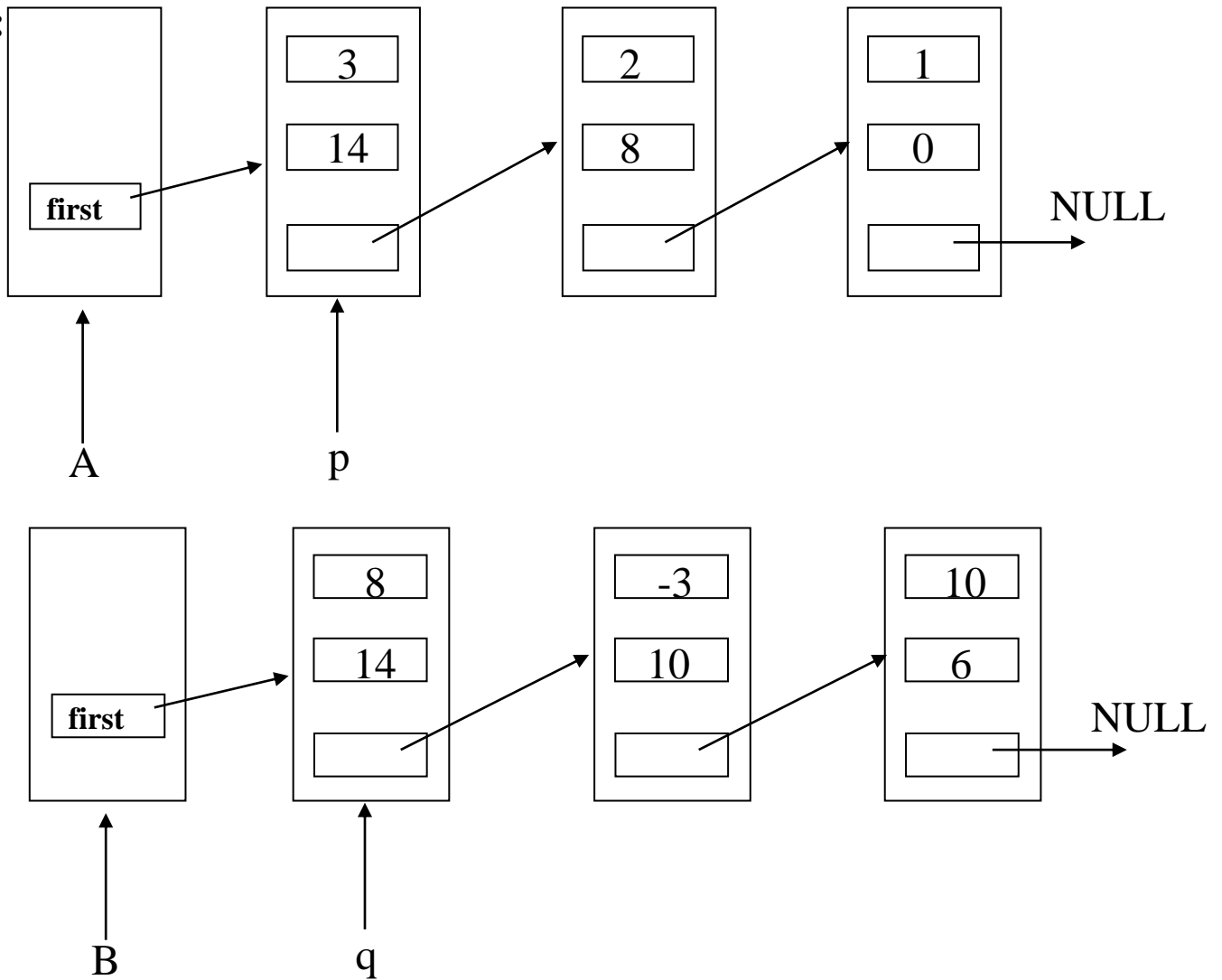
$$B(x) = 8x^{14} - 3x^{10} + 10x^6$$



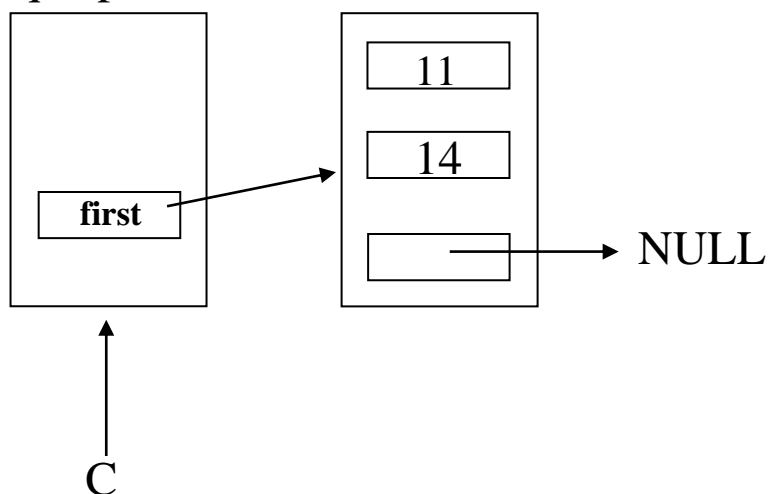
**Obtener la suma de A y B:**  $C = A + B$

### Pasos necesarios para la suma de polinomios

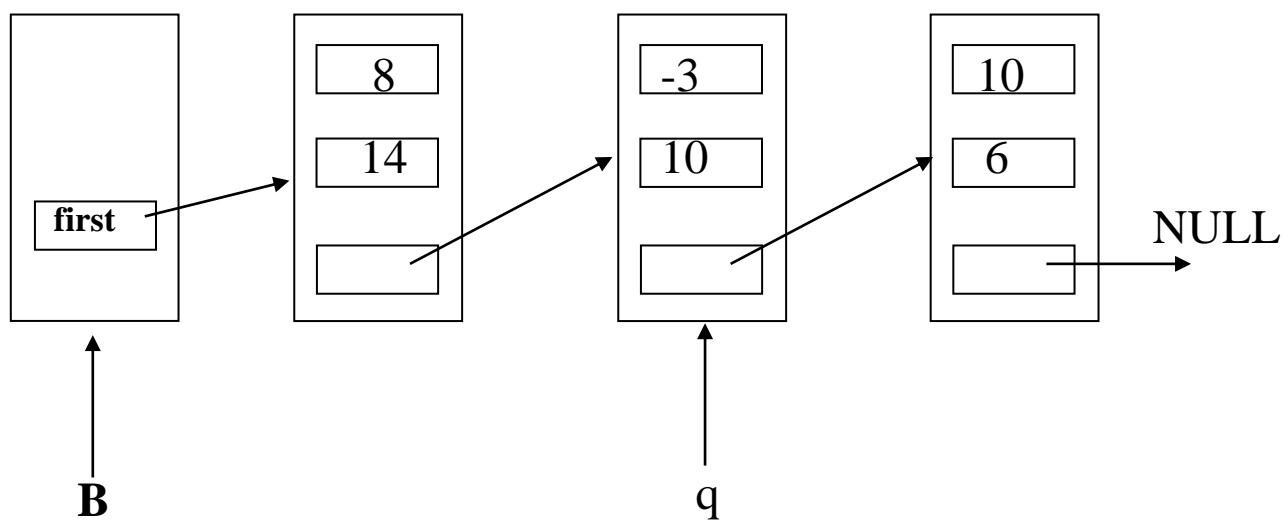
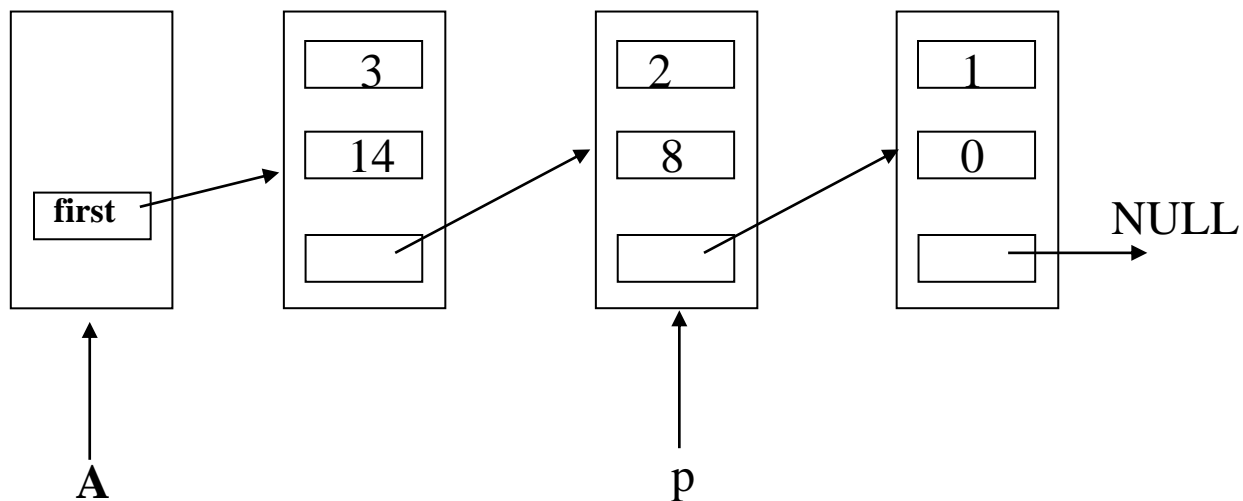
**Paso 1:**



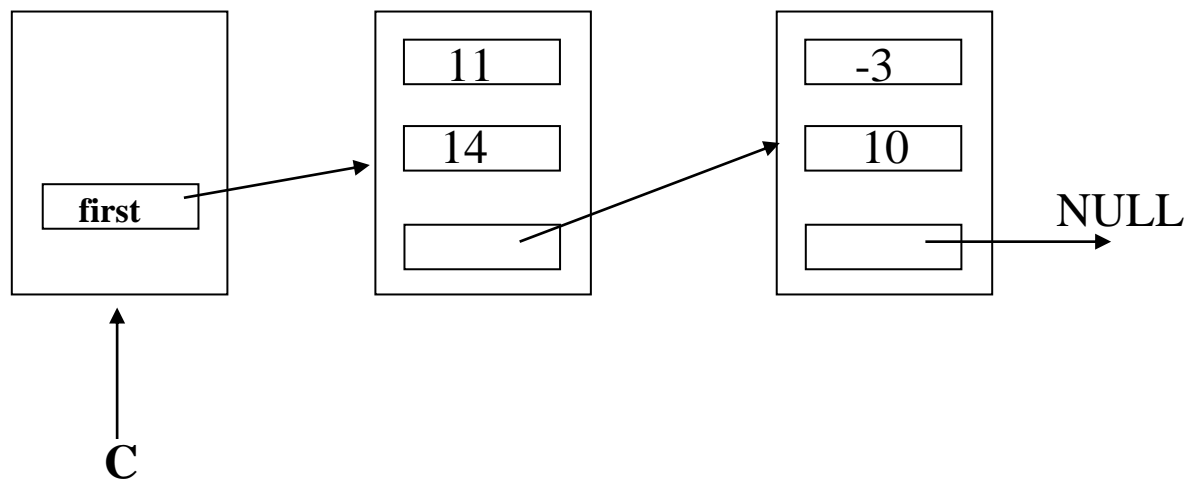
Como:  $p.exp = q.exp$

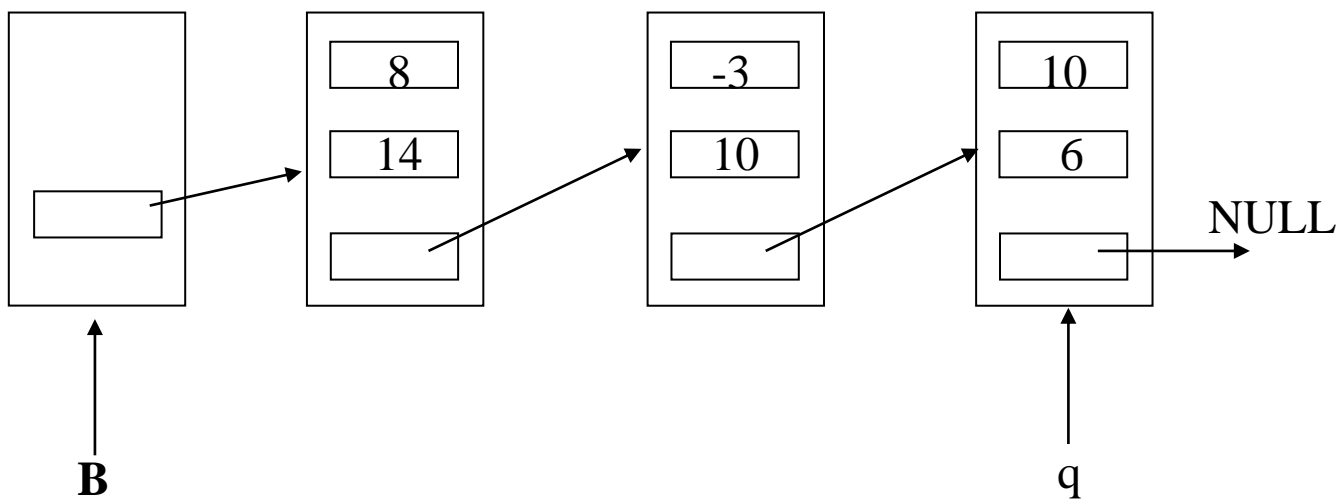
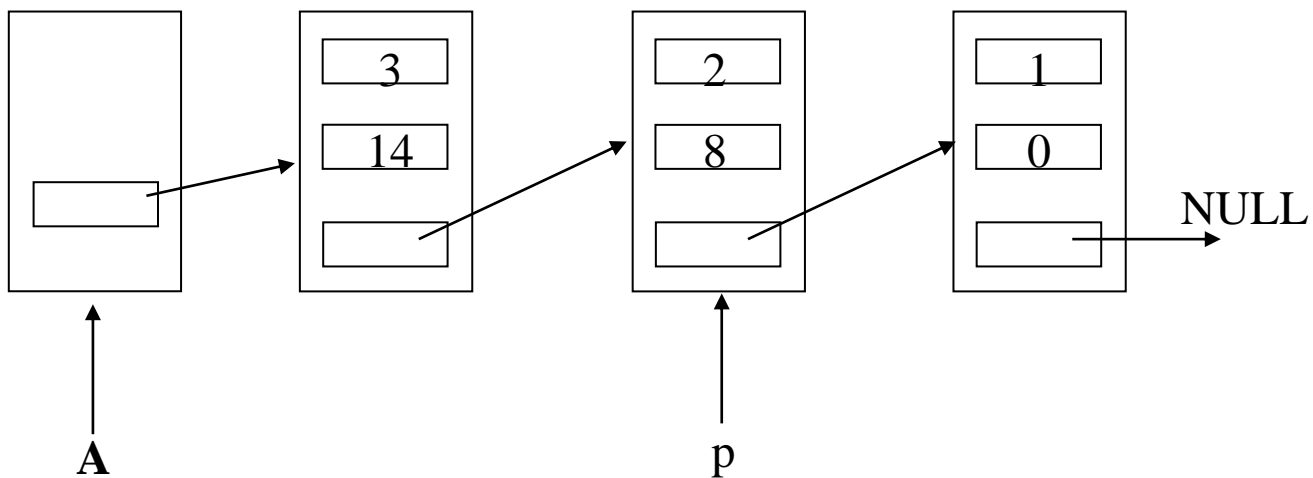


**Paso 2:**

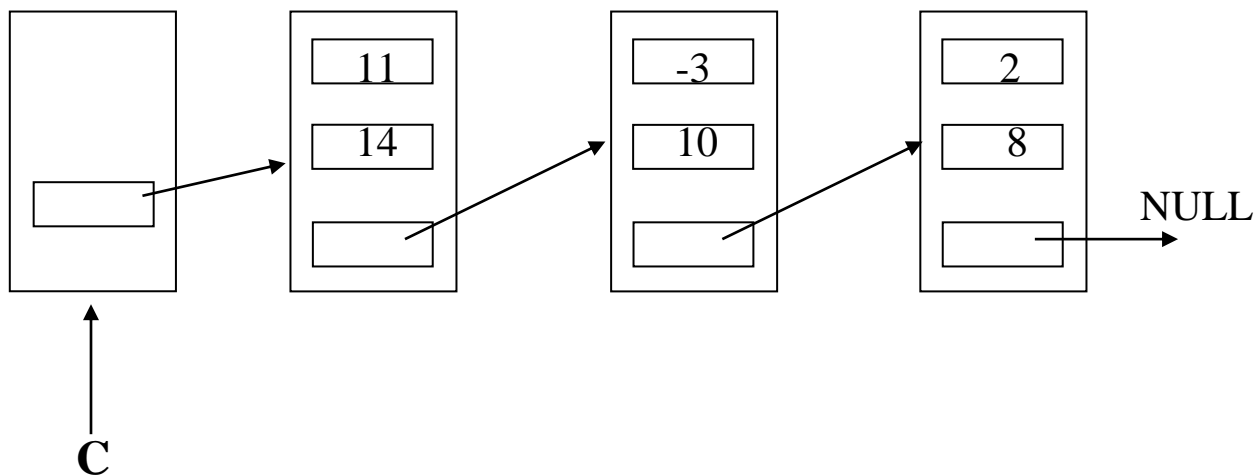


Como:  $p.exp < q.exp$



**Paso 3:**

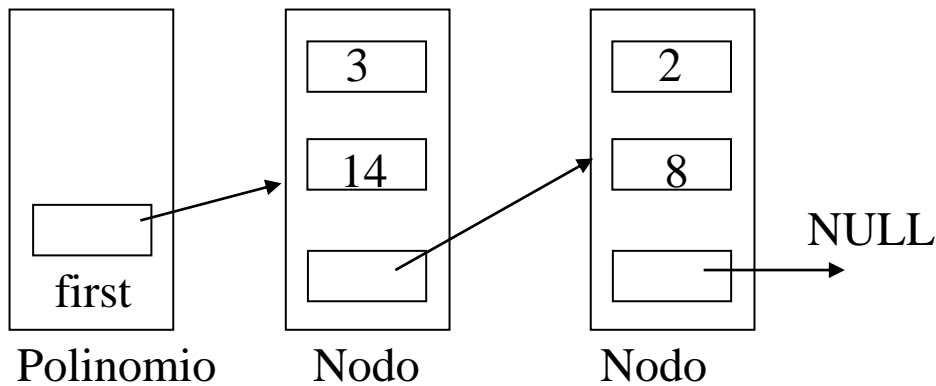
Como:  $p.exp > q.exp$





## Clases a considerar

- Polinomio
- Nodo
- Término



```

public class Termino {
    private int coeficiente;
    private int exponente;

    public Termino(int coeficiente, int exponente) {
        this.coeficiente = coeficiente;
        this.exponente = exponente;
    }
    //Métodos get y set
}
  
```

```

public class Nodo {
    private Termino termino;
    private Nodo next;

    public Nodo(int coeficiente, int exponente) {
        term = new Termino(coeficiente, exponente);
        next = null;
    }
    //Métodos get y set
}
  
```

Considerar que cada vez que se genera un nuevo nodo, éste se agrega al final de la lista resultado (**método ingresarTermFinal**). Para evitar buscar el último nodo del resultado, cada vez que se agrega un nuevo nodo a la lista, se utiliza una referencia, que direcciona al último nodo actualmente existente en el resultado (last).

|  |  |
|--|--|
| <pre> <b>public class Polinomio {</b>     private Nodo first;     private Nodo last;      <b>public Polonomio ( ){</b>         first = null;         last = null;     }      <b>public Polinomio suma(Polinomio polinomio ) {</b>         .....         .....     }  <b>//Fin clase Polinomio</b> </pre> | <pre>         <b>public void ingresarTermFinal(Termino termino)</b>             Nodo nodo = new Nodo (termino);             if (first == null){                 first = nodo;                 last = nodo;             }             else{                 last.setNext(nodo);                 last = nodo;             }         }     } </pre> |
|--|--|

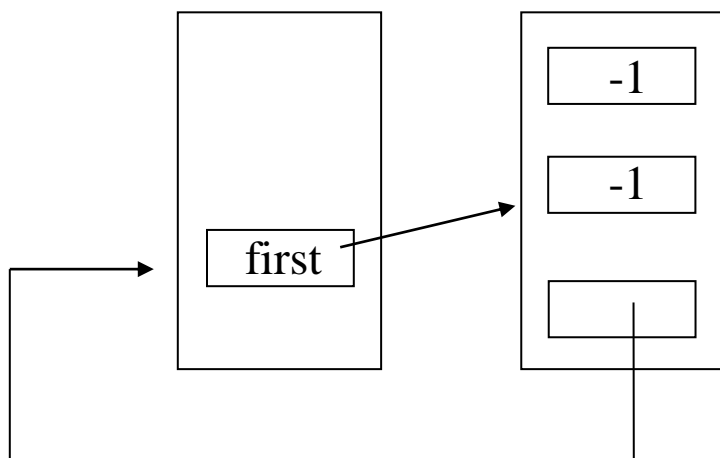
En el programa principal:

```

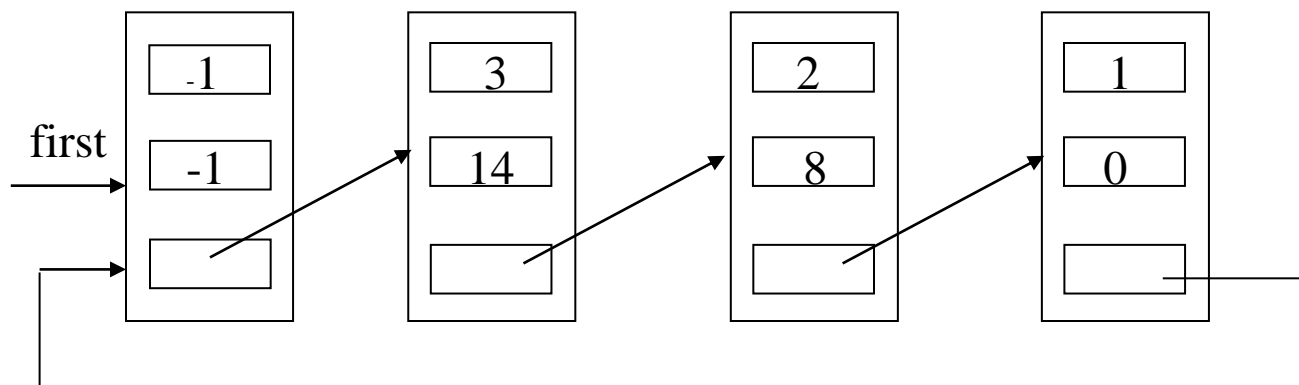
    Polinomio a, b, c;
    .....
    .....
    c = a.suma(b);
    .....
    .....

```

**Ejercicio:** Se puede representar a los polinomios como listas circulares con un nodo de encabezamiento.



**Caso de:**  $3 X^{14} + 2 X^8 + 1$



## Ejercicio Propuesto 1

Una asignatura es inscrita por muchos alumnos y a la vez, un alumno inscribe varias asignaturas. Existe una restricción por parte de la Jefatura de Carrera, que prohíbe a los alumnos inscribir más de 5 asignaturas simultáneas.

Los alumnos se caracterizan por un rut y un nombre.

Las asignaturas se clasifican como *normales* o *electivas*. Las asignaturas normales poseen un código, un nombre, cantidad de créditos asociados, nivel en que se dicta, y total de alumnos inscritos. Por otra parte, las asignaturas electivas poseen un código ficticio, un código real, un nombre, el nombre del departamento que la dicta y el total de alumnos inscritos.

Se solicita implementar las clases que permitan responder:

- Dado el código de una asignatura, mostrar el rut y nombre de todos los alumnos que la han inscrito, ordenados por rut de menor a mayor.
- Dado el rut de un alumno, mostrar el código real y el nombre de todas las asignaturas que ha inscrito.

Suponer que el ingreso de datos se realiza de la siguiente forma:

### Cantidad de asignaturas

Tipo de la asignatura 1

Datos de la asignatura 1

Datos de los alumnos que inscriben la asignatura 1

Tipo de la asignatura 2

Datos de la asignatura 2

Datos de los alumnos que inscriben la asignatura 2

.....

Considere que las listas se implementan con nexos. La lista de asignaturas con un nexo, circular y con una referencia al último elemento. La lista de alumnos con doble nexo, una referencia al primer elemento y una al último. Se pide: modelo del dominio, contratos, diagrama de clases del dominio de la aplicación, diagrama de clases y código

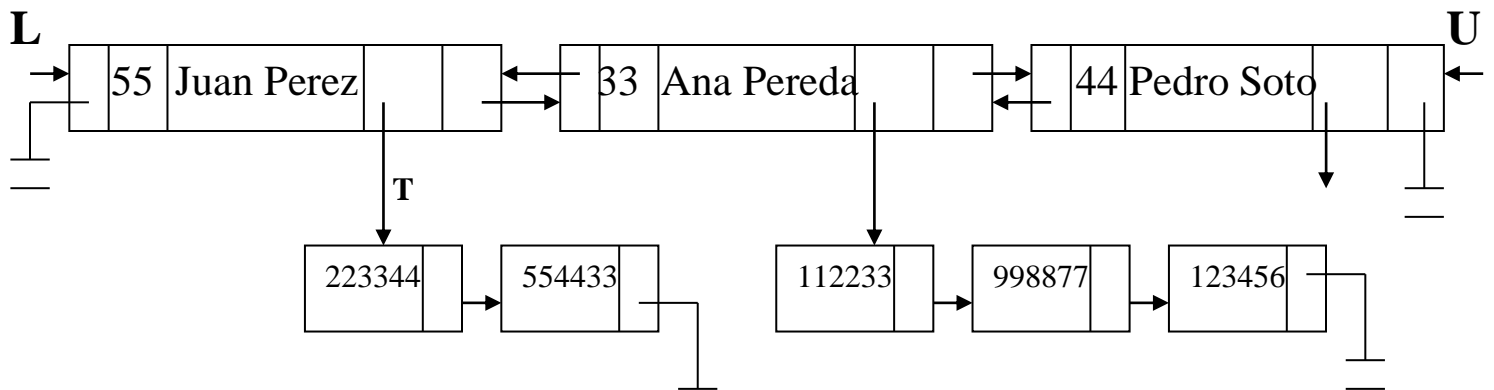
## Ejercicio Propuesto 2

Suponga que tiene información de los empleados de una empresa antofagastina, que considera el nombre, rut del empleado y todos sus teléfonos.

Se pide:

- Modelo del dominio, contratos, diagrama de clases del dominio de la aplicación, diagrama de clases y código.
- La aplicación debe:
  - Leer la información desde pantalla y cargue la estructura que se muestra a continuación. La información, para cada empleado, viene de la siguiente manera. (Fin de datos: rut = 111.):

rutEmpleado, nombreEmpleado, cantidadDeTelefonos  
 numeroTelefónico  
 ....  
 numeroTelefonico



- Ingresar un teléfono para un empleado. Debe leer el rut del empleado y el número telefónico. Si el empleado no está, no se debe ingresar el número telefónico.
- Eliminar un teléfono para un empleado. Debe leer el rut del empleado y el número telefónico.
- Eliminar a todos los empleados que no tienen teléfonos.

## 3.2 Contenedores implementados en Java (ArrayList y LinkedList)

Dentro de Java encontramos el paquete **java.util** el cual es uno de los paquetes más utilizados dentro del desarrollo de aplicaciones Java, ya que provee numerosas interfaces, así como también, un sin número de clases. Las clases **ArrayList** y **LinkedList** implementan la interface **List**. La interface **List** hereda desde la interface **Collection**, que es la raíz de todas las interfaces relacionadas con colecciones de elementos

Algunos métodos de la **interface List** son:

- **add(Element el)**: Inserta un elemento al final de la lista, retorna “true” si la inserción ha sido exitosa.
- **add(int index, Object el)**: Inserta un elemento en la posición dada. La primera posición es la 0.
- **clear()**: Elimina todos los elementos de la lista.
- **equals(Object ob)**: Compara dos objetos, retorna “true” si los objetos son los mismos y falso en caso contrario. Evidentemente estos objetos pueden ser listas y en este caso las listas pueden ser distintas, pero contener a los mismos objetos, por lo que resultaría true.
- **get(int index)**: Retorna un elemento de la posición indicada.
- **remove(int index)**: Elimina un elemento de la lista dada una posición, retorna “true” si la eliminación ha sido exitosa.
- **remove(Object ob)**: Elimina el objeto recibido por parámetro, retorna “true” si la eliminación ha sido exitosa.
- **contains(Object o)**: Retorna verdadero si la lista contine el objeto o.
- **indexOf(Object o)**: Retorna el índice de la primera ocurrencia del elemento especificado en la lista. Si la lista no lo contiene retorna un -1.

Al ser las clases **ArrayList** (lista implementada con arreglos) y **LinkedList** (lista implementada con nodos) “implementaciones” de la interface **List**, éstas poseen los mismos métodos que define dicha interface, la diferencia radica en que su implementación interna es diferente (como se ha visto en los capítulos anteriores)

**ArrayList** y **LinkedList** son listas genéricas. Se les puede colocar cualquier cosa, cualquier tipo de datos, pero se debe definir que va a contener. Se considera a la lista como un contenedor de propósito general. No se conoce el tipo de objeto que almacena, por lo que el tipo de los elementos es **OBJECT**. Un valor declarado como **OBJECT** es polimórfico, es decir, almacena valores de cualquier clase.

Cabe mencionar, que deben ser objetos, no tipos primitivos, por ejemplo como `int`.

### Ejemplo: manejo de ArrayList y LinkedList

```
public class Alumno{

    private String nombre;
    private int edad;
    public Alumno(String nombre, int edad){
        this.nombre = nombre;
        this.edad = edad;
    }
    //get y set ...
}
```

### Diagrama de clases

| Alumno                         |
|--------------------------------|
| - String nombre<br>- int edad  |
| + Alumno()<br>+ get y set()... |

```
import java.util.ArrayList;
import java.util.List;
public class App {
    public static void main(String[] args) {
        //Interface List implementada con ArrayList
        List <Alumno> listaAlumnos = new ArrayList<Alumno>();
        //agrego alumnos a la lista
        Alumno alumno1 = new Alumno("María",18);
        listaAlumnos.add(alumno1);
        Alumno alumno2 = new Alumno("Pedro",20);
        listaAlumnos.add(alumno2);
        //forma simplificada
        listaAlumnos.add(new Alumno("Pía",19));
        listaAlumnos.add(new Alumno("Juan",22));
    }
}
```

... = *new LinkedList<Alumno>()*;



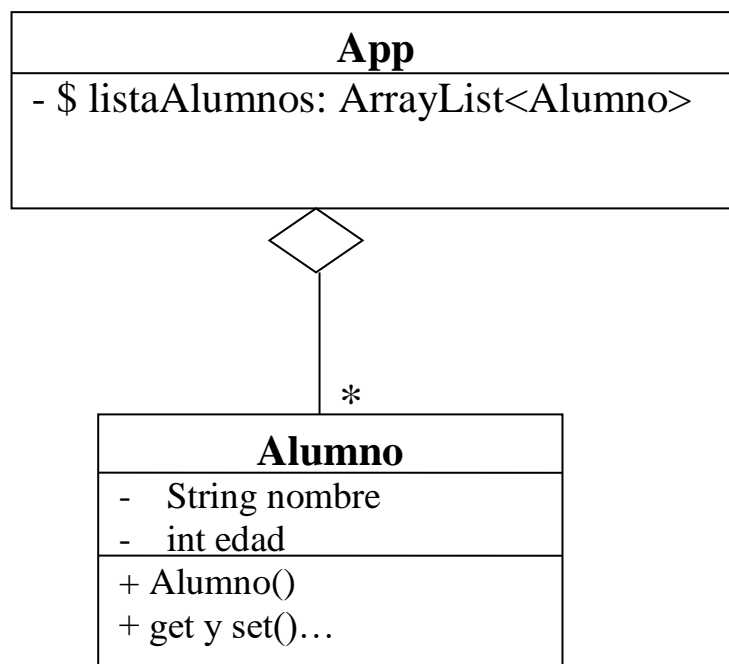
Si el código anterior fuese:

```
...
public class App {

    private static listaAlumnos = new ArrayList<Alumno>();
    //listaAlumnos es una variable de clase

    public static void main(String[] args) {
        //agrego alumnos a la lista
        Alumno alumno1 = new Alumno("María",18);
        listaAlumnos.add(alumno1);
        ...
    }
}
```

El diagrama de clases asociado sería:



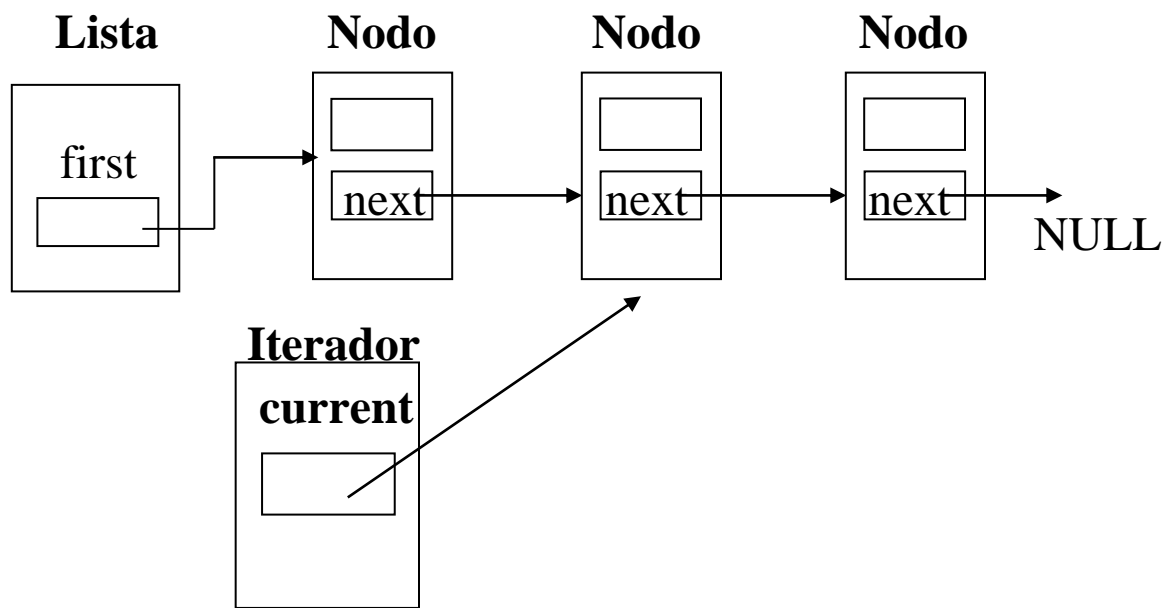


### 3.3 Iteradores

En el ejemplo anterior se ha utilizado la clase **ArrayList** incluida dentro del paquete **java.util** (import) en conjunto con la interface **List** para crear una lista de alumnos, ahora bien ¿cómo se podrían obtener los elementos (objetos) agregados a la lista? Para ello existe el **Iterador**.

Un iterador, cómo su nombre lo indica, es un objeto que opera sobre una colección de elementos (como una lista) seleccionando uno a la vez de manera secuencial. El objeto iterador esconde los detalles de la representación de la lista y proporciona una interface simple para acceder los elementos de ésta, uno por uno.

Los iteradores proporcionan control al usuario, respecto a los datos que se van accediendo, al recorrer la lista. El objeto iterador tiene aislada las operaciones para recorrer una lista.



Al igual que las clases **ArrayList** y **LinkedList** definidas dentro del paquete **java.util**, existe la interface **Iterator** la cual define los siguientes métodos:

- **hasNext():** retorna “true” si la iteración tiene más elementos.
- **next():** retorna el siguiente elemento en la iteración.
- **remove():** remueve el último elemento retornado por el iterador (posición actual del iterador en la iteración)

*El remove se debe usar cuando se trabaja con el iterador en forma explícita*

```

public interface Iterator {
    // Retorna true si hay un próximo elemento.
    boolean hasNext();

    // Obtiene el próximo elemento, si este no existe se
    // levanta la excepción "NoSuchElementException"
    Object next();
    ....
}

```

### // Ejemplo 1: Muestra el uso de los iteradores

```

public class App {

    public static void main(String[] args) {

        List <Alumno> lista1Alumnos = new LinkedList<Alumno>();
        lista1Alumnos.add(new Alumno("María",18));
        lista1Alumnos.add(new Alumno("Pedro",20));
        lista1Alumnos.add(new Alumno("Diego",19));
        lista1Alumnos.add(new Alumno("Tere",22));
    }
}

```

*El ciclo for tiene definido implícitamente la marca de fin de datos (hasNext) dentro de la sintáxis del cuerpo del ciclo*

```

//Utilizacion de un for de iterador
//El iterador no está creado en forma explícita
for(Alumno alumno: lista1Alumnos){
    StdOut.println("Alumno: "+alumno.getNombre());
}

```

*Cuando estamos con el for del iterador, no se puede trabajar con el remove del iterador, se cae:*

```

    lista1Alumnos.remove(); //SE CAE

```

*ya que estamos recorriendo y modificando la lista a la vez. Se debe usar un iterador explícito*

```

}
//Otra lista
ArrayList<Alumno> lista2Alumnos= new ArrayList<Alumno>();
lista2Alumnos.add(new Alumno("María",18));
lista2Alumnos.add(new Alumno("Pedro",20));
lista2Alumnos.add(new Alumno("Diego",19));
lista2Alumnos.add(new Alumno("Tere",22));

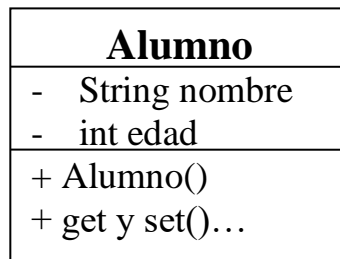
```

**//Uso del iterador**

```

Iterator<Alumno> it = lista2Alumnos.iterator();
while (it.hasNext()) {
    Alumno alumno = (Alumno) it.next();
    //next() devuelve un dato de tipo Object.
    //Hay que convertirlo a Alumno
    StdOut.println("Alumno: "+alumno.getNombre());
}

```

**Diagrama de clases**

*Dado que el ietrador se creó explícitamente, si se puede trabajar con el remove del iterador*

**it.remove()**

*Se sabe exactamente en que elemento va el iterador y ese es el que se elimina*

```

} //Fin main

```

```

} //Fin App

```

Si el código fuese

```

public class App {

```

```

    private static lista1Alumnos = new LinkedList<Alumno>();
    private static lista2Alumnos = new ArrayList<Alumno>();

```

```

    public static void main(String[] args) {

```

```

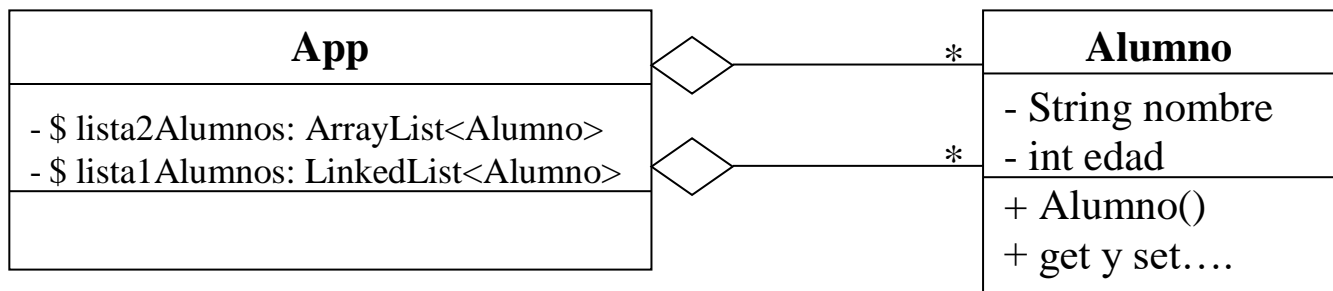
        lista1Alumnos.add(new Alumno("María",18));

```

```

        ....
    }
}

```

**Diagrama de clases**

## Ejemplo 2

## Diagrama de clases

```
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;
import ucn.Stdout;
```

```
public class App {
```

| Alumno           |
|------------------|
| - String nombre  |
| - int edad       |
| + Alumno()       |
| + get y set()... |

```
    public static void desplegarLista(List<Alumno> lista) {
        StdOut.println("");
        StdOut.println("Despliegue de la lista");
        Iterator<Alumno> it = lista.iterator();
        while (it.hasNext()) {
            Alumno alumno = (Alumno) it.next();
            //next() devuelve un dato de tipo Object.
            //Hay que convertirlo a Alumno
            StdOut.println("Alumno: "+alumno.getNombre());
        }
    }
```

```
    public static void main(String[] args) {
        List <Alumno> lista = new ArrayList <Alumno>();
        lista.add(new Alumno("juan"));
        Alumno alumno1 = new Alumno("pedro");
        lista.add(alumno1);
        lista.add(0, new Alumno("maria"));
        Alumno alumno2 = new Alumno("pedro");
        lista.add(alumno2);
        desplegarLista(lista);

        Alumno alumno = lista.get(0);
        StdOut.println("");
        StdOut.println("Primer alumno de la lista " +
                        alumno.getNombre());

        StdOut.println("");
        StdOut.println("Se elimina el elemento de la posicion 1
                        de la lista");

        lista.remove(1);

        desplegarLista(lista);
    }
```

```

StdOut.println("");
StdOut.println("Posicion donde esta alumno1 en lista: " +
               lista.indexOf(alumno1));

//Se elimina de la lista el objeto alumno1
StdOut.println("");
StdOut.println("Se quiere eliminar de la lista el
               objeto alumno1");

if(lista.remove(alumno1)){
    StdOut.println("Se elimino alumno1");
}
else{
    StdOut.println("No se elimino el alumno1,
                  porque no esta en la lista");
}
desplegarLista(lista);

StdOut.println("");
StdOut.println("Posicion donde esta alumno1 en lista: " +
               lista.indexOf(alumno1));

//Se elimina de la lista el objeto alumno1
StdOut.println("");
StdOut.println("Se quiere eliminar de la lista el
               objeto alumno1");

if(lista.remove(alumno1)){
    StdOut.println("Se elimino alumno1");
}
else{
    StdOut.println("No se elimino el alumno1,
                  porque no esta en la lista");
}
desplegarLista(lista);

//Se eliminan todos los elementos de la lista
StdOut.println("");
StdOut.println("Se eliminan todos los elementos de la
               lista");

lista.clear();

desplegarLista(lista);

StdOut.println("");

```

```

    if (alumno1.equals(alumno2)) {
        StdOut.println("alumno1 y alumno2 son el
                        mismo objeto");
    }
    else{
        StdOut.println("alumno1 y alumno2 son
                        distintos objetos");
    }

    List <Alumno> lista1 = new ArrayList <Alumno>();
    lista1.add(alumno1);

    List <Alumno> lista2 = new ArrayList <Alumno>();
    lista2.add(alumno1);

    StdOut.println("");
    if(lista1.equals(lista2)){
        StdOut.println("lista1 y lista2 son listas =");
    }
    else{
        StdOut.println("lista1 y lista2 son listas diferentes");
    }

    List <Alumno> lista3 = new ArrayList <Alumno>();
    lista3.add(new Alumno("pedro"));

    List <Alumno> lista4 = new ArrayList <Alumno>();
    lista3.add(new Alumno("pedro"));

    StdOut.println("");
    if(lista3.equals(lista4)){
        StdOut.println("lista3 y lista4 son listas =");
    }
    else{
        StdOut.println("lista3 y lista4 son listas diferentes");
    }

} //Fin main

} //Fin App

```

## Resultados

Despliegue de la lista

Alumno: maria

Alumno: juan

Alumno: pedro

Alumno: pedro

Primer alumno de la lista maria

Se elimina el elemento de la posicion 1 de la lista

Despliegue de la lista

Alumno: maria

Alumno: pedro

Alumno: pedro

Posicion donde esta alumno1 en lista: 1

Se quiere eliminar de la lista el objeto alumno1

Se elimino alumno1

Despliegue de la lista

Alumno: maria

Alumno: pedro

Posicion donde esta alumno1 en lista: -1

Se quiere eliminar de la lista el objeto alumno1

No se elimino el alumno1, porque no esta en la lista

Despliegue de la lista

Alumno: maria

Alumno: pedro

Se eliminan todos los elementos de la lista

Despliegue de la lista

alumno1 y alumno2 son distintos objetos

lista1 y lista2 son listas =

lista3 y lista4 son listas diferentes

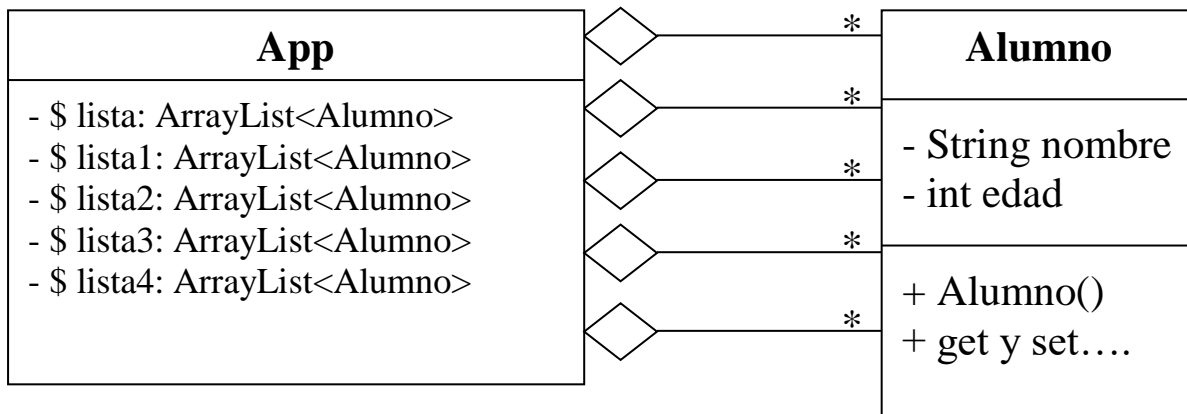
Si el código fuese:

```
public class App {

    private static lista = new ArrayList <Alumno>();
    private static lista1 = new ArrayList <Alumno>();
    private static lista2 = new ArrayList <Alumno>();
    private static lista3 = new ArrayList <Alumno>();
    private static lista4 = new ArrayList <Alumno>();

    public static void main(String[] args) {
        lista.add(new Alumno("juan"));
        Alumno alumno1 = new Alumno("pedro");
        lista.add(alumno1);
        ...
    }
}
```

El diagrama de clases asociado sería





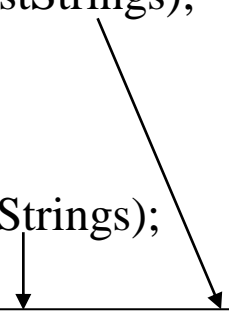
### 3.4. Ordenando una lista

La forma más simple es utilizando el método estático **Collections.sort()**, que ordena la lista especificada en orden ascendente.

#### Ejemplo 1

- **Código**

```
1 List<String> listStrings = new ArrayList<String>();
2 listStrings.add("D");
3 listStrings.add("C");
4 listStrings.add("E");
5 listStrings.add("A");
6 listStrings.add("B");
7
8 System.out.println("listStrings before sorting: " + listStrings);
9
10 Collections.sort(listStrings);
11
12 System.out.println("listStrings after sorting: " + listStrings);
```



*La lista se imprime entre corchetes, donde cada elemento está separado por “,”*

- **Resultados**

listStrings before sorting: [D, C, E, A, B]

listStrings after sorting: [A, B, C, D, E]

Notar que todos los elementos en la lista deben implementar la interface Comparable, de manera que si define su propio tipo debe asegurarse de implementar esa interface y su método compareTo().

## Ejemplo 2

- **Código**

```
public class Employee implements Comparable<Employee> {
    private String name;
    private int age;
    private int salary;
    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    // getters and setters

    @Override
    //El método compareTo es de la interface Comparable
    public int compareTo(Employee employee){
        //Se ordenara por sueldo
        return employee.salary - this.salary;
    }
    public String toString() {
        return String.format("(%s, %d, %d)", name, age, salary);
    }
}
```

### Parte de la App

```
List<Employee> listEmployees = new ArrayList<Employee>();
listEmployees.add(new Employee("Tom", 45, 80000));
listEmployees.add(new Employee("Sam", 56, 75000));
listEmployees.add(new Employee("Alex", 30, 120000));
listEmployees.add(new Employee("Peter", 25, 60000));
System.out.println("Before sorting: " + listEmployees);
Collections.sort(listEmployees);
System.out.println("After sorting: " + listEmployees);
```

### Resultados

Before sorting: [(Tom, 45, 80000), (Sam, 56, 75000), (Alex, 30, 120000), (Peter, 25, 60000)]

After sorting: [(Alex, 30, 120000), (Tom, 45, 80000), (Sam, 56, 75000), (Peter, 25, 60000)]

### 3.5. Ejercicio

Una de las características más usadas de un computador, es su gran capacidad de leer y procesar información. Un ejemplo de ello son los diccionarios, donde se puede rápidamente encontrar el significado de una palabra.

Se debe leer desde pantalla el significado de muchas palabras y su significado. Se lee la palabra y luego el significado. Fin de datos, palabra = 111. Las palabras vienen en cualquier orden

#### Ejemplos de algunos datos a leer:

|           |   |
|-----------|---|
| Imán      | Cuerpo que atrae al hierro, bien por naturaleza, bien por propiedades adquiridas. |
| Labadismo | Secta protestante pietista, fundada por Jean de Labadie.                          |
| Platicar  | Conversar, hablar unos con otros.   |
| Kimono    | Túnica larga y amplia usada en Japón.   |
| Pastaca   | Guiso de cerdo cocido con maíz.   |
| 111       |   |

Se desea construir una aplicación Java que almacene esta información del diccionario a través de las librerías de listas de Java.

Considere la siguiente **interface para el diccionario**:

- **public void insertarPalabra(String palabra, String significado)**  
Método que inserta una palabra en la estructura de datos.
- **public String buscarSignificado(String palabra)**  
Método que retorna el significado de una palabra, si no lo encuentra retorna “Sin Significado Asociado”.
- **public String listadoPalabras()**  
Método que entrega un String de todas las palabras del diccionario (separadas por coma) **ordenadas por orden alfabético**.
- **public String listadoPalabras(String letra)**  
Método que retorna un String de todas las palabras que comienzan con la letra.

## Se pide:

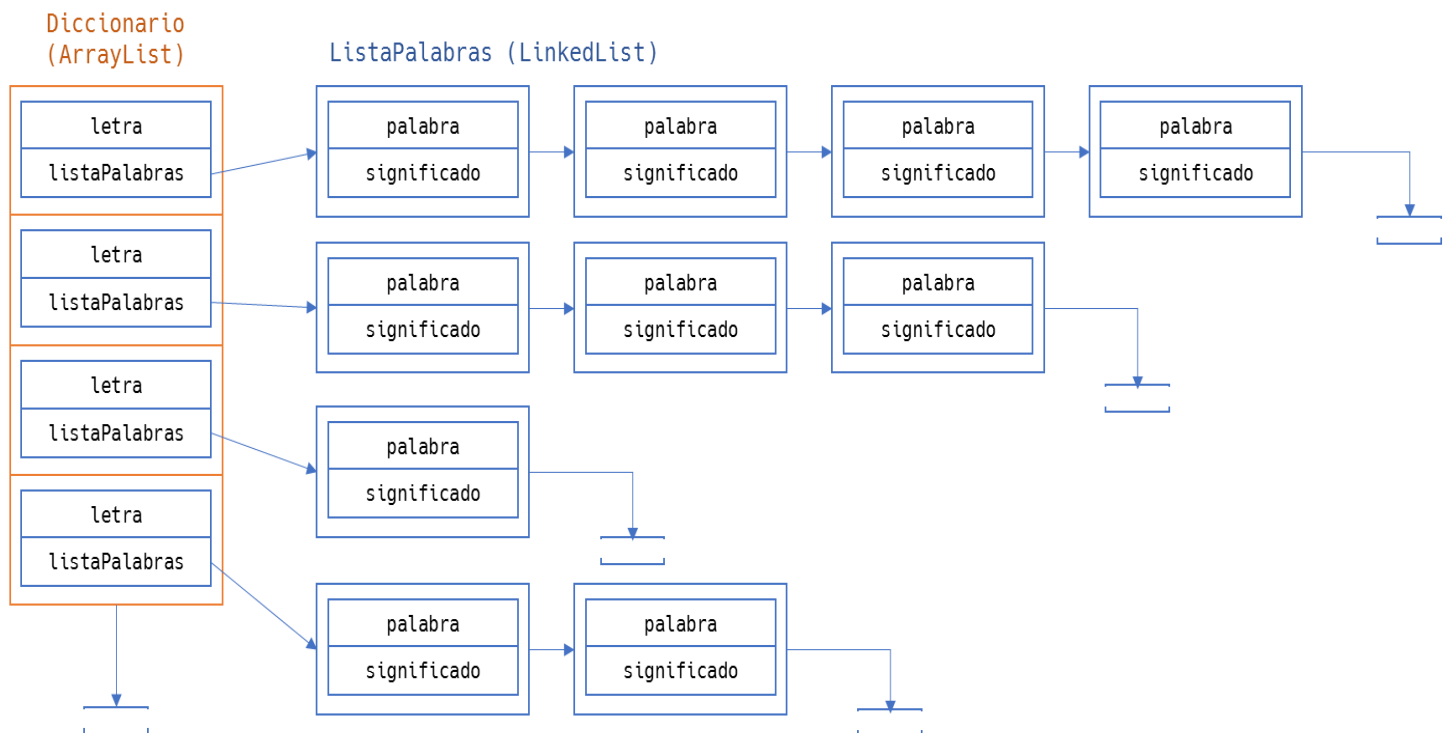
Dibuje el diagrama de clases y contruya el código Java para implementar el diccionario (todo el código menos la App).

## Nota:

- Utilice las Interfaces y Clases de Java para su código: **List**, **ArrayList**, **LinkedList** e **Iterator**.
- No es requisito que el diccionario se encuentre ordenado, sólo es necesario para el requerimiento listadoPalabras()
- Se tiene un método para un String llamado **substring** (valor inicial, valor final) que retorna el substring desde posición inicial a posición final del String

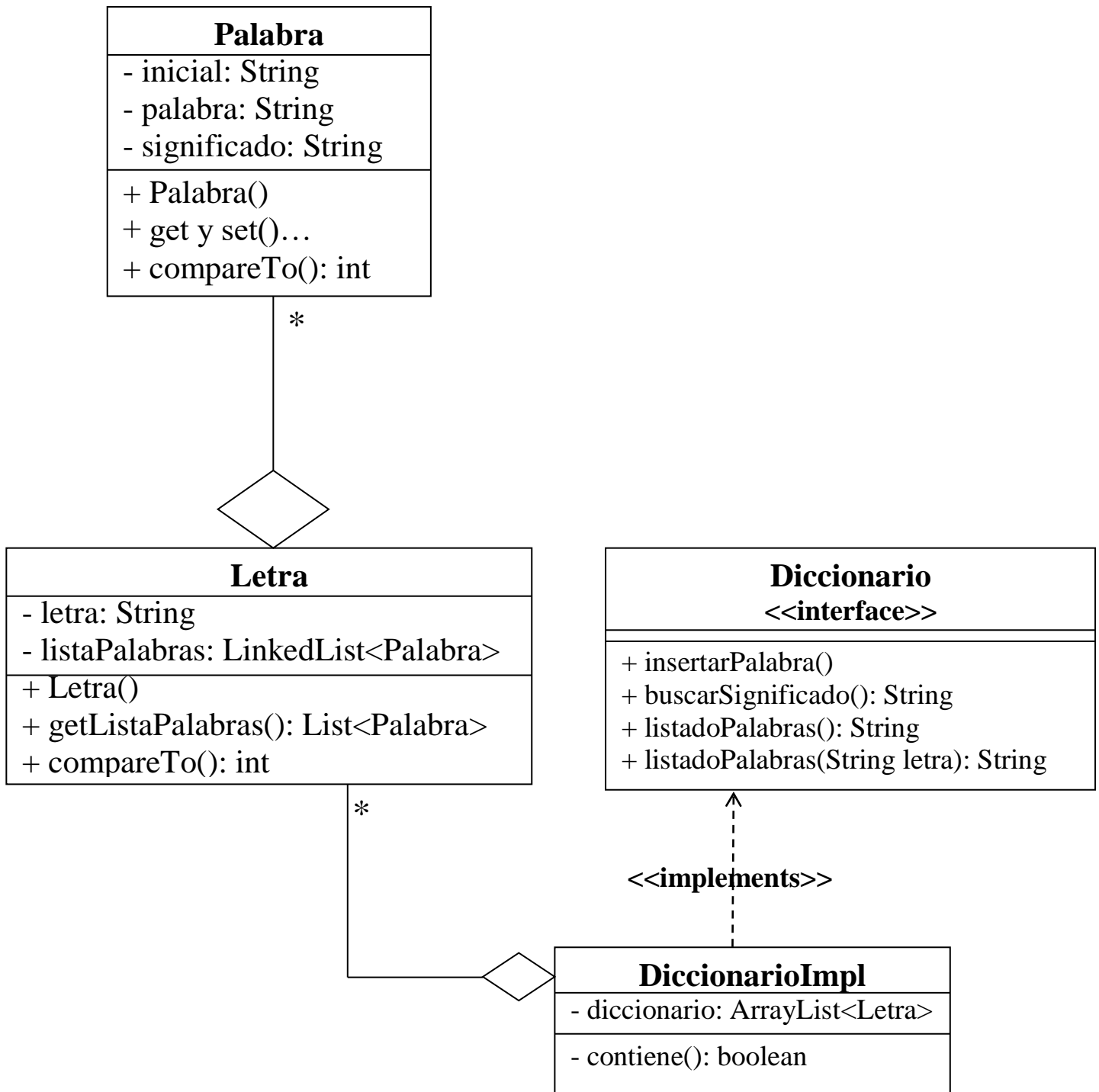
## Suponga:

- No se leen palabras repetidas
- La siguiente estructura para guardar los datos del diccionario



- **Letra**, clase que guarda una letra y una lista de palabras (LinkedList) de aquellas cuya inicial es letra.
- **Palabra**, clase que guarda la palabra y su significado.
- **Diccionario**, será un ArrayList de la clase Letra.

## Diagrama de clases



## Solución

```

import java.util.LinkedList;

public class Letra implements Comparable<Letra>{

    //Cada letra tiene una lista linkeada de cada una
    //de las palabras que comienzan con dicha letra
    private String letra;

    private LinkedList<Palabra> listaPalabras;
    //Idem lo anterior private List<Palabra> listaPalabras;

    public Letra(String letra){
        this.letra = letra;
        this.listaPalabras = new LinkedList<Palabra>();
    }

    public String getLetra() {
        return letra;
    }

    //public List<Palabra> getListaPalabras();
    public LinkedList<Palabra> getListaPalabras() {
        return listaPalabras;
    }

    /* (non-Javadoc)
     * @see java.lang.Comparable#compareTo(java.lang.Object)
     */
    @Override
    public int compareTo(Letra letra){
        return this.letra.compareTo(letra.getLetra());
    }
}

```

*Comparable es una interface que tiene solo el método compareTo*

```

public class Palabra implements Comparable<Palabra>{

    private String inicial;
    private String palabra;
    private String significado;

    //Cada palabra tiene la letra inicial, la palabra
    //propriadamente tal y su significado
    public Palabra(String palabra, String significado) {
        this.palabra = palabra;
        this.significado = significado;
        this.inicial = palabra.substring(0,1);
    }

    public String getSignificado() {
        return this.significado;
    }

    public void setSignificado(String significado) {
        this.significado = significado;
    }

    public String getPalabra() {
        return this.palabra;
    }

    public String getInicial(){
        return this.inicial;
    }

    /* (non-Javadoc)
    * @see java.lang.Comparable#compareTo(java.lang.Object)
    */
    @Override
    public int compareTo(Palabra palabra){
        return this.palabra.compareTo(palabra.getPalabra());
    }
}

```

```
public interface Diccionario {  
  
    /**  
     * Inserta una palabra en el diccionario  
     * @param palabra  
     * @param significado  
     */  
    public void insertarPalabra(String palabra, String significado);  
  
    /**  
     * @param palabra  
     * @return Significado de una palabra  
     */  
    public String buscarSignificado(String palabra);  
  
    /**  
     * @return Listado de todas las palabras del diccionario  
     */  
    public String listadoPalabras();  
  
    /**  
     * @param letra  
     * @return Listado de todas las palabras cuya inicial es letra  
     */  
    String listadoPalabras(String letra);  
}
```



```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
```

```
public class DiccionarioImpl implements Diccionario {
```

```
    /**
```

```
     * Variables de Clase
```

```
     * diccionario: Lista de letras (iniciales sólo de las palabras contenidas en el diccionario)
```

```
    */
```

```
private List<Letra> diccionario;
```

```
//Constructor del Diccionario (Vacio)
```

```
public DiccionarioImpl(){
```

```
    this.diccionario = new ArrayList<Letra>();
```

```
}
```

```
//Retorna verdadero si inicial está en el diccionario. Falso en caso contrario
```

```
//Es privado porque no es parte de la interface, lo usa un método de la interface
```

```
private boolean contiene(String inicial){
```

```
    //Variable resp en falso para retornar
```

```
    boolean resp = false;
```

```
    //Diccionario no vacío
```

```
    if(!this.diccionario.isEmpty()){
```

```
        //Tomamos sólo la primera letra de la entrada
```

```
        String inicialNuevaPalabra = inicial.substring(0, 1);
```

```
        //Recorremos el diccionario
```

```
        for(Letra letra: this.diccionario){ //Por cada elemento del diccionario
```

```
            //Si la letra es igual a la inicial que buscamos
```

```
            if(letra.getLetra().equals(inicialNuevaPalabra)){
```

```
                //Cambiamos el valor de resp a verdadero
```

```
                //(encontramos la letra en el diccionario)
```

```
                resp = true;
```

```
            }
```

```
        }
```

```
    }
```

```
    return resp; //Retornamos el valor de resp
```

```
}
```

```

public void insertarPalabra(String palabra, String significado) {

    //Creamos un objeto del tipo Palabra con la palabra y el significado
    Palabra nuevaPalabra = new Palabra(palabra, significado);

    //Obtenemos la inicial de la nueva palabra
    String nuevaInicial = nuevaPalabra.getInicial();

    //Verificamos si la inicial está en el diccionario
    boolean resp = this.contiene(palabra);

    //Si no está
    if(resp == false){
        //La agregamos al diccionario
        this.diccionario.add(new Letra(nuevaInicial));
    }

    //Buscamos la inicial en el diccionario
    for(Letra letra: this.diccionario){
        //Si la encontramos
        if(letra.getLetra().equals(nuevaInicial)){
            //Obtenemos la lista de palabras de la inicial y
            //agregamos la nueva palabra a la lista
            letra.getListaPalabras().add(nuevaPalabra);
        }
    }
}

```

```

public String buscarSignificado(String palabra) {
    // Se busca el significado de palabra
    //Variable resp en "Sin significado asociado"
    String resp = "Sin significado asociado";
    //Si el diccionario no está vacío
    if(!this.diccionario.isEmpty()){
        //Tomamos la inicial de la palabra
        String inicial = palabra.substring(0, 1);
        //Recorremos el diccionario
        for(Letra letra: this.diccionario){
            //Si la letra en el diccionario es igual a la inicial
            //de la palabra que buscamos
            if(inicial.equals(letra.getLetra())){
                //Obtenemos la lista de palabras de la letra en el diccionario
                LinkedList<Palabra> listaPalabras = letra.getListPalabras();
                //Creamos un Iterador para recorrer
                //la lista de palabras
                Iterator<Palabra> it = listaPalabras.iterator();
                //Mientras queden palabras en el iterador
                while(it.hasNext()){
                    //Obtenemos la palabra
                    Palabra pal = (Palabra) it.next();
                    //Si la palabra es igual a la palabra que buscamos
                    if(pal.getPalabra().equals(palabra)){
                        //Guardamos su significado en resp
                        resp= "Significado: " +
                            pal.getSignificado()+"\n";
                    }
                }
            }
        }
    }
    //Retornamos el valor de resp (significado de la palabra)
    return resp;
}

```

```

public String listadoPalabras() {

    //Retorna el diccionario completo

    //Variable resp en "Diccionario Vacío" para retornar
    String resp = "Diccionario Vacío";

    //Si el diccionario no está vacío
    if(!this.diccionario.isEmpty()){
        //Ordenamos el diccionario
        Collections.sort(this.diccionario);
        //Supuesto que esto funciona para ordenar el diccionario

        //Borramos el contenido de resp para guardar el listado de palabras
        resp = "";
        //Recorremos el diccionario
        for(Letra letra: this.diccionario){
            //Agregamos la letra a resp
            resp+=letra.getLetra() + "\n";

            //Obtenemos la lista de palabras de la letra
            LinkedList<Palabra> listaPalabras = letra.getListaPalabras();

            //Ordenamos la lista de palabras
            Collections.sort(listaPalabras);

            //Recorremos la lista de palabras
            for(Palabra palabra: listaPalabras){
                //Agregamos cada palabra a resp
                resp+=palabra.getPalabra() + " ";
            }
            //Agregamos un salto de línea luego de la última palabra de cada letra
            resp+="\n";
        }
    }
    //Retornamos el valor de resp
    return resp;
}

```

```

public String listadoPalabras(String letra) {

    //Para obtener todas las palabras que comienzan con letra

    // No se pide que salga ordenado

    //Variable lista en "No Existen Palabras con la Inicial Indicada"
    //para retornar
    String lista = "No Existen Palabras con la Inicial Indicada";
    //Si el diccionario no está vacío
    if(!this.diccionario.isEmpty()){

        //Si la letra está en el diccionario
        if(this.contiene(letra)){

            //Recorremos el diccionario
            for(Letra inicial: this.diccionario){

                //Si la letra es igual a la que buscamos
                if(inicial.getLetra().equals(letra)){

                    //Limpiamos la lista de respuesta
                    lista = "";

                    //Para cada palabra en la lista de palabras de la letra
                    for(Palabra pal: inicial.getListaPalabras()){
                        //Agregamos la palabra a resp
                        lista+=pal.getPalabra() + " ";
                    }

                }

            }

        }

    }

    //Retornamos el valor de resp
    return lista;

}

} //Fin class DiccionarioImpl

```