# Actividad 2: Listas enlazadas

Alejandro Ye, David Cuesta, Urtzi González, Zuhaitz Martínez

5 de noviembre de 2017

# Índice general

# Capítulo 1

# Introducción

El problema planteado es implementar una estructura de datos doblemente enlazada y comprobar su correcto funcionamiento en conjunto con la práctica anterior ('El buscador de páginas web').

- Se plantea especificar, diseñar e implementar en Java los métodos de las clases DoubleLinkedList y UnorderedDoubleLinkedList. Asimismo, se pide también el programa de pruebas que se haya diseñado y la complejidad de cada método.
- Además se deberá sustituir la nueva clase UnorderedDoubleLinkedList en alguna de las listas usadas en la fase 1 de la práctica, y comprobar que funciona correctamente (por ejemplo, en la lista de palabras clave de una web).
- También se propone la implementación de la clase OrderedDoubleLinkedList que es opcional.

# Capítulo 2

# Diseño de las clases

En esta sección se mencionarán las clases principales planteadas para la solución al problema. Por cada clase, se presentarán sus características, atributos y métodos, especificando por cada uno sus propiedades (visibilidad, parámetros, tipo de los resultados...). El gráfico siguiente muestra las clases y sus relaciones. (Véase figura 2.1.)

# Diagrama de clases

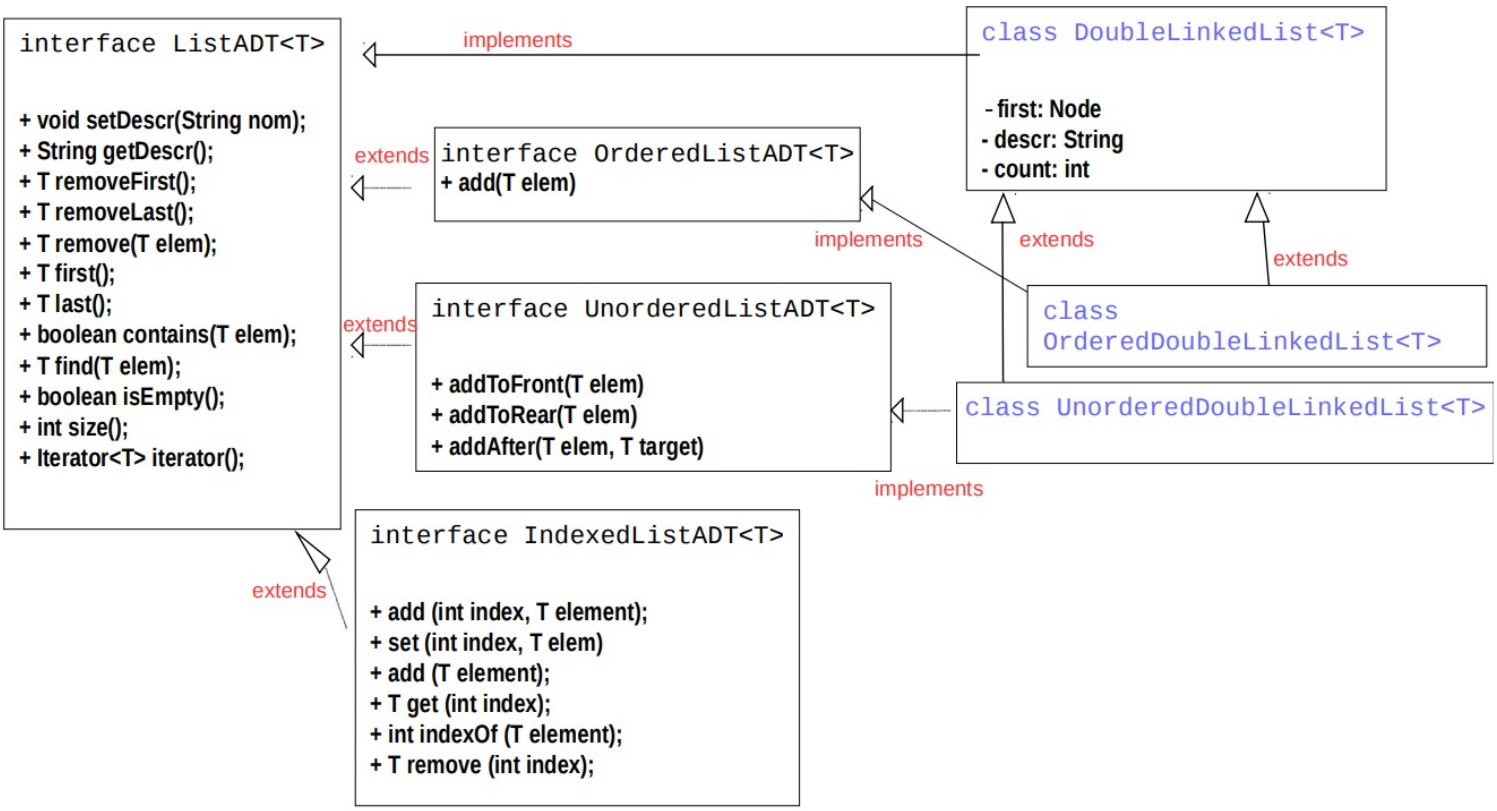


Figura 2.1: Diagrama de clases.

# Capítulo 3

# Descripción de las estructuras de datos principales

Hemos utilizado las estructuras seleccionadas por los profesores (DoubleLinkedList ordenada y desordenada).

La DoubledLinkedList unordered es una estructura eficiente respecto a memoria y para los metodos: addFirst, addLast, removeFirst y removeLast ya que esos métodos son de coste O(1). El aspecto negativo es que hay que tener cuidado con los punteros prev y next de cada nodo.

La DoubledLinkedList ordered es una estructura eficiente respecto a memoria y para los metodos: addFirst, addLast, removeFirst y removeLast ya que esos métodos son de coste O(1). Sin embargo, los metodos addAfter y find son de coste O(n), aunque el find se puede implementar para que sea mas eficiente teniendo en cuenta que la lista esta ordenada. El aspecto negativo es que hay que tener cuidado con los punteros prev y next de cada nodo.

La IndexedList es una estructura eficiente respecto a memoria y para lost metodos: addLast, removeLast y find ya que esos métodos son de coste O(1). Sin embargo metodos como: addFirst, removeFirst, addAfter... son de coste O(n) porque hay que actualizar los indices de las posiciones posteriores.

# Capítulo 4

# Diseño e implementación de los métodos principales

## 4.1. Clase DoubleLinkedList

### 4.1.1. Método removeFirst()

```
1     public T removeFirst() {
          // pre: la lista tiene al menos un elemento
3         // post: elimina y devuelve el primer elemento de la lista
          // coste: 0(1)
5
          T borrado = this.first.data;
7         if (this.count == 1){
              this.first = null;
9             this.count = this.count - 1;
          }else{
11            this.first.prev.next = this.first.next;
              this.first.next.prev = this.first.prev;
13            this.first = this.first.next;
              this.count = this.count - 1;
15        }
          return borrado;
17    }
```

### 4.1.2. Método removeLast()

```
public T removeLast() {
    // pre: la lista tiene al menos un elemento
    // post: elimina y devuelve el primer elemento de la lista
    // coste: O(1)

    T borrado = this.first.prev.data;
    if (this.count == 1){
        this.first = null;
        this.count = this.count - 1;
    }else{
        this.first.prev = this.first.prev.prev;
        this.first.prev.next = this.first;
        this.count = this.count - 1;
    }
    return borrado;
}
```

### 4.1.3. Método remove(T elem)

```
//Elimina un elemento concreto de la lista
// COMPLETAR EL CODIGO Y CALCULAR EL COSTE: O(n)
    boolean enc = false;
    Node<T> aux = this.first;
    if (this.count == 1 && aux.data.equals(elem)){
        this.first = null;
        this.count = this.count - 1;
    }else{
        while (!enc && !aux.equals(this.first.prev)){
            if (aux.data.equals(elem)){
                enc = true;
                aux.next.prev = aux.prev;
                aux.prev.next = aux.next;
            }else{
                aux = aux.next;
            }
        }
    }
    if (aux.data.equals(elem) && !enc){
        aux.next.prev = aux.prev;
        aux.prev.next = aux.next;
        enc = true;
    }
    if (enc){
        return aux.data;
    }
    return null;
}
```

### 4.1.4. Método contains(T elem)

```
public boolean contains(T elem) {
    //pre: -
    //post: devuelve un booleano en funcion de si la lista contiene elem
    //coste: O(n) siendo n el numero de elementos de la lista

    boolean enc = false;
    Node<T> aux = this.first;
    if (isEmpty())
        return false;
    else{
        while (!enc && !aux.equals(this.first.prev)){
            if (aux.data.equals(elem)){
                enc = true;
            }else{
                aux = aux.next;
            }
        }
    }
    if (aux.data.equals(elem) && !enc) enc = true;

    return enc;
}
```

### 4.1.5. Método find(T elem)

```
public T find(T elem) {
    // pre: -
    // post: determina si la lista contiene un elemento concreto, y devuelv
    // coste: O(n) siendo n el numero de elementos de la lista

    boolean enc = false;
    Node<T> aux = this.first;

    if (isEmpty()) {
        //System.out.println("entra");
        return null;

    }
    else{
        while (!enc && !aux.equals(this.first.prev)){
            if (aux.data.equals(elem)){
                enc = true;
            }else{
                aux = aux.next;
            }
        }
    }
```

```
        if (aux.data.equals(elem) && !enc) enc = true;

        if(enc) return aux.data;
        else return null;
    }
```

## 4.2. Clase UnorderedDoubleLinkedList

### 4.2.1. Método addToFront (T elem)

```
1   public void addToFront(T elem) {
        // pre: -
3       // post: anade un elemento al comienzo
        // coste: O(1)
5
        Node<T> n = new Node<T>(elem);
7       if (super.count == 0){
            super.first = n;
9           n.next = n;
            n.prev = n;
11          this.count++;
        }else{
13          n.next = super.first;
            n.prev = super.first.prev;
15          super.first.prev.next = n;
            super.first.prev = n;
17          this.first = n;
            this.count++;
19      }
    }
```

### 4.2.2. Método addToRear (T elem)

```
    public void addToRear(T elem) {
2       // pre: -
        // post: anade un elemento al final
4       // coste: O(1)

6       Node<T> n = new Node<T>(elem);
        if (super.count == 0){
8           super.first = n;
            n.next = n;
10          n.prev = n;
            this.count++;
12      }else{
            n.next = super.first;
14          n.prev = super.first.prev;
```

```
            super.first.prev.next = n;
            super.first.prev = n;
            this.count++;
        }
    }
```

### 4.2.3. Método addAfter (T elem)

```
public void addAfter(T elem, T target){
    //pre: target se encuentra en la lista
    //post: se anade elem detras de target
    //coste: O(n) siendo n el numero de elementos de la lista

    boolean enc = false;
    Node<T> aux = first;
    Node<T> nuevo = new Node<T>(elem);

    while(!enc && (aux.next != first)) {
        if (aux.data.equals(target)) {
            enc = true;
            nuevo.prev = aux;
            nuevo.next = aux.next;
            aux.next.prev = nuevo;
            aux.next = nuevo;
            count++;
        }
        else {
            aux = aux.next;
        }
    }
    if (first.prev.data.equals(target) && !enc) {
        enc = true;
        nuevo.prev = aux;
        nuevo.next = aux.next;
        aux.next.prev = nuevo;
        aux.next = nuevo;
        count++;
    }
}
```

## 4.3. Clase OrderedDoubleLinkedList

### 4.3.1. Método add (T elem)

```
public void add(T elem){
    // pre: -
    // post: si la lista esta vacia anade el elemento el principio, sino lo
    // coste: O(n) siendo n el numero de elementos de lista

    Node<T> nuevo = new Node<T>(elem);

    if (first == null) {
        nuevo.prev = nuevo;
        nuevo.next = nuevo;
        first = nuevo;
        count++;
    }else{
        Node<T> aux = first;
        boolean enc = false;
        int cont = 0;
        while ((cont < count) && !enc) {
            int comparacion = ((Comparable<T>) nuevo.data).compareTo(aux.da
            if (comparacion < 0) {
                nuevo.prev = aux.prev;
                nuevo.next = aux;
                aux.prev.next = nuevo;
                aux.prev = nuevo;
                if (aux == first)
                first = nuevo;
                enc = true;
                count++;
            }
            else {
                cont++;
                aux = aux.next;
            }
        }
        if (!enc) { //si no ha sido encontrado un elemento menor se anade a
            aux = aux.next;
            nuevo.prev = aux;
            nuevo.next = aux.next;
            aux.next.prev = nuevo;
            aux.next = nuevo;
            enc = true;
            count++;
        }
    }
}
```

## Casos de prueba (Véase el apartado 5.Código):

- Operaciones en lista vacía
- Operaciones en lista de un solo elemento:
  - es posible la operación
  - no es posible
- Operaciones en lista no vacía
  - Primer elemento de la lista
  - Último elemento de la lista
  - Por la mitad de la lista

# Capítulo 5

# Código

En este apartado se presentará el código de las clases, junto con los programas de prueba o Junits desarrollados.

## 5.1. Clase DoubleLinkedList

```
package eda;

import java.util.Iterator;
import java.util.NoSuchElementException;

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

    // ATRIBUTOS
    protected Node<T> first; // apuntador al primero
    protected String descr;  // descripcion
    protected int count;

    // CONSTRUCTORA
    public DoubleLinkedList() {
        first = null;
        descr = "";
        count = 0;
    }

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

    public String getDescr() {
        return descr;
    }

    public T removeFirst() {
        // pre: la lista tiene al menos un elemento
```

```
30          // post: elimina y devuelve el primer elemento de la lista
            // coste: O(1)
32
            T borrado = this.first.data;
34          if (this.count == 1){
                this.first = null;
36              this.count = this.count - 1;
            }else{
38              this.first.prev.next = this.first.next;
                this.first.next.prev = this.first.prev;
40              this.first = this.first.next;
                this.count = this.count - 1;
42          }
            return borrado;
44      }

46      public T removeLast() {
            // pre: la lista tiene al menos un elemento
48          // post: elimina y devuelve el primer elemento de la lista
            // coste: O(1)
50
            T borrado = this.first.prev.data;
52          if (this.count == 1){
                this.first = null;
54              this.count = this.count - 1;

56          }else{
                this.first.prev = this.first.prev.prev;
58              this.first.prev.next = this.first;
                this.count = this.count - 1;
60          }
            return borrado;
62      }

64      public T remove(T elem) {
            // pre: -
66          // post: elimina y devuelve el elemento de la lista especificado, s
            // coste: O(n) siendo n el numero de elementos de la lista
68
            boolean enc = false;
70          Node<T> aux = this.first;
            if (!isEmpty()){
72              if (this.count == 1 && aux.data.equals(elem)){
                    this.first = null;
74                  enc = true;
                }else{
76                  while (!enc && !aux.equals(this.first.prev)){
                        if (aux.data.equals(elem)){
78                          enc = true;
                            if (aux.equals(first)){
```

```
80                              first.prev.next = first.next;
                                first.next.prev = first.prev;
82                              first = aux.next;
                                //first.prev = first;
84
                            }else {
86                              aux.next.prev = aux.prev;
                                aux.prev.next = aux.next;
88                          }
                        }else{
90                          aux = aux.next;
                        }
92                  }
                }
94              if (aux.data.equals(elem) && !enc){
                    aux.next.prev = aux.prev;
96                  aux.prev.next = aux.next;
                    enc = true;
98              }
                if (enc){
100                 this.count = this.count - 1;
                    return aux.data;
102             }
            }
104
            return null;
106
        }
108
        public T first() {
110     //Da acceso al primer elemento de la lista
              if (isEmpty())
112               return null;
              else return first.data;
114     }

116     public T last() {
        //Da acceso al ultimo elemento de la lista
118
              if (isEmpty())
120               return null;
              else return first.prev.data;
122     }

124     public boolean contains(T elem) {
            //pre: -
126         //post: devuelve un booleano en funcion de si la lista contiene ele
            //coste: O(n) siendo n el numero de elementos de la lista
128
            boolean enc = false;
```

```
130         Node<T> aux = this.first;
            if (isEmpty())
132             return false;
            else{
134             while (!enc && !aux.equals(this.first.prev)){
                    if (aux.data.equals(elem)){
136                     enc = true;
                    }else{
138                     aux = aux.next;
                    }
140             }
            }
142         if (aux.data.equals(elem) && !enc) enc = true;

144         return enc;
        }
146
        public T find(T elem) {
148         // pre: -
            // post: determina si la lista contiene un elemento concreto, y dev
150         // coste: O(n) siendo n el numero de elementos de la lista

152         boolean enc = false;
            Node<T> aux = this.first;
154
            if (isEmpty()) {
156             //System.out.println("entra");
                return null;
158
            }
160         else{
                while (!enc && !aux.equals(this.first.prev)){
162                 if (aux.data.equals(elem)){
                        enc = true;
164                 }else{
                        aux = aux.next;
166                 }
                }
168         }
            if (aux.data.equals(elem) && !enc) enc = true;
170
            if(enc) return aux.data;
172         else return null;
        }
174
        public boolean isEmpty()
176     //Determina si la lista esta vacia
        { return first == null;};
178
        public int size()
```

```
180     //Determina el numero de elementos de la lista
        { return count;};
182
        /** Return an iterator to the stack that iterates through the items . *
184     public Iterator<T> iterator() { return new ListIterator(); }

186     // an iterator, doesn't implement remove() since it's optional
        private class ListIterator implements Iterator<T> {
188     // COMPLETAR EL CODIGO Y CALCULAR EL COSTE
            Node<T> act = first;
190         boolean end = false;

192             @Override
                public boolean hasNext() {  //O(1)
194                 if (act == null) return false;
                    else if(this.act.equals(first) && end) return false;
196                 else return true;
                }
198
                @Override
200             public T next() { //O(1)
                    if (!hasNext()) throw new NoSuchElementException("No hay mas
202                 else{
                        T elem = act.data;
204                     act = act.next;
                        if (act.equals(first)){
206                         end = true;
                        }
208                     return elem;
                    }
210             }


212
        } // private class
214
        public void visualizarNodos() {
216        System.out.println(this.toString());
        }
218
        @Override
220     public String toString() {
            String result = new String();
222         Iterator<T> it = iterator();
            while (it.hasNext()) {
224             T elem = it.next();
                result = result + "[" + elem.toString() + "] \n";
226         }

228         return "Numero de resultados: "  + this.count + "\n" + result;
        }
```

```
230
}
```

## 5.2. Clase UnorderedDoubleLinkedList

```
1  package segundaFase;
2
3  public class UnorderedDoubleLinkedList<T> extends DoubleLinkedList<T> imple
4
5      public void addToFront(T elem) {
6          // pre: -
7          // post: anade un elemento al comienzo
8          // coste: O(1)
9
10         Node<T> n = new Node<T>(elem);
11         if (super.count == 0){
12             super.first = n;
13             n.next = n;
14             n.prev = n;
15             this.count++;
16         }else{
17             n.next = super.first;
18             n.prev = super.first.prev;
19             super.first.prev.next = n;
20             super.first.prev = n;
21             this.first = n;
22             this.count++;
23         }
24     }
25
26     public void addToRear(T elem) {
27         // pre: -
28         // post: anade un elemento al final
29         // coste: O(1)
30
31         Node<T> n = new Node<T>(elem);
32         if (super.count == 0){
33             super.first = n;
34             n.next = n;
35             n.prev = n;
36             this.count++;
37         }else{
38             n.next = super.first;
39             n.prev = super.first.prev;
40             super.first.prev.next = n;
41             super.first.prev = n;
42             this.count++;
43         }
44     }
45
```

```
    public void addAfter(T elem, T target){
47      //OPCIONAL
    }
49
}
```

## 5.3. Clase OrderedDoubleLinkedList

```
package segundaFase;
2
public class OrderedDoubleLinkedList<T> extends DoubleLinkedList<T> impleme
4
    public void add(T elem){
6       // pre: -
        // post: si la lista esta vacia anade el elemento el principio, sin
8       // coste: O(n) siendo n el numero de elementos de lista

10      Node<T> nuevo = new Node<T>(elem);

12      if (first == null) {
            nuevo.prev = nuevo;
14          nuevo.next = nuevo;
            first = nuevo;
16          count++;
        }else{
18          Node<T> aux = first;
            boolean enc = false;
20          int cont = 0;
            while ((cont < count) && !enc) {
22              int comparacion = ((Comparable<T>) nuevo.data).compareTo(au
                if (comparacion < 0) {
24                  nuevo.prev = aux.prev;
                    nuevo.next = aux;
26                  aux.prev.next = nuevo;
                    aux.prev = nuevo;
28                  if (aux == first)
                        first = nuevo;
30                  enc = true;
                    count++;
32              }
                else {
34                  cont++;
                    aux = aux.next;
36              }
            }
38          if (!enc) { //si no ha sido encontrado un elemento menor se ana
                aux = aux.next;
40              nuevo.prev = aux;
                nuevo.next = aux.next;
42              aux.next.prev = nuevo;
```

```
                aux.next = nuevo;
44              enc = true;
                count++;
46          }
        }
48  }

50
}
```

## 5.4. Casos de prueba DoubleLinkedList

```
1  package segundaFase;

3  import java.util.Iterator;

5
   public class PruebaDoubleLinkedList {
7
       public static void visualizarNodos(UnorderedDoubleLinkedList<Integer> l
9          Iterator<Integer> it = l.iterator();
           System.out.println();
11         while (it.hasNext()) {
               Integer num = it.next();
13             System.out.println(num);
           }
15     }

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

19         UnorderedDoubleLinkedList<Integer> l = new UnorderedDoubleLinkedLis
           l.addToRear(1);
21         l.addToRear(3);
           l.addToRear(6);
23         l.addToRear(7);
           l.addToRear(9);
25         l.addToRear(0);
           l.addToRear(20);
27         l.addToFront(8);
           l.remove(new Integer(7));
29

31         System.out.print(" Lista ..............");
           visualizarNodos(l);
33         System.out.println("Num elementos: " + l.size());

35
           System.out.println("Prueba Find ..............");
37         System.out.println("9? " + l.find(9));
           System.out.println("0? " + l.find(0));
```

```
39          System.out.println("7? " + l.find(7));
41
            // CASOS DE PRUEBA NUESTROS
43
            UnorderedDoubleLinkedList<Integer> l2 = new UnorderedDoubleLinkedLi
45          int data;
47          // PRUEBAS addToRear //
49          // Anadir solo un elemento
            l2.addToRear(5);
51
            // Anadir dos veces el mismo elemento
53          l2.addToRear(6);
            l2.addToRear(6);
55
            // PRUEBAS addToFront //
57
            // Anadir solo un elemento
59          l2.addToFront(5);
61          // Anadir dos elementos
            l2.addToFront(5);
63          l2.addToFront(6);
65          // Anadir dos veces el mismo elemento
            l2.addToFront(5);
67          l2.addToFront(5);
            l2.addToFront(6);
69
            // Anadir mas de dos elementos
71          l2.addToFront(5);
            l2.addToFront(7);
73          l2.addToFront(6);
75          // PRUEBAS removeFirst //
77          // Eliminar primer elemento de una lista de solo un elemento
            l2.addToFront(10);
79          l2.removeFirst();
81          // Eliminar primer elemento de una lista de dos elementos
            l2.addToFront(10);
83          l2.addToFront(12);
            l2.removeFirst();
85
            // Eliminar primer elemento de una lista de mas de dos elementos
87          l2.addToFront(12);
            l2.addToFront(10);
```

```
89          l2.addToFront(15);
            l2.removeFirst();
91
            // PRUEBAS removeLast //
93
            // Eliminar ultimo elemento de una lista de solo un elemento
95          l2.addToFront(10);
            l2.removeLast();
97
            // Eliminar ultimo elemento de una lista de dos elementos
99          l2.addToFront(10);
            l2.addToFront(12);
101         l2.removeLast();

103         // Eliminar primer elemento de una lista de mas de dos elementos
            l2.addToFront(12);
105         l2.addToFront(10);
            l2.addToFront(15);
107         l2.removeLast();

109         // PRUEBAS remove //

111         // Eliminar un elemento de una lista de solo un elemento (esta el e
            l2.addToFront(10);
113         l2.remove(10);

115         // Eliminar un elemento de una lista de solo un elemento (no esta e
            l2.addToFront(10);
117         l2.remove(12);

119         // Eliminar un elemento (primero) de una lista de dos elementos (es
            l2.addToFront(10);
121         l2.addToFront(12);
            l2.remove(12);
123
            // Eliminar un elemento (ultimo) de una lista de dos elementos (est
125         l2.addToFront(10);
            l2.addToFront(12);
127         l2.remove(10);

129         // Eliminar un elemento de una lista de dos elementos (no esta el e
            l2.addToFront(10);
131         l2.addToFront(12);
            l2.remove(11);
133
            // Eliminar un elemento (primero) de una lista de mas de dos elemen
135         l2.addToFront(12);
            l2.addToFront(10);
137         l2.addToFront(15);
            l2.remove(15);
```

```
139
            // Eliminar un elemento (un elemento del medio) de una lista de mas
141         l2.addToFront(12);
            l2.addToFront(10);
143         l2.addToFront(15);
            l2.remove(10);
145
            // Eliminar un elemento (ultimo) de una lista de mas de dos element
147         l2.addToFront(12);
            l2.addToFront(10);
149         l2.addToFront(15);
            l2.remove(12);
151
            // PRUEBAS contains //
153
            // Lista vacia
155         System.out.print(l2.contains(5));

157         // Lista de un elemento
            l2.addToFront(2);
159         System.out.print(l2.contains(2));
            System.out.print(l2.contains(1));
161
            // Lista de dos o mas elementos
163         l2.addToFront(2);
            l2.addToFront(3);
165         System.out.print(l2.contains(2));
            System.out.print(l2.contains(3));
167
            // PRUEBAS find //
169
            // Lista vacia
171         data = l2.find(5);
            System.out.println(data);
173
            // Lista de un elemento
175         l2.addToFront(5);
            data = l2.find(5);
177         System.out.println(data);

179         // Encontrar un elemento (primero) de una lista de dos elementos
            l2.addToFront(5);
181         l2.addToFront(8);
            data = l2.find(8);
183         System.out.println(data);

185         // Encontrar un elemento (ultimo) de una lista de dos elementos
            l2.addToFront(5);
187         l2.addToFront(8);
            data = l2.find(8);
```

```
189         System.out.println(data);

191         // Encontrar un elemento (primero) de una lista de dos o mas elemen
            l2.addToFront(5);
193         l2.addToFront(8);
            l2.addToFront(12);
195         data = l2.find(12);
            System.out.println(data);
197
            // Encontrar un elemento (un elemento del medio) de una lista de do
199         l2.addToFront(5);
            l2.addToFront(8);
201         l2.addToFront(12);
            data = l2.find(8);
203         System.out.println(data);

205         // Encontrar un elemento (ultimo) de una lista de dos o mas element
            l2.addToFront(5);
207         l2.addToFront(8);
            l2.addToFront(12);
209         data = l2.find(5);
            System.out.println(data);
211
            visualizarNodos(l2);
213     }
}
```

## 5.5. Casos de prueba UnorderedDoubleLinkedList

```
package segundaFase;
2
import static org.junit.Assert.*;
4
import java.util.Iterator;
6
import org.junit.After;
8 import org.junit.Before;
import org.junit.Test;
10
/*Node<Integer> n = l2.first;
12 for(int i = 0; i<l2.count; i++){
    System.out.println(n.data);
14  n = n.prev;
}*/
16
public class UnorderedDoubleLinkedListTest {
18      public static void visualizarNodos(UnorderedDoubleLinkedList<Integer> l
            Iterator<Integer> it = l.iterator();
```

```
20          System.out.println();
            while (it.hasNext()) {
22              Integer num = it.next();
                System.out.println(num);
24          }
        }
26
        UnorderedDoubleLinkedList<Integer> l;
28
        @Before
30      public void setUp() throws Exception {
            l = new UnorderedDoubleLinkedList<Integer>();
32      }

34      @After
        public void tearDown() throws Exception {
36          l.first = null;
            l.count = 0;
38      }

40      @Test
        public void testAddToFront() {
42          System.out.println("\n---------testAddToFront()");
            visualizarNodos(l);
44          l.addToFront(1);
            visualizarNodos(l);
46          l.addToFront(2);
            visualizarNodos(l);
48      }

50      @Test
        public void testAddToRear() {
52          System.out.println("\n---------testAddToRear()");
            visualizarNodos(l);
54          l.addToRear(1);
            visualizarNodos(l);
56          l.addToRear(2);
            visualizarNodos(l);
58      }

60      @Test
        public void testRemoveFirst() {
62          System.out.println("\n---------testRemoveFirst()");
            System.out.println("Un elemento:");
64          l.addToRear(1);
            visualizarNodos(l);
66          l.removeFirst();
            visualizarNodos(l);
68          System.out.println("Varios elementos:");
            l.addToRear(2);
```

```
70          l.addToRear(3);
            l.addToRear(4);
72          l.addToRear(5);
            l.addToRear(6);
74          visualizarNodos(l);
            l.removeFirst();
76          visualizarNodos(l);
            l.removeFirst();
78          visualizarNodos(l);

80          System.out.println("\nInverso");
            Node<Integer> n = l.first.prev;
82          for(int i = 0; i<l.count; i++){
                System.out.println(n.data);
84              n = n.prev;
            }
86      }

88      @Test
        public void testRemoveLast() {
90          System.out.println("\n---------testRemoveLast()");
            System.out.println("Un elemento:");
92          l.addToRear(1);
            visualizarNodos(l);
94          l.removeLast();
            visualizarNodos(l);
96          System.out.println("Varios elementos:");
            l.addToRear(2);
98          l.addToRear(3);
            l.addToRear(4);
100         l.addToRear(5);
            l.addToRear(6);
102         visualizarNodos(l);
            l.removeLast();
104         visualizarNodos(l);
            l.removeLast();
106         visualizarNodos(l);

108         System.out.println("\nInverso");
            Node<Integer> n = l.first.prev;
110         for(int i = 0; i<l.count; i++){
                System.out.println(n.data);
112             n = n.prev;
            }
114     }

116     @Test
        public void testRemove() {
118         System.out.println("\n---------testRemove()");
            System.out.println("Vacio:");
```

```
120         System.out.println(l.remove(1));
            System.out.println("Un elemento:");
122         l.addToRear(1);
            visualizarNodos(l);
124         l.remove(1);
            visualizarNodos(l);
126         System.out.println("Varios elementos:");
            l.addToRear(2);
128         l.addToRear(3);
            l.addToRear(4);
130         l.addToRear(5);
            l.addToRear(6);
132         visualizarNodos(l);
            l.remove(2);
134         visualizarNodos(l);
            l.remove(5);
136         visualizarNodos(l);
            System.out.println("Varios elementos (no esta):");
138         l.remove(8);
            visualizarNodos(l);
140
            System.out.println("\nInverso");
142         Node<Integer> n = l.first.prev;
            for(int i = 0; i<l.count; i++){
144             System.out.println(n.data);
                n = n.prev;
146         }
        }
148
        @Test
150     public void testContains() {
            System.out.println("\n---------testContains()");
152         System.out.println("Vacio:");
            System.out.println(l.contains(1));
154         System.out.println("Un elemento:");
            l.addToRear(1);
156         visualizarNodos(l);
            System.out.println(l.contains(1));
158         System.out.println("Un elemento (no esta):");
            System.out.println(l.contains(2));
160         System.out.println("Varios elementos:");
            l.addToRear(2);
162         System.out.println(l.contains(2));
            System.out.println("Varios elementos (no esta):");
164         System.out.println(l.contains(3));
        }
166
        @Test
168     public void testFind() {
            System.out.println("\n---------testFind()");
```

```
170             System.out.println("Vacio:");
                System.out.println(l.find(1));
172             System.out.println("Un elemento:");
                l.addToRear(1);
174             visualizarNodos(l);
                System.out.println(l.find(1));
176             System.out.println("Un elemento (no esta):");
                System.out.println(l.find(2));
178             System.out.println("Varios elementos:");
                l.addToRear(2);
180             System.out.println(l.find(2));
                System.out.println("Varios elementos (no esta):");
182             System.out.println(l.find(3));
        }
184
    }
```

## 5.6. Casos de prueba DoubleLinkedList

```
1   package segundaFase;

3   public class PruebaOrderedDoubleLinkedList {

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

7               OrderedDoubleLinkedList<Integer> l = new OrderedDoubleLinkedLis
                l.add(1);
9               l.add(3);
                l.add(6);
11              l.add(7);
                l.add(9);
13              l.add(0);
                l.add(20);
15              l.visualizarNodos(); //
                l.remove(new Integer(7));
17

19              System.out.print(" Lista ..............");
                l.visualizarNodos();
21              System.out.println(" Num elementos: " + l.size());

23

                System.out.println("Prueba Find ..............");
25              System.out.println("20? " + l.find(20));
                System.out.println("9? " + l.find(9));
27              System.out.println("9? " + l.find(9));
                System.out.println("0? " + l.find(0));
29              System.out.println("7? " + l.find(7));

31
```

```
33              OrderedDoubleLinkedList<Persona> l2 = new OrderedDoubleLinkedLi
                l2.add(new Persona("jon", "1111"));
35              l2.add(new Persona("ana", "7777"));
                l2.add(new Persona("amaia", "3333"));
37              l2.add(new Persona("unai", "8888"));
                l2.add(new Persona("pedro", "2222"));
39              l2.add(new Persona("olatz", "5555"));

41              l2.remove(new Persona("", "8888"));

43

                System.out.print(" Lista ..............");
45              l2.visualizarNodos();
                System.out.println(" Num elementos: " + l2.size());
47

49              System.out.println("Prueba Find ..............");
                System.out.println("2222? " + l2.find(new Persona("", "2222")))
51              System.out.println("5555? " + l2.find(new Persona("", "5555")))
                System.out.println("7777? " + l2.find(new Persona("", "7777")))
53              System.out.println("8888? " + l2.find(new Persona("", "8888")))

55
    }
57  }
```

## 5.7. Implementación de UnorderedDoubleLinked-List en la práctica 1

```
package eda;
2
import java.util.HashMap;
4
//import java.util.Map;
6 //import java.util.Set;


8
public class Diccionario {
10      //Atributos
        private static Diccionario miDiccionario = null;
12      private HashMap<String, UnorderedDoubleLinkedList<String>> palabras = n

14      //Constructora
        private Diccionario(){
16          this.palabras = new HashMap<String, UnorderedDoubleLinkedList<Strin
        }
18
        public static Diccionario getDiccionario() {
```

```
20          if(miDiccionario == null) {
                miDiccionario = new Diccionario();
22          }
            return miDiccionario;
24      }

26      //Devuelve el hashmap.
        public HashMap<String, UnorderedDoubleLinkedList<String>> getPalabras()
28          return this.palabras;
        }
30
        //Esto sera para la constructora, para poder añadir las palabras al dic
32      //pre: Una palabra del diccionario no vacia.
        //post: Annade esa palabra como key en caso de no existir.
34      //Coste: O(1)
        public void anadirPalabra(String palabra){
36          if (!this.palabras.containsKey(palabra)){
                UnorderedDoubleLinkedList<String> l = new UnorderedDoubleLinked
38              this.palabras.put(palabra, l);
            }
40      }

42      //Metodo que se utilizara para introducir el nombre de una Web en las p
        //pre: Un nombre no vacio de una web.
44      //post: Annade el nombre de la web a todas las palabras que utiliza esa
        //Coste: O(n), siendo n la longitud de la palabra entrante.
46      public void añadirWeb(String web){
            String[] w = web.split("\\.");
48          //String[] w2 = w[0].split("[0-9]+");
            String palabra = w[0];
50          for (int l = 4; l<=palabra.length(); l++){
                for (int pos = 0; pos<=(palabra.length()-l); pos++) {
52                  String posible = palabra.substring(pos, pos+l);
                    if(palabras.containsKey(posible)){
54                      palabras.get(posible).addToRear(web);
                    }
56              }
            }
58
        }
60
        //Metodo que se utilizara para eliminar el nombre de una Web en las pal
62      //pre: Un nombre no vacio de una web.
        //post: Elimina el nombre de la web en todas las palabras que utiliza e
64      //Coste: O(n), siendo n la longitud de la palabra entrante.
        public void eliminarWeb(String web){
66          String[] w = web.split("\\.");
            //String[] w2 = w[0].split("[0-9]+");
68          String palabra = w[0];
            for (int l = 4; l<=palabra.length(); l++){
```

```
70              for (int pos = 0; pos<=(palabra.length()-l); pos++) {
                    String posible = palabra.substring(pos, pos+l);
72                  if(palabras.containsKey(posible)){
                        palabras.get(posible).remove(web);
74                  }
                }
76          }
        }
78
        //pre: Recibe lo que ha sido escrito por teclado.
80      //post: Devuelve la lista de paginas relacionadas con la palabra.
        //Coste: O(n*m*l), siendo n la longitud de la palabra entrante, y m la
82      public void buscarWeb(String[] busca){
            boolean sinCoincidencias = true;
84          for (int i = 0; i< busca.length; i++){
                for (int l = 4; l<=busca[i].length(); l++){
86                  for (int pos = 0; pos<=(busca[i].length()-l); pos++) {
                        String posible = busca[i].substring(pos, pos+l);
88                      if(palabras.containsKey(posible)){
                            System.out.println("PARA: "+ posible);
90                          palabras.get(posible).visualizarNodos();
                            sinCoincidencias = false;
92                      }

94                  }
                }
96          }
            if(sinCoincidencias){
98              System.out.println("No se han encontrado ninguna coincidencia")
            }
100     }


102


104     //Imprime todas las keys y sus listas.
        public void imprimir(){
106         for(String key: palabras.keySet()){
                System.out.println(key + " - " + palabras.get(key));
108         }

110     }

112     //Vacia el hashmap.
        public void reset(){
114         this.palabras.clear();
        }
116 }
```

# Capítulo 6

# Conclusiones

Después de finalizar la práctica hemos aprendido que existen estructuras más apropiadas que Arraylist, más eficientes en cuanto a tiempo y sobretodo memoria.

Asimismo hemos aprendido como están implementadas las listas que normalmente están ya implementadas en bibliotecas de java, que por gracia y desgracia, esta abstracción nos facilita su uso pero a su vez no vemos el código en sí así que no sabemos como está programado a menos que miremos detalladamente la documentación de la biblioteca. De esta manera sabemos como están implementados porque lo hemos hecho nosotros.