|  |
| --- |
|  |
| **Diseño de una estructura enlazada** |
| Estructuras de Datos y Algoritmos |
|  |
|  |
|  |

|  |
| --- |
|  |

**Fecha:** 26-X-2015

**Participantes**

XABIER CASADO

SERGIO ERLANTZ TOBAL

PEIO VALLE

Índice de contenido

[Introducción 3](#_Toc434138460)

[Descripción de las estructuras de datos principales 4](#_Toc434138461)

[Diseño de las clases 5](#_Toc434138462)

[Clases Node y Persona 6](#_Toc434138463)

[Interfaces 6](#_Toc434138464)

[Listas 6](#_Toc434138465)

[Casos de prueba 7](#_Toc434138466)

[public T removeFirst() 7](#_Toc434138467)

[public T removeLast() 7](#_Toc434138468)

[public T remove(T elem) 7](#_Toc434138469)

[public T first() 8](#_Toc434138470)

[public T last() 8](#_Toc434138471)

[public boolean contains(T elem) 8](#_Toc434138472)

[public T find(T elem) 8](#_Toc434138473)

[public boolean isEmpty() 9](#_Toc434138474)

[public int size() 9](#_Toc434138475)

[public void addToFront(T elem) 9](#_Toc434138476)

[public void addToRear(T elem) 9](#_Toc434138477)

[public void addAfter(T elem, T target) 10](#_Toc434138478)

[public void add(T elem) 10](#_Toc434138479)

[Código 11](#_Toc434138480)

[ListADT 11](#_Toc434138481)

[OrderedListADT 11](#_Toc434138482)

[UnorderedListADT 11](#_Toc434138483)

[Persona 12](#_Toc434138484)

[Node 12](#_Toc434138485)

[DoubleLinkedList 13](#_Toc434138486)

[Coste de los métodos principales 15](#_Toc434138487)

[OrderedDoubleLinkedList 16](#_Toc434138488)

[Coste de los métodos principales 16](#_Toc434138489)

[UnorderedDoubleLinkedList 17](#_Toc434138490)

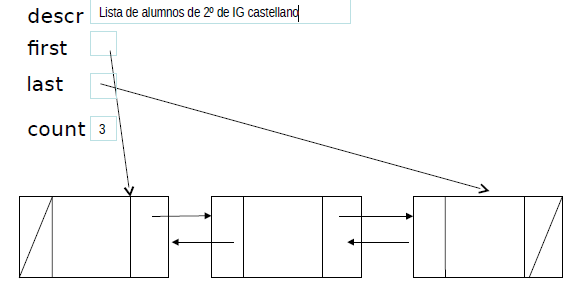
[Coste de los métodos principales 18](#_Toc434138491)

[Implementación de la estructura enlazada en la Práctica 1 18](#_Toc434138492)

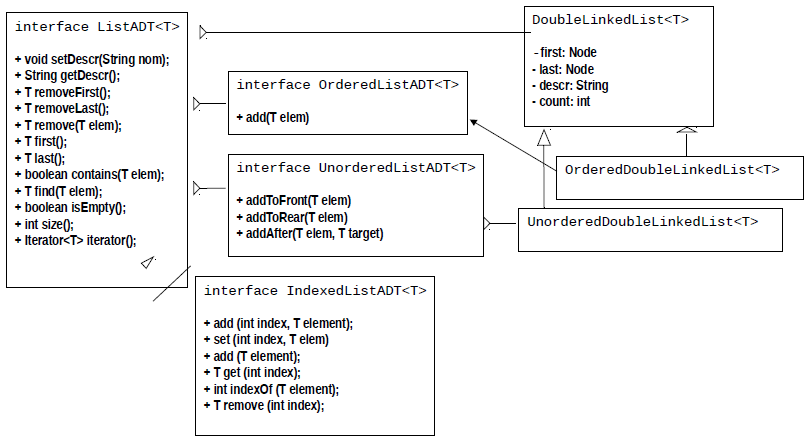
[Conclusiones 19](#_Toc434138493)

# Introducción

En esta segunda práctica se pretende desarrollar la clase *DoubleLinkedList*, la cual sigue una estructura doblemente enlazada, como puede apreciarse en la siguiente imagen:



Del mismo modo, también desarrollaremos dos subclases, *OrderedDoubleLinkedList* y *UnorderedDoubleLinkedList*, que diferencian inserciones de elementos de forma ordenada y desordenada, respectivamente. El esquema a seguir es el siguiente:



Por otra parte, dicho desarrollo se divide en tres fases, las cuales son:

* **Especificación**

Consistirá en la creación de los *tipos abstractos de datos (TAD)*, los cuales definirán las interfaces de la estructura a desarrollar, conteniendo las operaciones abstractas definidas sobre ellos.

* **Diseño**

En esta segunda fase definiremos los casos de prueba para cada uno de los métodos contenidos en la especificación. Un correcto análisis nos facilitará la posterior implementación, haciéndola más robusta y eficiente.

* **Implementación**

Por último, crearemos las tres clases anteriormente mencionadas e implementaremos cada uno de los métodos especificados en las interfaces y analizados en el diseño.

Obtendremos de cada método su coste, el cual quedará reflejado en esta documentación.

Tras finalizar, sustituiremos la clase *UnorderedDoubleLinkedList* en el primer proyecto, como lista de películas de un actor dado, sabiendo que ésta funciona de forma más eficaz en cuestiones de espacio.

# Descripción de las estructuras de datos principales

Las *estructuras enlazadas* han sido las escogidas para la realización de esta segunda práctica. Este tipo de estructura de datos se compone de una sucesión de *nodos*, utilizando *variables de referencia* a objetos con la finalidad de crear enlaces entre dichos nodos, conformando la lista.

Nuestra lista doblemente enlazada contendrá dos variables de referencia, *first* y *last*, que apuntarán al primer y último nodo de la estructura.

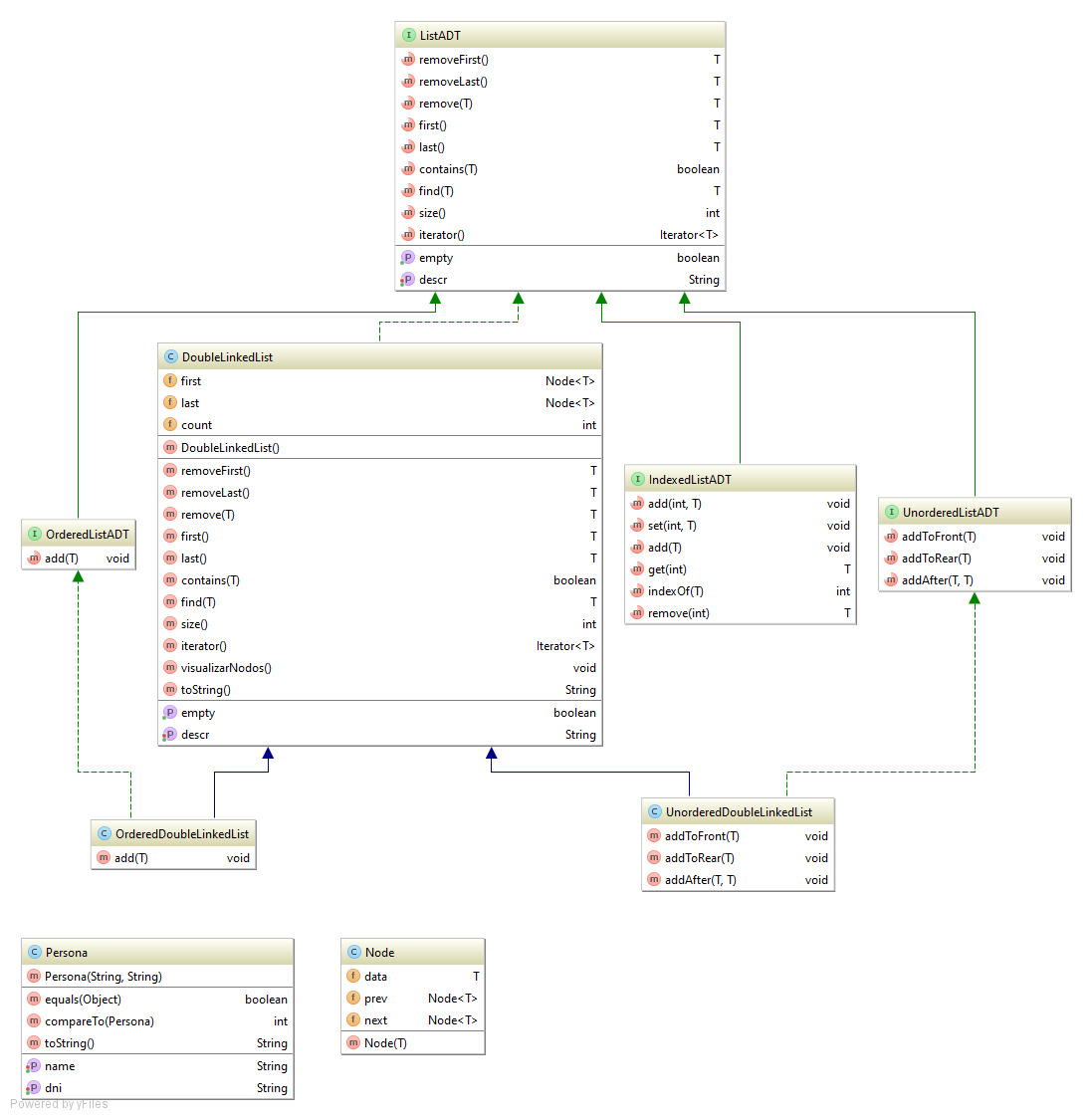
Un nodo por su parte es una clase auto-referencial, que contiene un campo de datos y una o más referencias a nodos, utilizadas para poder navegar por la lista (estas referencias son al anterior y siguiente nodo, en esta práctica).

Las ventajas que presentan las estructuras enlazadas respecto a los *ArrayList* utilizados en la anterior práctica, en cuanto a coste temporal, están reflejados en esta tabla:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **insertar al comienzo** | **eliminar al comienzo** | **insertar al final** | **eliminar al final** |
| **ArrayList** | O(n) | O(n) | O(1) | O(1) |
| **LinkedList** | O(1) | O(1) | O(n) | O(n) |
| **CircularLinkedList** | O(1) | O(1) | O(1) | O(n) |
| **DoubleLinkedList** | O(1) | O(1) | O(1) | O(1) |
| **CicularDoubleLinkedList** | O(1) | O(1) | O(1) | O(1) |

# Diseño de las clases

Para la resolución de la práctica hemos seguido el siguiente diagrama de clases:



## Clases Node y Persona

Los nodos, instancias de la clase **Node**, conforman la información básica de la que se compone la estructura enlazada de datos, con referencias al anterior y siguiente nodo así como un campo data de tipo T, que llevado a la práctica serán objetos **Persona**.

La clase **Persona** implementa la interfaz *Comparable,* para poder ordenar objetos de este tipo por nombre, mediante el uso del método **compareTo**. Sobrescribe también los métodos **equals** y **toString** de Object, para indicar que dos personas son la misma si tienen el mismo DNI e imprimir el nombre y DNI de una persona, respectivamente.

## Interfaces

Conforman la especificación del proyecto, el *qué* se quiere hacer, pero no el *cómo* (que se define posteriormente, en la implementación de las clases).

Contienen la cabecera de todos los métodos a implementar, y se dividen en dos grupos:

* **ListADT**

Es la estructura principal a seguir para una lista enlazada, sin diferenciar si esta va a ser ordenada o desordenada. Por tanto, contiene los métodos que comparten ambos tipos de estructuras enlazadas.

* **OrderedListADT** y **UnorderedListADT**

Cada una de ellas contiene la cabecera de los métodos de inserción que va a implementarse dependiendo de si la estructura enlazada está ordenada o no.

## Listas

Son las implementaciones de las interfaces anteriores, en las que se programa cada uno de los métodos para **OrderedDoubleLinkedList** y **UnorderedDoubleLinkedList**, con la particularidad de que ambas heredan de la genérica **DoubleLinkedList**.

Esta última clase alberga un **Iterator** propio para recorrer estructuras doblemente enlazadas, privado y por tanto solamente accesible por dicha clase.

La clase **UnorderedDoubleLinkedList** podrá insertar elementos al comienzo de la estructura enlazada, al final o tras otro elemento ya hallado en la lista, mientras que la clase **OrderedDoubleLinkedList** insertará cada nuevo elemento en su posición correspondiente, de acuerdo a los criterios de ordenación establecidos dependiendo del tipo de dato con el que trabajemos (de menor a mayor, alfabéticamente, comparando en el caso de objetos *Persona*…).

# Casos de prueba

Como consta en la introducción, un buen análisis de los casos de prueba antes de la implementación de los métodos es una técnica muy eficaz para agilizar dicha programación, visualizando los posibles casos dudosos o complicaciones que pudieran surgir posteriormente, de no seguir este procedimiento. Dicho esto, a continuación, se muestran todos los casos a tratar en esta práctica, indicando también los casos dudosos y las soluciones dadas (los ficheros JUnits serán análogos a estas tablas):

## public T removeFirst()

|  |  |  |
| --- | --- | --- |
| **Lista** | **Lista resultante** | **Return** |
| () | () | null |
| (x) | () | x |
| (x y z) | (y z) | x |

## public T removeLast()

|  |  |  |
| --- | --- | --- |
| **Lista** | **Lista resultante** | **Return** |
| () | () | null |
| (x) | () | x |
| (x y z) | (x y) | z |

## public T remove(T elem)

|  |  |  |  |
| --- | --- | --- | --- |
| **Lista** | **elem** | **Lista resultante** | **Return** |
| () | x | () | null (\*) |
| (x) | x | () | x |
|  | y | (x) | null (\*) |
| (x y z) | x | (y z) | x |
|  | y | (x z) | y |
|  | z | (x y) | z |
|  | t | (x y z) | null (\*) |

*(\*) Resultado de la operación en caso de que el* ***elem*** *a eliminar no exista en la lista: devuelve null.*

## public T first()

|  |  |
| --- | --- |
| **Lista** | **Return** |
| () | null |
| (x) | x |
| (x y z) | x |

## public T last()

|  |  |
| --- | --- |
| **Lista** | **Return** |
| () | null |
| (x) | x |
| (x y z) | z |

## public boolean contains(T elem)

|  |  |  |
| --- | --- | --- |
| **Lista** | **elem** | **Return** |
| () | x | False |
| (x) | x | True |
|  | y | False |

## public T find(T elem)

|  |  |  |
| --- | --- | --- |
| **Lista** | **elem** | **Return** |
| () | x | null |
| (x) | x | x |
|  | y | null |
| (x y z) | x | x |
|  | y | y |
|  | z | z |
|  | t | null |

## public boolean isEmpty()

|  |  |
| --- | --- |
| **Lista** | **Return** |
| () | True |
| (x) | False |

## public int size()

|  |  |
| --- | --- |
| **Lista** | **Return** |
| () | 0 |
| (x) | 1 |
| (1 .. n) | n |

## public void addToFront(T elem)

|  |  |  |
| --- | --- | --- |
| **Lista** | **elem** | **Lista resultante** |
| () | e | (e) |
| (x) | e | (e x) |
| (x y z) | e | (e x y z) |

## public void addToRear(T elem)

|  |  |  |
| --- | --- | --- |
| **Lista** | **elem** | **Lista resultante** |
| () | e | (e) |
| (x) | e | (x e) |
| (x y z) | e | (x y z e) |

## public void addAfter(T elem, T target)

|  |  |  |  |
| --- | --- | --- | --- |
| **Lista** | **elem** | **target** | **Lista resultante** |
| () | e | x | () (\*) |
| (x) | e | x | (x e) |
|  | e | y | (x) (\*) |
| (x y z) | e | x | (x e y z) |
|  | e | y | (x y e z) |
|  | e | z | (x y z e) |
|  | e | t | (x y z) (\*) |

*(\*) Resultado de la operación en caso de que el* ***target*** *a buscar no exista en la lista: El elemento no se añade.*

## public void add(T elem)

|  |  |  |
| --- | --- | --- |
| **Lista** | **elem** | **Lista resultante** |
| () | a | (a) |
| (b) | a | (a b) |
|  | b | (b b) (\*) |
|  | c | (b c) |
| (b d) | a | (a b d) |
|  | b | (b b d) (\*) |
|  | c | (b c d) |
|  | e | (b d e) |

*(\*) Resultado de la operación en caso de que el* ***elem*** *a añadir ya exista en la lista: El elemento se duplicará.*

# Código

En este apartado se incluyen todos los códigos correspondientes a cada una de las clases o interfaces del proyecto, así como el coste de los métodos principales implementados.

## ListADT

**public** **interface** ListADT<T> {

**public** **void** setDescr(String nom);

// Actualiza el nombre de la lista

**public** String getDescr();

// Devuelve el nombre de la lista

**public** T removeFirst();

// Elimina el primer elemento de la lista

**public** T removeLast();

// Elimina el último elemento de la lista

**public** T remove(T elem);

// Elimina un elemento concreto de la lista

**public** T first();

// Da acceso al primer elemento de la lista

**public** T last();

// Da acceso al último elemento de la lista

**public** **boolean** contains(T elem);

// Determina si la lista contiene un elemento concreto

**public** T find(T elem);

// Determina si la lista contiene un elemento concreto, y devuelve su

// referencia, null en caso de que no esté

**public** **boolean** isEmpty();

// Determina si la lista está vacía

**public** **int** size();

// Determina el número de elementos de la lista

**public** Iterator<T> iterator();

}

## OrderedListADT

**public** **interface** OrderedListADT<T **extends** Comparable<T>> **extends** ListADT<T>{

**public** **void** add(T elem);

// Añade un elemento a la lista (en el lugar de orden que le

corresponde)

}

## UnorderedListADT

**public** **interface** UnorderedListADT<T> **extends** ListADT<T> {

**public** **void** addToFront(T elem);

// añade un elemento al comienzo

**public** **void** addToRear(T elem);

// añade un elemento al final

**public** **void** addAfter(T elem, T target);

// Añade elem detrás de otro elemento concreto, target, que ya se

encuentra en la lista

}

## Persona

**public** **class** Persona **implements** Comparable<Persona> {

// Atributos

**private** String name;

**private** String dni;

**public** Persona(String pName, String pDni) { // Constructora

name = pName;

dni = pDni;

}

**public** String getName() { **return** name; }

**public** **void** setName(String name) { **this**.name = name; }

**public** String getDni() { **return** dni; }

**public** **void** setDni(String dni) { **this**.dni = dni; }

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Persona other = (Persona) obj;

**if** (dni == **null**) {

**if** (other.dni != **null**)

**return** **false**;

} **else** **if** (!dni.equals(other.dni))

**return** **false**;

**return** **true**;

}

@Override

**public** **int** compareTo(Persona arg0) {

**return** name.compareToIgnoreCase(arg0.name);

}

**public** String toString() {

**return** name + " " + dni;

}

}

## Node

**public** **class** Node<T> {

**public** T data; // dato del nodo

**public** Node<T> prev; // puntero al anterior nodo de la lista

**public** Node<T> next; // puntero al siguiente nodo de la lista

// -------------------------------------------------------------

**public** Node(T dd) {// constructor

data = dd;

next = **null**;

prev = **null**;

}

}

## DoubleLinkedList

**public** **class** DoubleLinkedList<T> **implements** ListADT<T> {

// Atributos

**protected** Node<T> first;

**protected** Node<T> last; // apuntador al ultimo

**protected** String descr; // descripciï¿½n

**protected** **int** count;

// Constructor

**public** DoubleLinkedList() {

first = **null**;

last = **null**;

descr = "";

count = 0;

}

**public** **void** setDescr(String nom) { descr = nom; }

**public** String getDescr() { **return** descr; }

// Elimina el primer elemento de la lista

**public** T removeFirst() {

**if** (isEmpty()) **return** **null**;

**else** {

T temp = first.data;

first = first.next;

**this**.count--;

**return** temp;

}

}

// Elimina el último elemento de la lista

**public** T removeLast() {

**if** (isEmpty()) **return** **null**;

**else** {

T temp = last.data;

**if** (size() > 1){

last.prev.next = **null**;

last = last.prev;

**this**.count--;

} **else** {

first = first.next;

**this**.count--;

}

**return** temp;

}

}

//Elimina un elemento concreto de la lista

**public** T remove(T elem) {

Node<T> node = first;

**boolean** enc = **false**;

**while** (node != **null** && !enc){

**if** (node.data.equals(elem)) enc = **true**;

**else** node = node.next;

}

**if** (node == **null**) **return** **null**;

elem = node.data;

**if** (node == first) elem = **this**.removeFirst();

**else** **if** (node == last) elem = **this**.removeLast();

**else** {

node.next.prev = node.prev;

node.prev.next = node.next;

count--;

}

**return** elem;

}

**public** T first() {

//Da acceso al primer elemento de la lista

**if** (isEmpty()) **return** **null**;

**else** **return** first.data;

}

**public** T last() {

//Da acceso al ï¿½ltimo elemento de la lista

**if** (isEmpty()) **return** **null**;

**else** **return** last.data;

}

// Determina si la lista contiene un elemento concreto

**public** **boolean** contains(T elem) {

**if** (isEmpty()) **return** **false**;

Node<T> current = first; // Empieza con el primero elemento

**while** ((current != **null**) && !elem.equals(current.data))

current = current.next;

**if** (current == **null**) **return** **false**;

**else** **return** elem.equals(current.data);

}

//Determina si la lista contiene un elemento concreto, y devuelve su

referencia, null en caso de que no está

**public** T find(T elem) {

Node<T> current;

**for** (current = first; current != **null**; current = current.next) {

**if** (elem.equals(current.data)) **return** current.data;

}

**return** **null**;

}

**public** **boolean** isEmpty() { **return** last==**null**; }

**public** **int** size() { **return** count; }

/\*\*

\* Return an iterator to the stack that iterates through the items.

\*/

**public** Iterator<T> iterator() {

**return** **new** ListIterator();

}

// an iterator, doesn't implement remove() since it's optional

**private** **class** ListIterator **implements** Iterator<T> {

**private** Node<T> current = first;//Inicializamos

@Override

**public** **boolean** hasNext() { **return** current != **null**; }

@Override

**public** T next() {

**if** (!hasNext())**throw** **new** NoSuchElementException();

T item = current.data;

current = current.next;

**return** item;

}

@Override

**public** **void** remove() {}

} // private class

**public** **void** visualizarNodos() { System.***out***.println(**this**.toString()); }

@Override

**public** String toString() {

String result = **new** String();

Iterator<T> it = iterator();

**while** (it.hasNext()) {

T elem = it.next();

result = result + elem.toString() + " ";

}

**return** result;

}

}

### Coste de los métodos principales

**removeFirst**, **removeLast**, **first** y **last:** Coste constante O(1).

**contains**, **find** y **remove:** En el peor de los casos tendrán que recorrer toda la lista, siendo su coste lineal O(n), para una lista con n número nodos.

## OrderedDoubleLinkedList

**public** **class** OrderedDoubleLinkedList<T **extends** Comparable<T>> **extends** DoubleLinkedList<T> **implements** OrderedListADT<T> {

**public** **void** add(T elem){

Node<T> nodoElem = **new** Node<T>(elem);

**if** (first == **null**) {

first = nodoElem;

last = nodoElem;

}**else** **if** (elem.compareTo(first.data) <= 0){// Si elem es el menor

nodoElem.prev = **null**;

nodoElem.next = first;

first.prev = nodoElem;

first = nodoElem;

} **else** **if** (elem.compareTo(last.data) >= 0) { //Si elem es mayor

nodoElem.prev = last;

nodoElem.next = **null**;

last.next = nodoElem;

last = nodoElem;

} **else** {//busqueda del nodo anterior al que se debe insertar

Node<T> current = first;

**while** ((elem.compareTo(current.data) > 0))

current = current.next;

nodoElem.next = current;

nodoElem.prev = current.prev;

current.prev.next = nodoElem;

current.prev = nodoElem;

}

**this**.count++;

}

}

### Coste de los métodos principales

**add:** Para una lista con n número de nodos, en el peor de los casos el elemento tendrá que insertarse una posición antes del último nodo (*last*), lo que incrementará el coste de la función a lineal O(n).

## UnorderedDoubleLinkedList

**public** **class** UnorderedDoubleLinkedList<T> **extends** DoubleLinkedList<T> **implements** UnorderedListADT<T> {

// Añade un elemento al comienzo

**public** **void** addToFront(T elem) {

Node<T> newLink = **new** Node<T>(elem);

**if** (**this**.size()==0){

last=newLink;

}**else** {

first.prev=newLink;

newLink.next=first;

}

first=newLink;

**this**.count++;

}

// Añade un elemento al final

**public** **void** addToRear(T elem) {

Node<T> newLink = **new** Node<T>(elem);

**if** (**this**.size()==0){

first = newLink;

}**else**{

last.next = newLink;

newLink.prev = last;

}

last = newLink;

**this**.count++;

}

// Añade elem detras de otro elemento concreto, target, que ya se

encuentra en la lista

**public** **void** addAfter(T elem, T target) {

Node<T> newLink = **new** Node<T>(elem);

Node<T> targetLink = first;

**boolean** enc = **false**;

**int** i = 1;

**if** (first != **null**){

**while** (!enc && i <= **this**.size()) {

**if** (targetLink.data.equals(target)){

**if** (**this**.size()!=1 && i != **this**.size()) {

targetLink.next.prev = newLink;

newLink.next = targetLink.next;

}

**else** {

last = newLink;

}

newLink.prev = targetLink;

targetLink.next = newLink;

**this**.count++;

enc = **true**;

}

targetLink = targetLink.next;

i++;

}

}

}

**public** **void** comprobacionInversa(){

Node<T> current = last;

**for** (**int** i = **this**.size(); i>0; i--){

System.***out***.println(current.data);

current = current.prev;

}

}

}

### Coste de los métodos principales

**addToFront**, **addToRear**: Coste constante O(1).

**addAfter**: para insertar después de un elemento dado tiene que buscar éste en una lista enlazada compuesta por n número de nodos, teniendo que recorrer la lista completa en el pro de los casos, siendo su coste lineal O(n).

## Implementación de la estructura enlazada en la Práctica 1

Como se comenta en la introducción, y como último paso de esta segunda práctica, sustituiremos el *ArrayList* que almacena la lista de películas de un actor dado por la clase *UnorderedDoubleLinkedList -* que implementa una estructura doblemente enlazada -, en el primer proyecto, sabiendo que ésta funciona de forma más eficiente en cuestiones de espacio.

**public** **class** Actor **implements** Comparable<Actor>{

// Atributos

**private** String nombre;

**private** UnorderedDoubleLinkedList<Pelicula> listaP;

// Constructor

**public** Actor(String nombre) {

**this**.nombre = nombre;

**this**.listaP = **new** UnorderedDoubleLinkedList<Pelicula>();

}

// Getters y Setters

**public** String getNombre(){ **return** nombre; }

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

**public** UnorderedDoubleLinkedList<Pelicula>getListaP() {**return** listaP;}

**public** **void** setListaP(UnorderedDoubleLinkedList<Pelicula> listaP) {

**this**.listaP = listaP;

}

// Métodos

**public** **void** imprimir(){ System.***out***.println(**this**.getNombre()); }

**public** **void** mostrarApariciones (){

**this**.imprimir();

System.***out***.println("Ha aparecido: "+**this**.listaP.tamano());

**this**.listaP.imprimir();

}

@Override

**public** **int** compareTo(Actor o) {

**return** **this**.getNombre().compareTo(o.getNombre());

}

}

Tras un cálculo simple en el que hemos hallado la *media de películas de un actor* y obtenido un promedio de dos películas*,* podemos cerciorar que la *UnorderedDoubleLikedList* será **mucho más eficiente en cuestiones de espacio**, ya que no contiene registros vacíos, a diferencia del *ArrayList* que se crea con 10 posiciones de las cuales, acorde a la media, ocho quedarán vacías.

# Conclusiones

La finalidad de esta práctica ha sido el aprendizaje y de las estructuras enlazadas / doblemente enlazadas, así como su utilidad en casos prácticos, sustituyendo esta estructura en la primera práctica de la asignatura.

Esta práctica también nos ha servido para diferenciar y separar las distintas fases del desarrollo de un proyecto, así como las ventajas de esta distribución del trabajo.

En lo que respecta a la especificación, es correcto y muy conveniente aislar el esquema de la estructura y la cabecera de sus métodos (dicho de otra forma, la interfaz) de la programación particular de cada método en cada una de sus implementaciones (podríamos haber implementado la estructura enlazada mediante *ArrayList* o *TreeMap*, por poner un ejemplo, sin suponer ningún problema porque la especificación seguiría siendo la misma para todas las implementaciones).

Siguiendo este orden hemos comprobado que, dadas las especificaciones, un correcto análisis y diseño de los casos de prueba simplifica la posterior fase de implementación.

Por último, hemos implementado la estructura doblemente enlazada en la fase de implementación, aprendiendo en este proceso el funcionamiento de ésta.