Aplicación para gestionar actores y películas

Fecha: 30-IX-2021

Participantes: Xabier Gabiña
Javier Criado

Contenido

1 Introducción

Para el laboratorio 2 se nos ha planteado definir un tipo de dato, una lista doblemente enlazada con un puntero apuntando al último elemento y el último elemento apuntando al primer elemento de la lista. Junto con la implementación de las estructuras de datos se ha pedido un programa de pruebas

2 Diseño de las clases

El diagrama de clases se compone de una interfaz principal, ListADT, que representa los métodos básicos de una lista cualquiera. Esta interfaz es hija de las interfaces OrderedListADT, UnorderedListADT y IndexedListADT. Estas interfaces representan sus métodos característicos por ser una clase concreta, además de extender los métodos comunes a todas las listas representados en ListADT.

Estas interfaces son implementadas por varias clases, cada una representado un tipo de lista. Estamos implementando una lista doblemente enlazada, con un puntero apuntando al último elemento.

DoubleLinkedList es la clase principal que implementa ListADT que contiene los métodos comunes a todas las listas doblemente enlazadas. Este método hereda las clases OrderedDoubleLinkedList y UnorderedDoubleLinkedList que comparten los métodos implementados en esa clase.

La clase DoubleLinkedList es una TAD abstracta. Tiene tres atributos. Last: de tipo Node que apunta al último elemento de la lista. Descr: de tipo String que describe la función de la lista. Count: de tipo int que indica cuantos elementos contiene la lista. Contiene una clase privada ListIterator que implementa Iterator. Es una TAD con dos atributos. Actual: de tipo Node y previo: de tipo Node, ambos apuntan al mismo elemento y nos servirán para recorrer la lista.

La clase UnorderedDoubleLinkedList es una TAD. No tiene atributos, comparte los de la clase DoubleLinkedList.

La clase OrderedDoubleLinkedList es una TAD. No tiene atributos, comparte los de la clase DoubleLinkedList.

La clase Node, es una TAD con tres atributos. next: de tipo Node, representa el nodo al que apunta este nodo. Prev: de tipo Node, apunta al nodo que prevalece al nodo actual. Data: de tipo T (genérico), es el dato que contiene el nodo. El nodo es el elemento que compone una lista enlazada.

3 Diseño e implementación de datos principales

En este laboratorio se nos ha pedido implementar una lista doblemente ligada o DoubleLinkedList y a partir de la misma generar una lista desordenada de elementos y otra ordenada usando como base la DoubleLinkedList. Para ello hemos definido en la DoubleLinkedList todos los métodos comunes para todas las listas que implementan estas estructuras menos la de añadir que será independiente de si queremos que el añadir sea ordenado o no.

3.1 Clase DoubleLinkedList

Public T remove()

//Pre:

Casos de prueba:

-Lista vacía, 1 elemento, varios elementos

Implementación:

Si el primer elemento de la lista es el buscado eliminarlo y devolver su data

Si el último elemento de la lista es el buscado eliminarlo y devolver su data

Si no buscar el elemento

Si lo encuentra se elimina y devuelve su data

Si no lo encuentra devuelve null

Coste: O(n) recorre todos los elemento en el peor de los casos.

Public T removeLast()

//Pre:

Casos de prueba:

-Lista vacía, lista no vacía

Implementación:

Actualizar el último elemento y reducir el contador

Devolver el valor del elemento eliminado

Coste: O(1) coste constante.

Public T removeFirst()

//Pre:

Casos de prueba:

-Lista vacía, lista de 1 elemento, lista de varios elementos

Implementación:

Actualizar el primer elemento y reducir el contador

Devolver el valor del elemento eliminado

Coste: O(1) coste constante.

Public boolean contains()

//Pre:

Casos de prueba:

-Lista vacía, lista no vacía, está el elemento en la lista y no está el elemento en la lista

Implementación:

Si la lista es vacía devuelve null

Si last es el elemento buscado devuelve True

Si no buscar el elemento

Si lo encuentra devuelve True

Si no lo encuentra devuelve False

Coste: O(n) recorre todos los elemento de la lista en el peor de los casos.

Public T find(T elem)()

//Pre:

Casos de prueba:

-Lista vacía, lista no vacía, que esté el elemento, que no esté el elemento

Implementación:

Si la lista es vacía devuelve null

Si last es el elemento buscado devuelve la data de last

Si no buscar el elemento

Si lo encuentra devuelve su data

Si no lo encuentra devuelve null

Coste: O(n) recorre todos los elemento de la lista en el peor de los casos.

Public boolean isEmpty()

//Pre:

Casos de prueba:

-Lista vacía, lista no vacía

Implementación:

True si last es igual a null

Coste:O(1) coste constante.

3.2 Clase OrderedDoubleLinkedList

public void add(T elem)

//Pre:

Casos de prueba:

-Lista vacía, lista no vacía, al principio, final y en la mitad

Implementación:

If lista vacia{

Se añade y se incrementa el contador

}

Else{

Recorrer la lista hasta encontrar el elemento y añadirlo, incrementar contador

}

Coste: El coste dependerá de la ubicación de la posición donde hay q añadirlo, siendo O(n) en el peor de los casos.

```
public void merge(DoubleLinkedList<T> lista)
```
//Pre:

Casos de prueba:

-Listas vacías, 1 lista vacía y otra no vacía, listas de distintos tamaños y listas del mismo tamaño.

Implementación:

Act=lista.last.next;

Ant=lista.alast.next;

While(ant!=act.last){

Ant=act;

Act=act.next;

Add(ant.data);

}

Coste: O(n) Se recorre toda la lista.

3.3 Clase UnorderedDoubleLinkedList

public void addToFront(T elem)

//Pre:

Casos de prueba:

-Lista vacía, lista no vacía

Implementación:

If lista vacia{

Last apunta al nodo de elem

}

Else{

Añadir al principio

}

Aumentar contador

Coste: O(1) constante

public void addToRear(T elem)

//Pre:

Casos de prueba:

--Lista vacía, lista no vacía

Implementación:

Añadir el elemento al final

Cambiar last

Coste: O(1) es constante.

public void addAfter(T elem, T target)

//Pre:

Casos de prueba:

-Lista vacía, lista con 1 elemento, lista con varios elementos

Implementación:

if lista vacía{

Añadir el elemento

}

Else{

If no se encuentra el target{

Añadir el elemento al final de la lista

}

Else {

Añadir el elemento

}

```
	}
```

Coste: Lineal porque hay que recorrer la lista hasta encontrar el target. n= número de elementos de la lista. Coste=O(n)

4 Código

4.1 DoubleLinkedList

```
public class DoubleLinkedList<T extends Comparable<T>> implements ListADT<T> {

	// Atributos
	protected Node<T> last;  // apuntador al ï¿½ltimo
	protected String descr;  // descripciï¿½n
	protected int count;

	// Constructor
	public DoubleLinkedList()
	{
		last = null;
		descr = "";
		count = 0;
	}

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

	public String getDescr()
	{
		return descr;
	}

	public T removeFirst()
	{
	// Elimina el primer elemento de la lista
	  // Precondiciï¿½n:
		// COMPLETAR EL CODIGO Y CALCULAR EL COSTE
		Node<T> actual=last.next;
		last.next.next.prev=last;
		last.next=actual.next;
		count--;
		return actual.data;
	}

	public T removeLast()
	{
		// Elimina el ï¿½ltimo elemento de la lista
		// Precondiciï¿½n:
		// COMPLETAR EL CODIGO Y CALCULAR EL COSTE
		last.prev.next=last.next;
		last.next.prev=last.prev;
		last=last.prev;
		count--;
```

```
        return last.data;
    }

    public T remove(T elem)
    {
        //Elimina un elemento concreto de la lista
        Node<T> actual=last.next;
        boolean enc=false;
        if(actual.data.equals(elem))
        {
            removeFirst();
            return actual.data;
        }
        else if(last.data.equals(elem))
        {
            removeLast();
            return last.data;
        }
        else
        {
            while (!actual.equals(last) && !enc)
            {
                if (actual.data.equals(elem))
                {
                    enc = true;
                }
                else
                {
                    actual = actual.next;
                }
            }
            if (!enc)
            {
                return null;
            }
            else
            {
                actual.prev.next = actual.next;
                actual.next.prev = actual.prev;
                count--;
                return actual.data;
            }
        }
    }

    public T first()
    {
        //Da acceso al primer elemento de la lista
        if (isEmpty())
            return null;
        else return last.next.data;
    }

    public T last()
    {
        //Da acceso al ï¿½ltimo elemento de la lista
        if (isEmpty())
            return null;
        else return last.data;
```

```
	}

	public boolean contains(T elem) {
		//Determina si la lista contiene un elemento concreto
		if (isEmpty())
		{
			return false;
		}
		else if(last.data.equals(elem))
		{
			return true;
		}
		else
		{
			Node<T> actual=last.next;
			boolean enc=false;
			while(!actual.equals(last)&&!enc)
			{
				if(actual.data.equals(elem))
				{
					enc=true;
				}
				else
				{
					actual=actual.next;
				}
			}
			return enc;
		}
	}

	public T find(T elem)
	{
		//Determina si la lista contiene un elemento concreto, y develve
su referencia, null en caso de que no esti¿½
		// COMPLETAR EL CODIGO Y CALCULAR EL COSTE
		if (isEmpty())
		{
			return null;
		}
		else if(last.data.equals(elem))
		{
			return last.data;
		}
		else
		{
			Node<T> actual=last.next;
			boolean enc=false;
			while(!actual.equals(last)&&!enc)
			{
				if(actual.data.equals(elem))
				{
					enc=true;
				}
				else
				{
					actual=actual.next;
				}
			}
```

```
                if(!enc)
                {
                        return null;
                }
                else
                {
                        return actual.data;
                }
        }
    }

    public boolean isEmpty()
    //Determina si la lista estï¿½ vacï¿½a
    {
        return last == null;
    }

    public int size()
    //Determina el nï¿½mero de elementos de la lista
    {
        return count;
    }

    /** Return an iterator to the stack that iterates through the items .
*/
    public Iterator<T> iterator()
    {
        return new ListIterator();
    }

    // an iterator, doesn't implement remove() since it's optional
    private class ListIterator implements Iterator<T>
    {
        private Node<T>    actual=last.next;
        private Node<T> previo=last.next;

        @Override
        public boolean hasNext()
        {
                return previo != last;
        }

        @Override
        public T next() {
                if(!hasNext())
                {
                        throw new NoSuchElementException();
                }
                else
                {
                        previo=actual;
                        T data = previo.data;
                        actual=actual.next;
                        return data;
                }
        }
    } // private class

    public void visualizarNodos()
```

```
	{
		System.out.print(this.toString());
	}

	@Override
	public String toString()
	{
		StringBuilder result = new StringBuilder("");
		Iterator<T> it = iterator();
		while (it.hasNext())
		{
			T elem = it.next();
			result.append("[").append(elem.toString()).append("]");
		}
		return result + "\n";
	}

}
```

4.2 IndexedListADT

```
public interface IndexedListADT<T> extends ListADT<T>
{
   /**
    * Inserts the specified element at the specified index.
    *
    * @param index   the index into the array to which the element is to be
    *                inserted.
    * @param element the element to be inserted into the array
    */
   public void add (int index, T element);

   /**
    * Sets the element at the specified index.
    *
    * @param index   the index into the array to which the element is to be
set
    * @param element the element to be set into the list
    */
   public void set (int index, T element);

   /**
    * Adds the specified element to the rear of this list.
    *
    * @param element  the element to be added to the rear of the list
    */
   public void add (T element);

   /**
    * Returns a reference to the element at the specified index.
    *
    * @param index  the index to which the reference is to be retrieved from
    * @return       the element at the specified index
    */
   public T get (int index);

   /**
    * Returns the index of the specified element.
    *
```

```
     * @param element  the element for the index is to be retrieved
     * @return         the integer index for this element
     */
    public int indexOf (T element);

    /**  Removes and returns the element at the specified index. */
    public T remove (int index);
}
```

4.3 ListADT

```
      public interface ListADT<T> {

public void setDescr(String nom);
// Actualiza el nombre de la lista

public String getDescr();
// Devuelve el nombre de la lista

public T removeFirst();
//Elimina el primer elemento de la lista

public T removeLast();
//Elimina el ï¿½ltimo elemento de la lista

public T remove(T elem);
//Elimina un elemento concreto de la lista

public T first();
//Da acceso al primer elemento de la lista

public T last();
//Da acceso al ï¿½ltimo elemento de la lista

public boolean contains(T elem);
//Determina si la lista contiene un elemento concreto

public T find(T elem);
//Determina si la lista contiene un elemento concreto, y develve su referencia,
null en caso de que no estï¿½

public boolean isEmpty();
//Determina si la lista estï¿½ vacï¿½a

public int size();
//Determina el nï¿½mero de elementos de la lista

public Iterator<T> iterator();

}
```

4.4 Node

```
public class Node<T extends Comparable<T>> {
            public T data;                    // dato del nodo
            public Node<T> next;      // puntero al siguiente nodo de la lista
            public Node<T> prev;      // puntero al anterior nodo de la lista
            // -----------------------------------------------------------
```

```
        public Node(T dd)         // constructor
        {
                data = dd;
                next = null;
                prev = null;
        }
}
```

4.5 OrderedDoubleLinkedList

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

        public void add(T elem)
        {
                if(last==null)
                {
                        last=new Node<>(elem);
                        last.prev=last;
                        last.next=last;
                        count++;
                }
                else
                {
                        Node<T> actual=last.next;
                        Node<T> previo=last;
                        if(previo.data.compareTo(elem)<0)
                        {
                                previo.next=new Node<>(elem);
                                previo.next.prev=previo;
                                previo.next.next=actual;
                                actual.prev=previo.next;
                                last=previo.next;
                                count++;
                        }
                        else
                        {
                                boolean enc=false;
                                while (!enc)
                                {
                                        if(actual.data.compareTo(elem)>0)
                                        {
                                                enc=true;
                                        }
                                        else
                                        {
                                                previo=actual;
                                                actual=actual.next;
                                        }
                                }
                                previo.next=new Node<>(elem);
                                previo.next.next=actual;
                                previo.next.prev=previo;
                                actual.prev=previo.next;
                                count++;
                        }
                }
```

```
	}
	public void merge(DoubleLinkedList<T> lista)
	{
		Node<T> actual=lista.last.next;
		Node<T> previo=lista.last.next;

		while(previo!=lista.last)
		{
			previo=actual;
			actual=actual.next;

			add(previo.data);
		}

	}


}
```

4.6 OrderedListADT

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

	public void add(T elem);
	// Aï¿½ade un elemento a la lista (en el lugar de orden que le
corresponde)

	public void merge(DoubleLinkedList<T> zerrenda);

}
```

4.7 Persona

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

	// atributos
	private String name;
    private String dni;

	public Persona(String pName, String pDni) { // Constructora
		name = pName;
		dni = pDni;
	}

	public String getName() { return name; }

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

	public String getDni() { return dni; }

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

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
```

```
                return false;
            if (getClass() != obj.getClass())
                return false;
            Persona other = (Persona) obj;
            if (dni == null) {
                if (other.dni != null)
                    return false;
            } else if (!dni.equals(other.dni))
                return false;
            return true;
        }

        @Override
        public int compareTo(Persona arg0) {
            return name.compareToIgnoreCase(arg0.name);
        }

        public String toString() {
            return name + " " + dni;
        }

}
```

4.8 PruebaDoubleLinkedList

```
public class PruebaDoubleLinkedList {
        public static void main(String[] args)
        {
            UnorderedDoubleLinkedList<Integer> l = new
UnorderedDoubleLinkedList<>();

            System.out.println("\nPrueba addToRear");
            l.addToRear(4);
            l.addToRear(5);
            l.addToRear(6);

            if(l.size()==3)
            {
                System.out.print("La lista creada es:
");l.visualizarNodos();
                System.out.println("La lista deveria ser: [4][5][6]");
            }
            else
            {
                System.out.println("Error en addToRear | Lista resultado:
");l.visualizarNodos();
            }

            System.out.println("\nPrueba addToFront");
            l.addToFront(3);
            l.addToFront(2);
            l.addToFront(1);

            if(l.size()==6)
            {
                System.out.print("La lista creada es:
");l.visualizarNodos();
```

```
                    System.out.println("La      lista      deveria      ser:
[1][2][3][4][5][6]");
              }
              else
              {
                    System.out.print ("Error en addToFront | Lista resultado:
");l.visualizarNodos();
              }

              System.out.println("\nPrueba addAfter");
              l.addAfter(7,6);
              l.addAfter(5,5);
              l.addAfter(9,10);

              if(l.size()==9)
              {
                    System.out.print("La        lista        creada        es:
");l.visualizarNodos();
                    System.out.println("La      lista      deveria      ser:
[1][2][3][4][5][5][6][7][9]");
              }
              else
              {
                    System.out.print("Error  en  addAfter  |  Nodo  resultado:
");l.visualizarNodos();
              }

              System.out.println("\nPrueba remove");
              l.remove(9);
              l.remove(5);

              if(l.size()==7)
              {
                    System.out.print("La        lista        creada        es:
");l.visualizarNodos();
                    System.out.println("La      lista      deveria      ser:
[1][2][3][4][5][6][7]");
              }
              else
              {
                    System.out.print("Error  en  remove  |  Nodo  resultado:
");l.visualizarNodos();
              }

              System.out.println("\nPrueba removeLast");
              l.removeLast();
              l.removeLast();

              if(l.size()==5)
              {
                    System.out.print("La        lista        creada        es:
");l.visualizarNodos();
                    System.out.println("La      lista      deveria      ser:
[1][2][3][4][5]");
              }
              else
              {
                    System.out.print("Error  en  removeLast  |  Nodo  resultado:
");l.visualizarNodos();
```

```
            }

            System.out.println("\nPrueba removeFirst");
            l.removeFirst();
            l.removeFirst();

            if(l.size()==3)
            {
                  System.out.print("La        lista        creada        es:
");l.visualizarNodos();
                  System.out.println("La lista deveria ser: [3][4][5]");
            }
            else
            {
                  System.out.print("Error en removeFirst | Nodo resultado:
");l.visualizarNodos();
            }

            System.out.println("\n Set & Get description");
            l.setDescr("Lista de numeros");
            System.out.println("La descripcion es: "+l.getDescr());
            System.out.println("La   descripcion   deveria   ser:   Lista   de
numeros");

            System.out.println("\nPrueba contains");

            if(l.size()==3)
            {
                  l.visualizarNodos();
                  if(l.contains(1))
                  {
                        System.out.println("Error");
                  }
                  else
                  {
                        System.out.println("No encontrado [1]");
                  }
                  if(l.contains(3))
                  {
                        System.out.println("Encontrado [3]");
                  }
                  else
                  {
                        System.out.println("Error");
                  }
                  if(l.contains(5))
                  {
                        System.out.println("Encontrado [5]");
                  }
                  else
                  {
                        System.out.println("Error");
                  }
                  if(l.contains(6))
                  {
                        System.out.println("Error");
                  }
                  else
                  {
```

```
                        System.out.println("No encontrado [6]");
                }

        }
        else
        {
                System.out.print("Error en find | Nodo resultado:
");l.visualizarNodos();
        }

        System.out.println("\nPrueba contains");

        if(l.size()==3)
        {
                l.visualizarNodos();
                if(l.find(1)!=null)
                {
                        System.out.println("Error");
                }
                else
                {
                        System.out.println("No encontrado [1]");
                }
                if(l.find(3)!=null)
                {
                        System.out.println("Encontrado [3]");
                }
                else
                {
                        System.out.println("Error");
                }
                if(l.find(5)!=null)
                {
                        System.out.println("Encontrado [5]");
                }
                else
                {
                        System.out.println("Error");
                }
                if(l.find(6)!=null)
                {
                        System.out.println("Error");
                }
                else
                {
                        System.out.println("No encontrado [6]");
                }

        }
        else
        {
                System.out.print("Error en find | Nodo resultado:
");l.visualizarNodos();
        }

        System.out.println("\nPrueba first & last");
        if(l.size()==3)
        {
                System.out.println("El primer elemento es: "+l.first());
```

```
                System.out.println("Y deberia de ser: [3]");
                System.out.println("El ultimo elemento es: "+l.last());
                System.out.println("Y deberia de ser: [5]");
            }
            else
            {
                System.out.print("Error en removeFirst | Nodo resultado:
");l.visualizarNodos();
            }

      }
}
```

4.9 PruebaOrderedLinkedList

```
public class PruebaOrderedDoubleLinkedList {

            public static void main(String[] args)  {

                OrderedDoubleLinkedList<Integer>      l       =       new
OrderedDoubleLinkedList<Integer>();
                l.add(1);
                l.add(3);
                l.add(6);
                l.add(7);
                l.add(9);
                l.add(0);
                l.add(20);
                l.remove(7);


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


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



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

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


                System.out.print(" Lista ..............");
```

```
                l2.visualizarNodos();
                System.out.println(" Num elementos: " + l2.size());


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


        }
        }
```

4.10 UnorderedDoubleLinkedList

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

        public void addToFront(T elem)
        {
                // aï¿½ade un elemento al comienzo
                // COMPLETAR EL CODIGO Y CALCULAR EL COSTE
                Node<T> nuevo = new Node<>(elem);
                if(last==null)
                {
                        last=nuevo;
                        last.next=last;
                        last.prev=last;
                }
                else
                {
                        nuevo.prev = last;
                        nuevo.next = last.next;
                        last.next.prev = nuevo;
                        last.next = nuevo;
                }
                count++;

        }

        public void addToRear(T elem)
        {
                // aï¿½ade un elemento al final
                // COMPLETAR EL CODIGO Y CALCULAR EL COSTE
                addToFront(elem);
                last=last.next;
        }

        public void addAfter(T elem, T target)
        {
                // Aï¿½ade elem detrï¿½s de otro elemento concreto, target,  que
ya se encuentra en la lista
                // ï¿½COMPLETAR OPCIONAL!
                count++;
```

```
            if(last==null)
            {
                  last=new Node<>(elem);
                  last.next=last;
                  last.prev=last;
            }
            else if(!contains(target))
            {
                  Node<T> primero=last.next;
                  last.next=new Node<>(elem);
                  last.next.next=primero;
                  last.next.prev=last;
                  primero.prev=last.next;
                  last=last.next;
            }
            else
            {
                  Node<T> actual=last.next;
                  Node<T> previo=last.next;
                  while(!previo.data.equals(target))
                  {
                        previo=actual;
                        actual=actual.next;
                  }
                  previo.next=new Node<>(elem);
                  previo.next.next=actual;
                  previo.next.prev=previo;
                  actual.prev=previo.next;
                  if(previo.equals(last))
                  {
                        last=last.next;
                  }
            }
      }

}
```

4.11 UnorderedListADT

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

      public void addToFront(T elem);
      // aï¿½ade un elemento al comienzo

      public void addToRear(T elem);
      // aï¿½ade un elemento al final

      public void addAfter(T elem, T target);
      // Aï¿½ade elem detrï¿½s de otro elemento concreto, target,  que ya se
encuentra en la lista

}
```

5 Conclusiones

Mediante este laboratorio hemos aprendido como implementar diferentes tipos de listas ligadas y un iterador y el funcionamiento interno de cada uno de los métodos que suelen implementar

estas clases analizando así sus costes y entendiendo cuales son las ventajas y desventajas de la implementación de estas clases frente a otras clases como podría ser un array.