

TEMA 4

CONCEPTOS DE JAVA PARA ESTRUCTURAS DE DATOS
Representación enlazada

Objetivos

El objetivo general de este tema es estudiar la representación enlazada de un grupo o colección de objetos como una alternativa a la representación secuencial:

- Representar en *Java* un grupo o colección de elementos mediante una *lista enlazada de nodos*. Comparación de la representación enlazada y la secuencial
- Incidir en el mecanismo de genericidad de *Java*
- Implementación de distintos tipos de listas en base a criterios de eficiencia y claridad en su diseño

Índice

1.- Introducción

2.- Listas enlazadas genéricas (*LEG*)

- La clase *NodoLEG*
- Inserción de elementos
- Borrado de elementos
- Representación enlazada vs. secuencial

3.- *LEG* con referencias al primer y último nodo

4.- *LEG* circular

5.- *LEG* doblemente enlazada

6.- *LEG* ordenada

1.- INTRODUCCIÓN

La representación secuencial

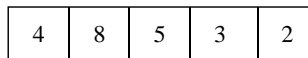
- La representación secuencial la proporciona el tipo *array*
 - Coste *constante* para acceder al *i-ésimo* elemento del grupo: $v[i]$
 - Coste de insertar un elemento
 - Ejemplo: insertar un elemento en la primera posición de un *array*. Es necesario desplazar todos los elementos del *array* para dejar un hueco en la primera posición
 - El coste está en función de la talla del *array* (*lineal* respecto al número de elementos del *array*)
 - Coste de eliminar un elemento: *en función de la talla del *array** (si se quiere mantener el orden de los elementos)

1.- INTRODUCCIÓN

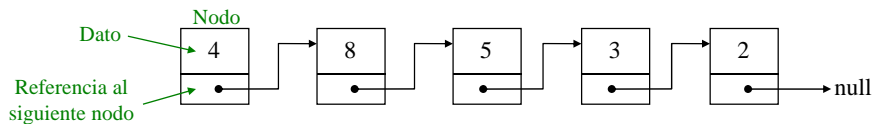
La representación enlazada

- Una representación enlazada de un grupo de elementos de un cierto tipo es una lista enlazada de nodos, es decir, una secuencia de nodos situados en la memoria dinámica y conectados entre sí
- Ejemplo: lista de enteros {4, 8, 5, 3, 2}

Representación secuencial:



Representación enlazada:



2.- LISTAS ENLAZADAS GENÉRICAS

La clase *NodoLEG*

- Como hemos visto, un nodo tiene dos atributos:
 - El **dato**, de tipo *genérico* (para poder guardar cualquier tipo de objeto)
 - Una referencia al nodo **siguiente**

```
class NodoLEG<E> {
    E dato;
    NodoLEG<E> siguiente;
    // Dos constructores
    NodoLEG(E dato, NodoLEG<E> sig) {
        this.dato = dato;
        this.siguiente = sig;
    }
    NodoLEG(E dato) {
        this(dato, null);
    }
}
```

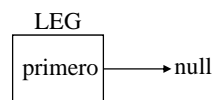
2.- LISTAS ENLAZADAS GENÉRICAS

- Una lista enlazada requiere, como mínimo, una referencia al primer nodo de la lista:

```
NodoLEG<E> primero;
```

- Cuando la lista está vacía, el atributo *primero* apunta a *null*

```
primero = null;
```



2.- LISTAS ENLAZADAS GENÉRICAS

Implementación de una *LEG*

```
package lineales;
```

```
public class LEG<E> {  
    protected NodoLEG<E> primero;  
    protected int talla;
```

```
    // El constructor crea la lista vacía
```

```
    public LEG() {  
        primero = null;  
        talla = 0;  
    }
```

```
    ...
```

```
}
```

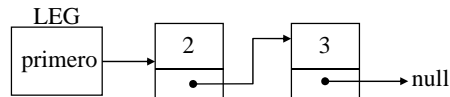
Referencia al primer nodo de la lista

En una lista (del tipo que sea) podemos incluir un atributo que indique el número de elementos de la lista. Así no es necesario recorrer toda la lista para averiguar cuántos elementos contiene.

2.- LISTAS ENLAZADAS GENÉRICAS

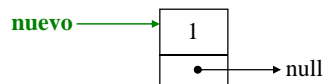
Insertar un elemento en la *LEG*

- Lo más eficiente es insertar al principio de la lista, pues tenemos una referencia al primer nodo
- Ejemplo: queremos insertar el elemento "1" en la siguiente lista:



Paso 1 Crear el nodo a insertar:

```
NodoLEG<Integer> nuevo = new NodoLEG<Integer>(new Integer(1));
```

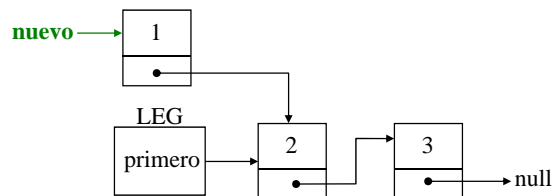


2.- LISTAS ENLAZADAS GENÉRICAS

Insertar un elemento en la *LEG*

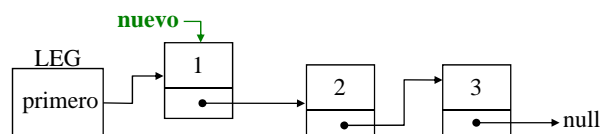
Paso 2 Hacer que el nodo siguiente del nuevo nodo sea el primero de la lista:

```
nuevo.siguiete = primero;
```



Paso 3 Hacer que primer nodo de la lista sea el nuevo nodo:

```
primero = nuevo;
```



2.- LISTAS ENLAZADAS GENÉRICAS

Implementación

```
public class LEG<E> {  
    ...  
    public void insertar(E x) {  
        primero = new NodoLEG<E>(x, primero);  
        talla++;  
    }  
    public String toString() {  
        String res = "";  
        for (NodoLEG<E> aux = primero; aux != null;  
            aux = aux.siguiente)  
            res += aux.dato.toString() + "\n";  
        return res;  
    }  
    ...  
}
```

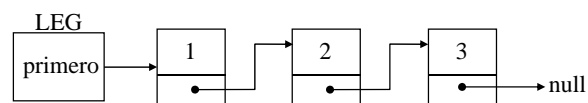
Esta instrucción resume los tres pasos que hemos visto para insertar un elemento en la lista

Este bucle muestra la forma de recorrer los nodos de una lista

2.- LISTAS ENLAZADAS GENÉRICAS

Eliminar un elemento de una *LEG*

- Ejemplo: eliminar el elemento "2" de la lista:



Para eliminar el "2" tenemos que hacer que el siguiente del "1" sea el "3"

Necesitamos averiguar, por lo tanto, cuál es el nodo anterior del que queremos eliminar

2.- LISTAS ENLAZADAS GENÉRICAS

Implementación

```
public class LEG <E> {
    ...
    public boolean eliminar(E x) {
        NodoLEG <E> aux = primero, ant = null;
        while (aux != null && !aux.dato.equals(x)) {
            ant = aux;
            aux = aux.siguiete;
        }
        if (aux == null) return false; // Elemento no encontrado
        if (ant == null) primero = aux.siguiete;
        else ant.siguiete = aux.siguiete;
        talla--;
        return true;
    }
}
```

Buscamos el elemento a borrar:
➤ aux apunta al nodo a borrar
➤ ant apunta al nodo anterior

Caso especial: cuando el nodo que vamos a borrar es el primero de la lista

2.- LISTAS ENLAZADAS GENÉRICAS

Representación enlazada vs. secuencial

Ventajas de la representación secuencial

- ❑ Acceso en tiempo constante a cualquier elemento del array

Desventajas de la representación secuencial

- ❑ La inserción y el borrado requieren el desplazamiento de parte (o todas) las componentes del array
- ❑ Reserva a-priori de una cantidad de memoria mayor que la talla del grupo

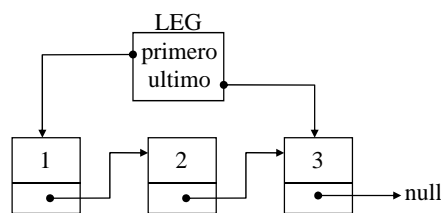
- ❑ Ejercicio 1: Añadir a la clase *LEG* un método que inserte un nuevo elemento al final de la lista:

```
public void insertarEnFin(E x);
```

¿Cuál es el coste del nuevo método?

3.- LEG CON REFERENCIAS AL PRIMER Y ÚLTIMO NODO

- ❑ ¿Cómo se puede mejorar la eficiencia al insertar en la última posición de la lista?
- ❑ Podemos guardar en la lista una referencia al último nodo:



3.- LEG CON REFERENCIAS AL PRIMER Y ÚLTIMO NODO

```
package lineales;

public class LEGConUltimo<E> {
    protected NodoLEG<E> primero, ultimo;
    protected int talla;

    public LEGConUltimo() {
        primero = ultimo = null;
        talla = 0;
    }
    ...
}
```

3.- LEG CON REFERENCIAS AL PRIMER Y ÚLTIMO NODO

```
public class LEGConUltimo <E> {
    ...
    public void insertarEnFin(E x) {           // Coste constante
        NodoLEG<E> nuevo = new NodoLEG<E>(x);
        if (ultimo != null) ultimo.siguiete = nuevo;
        else primero = nuevo;
        ultimo = nuevo;
        talla++;
    }
    public void insertar(E x) {
        primero = new NodoLEG <E> (x, primero);
        if (ultimo == null) ultimo = primero;
        talla++;
    }
    ...
}
```

3.- LEG CON REFERENCIAS AL PRIMER Y ÚLTIMO NODO

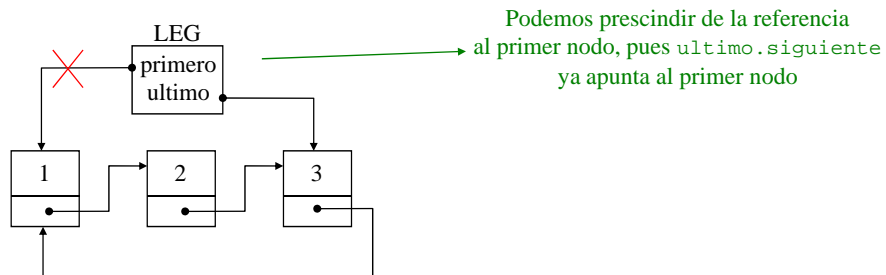
```
public class LEGConUltimo<E> {
    ...
    public boolean eliminar(E x) {
        NodoLEG <E> aux = primero, ant = null;
        while (aux != null && !aux.dato.equals(x)) {
            ant = aux;
            aux = aux.siguiete;
        }
        if (aux == null) return false; // No encontrado
        if (ant == null) primero = aux.siguiete;
        else ant.siguiete = aux.siguiete;
        if (aux.siguiete == null) ultimo = ant;
        talla--;
        return true;
    }
}
```

Caso especial: si borramos el último nodo, el último será ahora el nodo anterior

4.- LEG CIRCULAR

- Una alternativa a la *LEG* con referencias al primer y último nodo es la *LEG* circular: una *LEGConUltimo* en la que se obliga a que el siguiente al último nodo sea el primero:

```
ultimo.siguiete = primero
```



4.- LEG CIRCULAR

```
package lineales;
```

```
public class LEGCircular<E> {
```

```
    protected NodoLEG<E> ultimo; → Referencia al último nodo  
    protected int talla;
```

```
    public LEGCircular() {  
        ultimo = null;  
        talla = 0;  
    }  
}
```

4.- LEG CIRCULAR

```
public class LEGCircular <E> {
    ...
    public void insertarEnFin(E x) {
        NodoLEG<E> nuevo = new NodoLEG<E> (x);
        nuevo.siguiete = nuevo;
        if (ultimo != null) { // Lista no vacía
            nuevo.siguiete = ultimo.siguiete;
            ultimo.siguiete = nuevo;
        }
        ultimo = nuevo;
        talla++;
    }
    ...
}
```

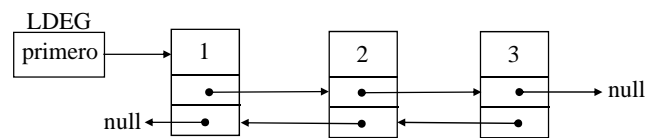
4.- LEG CIRCULAR

```
public class LEGCircular <E> {
    ...
    public void insertar(E x) {
        NodoLEG<E> nuevo = new NodoLEG<E> (x);
        nuevo.siguiete = nuevo;
        if (ultimo != null) { // Lista no vacía
            nuevo.siguiete = ultimo.siguiete;
            ultimo.siguiete = nuevo;
        }
        else ultimo = nuevo; // Lista vacía
        talla++;
    }
    ...
}
```

- Ejercicio 2: Escribir el método *eliminar* de la *LEGCircular*.

5.- LEG DOBLEMENTE ENLZADAS

- En las *LEG* doblemente enlazadas cada nodo, además de una referencial a nodo siguiente, tiene una referencia al nodo anterior



5.- LEG DOBLEMENTE ENLZADAS

Nodos doblemente enlazados

```
package lineales;
class NodoLDEG<E> {
    // Tres atributos
    E dato;
    NodoLDEG<E> siguiente, anterior;
    // Dos constructores
    NodoLDEG(E dato) {
        this(dato, null, null);
    }
    NodoLDEG(E dato, NodoLDEG<E> s, NodoLDEG<E> a) {
        this.dato = dato;
        this.siguiente = s;
        this.anterior = a;
    }
}
```

5.- LEG DOBLEMENTE ENLAZADAS

```
public class LDEG<E> {
    protected NodoLDEG<E> primero;
    protected int talla;
    public LDEG() {
        primero = null;
        talla = 0;
    }
    public void insertar(E x) {
        NodoLDEG<E> nuevo = new NodoLDEG<E>(x, primero, null);
        if (primero != null) primero.anterior = nuevo;
        primero = nuevo;
        talla++;
    }
    ...
}
```

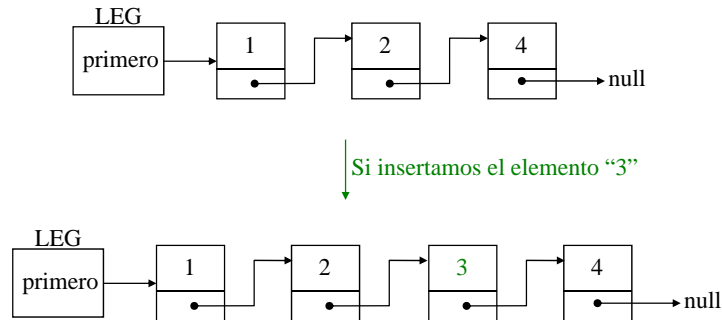
5.- LEG DOBLEMENTE ENLZADAS

```
public boolean eliminar(E x) {
    NodoLDEG<E> aux = primero;
    while (aux != null && !aux.dato.equals(x))
        aux = aux.siguiete;
    if (aux == null) return false; // Elemento no encontrado
    if (aux.anterior != null) aux.anterior.siguiete = aux.siguiete;
    else primero = aux.siguiete;
    if (aux.siguiete != null) aux.siguiete.anterior = aux.anterior;
    talla--;
    return true;
}
...
}
```

Ahora no es necesario una variable ant que apunte al nodo anterior

6.- LEG ORDENADAS

- En una *LEG* ordenada, los nodos de la lista se mantienen ordenados de acuerdo con el método *compareTo*:



6.- LEG ORDENADAS

```
public class LEGordenada <E extends Comparable<E>>
    extends LEG <E> {
    ...
    public void insertar(E x) {
        NodoLEG<E> nuevo = new NodoLEG<E>(x);
        NodoLEG<E> ant = null, aux = primero;
        while (aux != null && aux.dato.compareTo(x) < 0) {
            ant = aux;
            aux = aux.siguiete;
        }
        nuevo.siguiete = aux;
        if (ant != null) ant.siguiete = nuevo;
        else primero = nuevo;
        talla++;
    }
    ...
}
```

EJERCICIOS

- Ejercicio 3: Escribir un método `toString()` de *LDEG* que obtenga una representación textual de los nodos de la lista en orden descendente, del último al primero
- Ejercicio 4: Diseña el método `insertarEnFin(E x)` de la clase *LDEG*
- Ejercicio 5: ¿Es conveniente que *LEGOrdenada* sobrescriba el método `eliminar` de *LEG*? En caso afirmativo, indíquese por qué y realícense las modificaciones oportunas

EJERCICIOS

- Ejercicio 6: Diseña el método `boolean eliminarMayor(E x)`, que borra de la lista todos los elementos mayores que `x`, en las clases *LEG* y *LEGOrdenada*
- Ejercicio 7: Escribe el método `E buscarMin()` en la clase *LEGCircular*, que devuelve el menor elemento de la lista
- Ejercicio 8: Escribe el método `E recuperar(int indice)` en la clase *LEG*, que devuelve el elemento de la lista que ocupa dicha posición. El método deberá lanzar la excepción *ElementoNoEncontrado* si el índice está fuera de rango.