Listas Enlazadas

Fecha: 04-11-2024
Participantes: Alex Isasi
Liviu Deleanu

1. Introducción

En esta práctica se nos pide especificar, diseñar e implementar en Java los métodos de la clase DoubleLinkedList y UnorderedDoubleLinkedList que figuran en la tabla adjunta. 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 (la sustitución y las pruebas de que funcione se encuentran en el apartado 5).

Por último, de manera opcional, se nos pide implementar los métodos de la clase OrderedDoubleLinkedList.

2. Diseño de las clases

Tenemos un total de 6 archivos .java, de los cuales 2 son interfaces y el resto clases. También tenemos una clase privada definida en el archivo .java para la clase DoubleLinkedList.

La interfaz que abarca a todas las clases es la interfaz ListADT. En ella se presentan todos los métodos que tiene que incluir todas las clases que consideren usar una lista enlazada. También tenemos la interfaz UnorderedListADT que contiene todos los métodos de ListADT y 3 métodos más (más tarde se especificarán todos los métodos).

Por otro lado, tenemos una clase madre DoubleLinkedList que implementa todos los métodos de ListADT y por último, la clase UnorderedDoubleLinkedList, que es una clase hija de DoubleLinkedList y además implementa los métodos de la interfaz UnorderedListADT (a su vez contiene los métodos de la interfaz de ListADT).

Para las listas enlazadas, vamos a usar la clase Node<T>, para poder representar cada elemento de la lista y enlazarlo con el siguiente y con el anterior.

Para las pruebas, tenemos el archivo .java con el nombre de PruebaDoubleLinkedList, que probamos todos los métodos, tanto de la clase DoubleLinkedList como de UnorderedDoubleLinkedList.

ListADT<T> (Interfaz)

-Métodos:

```
public void setDescr(String nom);
public String getDescr();
public T removeFirst();
public T removeLast();
public T remove(T elem);
public void removeAll(T elem);
public T first();
public T last();
public ListADT<T> clone();
public boolean contains(T elem);
public T find(T elem);
```

public boolean isEmpty();
public int size();
public Iterator<T> iterator();

UnorderedListADT<T> (Interfaz) extends ListADT<T>

-Métodos:

public addToFront(T elem);
public addToRear(T elem);
public addAfter(T elem, T target);

DoubleLinkedList (TAD) implements ListADT<T>

-Atributos:

protected Node<T> first:
protected Node<T> last;
protected String descr;
protected int count;

-Métodos:

(Implementa los métodos de la interfaz ListADT)
public DoubleLinkedList(); //constructora
protected DoubleLinkedList<T> createList(); //método que nos proporcionó el profesor para que pudiera funcionar clone() en la clase UnorderedDoubleLinkedList
public void visualizarNodos();
public String toString();

ListIterator (private, TAD) implements Iterator<T>

-Atributos:

private Node<T> actual;

-Métodos:

public boolean hasNext();
public T next();

UnorderedDoubleLinkedList (TAD) extends DoubleLinkedList<T> implements UnorderedListADT<T>

-Métodos:

(Implementa los métodos de la interfaz ListADT y UnorderedListADT)

protected DoubleLinkedList<T> createList();

Node<T> (TAD)

-Atributos:

public T data;

public Node<T> prev;

public Node<T> next;

-Métodos:

public Node (T dd); //constructora

Clases no utilizadas para nuestra solución:

IndexedListADT<T> (interfaz) extends ListADT<T>

-Métodos:

public void add (int index, T element);

public void set (int index, T element);

public void add (T element);

public T get (int index);

public int indexOf (T element);

public T remove (int index);

OrderedListADT<T> (interfaz) extends ListADT<T>

-Métodos:

public void add(T elem);

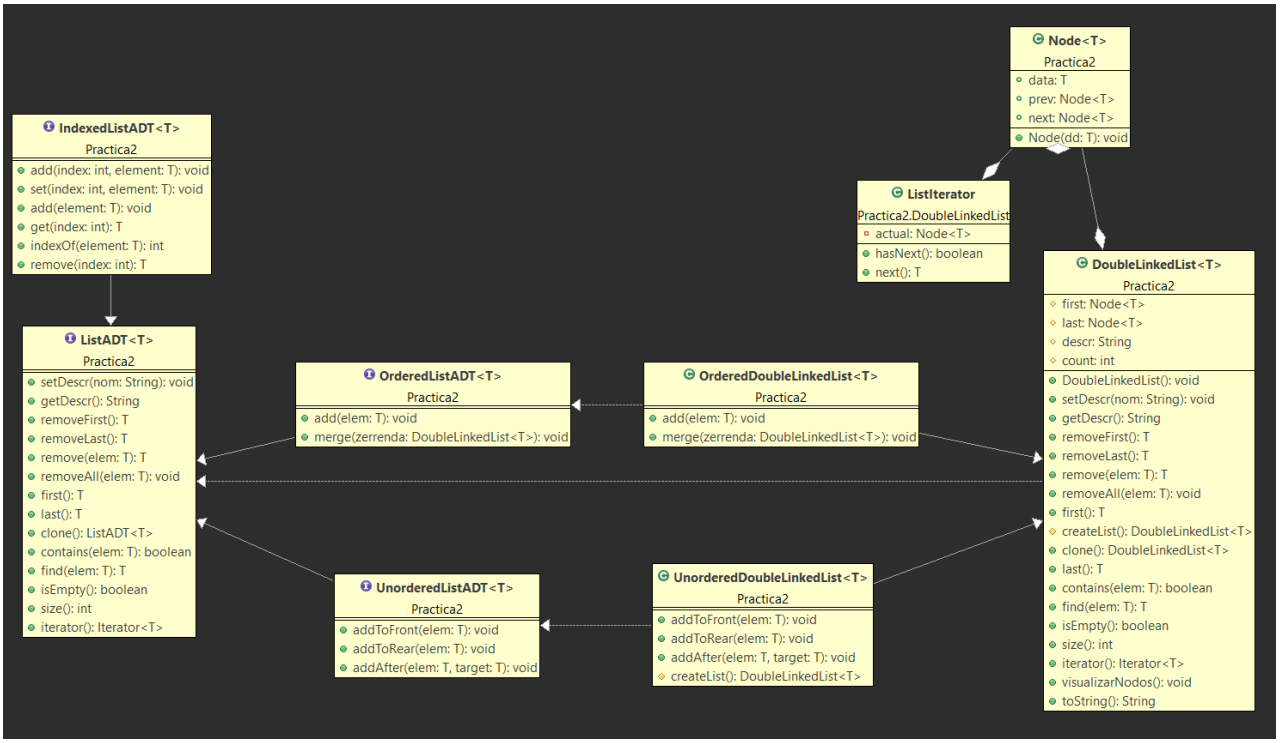
public void merge(DoubleLinkedList<T> zerrenda);

OrderedDoubleLinkedList<T> extends DoubleLinkedList<T> implements OrderedListADT<T>

-Métodos:

public void add(T elem);

public void merge(DoubleLinkedList<T> zerrenda);



3. Descripción de las estructuras de datos principales

La estructura principal de esta actividad es el Nodo o Node. Mediante este, se puede guardar tanto el dato que queramos almacenar como el nodo siguiente y el anterior a dicho nodo. Uniendo varios nodos podemos formar una lista ligada o enlazada la cual será la solución a nuestro ejercicio.

Para acceder a esta lista de nodos solo es necesario tener la referencia a uno de ellos, ya que al estar enlazados doblemente podemos recorrerlos en ambas direcciones, pero para la solución planteada, guardaremos tanto el primer nodo, es decir, el nodo que no tiene un nodo anterior, como el último nodo, es decir, el nodo que no tiene un nodo posterior. De esta manera, tendremos mayor facilidad a la hora de añadir o eliminar información tanto al principio como al final de la lista.

4. Diseño e implementación de los métodos principales

Todas las implementaciones que se explican a continuación pertenecen tanto a la clase DoubleLinkedList como a la clase UnorderedDoubleLinkedList.

4.1 DoubleLinkedList<T>

4.1.1 método setDescr()

```
public void setDescr(String nom)
          -Precondición:
          -Postcondición: el atributo Descr tomará por valor
el que le hayamos pasado como parámetro.
```

Casos de prueba:

- (No hay ningún caso de prueba real puesto que es una asignación al atributo descr)

Coste: Siempre es una asignación. Coste constante. O(1)

Implementación del algoritmo:

```
descr = nom;
```

4.1.2 método getDescr()

```
public String getDescr()
          -Precondición:
          -Postcondición: nos devuelve el valor que toma el
atributo descr.
```

Casos de prueba:

- (No hay ningún caso de prueba real puesto que es un return, devuelve un valor)

Coste: Devolver el valor asignado a una variable o atributo siempre es coste constante. O(1)

Implementación del algoritmo:

```
return descr;
```

4.1.3 método removeFirst()

```
public T removeFirst()
```

```
          -Precondición:
          -Postcondición: se borrará el primer elemento de la
lista y devuelve su referencia.
```

Casos de prueba:

- Borrar el primer elemento con 1 elemento en la lista.
- Borrar el primer elemento con varios elementos en la lista.

Coste: Puesto que tenemos la referencia del primer elemento con el atributo first, el algoritmo actualiza los punteros y lo que hace es apuntar al siguiente y los demás elementos que dejen de apuntar al elemento borrado. Coste constante. O(1)

Implementación del algoritmo:

```
T d = null;
si (está vacía la lista) {}
si no si (hay un solo elemento en la lista) {
     d = first.data;
     first = null;
     last = null;
     count--;
     }
     si no {
          d = first.data;
          first = first.next;
          first. prev = null;
          count--;
     }
return d;
```

4.1.4 método removeLast()

```
public T removeLast()
```

```
          -Precondición:
          -Postcondición: se borrará el último elemento de la
lista y devuelve su referencia.
```

Casos de prueba:

- Borrar el último elemento con 1 elemento en la lista.
- Borrar el último elemento con varios elementos en la lista.

Coste: Puesto que tenemos la referencia del último elemento con el atributo last, el algoritmo actualiza los punteros y lo que hace es apuntar al anterior y los demás elementos que dejen de apuntar al elemento borrado. Coste constante. O(1)

Implementación del algoritmo:

```
T d = null;
si (está vacía la lista) {}
si no si (hay un solo elemento en la lista) {
     d = first.data;
     last = null;
     first = null;
     count--;
     }
si no {
     d = last.data;
     last = last.prev;
     last.next = null;
     count--;
}
return d;
```

4.1.5 método remove()

```
public T remove(T elem)
```

```
          -Precondición:
          -Postcondición: se borrará el primer elemento que
sea elem y devuelve su referencia.
```

Casos de prueba:

- Borrar el elemento con 1 elemento en la lista.
- Intentar borrar el elemento que no está con 1 elemento en la lista
- Borrar el elemento que está por en medio con varios elementos en la lista.
- Borrar el elemento que está al final con varios elementos en la lista.
- Borrar el elemento que está al principio con varios elementos en la lista.
- Borrar el elemento, el cual está repetido. Hay más elementos en la lista. Se tiene que borrar la primera aparición

Coste: En el peor de los casos, si se encuentra en la última posición la primera aparición del elemento que queremos borrar, entonces habrá que recorrer toda la lista hasta encontrarlo y borrarlo. También si el elemento que queremos borrar no está en la lista. Coste lineal. O(n)

Implementación del algoritmo:

```
Node<T> actual;
Node<T> anterior;
boolean encontrado = false;
T d = null;

si (está vacía la lista) {}
si no si (el primer elemento es igual que elem) {
     d = this.removeFirst();
}
si no {
     actual = this.first;
     anterior = null;
     mientras(haya un siguiente y no se haya encontrado){
          si (el actual es igual que elem) {
               d = actual.data;
               actualizar punteros para que nada apunte
               al elemento a borrar;
```

```
                encontrado = true;
                count--;
           {
           si no {
                anterior = actual;
                actual = actual.next;
           }
     si (el actual es igual que elem){
           //hay que revisar la última posición
           this.removeLast();
           encontrado = true;
     }
return d;
```

4.1.6 método removeAll()

```
public void removeAll(T elem)
          -Precondición:
          -Postcondición: se borrará todas las apariciones de
elem
```

Casos de prueba:

- Borrar el elemento con 1 elemento en la lista.
- Intentar borrar el elemento que no está con 1 elemento en la lista
- Borrar el elemento que está por en medio con varios elementos en la lista.
- Borrar el elemento que está al final con varios elementos en la lista.
- Borrar el elemento que está al principio con varios elementos en la lista.
- Borrar el elemento, el cual aparece repetido uno después del otro. Hay más elementos en la lista.
- Borrar el elemento, el cual aparece repetido uno después del otro hasta el final de la lista (toda la lista llena del mismo elemento)
- Intentar borrar el elemento que no está con todos los elementos de la lista repetidos.
- Borrar el elemento, el cual aparece repetido uno después del otro hasta el final de la lista, pero el primer elemento es distinto.

Coste: Siempre va a tener que recorrer toda la lista para borrar todas las apariciones. El coste siempre es lineal. O(n)

Implementación del algoritmo:

```
Node<T> first;
Node<T> last;
si (está vacía la lista) {}
si no {
     act = first;
     ant = first;
     mientras (haya un siguiente) {
          si (el actual es igual que elem) {
               si (el actual está en el primero) {
                    eliminar primero(this.removeFirst());
               } si no {
                    eliminar actual;
                    actualizar punteros;
                    avanzar posición;
```

```
                this.count = this.count - 1;
            }
        } si no {
            avanzar posición (actualizar punteros)
        }
}
si (actual es igual que elem) {
     borrar último(this.removeLast);
}
```

4.1.7 método first()

```
public T first()
          -Precondición:
          -Postcondición: da acceso al primer elemento de la
lista.
```

Casos de prueba:

- Hay 1 elemento en la lista
- La lista está vacía.
- Hay varios elementos en la lista.

Coste: Puesto que tenemos la referencia al primero, podemos ver directamente cuál es el elemento de la primera posición. Coste constante. O(1)

Implementación del algoritmo:

```
si (está vacía la lista) {
return null;
}
si no {
return first.data;
}
```

4.1.8 método clone()

```
public DoubleLinkedList<T> clone()
```

```
-Precondición:
```

```
-Postcondición: Devuelve una copia de la lista (copia todos los nodos).
```

Casos de prueba:

- Copiar la lista con un elemento.
- Copiar la lista vacía.
- Copiar la lista con varios elementos.

Coste: Siempre recorre toda la lista para ir copiándola, por tanto si 'n' son los elementos, tendrá coste lineal. O(n)

Implementación del algoritmo:

```
DoubleLinkedList<T> resultado
Node<T> act = this.first;
Node<T> ult = null;
si (está vacía la lista) {}
si no {
     mientras(no se hayan recorrido todos los elementos){
          Node<T> nuevo = new Node<T>(act.data);
          si (la lista resultado está vacía) {
               el primero y el último apuntan al nuevo de
               la lista resultado;
               ult apunta al primero;
          }
          si no {
               el siguiente del último apunta al nuevo;
               el anterior del nuevo apunta al último;
               ult apunta al siguiente (que es el nuevo);
          }
          act = act.next;
     }
}
return resultado;
```

4.1.9 método last()

```
public T last()
          -Precondición:
          -Postcondición: da acceso al último elemento de la
lista.
```

Casos de prueba:

- Hay 1 elemento en la lista
- La lista está vacía.
- Hay varios elementos en la lista.

Coste: Puesto que tenemos la referencia al último, podemos ver directamente cuál es el elemento de la última posición. Coste constante. O(1)

Implementación del algoritmo:

```
si (está vacía la lista) {
return null;
}
si no {
return last.data;
}
```

4.1.10 método find()

Determina si la lista contiene un elemento concreto, y devuelve su referencia, null en caso de que no esté

```
public T find(T elem)
```

```
-Precondición:
-Postcondición: Determina si la lista contiene un elemento concreto, y devuelve su referencia, null en caso de que no esté.
```

Casos de prueba:

- Hay 1 elemento en la lista y ese es el que queremos encontrar.
- Hay 1 elemento en la lista y no es el que queremos encontrar.
- Hay varios elementos en la lista y el elemento que queremos encontrar está por en medio.
- Hay varios elementos en la lista y el elemento que queremos encontrar está al principio.
- Hay varios elementos en la lista y el elemento que queremos encontrar está al final.

Coste: En el peor de los casos, si se ubica al final, entonces habrá que recorrer toda la lista. Coste lineal. O(n)

Implementación del algoritmo:

```
d = null;
Node<T> actual;
boolean enc = false;
si (está vacía la lista) {
     return null;
}si no {
     actual = first;
     mientras(haya un siguiente y no se haya encontrado){
          si (el actual es el mismo que elem) {
               d = actual.data;
               encontrado = true;
          } si no {
               pasar al siguiente;
          }
     }
```

```
}
return d;
```

4.1.11 método contains()

```
public boolean contains(T elem)
```

-Precondición:

-Postcondición: Determina si la lista contiene un elemento concreto, indicando true si está y false si no está.

Casos de prueba:

- Hay 1 elemento en la lista y ese es el que queremos encontrar.
- Hay 1 elemento en la lista y no es el que queremos encontrar.
- Hay varios elementos en la lista y el elemento que queremos encontrar está por en medio.
- Hay varios elementos en la lista y el elemento que queremos encontrar está al principio.
- Hay varios elementos en la lista y el elemento que queremos encontrar está al final.

Coste: En el peor de los casos, si se ubica al final o no está, entonces habrá que recorrer toda la lista, ya que hace una llamada al método find. Coste lineal. O(n)

Implementación del algoritmo:

```
boolean encontrado = false;
si (encontramos el elemento) {
     encontrado = true;
}
return encontrado;
```

4.1.12 método isEmpty()

```
public boolean isEmpty()
          -Precondición:
          -Postcondición: indica si la lista está vacía o no.
```

Casos de prueba:

- La lista sí esté vacía (0 elementos)
- La lista con 1 elemento
- La lista con varios elementos

Coste: Mirar la condición de que first valga null o no es de tiempo constante en todas las ocasiones. O(1)

Implementación del algoritmo:

```
devuelve true si first == null, false si no;
```

4.1.13 método size()

```
public int size()
          -Precondición:
          -Postcondición: devuelve el número de elementos que
hay en la lista
```

Casos de prueba:

- La lista sí esté vacía (0 elementos)
- La lista con 1 elemento
- La lista con varios elementos

Coste: Devolver el valor asignado a una variable o atributo siempre es coste constante. O(1)

Implementación del algoritmo:

```
return count;
```

4.1.14 método iterator()

(Nuestro iterador se utiliza en el método toString para recorrer toda la lista e ir convirtiendo la información de los nodos a un String).

```
public Iterator<T> iterator()
          -Precondición:
          -Postcondición: nos devuelve el iterador de la lista.
```

Casos de prueba: (se prueba indirectamente en el método visualizar nodos al realizar los casos de prueba del resto de métodos)

- Lista vacía
- Lista con algún elemento

Coste: El coste es O(1) ya que crear un objeto y devolverlo es de coste constante.

Implementación del algoritmo:

```
return new ListIterator();
```

4.1.15 método visualizarNodos()

(Es un print de una llamada del método toString())

```
public void visualizarNodos()
          -Precondición:
          -Postcondición: se escribe por consola el toString de la lista.
```

Casos de prueba: (se prueba indirectamente al realizar los casos de prueba del resto de métodos)

- Lista vacía
- Lista con algún elemento

Coste: El coste de escribir por consola es de O(1) .

Implementación del algoritmo:

```
System.out.println(this.toString());
```

4.1.16 método toString()

```
public String toString()
          -Precondición:
          -Postcondición: devuelve un String con la lista
convertida en String.
```

Casos de prueba:

- La lista esté vacía (0 elementos)
- La lista con 1 elemento
- La lista con varios elementos

Coste: Siempre recorre toda la lista de elementos (n), pero para ir concatenando el String se hace una copia del String anterior más el elemento añadido con unos paréntesis por cada elemento, por tanto el coste total es O(n^2).

Implementación del algoritmo:

```
     String result = new String();
     Iterator<T> it = this.iterator();
     mientras (haya un siguiente) {
          T elem = siguiente;
          result = result + el elemento convertido en String
          //(usando el toString() de la clase Object)
     }
return "DoubleLinkedList" + ":" + "[" + result + "]";
```

4.2 UnorderedDoubleLinkedList<T>

4.2.1 método addToFront()

```
public void addToFront(T elem)
        -Precondición:
        -Postcondición: se ha añadido el elemento al principio de la lista.
```

Casos de prueba:

- Lista vacía
- Lista con un elemento
- Lista con varios elementos

Coste: El coste es O(1) ya que no tiene que recorrer la lista.

Implementación del algoritmo:

```
Node<T> nuevo = new Node<T>(elem);
si (el first está vacío) {
     añadir nuevo y cambiar first y last;
} si no {
     añadir nuevo al principio y cambiar first;
}
this.count++;
```

4.2.2 método addToRear()

```
public void addToRear(T elem)
          -Precondición:
          -Postcondición: se ha añadido el elemento al final
de la lista.
```

Casos de prueba:

- Lista vacía
- Lista con un elemento
- Lista con varios elementos

Coste: El coste es O(1) ya que no tiene que recorrer la lista.

Implementación del algoritmo:

```
Node<T> nuevo = new Node<T>(elem);
si (el first está vacío) {
     añadir nuevo y cambiar first y last;
} si no {
     añadir nuevo al final y cambiar last;
}
this.count++;
```

4.2.3 método addAfter()

```
public void addToAfter(T elem, T target)
        -Precondición: target se encuentra en la lista
        -Postcondición: se ha añadido el elemento elem después de target.
```

Casos de prueba:

- Lista con un elemento (se añade al final)
- Lista con varios elementos (se añade por el medio)

Coste: En el peor de los casos el coste es O(n) ya que tendría que recorrer toda la lista hasta llegar al último elemento si ese fuese el target. El coste medio sería n/2.

Implementación del algoritmo:

```
Node<T> nuevo = new Node<T>(elem);
Node<T> act = this.first;
boolean enc = false;
mientras (no encontrado) {
     si (actual es el target y no hay siguiente) {
          añadir elem al final (this.addToRear(elem));
          enc = true;
     }
     si no si (actual es el target y si hay siguiente) {
          añadir elem entre actual y siguiente (actualizar punteros);
          enc = true;
          this.count = this.count+1;
     }
     si no{
          avanzar posición;
     }
}
```

4.3 ListIterator

4.3.1 método hasNext()

```
public boolean hasNext()
          -Precondición:
          -Postcondición: devuelve un booleano indicando si
hay siguiente elemento.
```

Casos de prueba: (probados indirectamente con el método visualizarNodos())

- Si hay siguiente elemento.
- No hay siguiente elemento.

Coste: El coste de comprobar si el siguiente es null o no es constante: O(1).

Implementación del algoritmo:

```
return actual!=null;
```

4.3.2 método next()

```
public T next()
          -Precondición:
          -Postcondición: devuelve el siguiente elemento de la
lista.
```

Casos de prueba: (probados indirectamente con el método visualizarNodos())

- Si hay siguiente elemento.
- No hay siguiente elemento. (devuelve null)

Coste: El coste de devolver el valor del nodo actual es constante: O(1).

Implementación del algoritmo:

```
si(actual está vacío) {
     return null;
} si no {
     T aux = actual.data;
     actual = actual.next;
     return aux;
}
```

5. Código

```
import java.util.Iterator;
public class DoubleLinkedList<T> implements
ListADT<T> {

// Atributos
protected Node<T> first;  // apuntador al primero
protected Node<T> last;  // apuntador al último
protected String descr;  // descripción
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;
}

public T removeFirst() {
      T d = null;

      if (this.isEmpty()) {}
      else if (this.first.next == null) {
            d = this.first.data;
            this.first = null;
            this.last = null;
            this.count--;
      }
      else {
            d = this.first.data;
            this.first = this.first.next;
            this.first.prev = null;
            this.count--;
      }
      return d;
}
```

```
public T removeLast() {
      T d = null;

      if (this.isEmpty()) {}
      else if (this.last.prev == null) {
            d = this.last.data;
            this.last = null;
            this.first = null;
            this.count--;
      }
      else {
            d = this.last.data;
            this.last = this.last.prev;
            this.last.next = null;
            this.count--;
      }
      return d;
}
```

```
public T remove(T elem) {
      Node<T> actual;
      Node<T> anterior;
      boolean enc = false;
      T d = null;

      if (this.isEmpty()) {}
      else if (this.first.data.equals(elem)) {
            d = this.removeFirst();
            enc = true;
      }
      else {
            actual = this.first;
            anterior = null;

            while (actual.next != null && !enc) {
                  if (actual.data.equals(elem)) {
                        d = actual.data;
                        anterior.next = actual.next;
                        actual.next.prev = anterior;
                        actual = actual.next;
                        enc = true;
                        this.count--;
                  }
                  else {
                        anterior = actual;
                        actual = actual.next;
                  }
            }
            if (actual.data.equals(elem)) {
                  this.removeLast();
                  enc = true;
            }
      }
      return d;
}
```

```
public void removeAll(T elem) {
      Node<T> act;
      Node<T> ant;

      if (this.isEmpty()) {}
      else {
            act = this.first;
            ant = this.first;

            while(act.next != null) {
                  if (act.data.equals(elem)) {
                        if (this.first.equals(act)) {
                              this.removeFirst();
                              act = this.first;
                        } else {
                              ant.next = act.next;
                              act.next.prev = ant;
                              act = act.next;
                              this.count = this.count - 1;
                        }
                  }
                  else {
                        ant = act;
                        act = act.next;
                  }
            }
            if (act.data.equals(elem)) {
                  this.removeLast();
            }
      }
}

public T first() {
      //Da acceso al primer elemento de la lista
      if (isEmpty()) {
            return null;
      }
      else {
            return first.data;
      }
}
protected DoubleLinkedList<T> createList() {
      return new DoubleLinkedList<T>();
}
```

```
public DoubleLinkedList<T> clone() {
	DoubleLinkedList<T> resultado = createList();
	Node<T> act = this.first;
	Node<T> ult = null;

	if (this.isEmpty()) {}
	else {
		while (act != null) {
			Node<T> nuevo = new Node<T>(act.data);
			if (resultado.isEmpty()) {
				resultado.first = nuevo;
				resultado.last = nuevo;
				ult = resultado.first;
			}
			else {
				ult.next = nuevo;
				nuevo.prev = ult;
				ult = ult.next;
			}
			act = act.next;
		}
	}
	return resultado;
}

public T last() {
	if(this.isEmpty()) {
		return null;
	}
	return last.data;
}

public boolean contains(T elem) {
	boolean enc = false;

	if (this.find(elem) != null) {
		enc = true;
	}
	return enc;
}
```

```
public T find(T elem) {
      T d = null;
      Node<T> actual;
      boolean enc = false;

      if(this.isEmpty()) {
            return null;
      }
      else {
            actual = this.first;
            while (actual != null & !enc) {
                  if (actual.data.equals(elem)) {
                        d = actual.data;
                        enc = true;
                  }
                  else {
                        actual = actual.next;
                  }
            }
      }
      return d;
}

public boolean isEmpty() {
      return first == null;
}

public int size() {
      return count;
}

public Iterator<T> iterator() {
      return new ListIterator();
}
```

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

		private Node<T> actual = first;

		@Override
		public boolean hasNext() {
			return actual != null;
		}

		@Override
		public T next() {
			if (actual == null) {
				return null;
			}
			else
			{
				T aux = actual.data;
				actual = actual.next;
				return aux;
			}
		}
	}

	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 "DoubleLinkedList" + ":" + "[" + result + "]";
	}
}
```

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

      public void addToFront(T elem)
      {
            Node<T> nuevo = new Node<T>(elem);
            if (this.first == null) {
                  this.first = nuevo;
                  this.last = nuevo;
            }
            else {
                  nuevo.next = this.first;
                  this.first.prev = nuevo;
                  this.first = nuevo;
            }
            this.count++;
      }

      public void addToRear(T elem) {
                  Node<T> nuevo = new Node<T>(elem);

                  if (this.isEmpty()) {
                        this.first = nuevo;
                        this.last = nuevo;
                  }
                  else {
                        this.last.next = nuevo;
                        nuevo.prev = this.last;
                        this.last = this.last.next;
                  }
                  this.count = this.count + 1;
            }
```

```
	public void addAfter(T elem, T target) {
		//Pre: target ya se encuentra en la lista
		Node<T> nuevo = new Node<T>(elem);
		Node<T> act = this.first;
		boolean enc = false;

		while (!enc) {
			if (act.data == target && act.next == null) {
				this.addToRear(elem);
				enc = true;
			}
			else if (act.data == target && act.next != null) {
				nuevo.prev = act;
				nuevo.next = act.next;
				act.next.prev = nuevo;
				act.next = nuevo;
				enc = true;
				this.count++;
			}
			else {
				act = act.next;
			}
		}
	}

	@Override
	protected DoubleLinkedList<T> createList() {
		return new UnorderedDoubleLinkedList<T>();
	}
}
```

```
public class PruebaDoubleLinkedList {

     public static void main(String[] args)  {
          //Métodos de la clase UnorderedDoubleLinkedList
          UnorderedDoubleLinkedList<Integer> l = new
UnorderedDoubleLinkedList<Integer>();

          System.out.println("Primero se hacen las pruebas de los
métodos de la clase DoubleLinkedList.");
          System.out.println();

          //Prueba de set y de get
          System.out.println("Prueba de los métodos get() y set():
");
          l.setDescr("Mi lista favorita");
          System.out.println("Tiene que poner: 'Mi lista favorita'
y pone: " + l.getDescr());
          System.out.println();

          System.out.println("----------------");
          System.out.println();
```

```
            //Prueba de removeFirst
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método removeFirst(): ");
            System.out.println();
            System.out.println("Caso 1: único elemento. Añadimos
primero un elemento");
            l.addToFront(5);
            l.visualizarNodos();
            System.out.println("Lo eliminamos");
            l.removeFirst();
            l.visualizarNodos();
            System.out.println();

            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Caso 2: varios elementos. Añadimos
elementos");
            l.addToRear(6);
            l.addToRear(7);
            l.addToRear(8);
            l.addToRear(9);
            l.visualizarNodos();
            System.out.println("Eliminamos el primero. Tiene que ser
7 8 9 y es: ");
            l.removeFirst();
            l.visualizarNodos();
            System.out.println();

            System.out.println("-----------------");
            System.out.println();
```

```
            //Prueba de removeLast
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método removeLast(): ");
            System.out.println();
            System.out.println("Caso 1: único elemento. Añadimos
primero un elemento");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Lo eliminamos");
            l.removeLast();
            l.visualizarNodos();
            System.out.println();

            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Caso 2: varios elementos. Añadimos
elementos");
            l.addToRear(6);
            l.addToRear(7);
            l.addToRear(8);
            l.addToRear(9);
            l.visualizarNodos();
            System.out.println("Eliminamos el último. Tiene que ser
6 7 8 y es: ");
            l.removeLast();
            l.visualizarNodos();
            System.out.println();

            System.out.println("-----------------");
            System.out.println();
```

```
            //Prueba de remove
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método remove(): ");
            System.out.println();
            System.out.println("Caso 1: único elemento. Ese elemento
sí que hay que eliminarlo. Añadimos primero un elemento");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Se elimina el 5");
            l.remove(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 2: único elemento. Ese elemento
no que hay que eliminarlo. Añadimos primero un elemento");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Se intenta eliminar el 2. No se
elimina ninguno");
            l.remove(2);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 3: varios elementos. El
elemento a eliminar está en medio. Añadimos elementos");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Se elimina el 13");
            l.remove(13);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 4: varios elementos. El
elemento a eliminar está al final. Añadimos elementos");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Se elimina el 14");
            l.remove(14);
            l.visualizarNodos();
```

```
            System.out.println();

            System.out.println("Caso 5: varios elementos. El
elemento a eliminar está al principio. Añadimos elementos");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Se elimina el 5");
            l.remove(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 6: varios elementos. Hay
elementos repetidos, que es el que queremos borrar. Añadimos
elementos");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(5);
            l.addToRear(6);
            l.addToRear(9);
            l.visualizarNodos();
            System.out.println("Se elimina la primera aparición de
5");
            l.remove(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("-----------------");
            System.out.println();
```

```
            //Prueba de removeAll
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método removeAll(): ");
            System.out.println();
            System.out.println("Caso 1: único elemento. Ese elemento
sí que hay que eliminarlo. Añadimos primero un elemento");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Se eliminan todas las apariciones de
5");
            l.removeAll(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 2: único elemento. Ese elemento
no que hay que eliminarlo. Añadimos primero un elemento");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Se intenta eliminar el 2. No se
elimina ninguno.");
            l.removeAll(2);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 3: varios elementos. El
elemento a eliminar es el 12. Añadimos elementos");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Se eliminan todas las apariciones de
12");
            l.removeAll(12);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 4: varios elementos. El
elemento a eliminar es el 14. Añadimos elementos");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
```

```
            System.out.println("Se eliminan todas las apariciones de
14");
            l.removeAll(14);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 5: varios elementos. El
elemento a eliminar está al principio. Añadimos elementos");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Se eliminan todas las apariciones de
5");
            l.removeAll(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 6: varios elementos. Hay
elementos repetidos, que es el que queremos borrar. Añadimos
elementos");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(5);
            l.addToRear(6);
            l.addToRear(9);
            l.visualizarNodos();
            System.out.println("Se eliminan todas la apariciones de
5");
            l.removeAll(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 7: varios elementos. Todos los
elementos son repetidos y son los que queremos borrar");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(5);
            l.addToRear(5);
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Se eliminan todas la apariciones de
5");
            l.removeAll(5);
            l.visualizarNodos();
            System.out.println();
```

```
            System.out.println("Caso 8: varios elementos. Todos los
elementos son repetidos y pero ninguno de ellos son los que se
quieren borrar");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(5);
            l.addToRear(5);
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Se intenta eliminar todas las
apariciones de 6");
            l.removeAll(6);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 9: varios elementos. El primer
elemento es diferente del resto y el el resto son todos repetidos,
los cuales se quieren borrar.");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(6);
            l.addToRear(5);
            l.addToRear(5);
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Se eliminan todas las apariciones de
5");
            l.removeAll(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("----------------");
            System.out.println();
```

```
            //Prueba de first
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método first(): ");
            System.out.println();

            System.out.println("Caso 1: único elemento.");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("El primero tiene que ser 5 y es: " +
l.first());
            System.out.println();

            System.out.println("Caso 2: lista vacía");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.visualizarNodos();
            System.out.println("No hay primero. Será null y es: " +
l.first());
            System.out.println();

            System.out.println("Caso 3: varios elementos.");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("El primero tiene que ser el 5 y es:
" + l.first());
            System.out.println();

            System.out.println("-----------------");
            System.out.println();
```

```
            //Prueba de last
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método last(): ");
            System.out.println();

            System.out.println("Caso 1: único elemento.");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("El último tiene que ser 5 y es: " +
l.last());
            System.out.println();

            System.out.println("Caso 2: lista vacía");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.visualizarNodos();
            System.out.println("No hay último. Será null y es: " +
l.last());
            System.out.println();

            System.out.println("Caso 3: varios elementos.");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("El último tiene que ser el 14 y es:
" + l.last());
            System.out.println();

            System.out.println("-----------------");
            System.out.println();
```

```
            //Prueba de clone
            UnorderedDoubleLinkedList<Integer> l1 = new
UnorderedDoubleLinkedList<Integer>();
            DoubleLinkedList<Integer> l2 = new
DoubleLinkedList<Integer>();

            System.out.println("Prueba del método clone(): ");
            System.out.println();

            System.out.println("Caso 1: único elemento.");
            l1.addToRear(5);
            l1.visualizarNodos();
            System.out.println("Se creará una nueva lista con el
elemento 5");
            l2 = l1.clone();
            l2.visualizarNodos();
            System.out.println();

            l1 = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Caso 2: lista vacía.");
            l1.visualizarNodos();
            System.out.println("Se creará una nueva lista que será
vacía");
            l2 = l1.clone();
            l2.visualizarNodos();
            System.out.println();

            l1 = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Caso 3: lista con varios
elementos.");
            l1.addToRear(5);
            l1.addToRear(6);
            l1.addToRear(7);
            l1.addToRear(8);
            l1.visualizarNodos();
            System.out.println("Se creará una nueva lista con los
elementos 5, 6, 7 y 8");
            l2 = l1.clone();
            l2.visualizarNodos();
            System.out.println();

            System.out.println("----------------");
            System.out.println();
```

```
            //Prueba de contains
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método contains(): ");
            System.out.println();

            System.out.println("Caso 1: único elemento y esa lista
sí que contiene ese elemento buscado.");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Buscamos el 5. Tiene que responder
true y responde: " + l.contains(5));
            System.out.println();

            System.out.println("Caso 2: único elemento y esa lista
no contiene ese elemento buscado");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(7);
            l.visualizarNodos();
            System.out.println("Buscamos el 5. Tiene que responder
false y responde: " + l.contains(5));
            System.out.println();

            System.out.println("Caso 3: varios elementos y esa lista
sí que contiene ese elemento buscado. El elemento buscado está por
el medio");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Buscamos el 12. Tiene que responder
true y responde: " + l.contains(12));
            System.out.println();

            System.out.println("Caso 4: varios elementos y esa lista
sí que contiene ese elemento buscado. El elemento buscado es el
primero");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Buscamos el 5. Tiene que responder
true y responde: " + l.contains(5));
            System.out.println();
```

```
            System.out.println("Caso 5: varios elementos y esa lista
sí que contiene ese elemento buscado. El elemento buscado es el
último");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Buscamos el 14. Tiene que responder
true y responde: " + l.contains(14));
            System.out.println();

            System.out.println("-----------------");
            System.out.println();
```

```
            //Prueba de find
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método find(): ");
            System.out.println();

            System.out.println("Caso 1: único elemento y esa lista
sí que contiene ese elemento buscado.");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Buscamos el 5. Tiene que devolver 5
y devuelve: " + l.find(5));
            System.out.println();

            System.out.println("Caso 2: único elemento y esa lista
no contiene ese elemento buscado");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(7);
            l.visualizarNodos();
            System.out.println("Buscamos el 5. Tiene que devolver
null y devuelve: " + l.find(5));
            System.out.println();

            System.out.println("Caso 3: varios elementos y esa lista
sí que contiene ese elemento buscado. El elemento buscado está por
el medio");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Buscamos el 12. Tiene que devolver
12 y devuelve: " + l.find(12));
            System.out.println();

            System.out.println("Caso 4: varios elementos y esa lista
sí que contiene ese elemento buscado. El elemento buscado es el
primero");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Buscamos el 5. Tiene que devolver 5
y devuelve: " + l.find(5));
            System.out.println();
```

```
            System.out.println("Caso 5: varios elementos y esa lista
sí que contiene ese elemento buscado. El elemento buscado es el
último");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Buscamos el 14. Tiene que devolver
14 y devuelve: " + l.find(14));
            System.out.println();

            System.out.println("----------------");
            System.out.println();

            //Prueba de isEmpty
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método isEmpty(): ");
            System.out.println();

            System.out.println("Caso 1: lista vacía.");
            System.out.println("Tiene que responder true y responde:
" + l.isEmpty());
            System.out.println();

            System.out.println("Caso 2: lista con un elemento");
            l.addToRear(2);
            l.visualizarNodos();
            System.out.println("Tiene que responde false y responde:
" + l.isEmpty());
            System.out.println();

            System.out.println("Caso 3: lista con varios
elementos.");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Tiene que responde false y responde:
" + l.isEmpty());
            System.out.println();

            System.out.println("----------------");
            System.out.println();
```

```
            //Prueba de size
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método size(): ");
            System.out.println();

            System.out.println("Caso 1: lista vacía.");
            System.out.println("Tiene un total de: " + l.size() + "
elementos");
            System.out.println();

            System.out.println("Caso 2: lista con un elemento");
            l.addToRear(2);
            l.visualizarNodos();
            System.out.println("Tiene un total de: " + l.size() + "
elemento");
            System.out.println();

            System.out.println("Caso 3: lista con varios
elementos.");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            l.visualizarNodos();
            System.out.println("Tiene un total de: " + l.size() + "
elementos");
            System.out.println();

            System.out.println("----------------");
            System.out.println();
```

```
            //Prueba de toString
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método toString(): ");
            System.out.println();

            System.out.println("Caso 1: lista vacía.");
            System.out.println(l.toString());
            System.out.println();

            System.out.println("Caso 2: lista con un elemento. Se
añade el 2");
            l.addToRear(2);
            System.out.println("Se tiene que ver el 2");
            System.out.println(l.toString());
            System.out.println();

            System.out.println("Caso 3: lista con varios elementos.
Se añade el 5, 12, 13 y 14");
            l = new UnorderedDoubleLinkedList<Integer>();
            l.addToRear(5);
            l.addToRear(12);
            l.addToRear(13);
            l.addToRear(14);
            System.out.println("Se tiene que ver el 5, 12, 13 y el
14");
            System.out.println(l.toString());
            System.out.println();

            System.out.println("----------------");
            System.out.println();
```

```
            //Prueba de iterator
            System.out.println("Prueba del método iterator(): ");
            System.out.println();

            System.out.println("Puesto que en todas las demás
pruebas se utiliza el método visualizarNodos enseña por pantalla
los elementos de la lista, este método está implementado con un
iterator, con su next() y su hasNext().");
            System.out.println("Por tanto, está correcto ya que
siempre imprime los elememntos de una manera correcta");

            System.out.println("----------------");
            System.out.println();


            System.out.println("Ahora se harán las pruebas de
UnorderedDoubleLinkedList.");
            System.out.println();

            //Prueba de addToFront
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método addToFront(): ");
            System.out.println();
            System.out.println("Caso 1: lista vacía.");
            System.out.println("Se tiene que añadir el 5");
            l.addToFront(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 2: lista con un elemento.");
            System.out.println("Se tiene que añadir el 6 antes que
el 5");
            l.addToFront(6);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 3: lista con varios
elementos.");
            System.out.println("Se tiene que añadir el 7 antes que
el 6");
            l.addToFront(7);
            l.visualizarNodos();
            System.out.println();

            System.out.println("----------------");
            System.out.println();
```

```
            //Prueba de addToRear
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método addToRear(): ");
            System.out.println();
            System.out.println("Caso 1: lista vacía.");
            System.out.println("Se tiene que añadir el 5");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 2: lista con un elemento.");
            System.out.println("Se tiene que añadir el 6 después que
el 5");
            l.addToRear(6);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 3: lista con varios
elementos.");
            System.out.println("Se tiene que añadir el 7 después que
el 6");
            l.addToRear(7);
            l.visualizarNodos();
            System.out.println();

            System.out.println("----------------");
            System.out.println();
```

```
            //Prueba de addAfter
            l = new UnorderedDoubleLinkedList<Integer>();
            System.out.println("Prueba del método addAfter(): ");
            System.out.println();

            System.out.println("Caso 1: lista con un elemento (ya
que la precondición es que target ya se encuentra en la lista,
esto implica que haya mínimo un elemento).");
            System.out.println("Queremos añadir el 6 después del
5");
            l.addToRear(5);
            l.visualizarNodos();
            System.out.println("Se añade el 6. La lista tiene que
quedar 5 y 6");
            l.addAfter(6, 5);
            l.visualizarNodos();
            System.out.println();

            System.out.println("Caso 2: lista varios elementos.");
            System.out.println("Queremos añadir el 7 después del
5");
            l.visualizarNodos();
            System.out.println("Se añade el 7. La lista tiene que
quedar 5, 7, y 6");
            l.addAfter(7, 5);
            l.visualizarNodos();
            System.out.println();

      }
}
```

```
package Practica1;


import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

import Practica2.UnorderedDoubleLinkedList;

public class ListaPalabras {

	private UnorderedDoubleLinkedList<String> lista;

	public ListaPalabras() {
		this.lista = new UnorderedDoubleLinkedList<String>();
	}

	private Iterator<String> getIterador(){
		return this.lista.iterator();

	}

	public void leerPalabras(String pNomFichero) throws
FileNotFoundException {
		Scanner entrada = new Scanner(new
FileReader(pNomFichero));

		String linea;
		while (entrada.hasNext()) {

			linea = entrada.nextLine();
			String nuevaPalabra = linea;
			this.lista.addToRear(nuevaPalabra);

		}
	}

	public UnorderedDoubleLinkedList<String> getPalabras() {
		return this.lista;
	}
```

```
public ArrayList<String> web2words(String pWeb){
          ArrayList<String> listaPalabras = new
ArrayList<String>();
          Iterator<String> itr = this.getIterador();

          String palabraActual;
          while (itr.hasNext()) {
               palabraActual = itr.next();
               if (pWeb.contains(palabraActual)) {
                    listaPalabras.add(palabraActual);
               }
          }

          return listaPalabras;
     }

}
```

```
package Practica1;

import static org.junit.jupiter.api.Assertions.*;

import java.io.FileNotFoundException;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class ListaPalabrasTest {

	@BeforeEach
	void setUp() throws Exception {
	}

	@AfterEach
	void tearDown() throws Exception {
	}

	@Test
	void testListaPalabras() {
		ListaPalabras l1 = new ListaPalabras();
	}

	@Test
	void getPalabras() {
		//Metodo getter, solo lo usamos para las JUnit
	}
```

```
	@Test
	void testLeerPalabras() {

		//Leer palabras de un fichero que no existe
		//Leer palabras de un fichero vacio
		//Leer palabras de un fichero que existe pero solo tiene
1 palabra
		//Leer palabras de un fichero que existe y tiene varias
palabras

		//La excepción se trata en la clase TodasLasPalabras


		//Leer palabras de un fichero que no existe
		ListaPalabras l1 = new ListaPalabras();
		try {
			l1.leerPalabras("ficheroInexistente");
			fail("El programa ha fallado");

		}
		catch(FileNotFoundException fnfe) {
			System.out.println("El fichero no existe, con lo
cual, este caso funciona");
		}

		//Leer palabras de un fichero vacio
		ListaPalabras l2 = new ListaPalabras();
		try {
			l2.leerPalabras("ficheroVacio.txt");
			System.out.println("Se ha leido correctamente");

assertEquals(l2.getPalabras().toString(),"DoubleLinkedList:[]");

		}
		catch(FileNotFoundException fnfe) {
			fail("El programa ha fallado");
		}
```

```
            //Leer palabras de un fichero que existe pero solo tiene
      1 palabra
            ListaPalabras l3 = new ListaPalabras();
            try {
                  l3.leerPalabras("fichero1Palabra.txt");
                  System.out.println("Se ha leido correctamente");

assertEquals(l3.getPalabras().toString(),"DoubleLinkedList:[(hola)
]");

            }
            catch(FileNotFoundException fnfe) {
                  fail("El programa ha fallado");
            }

            //Leer palabras de un fichero que existe y tiene varias
palabras
            ListaPalabras l4 = new ListaPalabras();
            try {
                  l4.leerPalabras("ficheroVariasPalabras.txt");
                  System.out.println("Se ha leido correctamente");

assertEquals(l4.getPalabras().toString(),"DoubleLinkedList:[(hola)
(adios)(jamon)(tortilla)]");

            }
            catch(FileNotFoundException fnfe) {
                  fail("El programa ha fallado");
            }


      }
```

```
	@Test
	void testWeb2words() {

		//En teoria solo se pide el método word2Webs, este
método sale en el enunciado pero no se pide

		//Web que no contiene ninguna palabra del fichero
		//Web que contiene 1 palabra del fichero
		//Web que contien varias palabras del fichero

		//Web que no contiene ninguna palabra del fichero
		try {
			ListaPalabras l1 = new ListaPalabras();
			l1.leerPalabras("words.txt");
			assertEquals(l1.web2words("").toString(),"[, ]");


		}
		catch(FileNotFoundException fnfe) {
			fail("El programa ha fallado");
		}

		//Web que contiene 1 palabra del fichero
		try {
			ListaPalabras l2 = new ListaPalabras();
			l2.leerPalabras("words.txt");
			assertEquals(l2.web2words("z").toString(),"[, ,
z]");


		}
		catch(FileNotFoundException fnfe) {
			fail("El programa ha fallado");
		}
```

```
            //Web que contien varias palabras del fichero
            try {
                  ListaPalabras l3 = new ListaPalabras();
                  l3.leerPalabras("words.txt");

assertEquals(l3.web2words("ultimate.es").toString(),"[a, at, ate,
e, es, , i, im, l, lt, m, ma, mat, mate, s, t, te, ti, tim, u,
ult, ultima, ultimate, ]");


            }
            catch(FileNotFoundException fnfe) {
                  fail("El programa ha fallado");
            }

      }

}
```

6. Conclusiones

En esta práctica hemos podido aprender más en detalle cómo funcionan las listas enlazadas y también hemos podido identificar algunas de las ventajas que tiene usar dichas listas.

La primera de ellas es que a diferencia de los arrays, no hace falta definir un tamaño fijo para nuestra lista. De esta forma, se puede ajustar su tamaño en tiempo de ejecución según sea necesario, optimizando el uso de la memoria.

La segunda de ellas es que no es necesario que en la memoria estén almacenados de manera contigua los elementos (nodos). De esta manera se puede aprovechar de manera más eficiente espacios de memoria dispersos.

Por último y lo más importante, hemos aplicado nuestra implementación de una lista enlazada a nuestra práctica anterior, ya que las listas enlazadas se pueden adaptar fácilmente para representar otras estructuras de datos más complejas, como pilas, colas y grafos (aunque en la práctica anterior usamos ArrayList).