Universidad del País Vasco

Escuela Universitaria de Ingeniería Técnica Industrial de Bilbao

Estructuras Datos y Algoritmos

Laboratorío 2:DoubleLinkedList

Andrea Siles
Ivan Salazar

Ingeniería Informática de Gestión y Sistemas de Información

November 2, 2015

Contents

1 Introducción

Conforme ha ido evolucionando la algoritmica se ha ido creando estructuras mas rapidas como por ejemplo las listas enlazas que nos permite que los metodos basicos se realicen en un tiempo muy reducido y esto nos permite procesar grandes cantidades de datos.

2 Diseño de clases

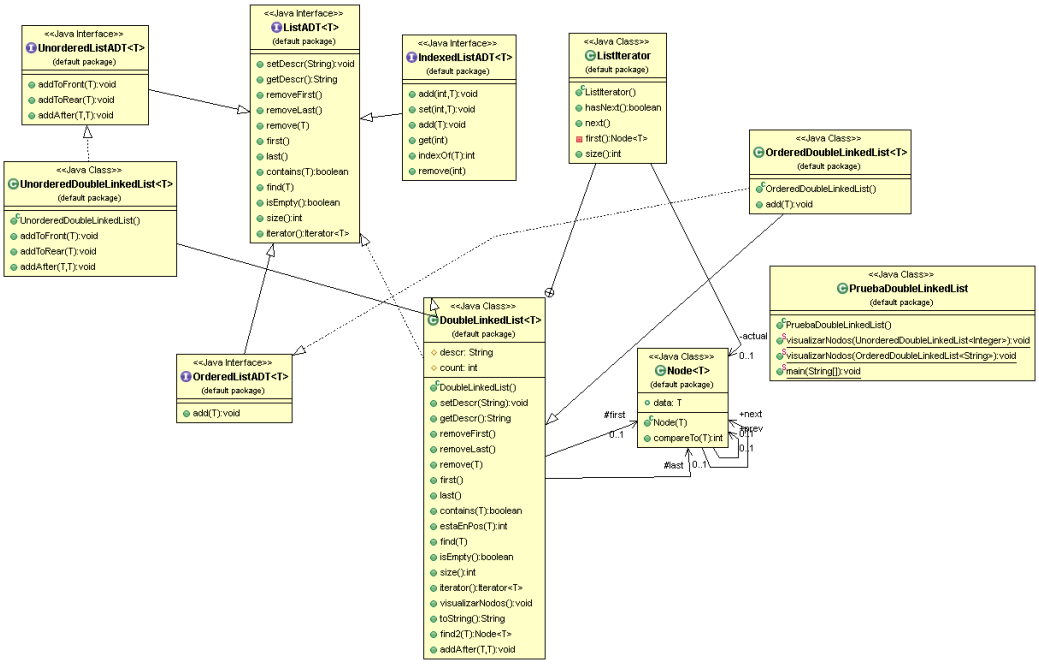

Figure 1: Diagrama de clases del proyecto

3 Descripción de las estructuras de datos principales

Hemos utilizado la siguiente estructura de datos: DoubleLinkedList.

una lista doblemente enlazada es una estructura de datos vinculados que consiste en un conjunto de forma secuencial vinculando registros llamados nodos. Cada nodo contiene dos campos, llamados enlaces, que son referencias a la anterior y al siguiente nodo en la secuencia de nodos. Ademas la lista contiene 2 referencias, el principio (First) y el final (Last) para facilitar el recorrido de la lista. Estas referencias apuntan al primer y último nodo respectivamente.

4 Diseño e implementación de los métodos principales

4.1 Método añadir al comienzo

Casos de prueba

l.inicial	elem	l.final
[ ]	x	[x]
[z]	x	[x,z]
[y,z,k]	x	[x,y,z,k]

El coste de añadir un elemento al comienzo de una lista doblemente ligada es O(1)

Código

```java
public class UnorderedDoubleLinkedList<T> extends DoubleLinkedList<T> implements
        UnorderedListADT<T> {

    public void addToFront(T elem) {
        // añade un elemento al comienzo
        // COMPLETAR EL CODIGO Y CALCULAR EL COSTE
        Node<T> nuevo = new Node<T>(elem);
        nuevo.data = elem;
        if (this.isEmpty()) {

            nuevo.next = null;
            nuevo.prev = null;
            first = nuevo;
            last = nuevo;

            this.count++;

        } else {
            Node<T> Actual = first;
            Actual.prev = nuevo;
            nuevo.next = Actual;
            nuevo.prev = null;
            first = nuevo;
            this.count++;

        }
    }

```

Figure 2: Código addToFront

```java
public static void visualizarNodos(UnorderedDoubleLinkedList<Integer> l2) {
    Iterator<Integer> it = l2.iterator();
    System.out.println("");
    while (it.hasNext()) {
        Integer num = it.next();
        System.out.println(num);
    }
}
    public static void visualizarNodos(OrderedDoubleLinkedList<String> l) {
        Iterator<String> it = l.iterator();
        System.out.println("");
        while (it.hasNext()) {
            String k = it.next();
            System.out.println(k);
        }


}

public static void main(String[] args) {

    UnorderedDoubleLinkedList<Integer> l2 = new UnorderedDoubleLinkedList<Integer>();
    //l2.addToFront(1);
    //l2.addToFront(3);
    //l.addToFront(6);
    // l2.addToRear(7);
    //l2.addToRear(9);
    // l2.addToRear(6);
    //l.addToRear(20);
     System.out.print(" Lista ...............");
    visualizarNodos(l2);



}
```

```
<terminated> PruebaDoubleLink
 Lista ...............
```

Figure 3: Pruebas Eclipse addToFront

```java
public static void visualizarNodos(UnorderedDoubleLinkedList<Integer> l2) {
    Iterator<Integer> it = l2.iterator();
    System.out.println("");
    while (it.hasNext()) {
        Integer num = it.next();
        System.out.println(num);
    }
}
    public static void visualizarNodos(OrderedDoubleLinkedList<String> l) {
        Iterator<String> it = l.iterator();
        System.out.println("");
        while (it.hasNext()) {
            String k = it.next();
            System.out.println(k);
        }


}

public static void main(String[] args) {

    UnorderedDoubleLinkedList<Integer> l2 = new UnorderedDoubleLinkedList<Integer>();
    l2.addToFront(1);
    l2.addToFront(3);
    //l.addToFront(6);
    // l2.addToRear(7);
    //l2.addToRear(9);
    // l2.addToRear(6);
    //l.addToRear(20);
     System.out.print(" Lista ...............");
    visualizarNodos(l2);
```

```
<terminated> PruebaDoubleLinke
 Lista ...............
3
1
```

Figure 4: Pruebas Eclipse addToFront

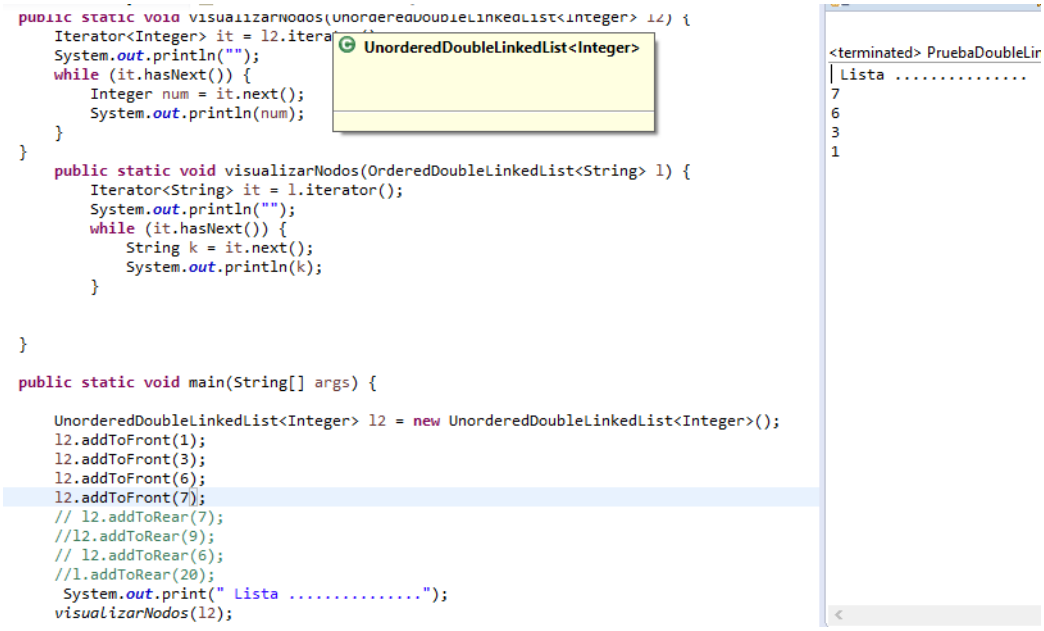

Figure 5: Pruebas Eclipse addToFront

4.2 Método añadir al final

Casos de prueba

l.inicial	elem	l.final
[ ]	x	[x]
[z]	x	[z,x]
[y,z,k]	x	[y,z,k,x]

El coste de añadir un elemento al final de una lista doblemente ligada es O(1)

```java
public void addToRear(T elem) {
    // añade un elemento al final
    // COMPLETAR EL CODIGO Y CALCULAR EL COSTE
    Node<T> nuevo = new Node<T>(elem);
    nuevo.data = elem;
    if (this.isEmpty()) {
        nuevo.next = null;
        nuevo.prev = null;
        first = nuevo;
        last = nuevo;
        this.count++;

    } else {
        Node<T> actual = last;

        nuevo.prev = actual;
        actual.next = nuevo;
        nuevo.next = null;
        last = nuevo;
        count++;

    }

}
```

Figure 6: Codigo addToRear

```
public static void visualizarNodos(UnorderedDoubleLinkedList<Integer> l2) {
    Iterator<Integer> it = l2.iterator();
    System.out.println("");
    while (it.hasNext()) {
        Integer num = it.next();
        System.out.println(num);
    }
}
    public static void visualizarNodos(OrderedDoubleLinkedList<String> l) {
        Iterator<String> it = l.iterator();
        System.out.println("");
        while (it.hasNext()) {
            String k = it.next();
            System.out.println(k);
        }


}

public static void main(String[] args) {

    UnorderedDoubleLinkedList<Integer> l2 = new UnorderedDoubleLinkedList<Integer>();
    //l2.addToFront(1);
    //l2.addToFront(3);
    //l2.addToFront(6);
    //l2.addToFront(7);
     l2.addToRear(7);
    //l2.addToRear(9);
    // l2.addToRear(6);
    //l.addToRear(20);
     System.out.print(" Lista ..............");
    visualizarNodos(l2);
```

```
<terminated> PruebaDoubleLin
Lista ..............
7
```

Figure 7: Pruebas Eclipse addToRear

```
public static void visualizarNodos(UnorderedDoubleLinkedList<Integer> l2) {
    Iterator<Integer> it = l2.iterator();
    System.out.println("");
    while (it.hasNext()) {
        Integer num = it.next();
        System.out.println(num);
    }
}
    public static void visualizarNodos(OrderedDoubleLinkedList<String> l) {
        Iterator<String> it = l.iterator();
        System.out.println("");
        while (it.hasNext()) {
            String k = it.next();
            System.out.println(k);
        }


}

public static void main(String[] args) {

    UnorderedDoubleLinkedList<Integer> l2 = new UnorderedDoubleLinkedList<Integer>();
    //l2.addToFront(1);
    //l2.addToFront(3);
    //l2.addToFront(6);
    //l2.addToFront(7);
     l2.addToRear(7);
    l2.addToRear(9);
    // l2.addToRear(6);
    //l.addToRear(20);
     System.out.print(" Lista ..............");
    visualizarNodos(l2);
```

```
<terminated> PruebaDoubleLi
Lista ..............
7
9
```

Figure 8: Pruebas Eclipse addToRear

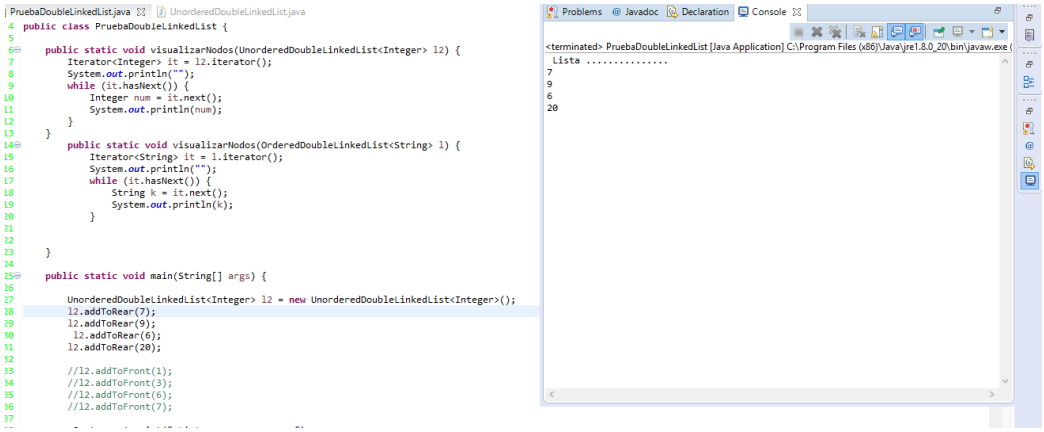

Figure 9: Pruebas Eclipse addToRear

4.3 Método añadir un elemento detrás de otro

Casos de prueba

l.inicial	elem	elem2	l.final
[ ]	a	c	”lista vacia”
[a]	a	c	[a,c]
[b,c,d]	k	e	[b,c,d]
[k,d,c]	k	e	[k,e,d,c]
[b,k,g]	k	e	[b,k,e,g]
[b,c,k]	k	e	[b,c,k,e]

El coste de addAfter es de coste O(n/2)

aunque el código del método sea constante, ya que hace uso del programa find2 que es de coste O(n/2)

```java
public void addAfter(T elem, T pTarget) {
    // Añade elem detrás de otro elemento concreto, target, que ya se
    // encuentra en la lista
    // ¡COMPLETAR OPCIONAL!

    Node<T> miNode = this.find2(pTarget);
    Node<T> nNodo = new Node<T>(elem);
    if (miNode != null) {
        nNodo.next = miNode.next;
        nNodo.prev = miNode;
        if (miNode.next != null) {
            miNode.next.prev = nNodo;
        }
        miNode.next = nNodo;

    }


}
```

Figure 10: Codigo addAfter

```java
UnorderedDoubleLinkedList<Integer> l2 = new UnorderedDoubleLinkedList<Integer>();
l2.addAfter(4, 3); //En este caso probamos el caso en el que la lista esta vacia.
 System.out.print(" Lista ..............");
visualizarNodos(l2);  //Al visualizar los nodos no se debe añadir ningun nodo
l2.addToFront(3); // Añadimos un nodo
l2.addAfter(4, 3);  // Ahora realizamos el metodo añadiendo un nodo de elemento 4 despues del unico nodo de la lista
System.out.print(" Lista ..............");
visualizarNodos(l2);  //Al visualizar los nodos se debe añadir el nodo y visualizar 3,4
l2.addToFront(1); // Añadimos un nodo de elemneto 1 al principio de la lista y la lista inicial seria 1,3,4
l2.addAfter(5, 1);  // Añadimos un nodo de elemento 5 despues del primer nodo de una lista de n nodos
System.out.print(" Lista ..............");
visualizarNodos(l2);  //Al visualizar los nodos se escribira por pantalla 1,5,3,4
l2.addAfter(6, 5);  // Añadimos un nodo de elemento 5 despues de un nodo ubicado en el medio de una lista de n nodos
System.out.print(" Lista ..............");
visualizarNodos(l2);  //Al visualizar los nodos se debe ver  1,5,6,3,4
l2.addAfter(7, 4);  // Añadimos un nodo de elemento 5 despues del ultimo nodo de una lista de n nodos
System.out.print(" Lista ..............");
visualizarNodos(l2);  //Al visualizar los nodos se debe ver  1,5,6,3,4,7
```

Figure 11: Pruebas Eclipse

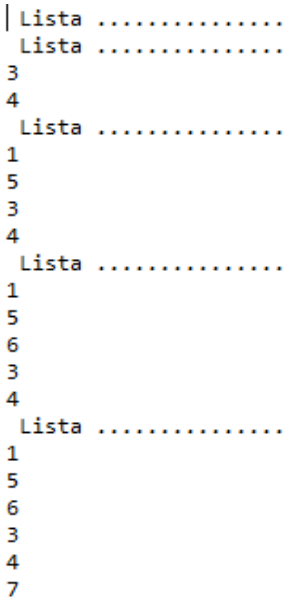

Figure 12: Resultados

4.4 Método añadir en lista ordenada

Casos de prueba

l.inicial	elem	l.final
[ ]	a	[a]
[a]	b	[a,b]
[b,c,d]	a	[a,b,c,d]
[b,d,e]	c	[b,c,d,e]
[b,d,e]	z	[b,d,e,z]

Coste: En este metodo si la lista es vacia añadir el elemento es O(1); si el elemento a añadir es menor que el primero de la lista el tiempo es O(1); si el elemento a añadir es mayor que el ultimo elementode la lista el tiempo es O(1); En el peor de los casos el elemento a añadir puede que este en el medio y entoces el coste seria O(n/2)

```java
public void add(T elem) {
    // COMPLETAR EL CODIGO Y CALCULAR EL COSTE

    boolean enc = false;
    Node<T> nuevo = new Node<T>(elem);
    Node<T> anterior = null;
    Node<T> actual = this.first;

    if (this.isEmpty()) {
        nuevo.next = null;
        nuevo.prev = null;
        first = nuevo;
        last = nuevo;

        this.count++;
        System.out.println("" + count);
    }else {

        anterior = null;

        while (actual != null && !enc) {

            if (nuevo.compareTo(actual.data) < 0) {
                enc = true;

            } else {
                anterior = actual;
                actual = actual.next;

            }
        }
```

Figure 13: Código de add

```
        if (!enc) {// añadir al final
            nuevo.next = null;
            nuevo.prev = anterior;
            anterior.next = nuevo;
            last = nuevo;

            count++;

        }

        else if (actual == this.first) {
            Node<T> Actual = first;
            Actual.prev = nuevo;
            nuevo.next = first;
            nuevo.prev = null;
            first = nuevo;
            this.count++;
        }

        else {// insertar en el medio

            nuevo.data = elem;
            nuevo.next = actual;
            anterior.next = nuevo;
            this.count++;
        }

    }
    // distinguimos 3 casos
    // esta en el principio
    // esta en el medio
    // esta en el final
    // insertar en lista vacia
```

Figure 14: Código de add

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

        OrderedDoubleLinkedList<String>l=new OrderedDoubleLinkedList<String>();
        l.add("bela");
        System.out.print(" Lista ..............");
        visualizarNodos(l);
        l.add("era");
        System.out.print(" Lista ..............");
        visualizarNodos(l);
        l.add("steve");
        l.add("andre");
        System.out.print(" Lista ..............");
        visualizarNodos(l);
        l.add("fran");
        System.out.print(" Lista ..............");
        visualizarNodos(l);
        l.add("tony");
        System.out.print(" Lista ..............");
        visualizarNodos(l);

    }
}
```

Figure 15: Pruebas de add

```
1
 Lista ..............
bela
 Lista ..............
bela
era
 Lista ..............
andre
bela
era
steve
 Lista ..............
andre
bela
era
fran
steve
 Lista ..............
andre
bela
era
fran
steve
tony
```

Figure 16: Resultados

4.5 Método borrar el primero

Casos de prueba

l.inicial	l.final	resultado
[ ]	[ ]	null
[y]	[ ]	null
[y,z,e]	[z,e]	y

Coste: El coste de este metodo es constante porque tenemos un puntero(first) que apunta al primer nodo

Código

```
public T removeFirst() {
    // Elimina el primer elemento de la lista
    // Precondición: la lista tiene al menos un elemento
    // COMPLETAR EL CODIGO Y CALCULAR EL COSTE
    Node<T> actual = this.first;
    if (actual.next == null) {
        this.first = null;
        this.last = null;
        return actual.data;

    } else {
        this.first = actual.next;
        actual.next.prev = null; // first.prev=null
        System.out.println("te he borrado");

        return actual.next.data;

    }
}
```

Figure 17: Codigo de removeFirst

```
import java.util.Iterator;

public class PruebaDoubleLinkedList {

    public static void visualizarNodos(UnorderedDoubleLinkedList<Integer> l2) {
        Iterator<Integer> it = l2.iterator();
        System.out.println("");
        while (it.hasNext()) {
            Integer num = it.next();
            System.out.println(num);
        }
    }
        public static void visualizarNodos(OrderedDoubleLinkedList<String> l) {
            Iterator<String> it = l.iterator();
            System.out.println("");
            while (it.hasNext()) {
                String k = it.next();
                System.out.println(k);
            }
        }

    public static void main(String[] args) {
        UnorderedDoubleLinkedList<Integer> l2 = new UnorderedDoubleLinkedList<Integer>();
        l2.addToRear(7);
        System.out.print(" Lista ...............");
        visualizarNodos(l2);
        l2.removeFirst();
        System.out.print(" Lista ...............");
        visualizarNodos(l2);
        /**
        l2.addToRear(9);
         l2.addToRear(6);
        l2.addToRear(20);
```

```
<terminated> PruebaDoubleLinke
 Lista ...............
7
 Lista ...............
```

Figure 18: Pruebas de removeFirst

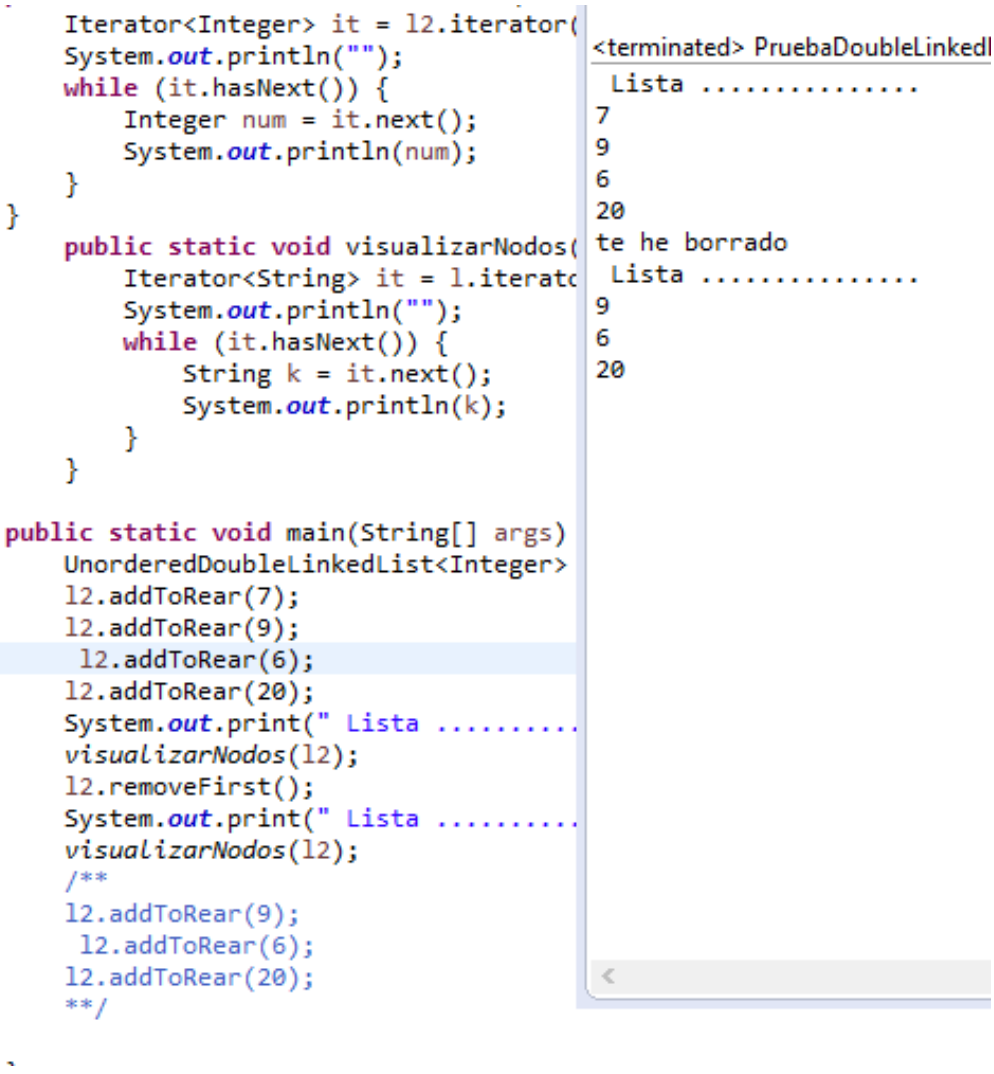

Figure 19: Pruebas de removeFirst

4.6 Método borrar el ultimo

Casos de prueba

l.inicial	l.final	resultado
[ ]	[ ]	null
[y]	[ ]	null
[y,z,e]	[y,z]	e

Coste: Coste: El coste de este metodo es constante porque tenemos un puntero(last) que apunta al ultimo nodo

Código

```
public T removeLast() {// /corregidooooooo
    // Elimina el último elemento de la lista
    // Precondición: la lista tiene al menos un elemento
    // COMPLETAR EL CODIGO Y CALCULAR EL COSTE
    Node<T> actual = this.last;
    if (actual == this.first) {

        this.last = null;
        this.first = null;
        this.count--;
        return actual.data;

    } else {

        actual = actual.prev;
        actual.next = null;
        last = actual;
        this.count--;
        return actual.data;

    }

}
```

Figure 20: Codigo de removeFirst

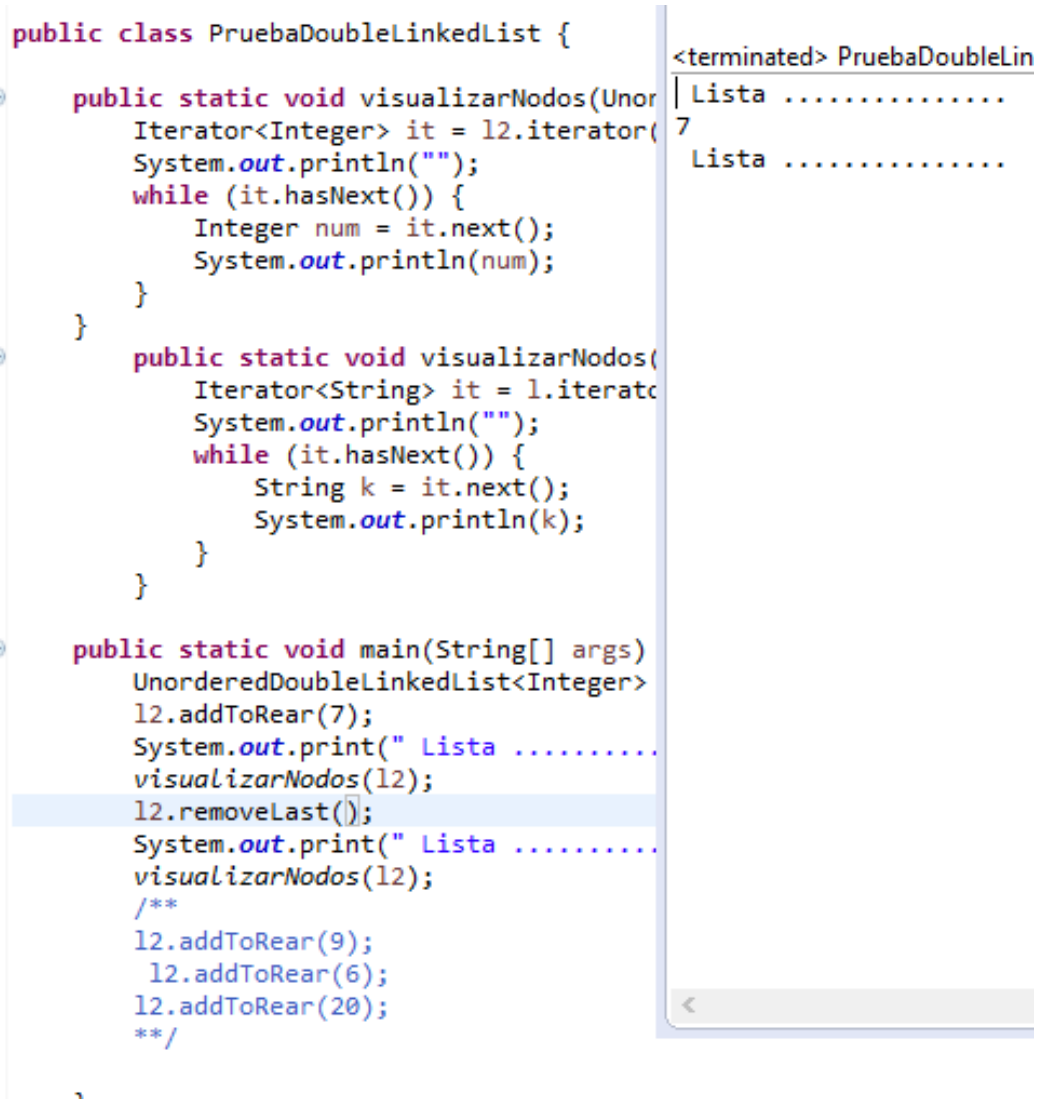

Figure 21: Pruebas de removeFirst

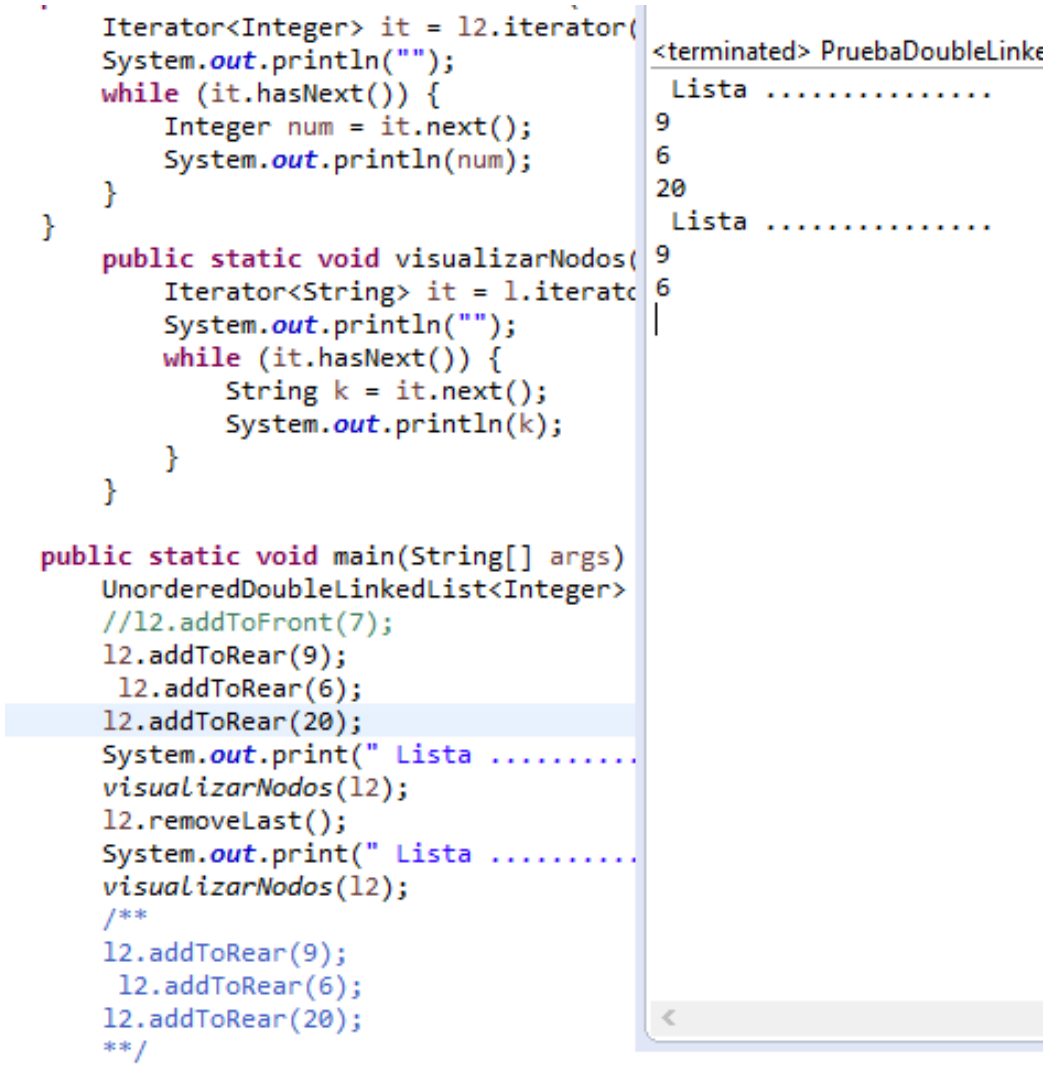

Figure 22: Pruebas de removeFirst

4.7 Método borrar un elemento concreto de la lista

Casos de prueba

l.inicial	elem	l.final	resultado
[ ]	x	[ ]	”mensaje ”
[y,z,e]	x	[y,z,e]	”mensaje ”
[x]	x	[]	x
[x,z,e]	x	[z,e]	x
[y,x,e]	x	[y,e]	x
[y,z,x]	x	[y,z]	x

El coste del metodo remove es de O(n)

ya que se suma su propio coste O(n/2) con el coste de estaEnPos que es tambien de O(n/2) y al simplificarlo da como resultado el coste de O(n)

Código

```
public T remove(T elem) {
    // Elimina un elemento concreto de la lista
    // COMPLETAR EL CODIGO Y CALCULAR EL COSTE
    int posi = this.estaEnPos(elem);
    Node<T> actual = this.first;

    // Node<T> f =this.last;
    int cont = 1;
    if (posi == -1) {
        System.out.println("no hay nada que borrar,o no exite elem");
    }

    else if (posi == 1) {
        this.removeFirst();
        return actual.data;
    }

    else {

        while (actual != last) {

            if (cont == posi) {
                actual.prev.next = actual.next;

                actual.next = actual.prev.next;
                return actual.data;
            }
            cont++;
            actual = actual.next;
        }
        this.removeLast();
        return actual.data;

    }
    return null;
}
```

Figure 23: Código de Remove

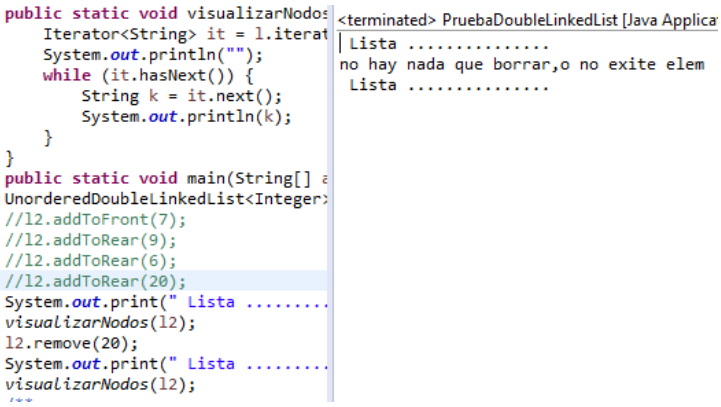

Figure 24: Pruebas de Remove

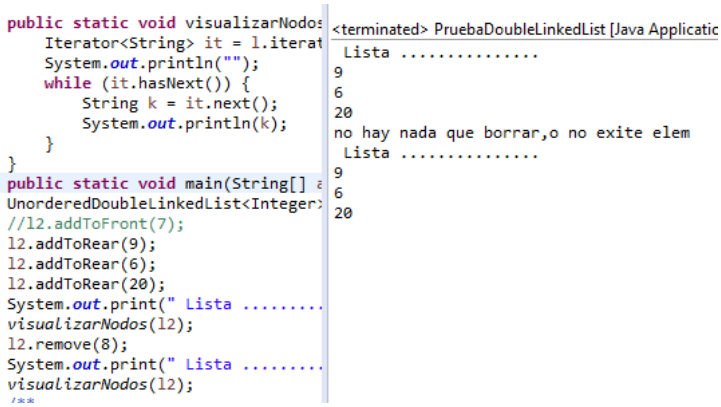

Figure 25: Pruebas de Remove

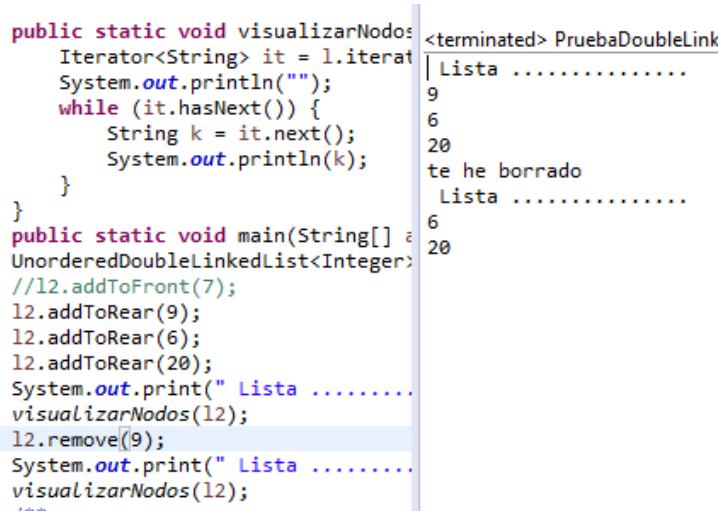

Figure 26: Pruebas de Remove

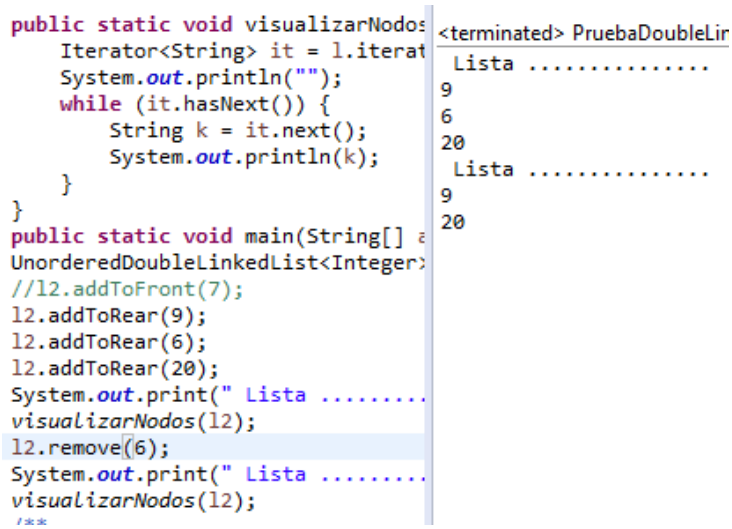

Figure 27: Pruebas de Remove

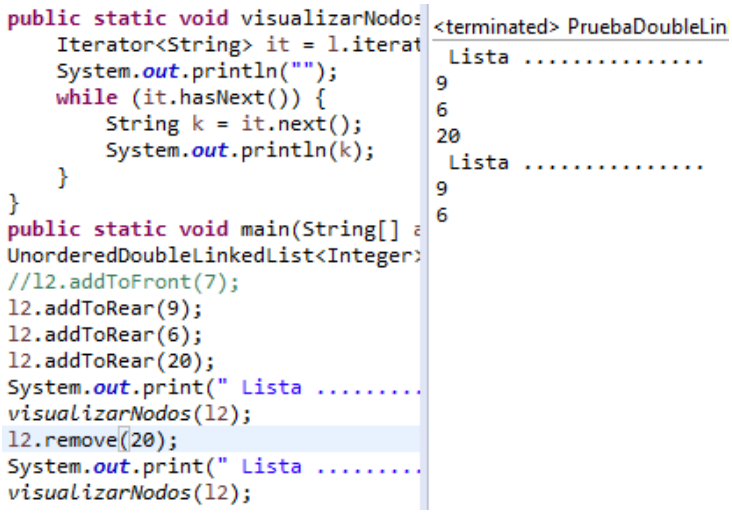

Figure 28: Pruebas de Remove

4.8 Método buscar

Casos de prueba

l.inicial	elem	resultado
[ ]	x	null
[y,z,e]	x	null
[x]	x	x
[x,z,e]	x	x
[y,x,e]	x	x
[y,z,x]	x	x

El coste del método find es de O(n/2)

Código

```
public T find(T elem) {
    // Determina si la lista contiene un elemento concreto, y develve su
    // referencia, null en caso de que no esté
    // COMPLETAR EL CODIGO Y CALCULAR EL COSTE

    if (isEmpty())
        return null;

    Node<T> actual = first; // Empieza con el segundo elemento

    while ((actual != null) && !elem.equals(actual.data))
        actual = actual.next;
    if (actual == null)
        return null;
    else
        return actual.data;

}
```

Figure 29: Código de find

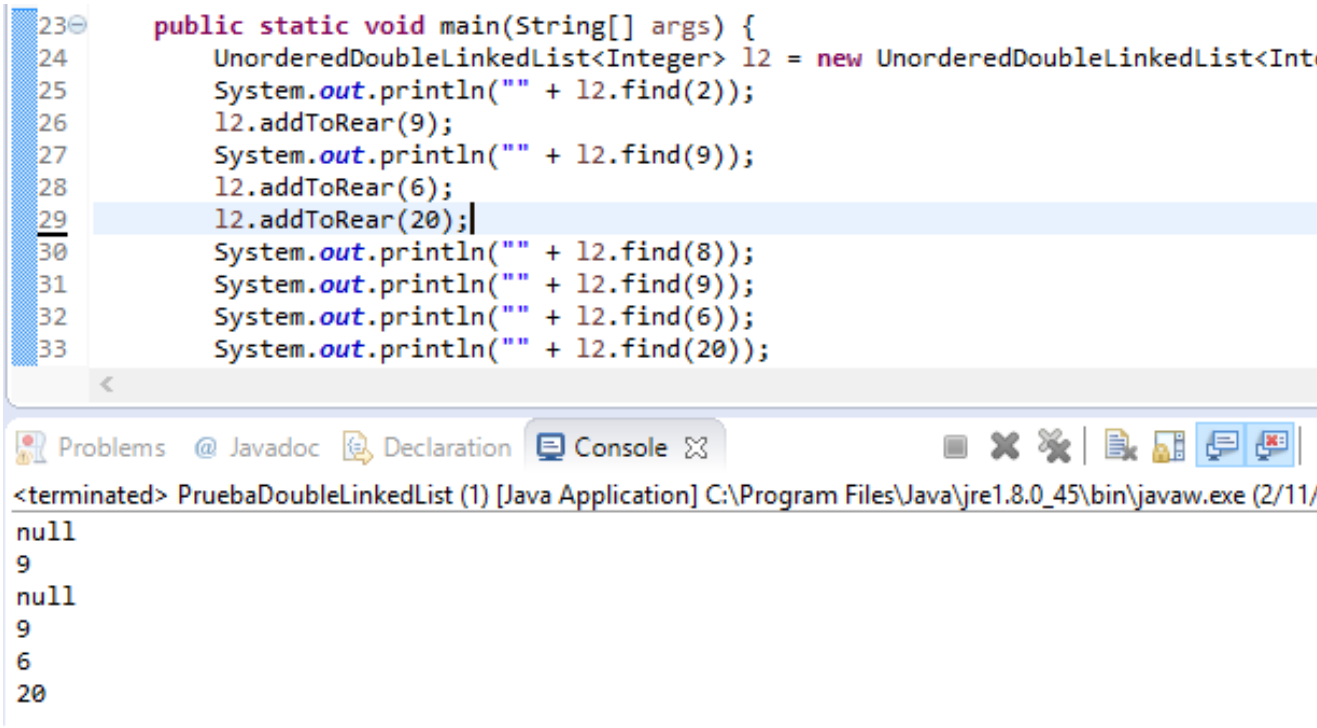

Figure 30: Pruebas de find

4.9 Cambios Realizados en Laboratorio 1

```java
import java.util.Iterator;

public class Actor implements Comparable<Actor> {
    private String nombre;

    private UnorderedDoubleLinkedList<String> ListaPe;///esto hemos cambiado

    public Actor(String o) {
        this.nombre = o;
        ListaPe = new UnorderedDoubleLinkedList<String>();

    }

    public String getNombre() {
        return nombre;
    }

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

    public LisPeli getListaP() {
        return null;
    }

    public UnorderedDoubleLinkedList<String> getDouble() {
        return this.ListaPe;
    }

    @Override
    public int compareTo(Actor A) {
        // TODO Auto-generated method stub

        return this.nombre.compareTo(A.nombre);
    }
```

Figure 31: Cambios laboratorio 1

Figure 32: Cambios laboratorio 1

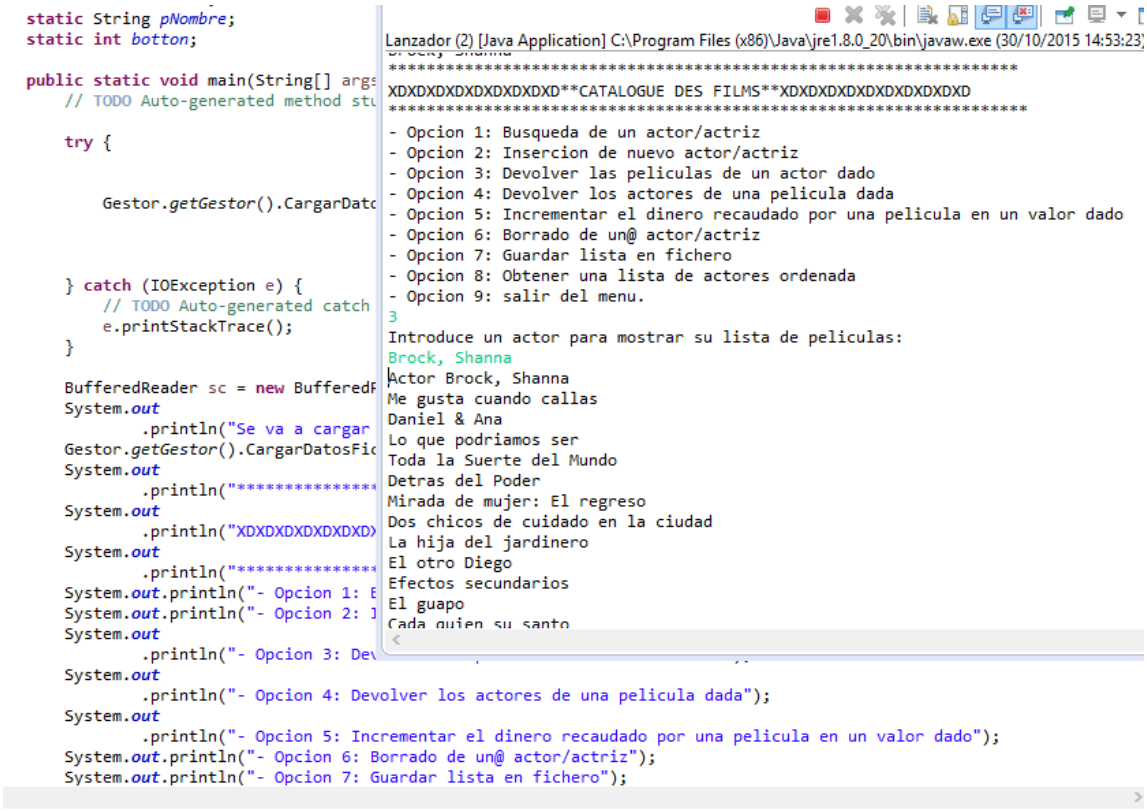

Figure 33: Prueba de cargar

5 Conclusiones

Como resultado de los cambios podemos decir que el coste de añadir y borrar en una lista doblemente enlazada es igual de eficiente que las estructura que habíamos utilizado en un principio (tablas hash)

Durante la realización de este laboratorio nos han surgido las siguientes dificultades: No actualizar correctamente los punteros first y last. La utilización del compareTo en el método add de ListaOrdenada, pero en la medida de lo posible lo hemos salvaguardado

En un futuro el método cargar datos se podra hacer mas eficiente en cuanto al espacio utilizado en memoria y se podrian guardar referencias en vez de strings. Esto es una idea teórica y habría que comprobarlo

6 correcciones laboratorio 1

Comentario Koldo: El diagrama de clases está incompleto. En el fichero pdf entregado, se ven flechas que aparecen en la parte de abajo del gráfico, por lo que parece que hay clases que no se ven. Comentario Koldo: ?Coste: el

algoritmo, en el caso peor ,puede que el coste sea O(n),Pero al ser hashMap nos presentan algoritmos de búsqueda muy rápidas y normalmente su coste suele ser O(1).? Al volver a analizar el coste nos retratamos y nos damos

cuenta que el coste es O(1) gracias a la estructura hash.

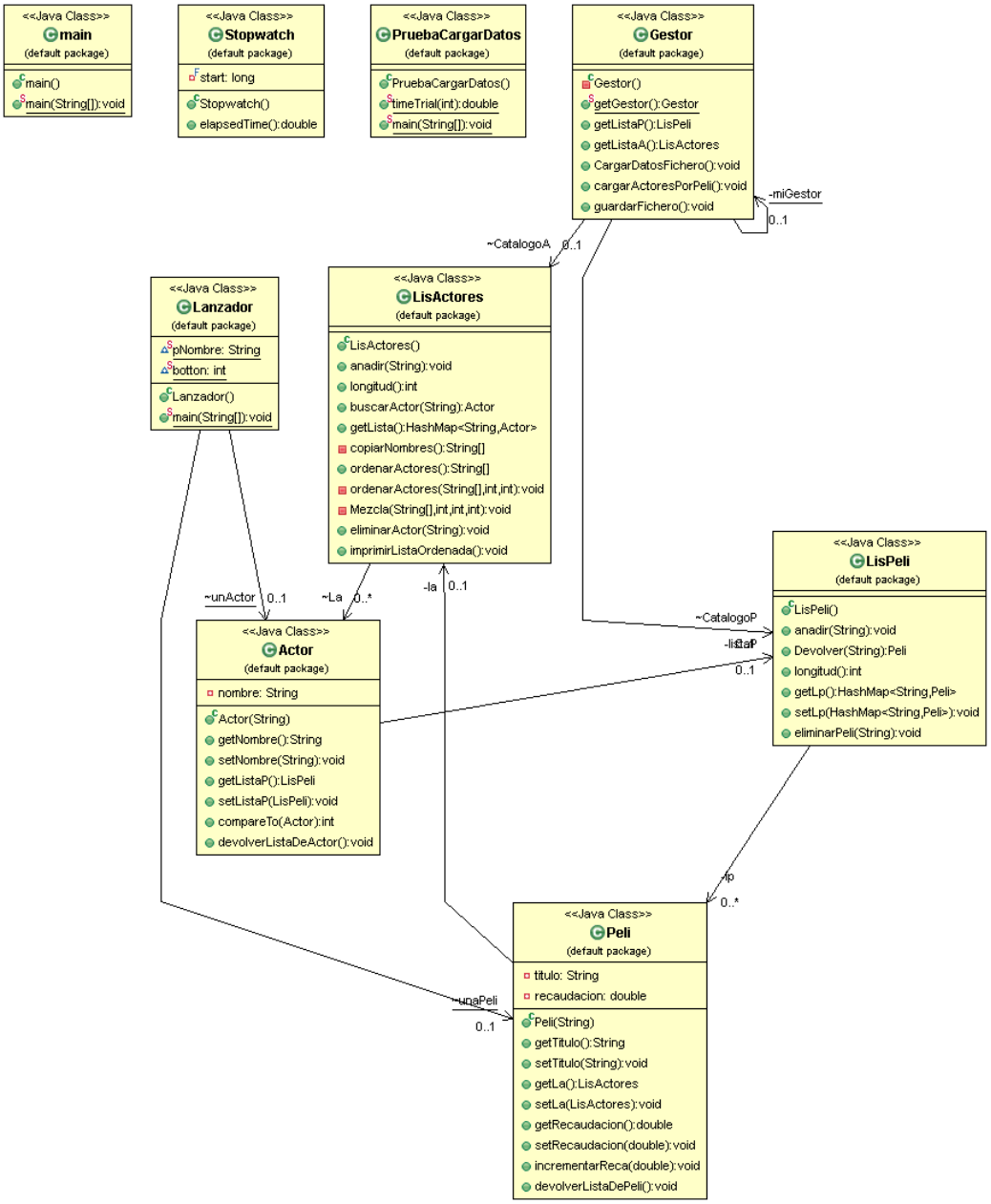

Figure 34: Corrección Diagrama

Comentario Koldo: En la página 4 de la memoria (buscarActor) no aparece el algoritmo. El coste solo se puede calcular si se ve el algoritmo. Si no hay que creérselo, y eso no vale.

4.1 Método buscarActor

public Actor buscarActor(String A) {
Precondición: La lista estará con al menos un actor ,podemos pensar que la lista este vacia tambien.
Postcondición: se devolverá el actor cuyo nombre es pasado por parámetro
casos de prueba:

Lista Inicial	"andrea"	Resultdo
[]	"andrea"	"lista vacia"
["pepe","mario"]	"andrea"	Null
["ana","karen","joe"]	"andrea"	Null
["andrea","karen","joe"]	"andrea"	Actor entero("andrea",Actor)
["karen","andrea","joe"]	"andrea"	Actor entero("andrea",Actor)
["joe","karen","andrea"]	"andrea"	Actor entero("andrea",Actor)

Coste: el algoritmo, en el caso peor ,puede que el coste sea O(n),Pero al ser hashMap nos presentan algoritmos de búsqueda muy rápidas y normalmente su coste suele ser O(1).

Figure 35: Código buscar

```
public Actor buscarActor(String A)

{
    Actor unActor = null;
    if (!this.La.isEmpty()) {


            unActor = this.La.get(A);
        }
    return unActor;
    }
```

Figure 36: Código buscar

4.2 Método ordenarActores

public String[] ordenarActores() {

Precondición: La lista contendrá al menos un actor, y estará desordenada
Postcondición: el array estará ordenado alfabéticamente

He utilizado el algoritmo de ordenación mergeSort() por John Von Neumann

1. Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:
2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.

Casos Prueba:

Lista Inicial	Lista Final
[a]	[a]
[b,a,c,d,e]	[a,b,c,d,e]
[e,b,d,c,a]	[a,b,c,d,e]

La primera partición tendrá un coste O(n),.luego n/2...n/4 ..etc

En el caso peor, medio y mejor este algoritmo tendrá un coste :O(n.logn)

Figure 37: Código ordenar

```java
private void Mezcla(String[] actores, int inicio, int centro, int fin) {
    String[] laMezcla = new String[fin - inicio + 1];

    int izq = inicio;
    int der = centro + 1;
    int k = 0;
    while (izq <= centro && der <= fin) {
        if (actores[izq].compareTo(actores[der]) <= 0) {
            laMezcla[k] = actores[izq];
            k++;
            izq++;
        } else {

            laMezcla[k] = actores[der];
            k++;
            der++;
        }
    }
    if (izq > centro) {
        while (der <= fin) {
            laMezcla[k] = actores[der];
            k++;
            der++;
        }
    } else {
        while (izq <= centro) {
            laMezcla[k] = actores[izq];
            k++;
            izq++;
        }
    }
    for (int j = inicio; j <= fin; j++) {
        actores[j] = laMezcla[j - inicio];
    }
}
```

Figure 38: Código mezclar

```java
public String[] ordenarActores() {
    String[] actores = this.copiarNombres();
    ordenarActores(actores, 0, actores.length - 1);
    return actores;
}

private void ordenarActores(String[] actores, int inicio, int fin) {
    if (inicio < fin) {
        ordenarActores(actores, inicio, ((inicio + fin) / 2));
        ordenarActores(actores, ((inicio + fin) / 2) + 1, fin);
        Mezcla(actores, inicio, ((inicio + fin) / 2), fin);
    }

}
```

Figure 39: Código ordenar