Universidad del País Vasco
Euskal Herriko Unibertsitatea

Ingeniería Informática de gestión y sistemas de la información
Estructura de Datos y Algoritmos
2024-25

Diseño e implementación de estructuras enlazadas: DoubleLinkedList

Iker Fernández, Urko Horas y Eneko Rodríguez

PROFESOR:
Koldo Gojenola

CURSO: 2

GRUPO: 01

Índice

Apartado 1

Introducción

En esta segunda práctica de Estructura de Datos y Algoritmos, se presentan nuevas estructuras de datos, sobre las que debemos aprender a trabajar e implementar código. Es por ello, que esta práctica propone diferentes métodos a implementar sobre estas nuevas estructuras.

Durante la práctica, hemos implementado métodos empleando nodos para recorrer y modificar listas de datos enlazadas que podrían ser simples o dobles, y circulares o lineales. En este caso serán doblemente enlazadas, y lineales.

Apartado 2

Diseño de las clases

Como solución al problema para el que se nos ha propuesto buscar un algoritmo, hemos planteado las siguientes clases, que serán explicadas posteriormente.

DoubleLinkedList, ListaWebs, Node, OrderedDoubleLinkedList, UnorderedDoubleLinkedList, Web y las clases de pruebas (PruebaConP1, PruebaDoubleLinkedList y PrubeaOrderedDoubleLinkedList), que únicamente se encargan de ejecutar el programa.

Los enlaces, programas y atributos quedan resumidos en diagrama de clases que se muestra en la figura 2.1.

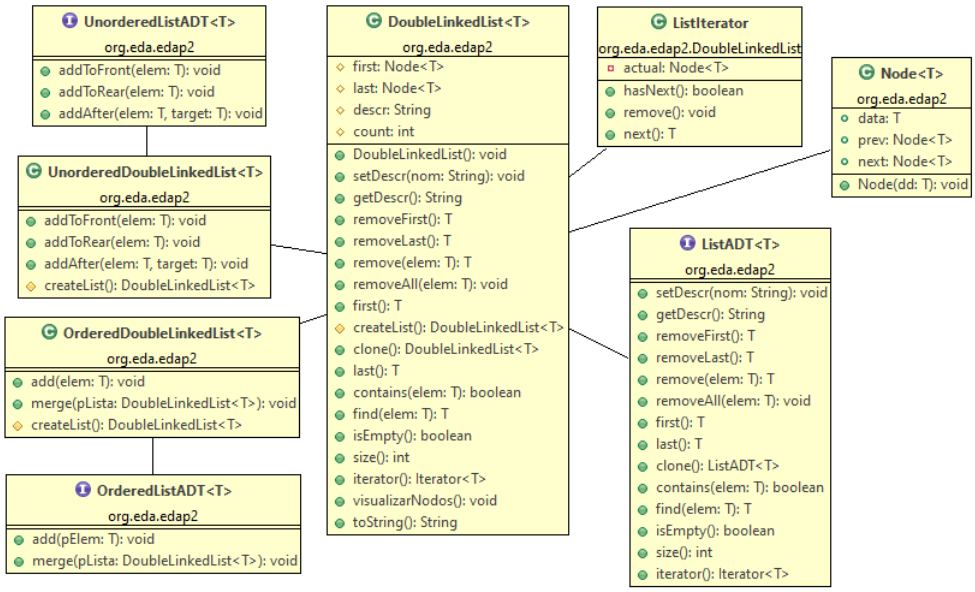

Figure 2.1: Diagrama de clases.

2.1 Clase DoubleLinkedList$< T >$

La clase DoubleLinkedList es la base de las listas doblemente enlazadas de este proyecto. Esta clase consta de 3 atributos:

- Node$< T >$ first: Nodo que representa el inicio de la lista.
- Node$< T >$ last: Nodo que representa el final de la lista.
- int count: Contador de elementos en la lista.

Además, cuenta con 4 métodos:

- isEmpty(): Devuelve "true" si la lista está vacía.
- int size(): Retorna el número de elementos en la lista.
- visualizarNodos(): Imprime todos los nodos de la lista y su contenido.
- createList(): Crea instancias de listas doblemente enlazadas.

2.2 Clase Node$< T >$

La clase Node representa cada nodo de una lista doblemente enlazada. Esta clase consta de 3 atributos:

- T data: Dato almacenado en el nodo.
- Node$< T >$ next: Referencia al nodo siguiente en la lista.
- Node$< T >$ prev: Referencia al nodo anterior en la lista.

2.3 Clase OrderedDoubleLinkedList$< T >$

La clase OrderedDoubleLinkedList, heredada de DoubleLinkedList, representa una lista doblemente enlazada cuyos elementos están ordenados ascendentemente. Esta clase consta de 3 atributos heredados de la clase DoubleLinkedList (first, last y count). Además, cuenta con 7 métodos:

- add(T elem): Añade un elemento a la lista de forma que se mantenga el orden ascendente.
- find(T elem): Devuelve "true" si el elemento especificado se encuentra en la lista.
- remove(T elem): Elimina un elemento de la lista si está presente.
- visualizarNodos(): Imprime todos los elementos en la lista, en el orden en que se encuentran.

- clone(): Devuelve una copia superficial de la lista actual.
- merge(DoubleLinkedList¡T¿ otraLista): Fusiona otra lista en la lista actual, manteniendo el orden ascendente.
- createList(): Crea una instancia de OrderedDoubleLinkedList para operaciones de clonación y mezcla.

2.4 Clase UnorderedDoubleLinkedList$< T >$

La clase UnorderedDoubleLinkedList, heredada de DoubleLinkedList, representa una lista doblemente enlazada cuyos elementos no siguen un orden específico. Esta clase consta de 3 atributos heredados de la clase DoubleLinkedList (first, last y count). Además cuenta con 4 métodos:

- addToFront(T elem): Añade un elemento al principio de la lista.
- addToRear(T elem): Añade un elemento al final de la lista.
- addAfter(T elem, T target): Inserta un nuevo elemento después de otro elemento especificado en la lista. Si el elemento objetivo no se encuentra, lanza una excepción.
- createList(): Crea una nueva instancia de UnorderedDoubleLinkedList.

2.5 Clase Persona

La clase Persona representa una persona con un nombre y un identificador único (DNI). Esta clase consta de 2 atributos:

- String nombre: Nombre de la persona.
- String dni: Identificador único (DNI) de la persona.

Además, cuenta con 6 métodos:

- Persona(String nombre, String dni): Constructor que inicializa los atributos de nombre y DNI.
- getNombre(): Devuelve el nombre de la persona.
- getDni(): Retorna el DNI.
- setNombre(String nombre): Cambia el nombre de la persona.
- setDni(String dni): Establece un nuevo DNI.
- equals(Object o): Método sobrescrito que compara dos personas por su DNI, útil en operaciones de búsqueda y comparación.

2.6 Clase Web

La clase Web pertenece a la práctica 1, aunque ha sufrido un cambio en sus atributos. Esta clase consta de 4 atributos:

```
private UnorderedDoubleLinkedList<Web> salientes;

//Lista de enlaces salientes hacia otras webs.
//Se implementa con UnorderedDoubleLinkedList
//para anadir enlaces de manera eficiente.
```

Apartado 3

Descripción de las estructuras de datos principales

Hemos empleado una tipo de estructura enlazada, DoubleLinkedList (Lista Doblemente Enlazada) y a partir de esta, hemos obtenido otras dos subestructuras, OrderedDoubleLinkedList (Lista Ordenada Doblemente Enlazada) y UnorderedDoubleLinkedList (Lista Desordenada Doblemente Enlazada).

3.1 DoubleLinkedList

Una lista doblemente enlazada es una estructura de datos lineal que, como una lista enlazada, almacena elementos de forma secuencial pero con una mejora: cada nodo de la lista contiene un enlace tanto al siguiente nodo como al anterior. Esto permite moverse en ambas direcciones a través de la lista, haciendo más sencilla la navegación entre los nodos que conforman la lista.

Además, la lista tiene un enlace directo al primer y al último elemento. Gracias a esto, los costes se ven reducidos para apuntar al último elemento. El coste se reduce de O(n), al recorrer toda la lista para llegar al último elemento, a coste O(1), constante.

3.2 Nodo

Un nodo es la unidad principal para crear las listas enlazadas. Tiene una función de almacenaje, ya que guarda un dato. Además, tiene función conectiva, ya que establece relaciones entre otros nodos permitiendo generar estructuras complejas; Gracias a esto también se guarda el recorrido para recorrer toda la lista.

3.3 UnorderedDoubleLinkedList

El concepto de esta lista es el mismo que la lista doblemente enlazada, de hecho, tiene los mismos métodos, aunque implementa algunos nuevos. Esta estructura no guarda ningún orden, los datos están desordenados.

Los métodos nuevos que implementa son el de añadir elementos al final, otro de añadirlo al principio y un último, tras otro elemento ya existente. Se puede añadir el elemento en cualquier lugar puesto que no tiene que guardar ningún orden.

3.4 OrderedDoubleLinkedList

Al igual que la anterior estructura, esta implementa los métodos de la lista doblemente enlazada y otros característicos de su clase. Además, esta lista tiene los datos ordenados, por ejemplo, si los datos son Integer, de manera ascendente.

Los métodos nuevos que implementa son el de añadir un elemento, que lo añade respetando el orden y el merge, el cual fusiona dos listas, pero la nueva lista es también ordenada.

3.5 Interfaces

Las interfaces son una forma de especificar un conjunto de métodos que las clases que las implementan deben tener. A diferencia de las clases, las interfaces no contienen implementaciones de métodos.

Son útiles para diseñar un grupo de clases que tienen que implementar un método, pero cada una respetando su estructura interna. Un ejemplo son los ArrayList, LinkedList y Vectores, que todas implementan la interfaz List, que especifica que tienen que proporcionar métodos como size() o add(); sin embargo, cada una lo implementará según lo requiera su estructura.

Apartado 4

Diseño e implementación de los métodos principales

4.1 Clase DoubleLinkedList

4.1.1 Método removeFirst()

```
public T removeFirst() {
//Precondicion: -
//Postcondicion: Se ha eliminado el primer elemento
//          de la lista
```

Casos de prueba:

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

Este es el código del algoritmo resultante:

```
T data=null;
Si la lista no esta vacia {
    data toma el valor del primer nodo;
    first=siguiente elemento de first;
    Si first es distinto de null {
        el nodo a eliminar deja de apuntar al nuevo first;
        first deja de apuntar al nodo eliminado;
    } Si no last=null;
    decrementa el tamano de la lista;
} devuelve data;
```

Coste: Coste O(1), constante. Trata los casos de que sea vacía, sólo haya 1 elemento o que haya varios.

4.1.2 Método removeLast()

```
public T removeLast() {
//Precondicion: -
//Postcondicion: Se ha eliminado el ultimo elemento
//          de la lista
```

Casos de prueba:

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

Este es el código del algoritmo resultante:

```
T data=null;
Si la lista no es vacia {
    Si la lista solo tiene un elemento {
        data=this.removeFirst();
    } Si no {
        Node<T> actual=last;
        data toma el valor del ultimo nodo;
        el ultimo elemento es el que antes era penultimo;
        el nuevo ultimo elemento deja de apuntar al nodo a eliminar;
        el nodo eliminado no apunta a nada;
        decrementa el tamano de la lista;
    }
}
devuelve data;
```

Coste: Coste O(1), constante. removeFirst() es constante, que se usa en el caso de que solo haya un elemento y las instrucciones en el else son constantes también.

4.1.3 Método remove()

```
public T remove(T elem) {
//Precondicion: -
//Postcondicion: Se ha eliminado la primera
//          aparicion de elem
```

Casos de prueba:

- Lista vacía.
- Lista con un elemento, que es el que buscamos.
- Lista con un elemento, que no es el que buscamos.
- Lista con varios elementos, que tiene el que buscamos y está en medio.
- Lista con varios elementos, que tiene el que buscamos y está al final.
- Lista con varios elementos, que no tiene el que buscamos.

Este es el código del algoritmo resultante:

```
T data=null;
Si la lista no es vacia {
    Node<T> actual=first;

    Si el elemento esta al inicio {
        devuelve this.removeFirst();
    }

    Si el elemento esta al final{
        devuelve this.removeLast();
    }

    boolean encontrado=false;
    Mientras no se haya encontrado y actual no sea nulo {
        data=actual.data;
        Si el dato de actual y elem son iguales {
            encontrado=true;
        } Si no {
            se avanza al siguiente nodo;
        }
    }
    Si se ha encontrado {
        el nodo anterior apunta al siguiente de actual;
        el siguiente nodo apunta al anterior de actual;
        actual no apunta a nada, ni previo ni posterior;
        decrementa el tamano de la lista
    }
}
devuelve data;
```

Coste: En el caso de que el elemento a borrar sea el primero o el último el coste será O(1), constante, que sería el mejor de los casos. El peor de los casos sería que fuese el penúltimo elemento, entonces el coste sería de O(n-1), lineal. Por lo que su coste medio es O(n), lineal.

4.1.4 Método iterator()

```
public Iterator<T> iterator()
```

Casos de prueba:

- Probar que funcione correctamente en algún método

Este es el código del algoritmo resultante:

```
devuelve una nueva ListIterator();

ListIterator es una clase privada dentro del metodo,
que implementa la interfaz Iterator<T>{

    atributo privado Node<T> actual=first;

    public boolean hasNext(){
        devuelve si actual es null o no;
    }

    public T next(){
        Si actual vale null lanza una excepcion
        NoSuchElementException;
        Pero si no es asi devuelve el dato de actual
        y actual avanza al siguiente nodo;
    }
```

Coste: Todos los métodos de ListIterator son constantes O(1).

4.1.5 Método first()

```
public T first(){
//Precondicion: --
//Postcondicion: Se devuelve el primer elemento de la lista
```

Casos de prueba:

- Lista con elementos
- Lista vacía

Este es el código del algoritmo resultante:

```
Si la lista esta vacia{
    devolver null;
}
Si no{
    devolver first.data;
}
```

Coste: El coste es O(1), por lo que el método tiene coste constante.

4.1.6 Método last()

```
public T last(){
//Precondicion: -
//Postcondicion: Se devuelve el ultimo elemento de la lista
```

Casos de prueba:

- Lista con elementos
- Lista vacía

Este es el código del algoritmo resultante:

```
Si la lista esta vacia{
    devolver null
}
Si no{
    devolver last.data
}
```

Coste: El coste es O(1), por lo que el método tiene coste constante.

4.1.7 Método contains()

```
public boolean contains(T elem){
//Precondicion: Se da un elemento
//Postcondicion: Se devuelve un booleano que
//        indica si el elemento esta en la lista o no
```

Casos de prueba:

- El elemento está en la lista
- El elemento no está en la lista
- Lista vacía

Este es el código del algoritmo resultante:

```
Si la lista esta vacia{
    devolver null;
} Si no {
    T data=this.find(elem);
    Si data==null devolver false;
    Si no devolver true;
}
```

Coste: El coste es O(n) donde n es el número de elementos de la lista. El coste es lineal

4.1.8 Método clone()

```
public DoubleLinkedList<T> clone(){
//Precondicion:-
//Postcondicion: Se devuelve una copia de la lista
```

Casos de prueba:

- Lista no vacía
- Lista vacía

Este es el código del algoritmo resultante:

```
DoubleLinkedList<T> clonado = createList();
Node<T> actualaux=null;
Para actual desde this.first hasta actual no sea null{
    Si clonado esta vacio {
        clonado.first=actual;
        clonado.last=actual;
        actualaux=clonado.first;
    } Si no {
        actualaux.next=actual;
        actual.prev=actualaux;
        clonado.last=actual;
        actualaux=actualaux.next;
    }
    incrementa el tamano del clonado;
}
devolver clonado;
```

Coste: El coste es O(n) siendo n el número de nodos de la lista. El coste es lineal.

4.1.9 Método find()

```
public T find(T elem){
//Precondicion: Se da un elemento
//Postcondicion: Devuelve la referencia del
//          elemento en caso de que la lista lo contenga
```

Casos de prueba:

- El elemento se encuentra en la lista
- El elemento no se encuentra en la lista
- La lista está vacía

Este es el código del algoritmo resultante:

```
T data=null;
boolean encontrado=false;
Si la lista no esta vacia {
    Node<T> actual=first;
    Mientras no se encuentre y actual!=null {
        data=actual.data;
        Si data es igual que elem {
            encontrado=true;
        } Si no {
            actual=actual.next;}
    }
}
si no se ha encontrado entonces data=null;
devolver data;
```

Coste: El coste es O(n) siendo n el número de elementos de la lista. El coste es lineal

4.2 OrderedDoubleLinkedList

4.2.1 Método add(T elem)

```
public void add(T elem) {
//Precondicion: Se da un elemento elem de tipo T
//Postcondicion: Se ha anadido el elemento elem
//                en la posicion correspondiente
```

Casos de prueba:

- Añadir cualquier elemento a una lista vacía
- Añadir cualquier elemento a una lista con más elementos

Este es el código del algoritmo resultante:

```
Node<T> nuevo=new Node<T>(elem);
Si la lista no esta vacia {
    Node<T> actual=first;
    Mientras actual no sea nulo y el dato de elem
    sea mayor o igual que el dato de actual {
        se avanza al siguiente nodo;
    }
    Si actual no es nulo {
        Si actual no apunta al primer nodo {
            el nodo anterior a actual apunta a nuevo;
            el nodo nuevo apunta al anterior de actual;
        } Si no {
            el puntero first apunta a nuevo;
        }
        actual apunta a nuevo;
        nuevo apunta a actual;
    } Si no {
        el siguiente nodo al apuntado por el puntero last apunta a nuevo;
        nuevo apunta al puntero last;
        el puntero last apunta a nuevo;
    }
} Si no {
    el puntero first apunta a nuevo;
    el puntero last apunta a nuevo;
}
Se incrementa el tamano de la lista;
```

Coste: Coste O(n), es decir, coste lineal, donde n, en el peor de los casos, corresponderá a la posición en la que deberá ir el nodo en la lista.

4.2.2 Método merge(DoubleLinkedList< T > pLista)

```
public void merge(DoubleLinkedList<T> pLista){
//Precondicion: Se da una lista pLista de tipo DoubleLinkedList
//Postcondicion: pLista esta ordenada ascendentemente
```

Casos de prueba:

- Ordenar una lista desordenada.

Este es el código del algoritmo resultante:

```
Node<T> actual=pLista.first;
Mientras actual no sea nulo {
    se anade el elemento actual a la lista;
    se pasa al siguiente nodo;
}
```

Coste: add tiene coste O(n), lineal (n número de nodos de this), y como se recorre pLista (m número de nodos) para añadir elemento a elemento el coste acaba resultando en O(n*m), cuadrático.

4.3 UnorderedDoubleLinkedList

4.3.1 Método addToFront(T elem)

```
public void addToFront(T elem) {
//Precondicion: Se da un elemento elem de tipo T
//Postcondicion: Se ha anadido el elemento elem
//                al principio de la lista
```

Casos de prueba:

- Añadir cualquier elemento al principio de cualquier lista.
- Añadir cualquier elemento al principio de una lista vacía.

Este es el código del algoritmo resultante:

```
Node<T> nuevo=new Node<T>(elem);
Si la lista esta vacia {
    el puntero first apunta al nodo nuevo;
    el puntero last apunta al nodo nuevo;
```

```
} Si no {
    nuevo apunta al nodo apuntado por el puntero first;
    el primer elemento de la lista apunta a nuevo;
    el puntero first apunta a nuevo;
}
Se incrementa el tamano de la lista;
```

Coste: Coste O(1), coste constante.

4.3.2 Método addToRear(T elem)

```
public void addToRear(T elem) {
//Precondicion: Se da un elemento elem de tipo T
//Postcondicion: Se ha anadido el elemento elem
//                al principio de la lista
```

Casos de prueba:

- Añadir cualquier elemento al final de cualquier lista.
- Añadir cualquier elemento al final de una lista vacía.

Este es el código del algoritmo resultante:

```
Node<T> nuevo=new Node<T>(elem);
Si la lista esta vacia {
    el puntero first apunta al nodo nuevo;
    el puntero last apunta al nodo nuevo;
}
Si no {
    nuevo apunta al nodo apuntado por el puntero last;
    el ultimo elemento de la lista apunta a nuevo;
    el puntero last apunta a nuevo;
}
Se incrementa el tamano de la lista;
```

Coste: Coste O(1), coste constante.

4.3.3 Método addAfter(T elem, T target)

```
public void addToRear(T elem) {
//Precondicion: Se da un elemento elem, y target de tipo T
//Postcondicion: Se ha anadido el elemento elem
//                detras del elemento target
```

Casos de prueba:

- Añadir cualquier elemento detrás de cualquier otro elemento.
- Añadir cualquier elemento en una lista vacía.

Este es el código del algoritmo resultante:

```
Si la lista esta vacia {
    lanzar una nueva IllegalStateException("La lista esta vacia");
}
Node<T> nuevo= new Node<T>(elem);
Node<T> actual=first;
boolean enc=false;
Mientras actual sea nulo y enc sea falso {
    Si el dato actual es igual a target{
        enc es true;
    }
    Si no{
        Se avanza al siguiente nodo;
    }
}
Si enc es true {
    nuevo apunta al siguiente nodo al que apunta el puntero actual;
    nuevo apunta a actual;
    Si el siguiente nodo al que apunta el puntero actual es nulo {
        el siguiente nodo a nuevo apunta a nuevo;
    }
    Si no {
        el puntero last apunta a nuevo;
    }
    actual apunta a nuevo;
    Se incrementa el tamano de la lista;
}
Si no {
    lanzar una nueva IllegalArgumentException
    ("El elemento no se encuentra en la lista");
}
```

Coste: Coste O(1), coste constante.

Apartado 5

Código

En este apartado se presentará el código de las clases.

5.1 Clase DoubleLinkedList

```
public T removeFirst() {
                T data=null;
                if(!this.isEmpty()) {
                        data=this.first();
                        first=first.next;
                        if(first!=null) {
                                first.prev.next=null;
                                first.prev=null;
                        }
                        else {

                                last=null;
                        }
                        count--;
                }
                return data;

        }


public T removeLast() {
        T data=null;
        if(!this.isEmpty()) {
                if(this.count==1) {
                        data=this.removeFirst();
                }
                else{
                        Node<T> actual=last;
                        data=this.last();
                        last=last.prev;
```

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

            public T remove(T elem) {
                T data=null;
                if(!this.isEmpty()) {
                    Node<T> actual=first;
                    if(elem.equals(first.data)) {
                        return this.removeFirst();
                    }
                    else if(elem.equals(last.data)){
                        return this.removeLast();
                    }
                        boolean enc=false;
                        while(!enc&&actual!=null) {
                            if(actual.data.equals(elem)) {
                                data=actual.data;
                                enc=true;
                            }
                            else {
                                actual=actual.next;
                            }
                        }
                        if(enc) {
                            actual.prev.next=actual.next;
                            actual.next.prev=actual.prev;
                            actual.next=null;
                            actual.prev=null;
                            count--;
                        }
                }
                return data;
            }

            public void removeAll(T elem) {
                if(!this.isEmpty()) {

                    while(first!=null&&elem.equals(first.data)) {
                        this.removeFirst();
                    }

                    Node<T> actual=first;
                    while(actual!=null) {
                        if(elem.equals(actual.data)){
```

```
                    if(actual==last){
                        this.removeLast();
                        actual=null;
                    }else{
                        Node<T> nodoAEliminar=null;
                        nodoAEliminar=actual;
                        actual=actual.next;
                        nodoAEliminar.prev.next=nodoAEliminar.next;
                        nodoAEliminar.next.prev=nodoAEliminar.prev;
                        nodoAEliminar.next=null;
                        nodoAEliminar.prev=null;
                        count--;
                    }
                }else{
                    actual=actual.next;
                }
            }
        }
    }

    public T first() {
        if (isEmpty()){
            return null;
        } else return first.data;
    }

    protected DoubleLinkedList<T> createList() {
            return new DoubleLinkedList<T>();
    }

    public DoubleLinkedList<T> clone() {
        DoubleLinkedList<T> clonado = createList();
        Node<T> actualaux=null;
        for (Node<T> actual = this.first; actual != null;
        actual = actual.next) {
            if(clonado.isEmpty()) {
                clonado.first=actual;
                clonado.last=actual;
                actualaux=clonado.first;
            } else {
                actualaux.next=actual;
                actual.prev=actualaux;
                clonado.last=actual;
                actualaux=actualaux.next;
            }
            clonado.count++;
        }
        return clonado;
    }
```

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

public boolean contains(T elem) {
        if (isEmpty()) return false;
        else {
                T data=this.find(elem);
                if(data==null)return false;
                else return true;
        }
}

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

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;

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

```
        public void remove() {
                throw new UnsupportedOperationException();
        }
        public T next() {
                if(!hasNext()) {
                        throw new NoSuchElementException();
                }
                T data=actual.data;
                actual=actual.next;
          return data;
        }
}


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

public String toString() {
        String result = new String();
        Iterator<T> itr = iterator();
        while (itr.hasNext()) {
                T elem = itr.next();
                result = result + "[" + elem.toString() + "]\n";
        }
        return "Lista.....\n" + result ;
}
```

5.2 Clase OrderedDoubleLinkedList

```
public void add(T elem){
    Node<T> nuevo=new Node<T>(elem);
    if(!this.isEmpty()) {
        Node<T> actual=first;
        while(actual!=null&&
        ((Comparable<T>)elem).compareTo(actual.data)>=0){
            actual=actual.next;
        }
        if(actual!=null) {
            if(actual!=first) {
                actual.prev.next=nuevo;
                nuevo.prev=actual.prev;
            }
            else {
                first=nuevo;
            }
            actual.prev=nuevo;
            nuevo.next=actual;
        }
        else {
            last.next=nuevo;
            nuevo.prev=last;
            last=nuevo;
        }
    }
    else {
        first=nuevo;
        last=nuevo;
    }
    count++;
}

public void merge(DoubleLinkedList<T> pLista){
    Node<T> actual=pLista.first;
    while(actual!=null) {
        this.add(actual.data);
        actual=actual.next;
    }
}
```

5.3 Clase UnorderedDoubleLinkedList

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

public void addAfter(T elem, T target) {

    if(this.isEmpty()) throw new IllegalStateException
    ("La lista esta vacia");

    Node<T> nuevo= new Node<T>(elem);
    Node<T> actual=first;
    boolean enc=false;
    while(actual!=null && !enc) {
        if(actual.data.equals(target)){
            enc=true;
        } else {
            actual=actual.next;
        }
    }
    if(enc) {
        nuevo.next=actual.next;
        nuevo.prev=actual;
        if(actual.next!=null){
            nuevo.next.prev=nuevo;
```

```
        } else {
            last=nuevo;
        }
        actual.next=nuevo;
        this.count++;
    } else {
        throw new IllegalArgumentException
        ("El elemento no se encuentra en la lista");
    }
}

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

5.4 Prueba de OrderedDoubleLinkedList

```
public static void main(String[] args)  {
    System.out.println("-----------------------------------------");
    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);

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

    OrderedDoubleLinkedList<Persona> lvacia = new
    OrderedDoubleLinkedList<Persona>();

    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));

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

```
l.visualizarNodos();
OrderedDoubleLinkedList<Integer> lcopia
=(OrderedDoubleLinkedList<Integer>)l.clone();

System.out.println("-Resultado,-num-elementos:-" + lcopia.size());

System.out.println("\nPrueba-clone-.............");
lcopia.visualizarNodos();

System.out.println("-----------------------------------------------");
OrderedDoubleLinkedList<Persona> l2 =
new OrderedDoubleLinkedList<Persona>();
System.out.println("\n");
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"));

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")));
System.out.println("7777(lvacia)?-" +
lvacia.find(new Persona("", "7777")));

System.out.println("-----------------------------------------------");
UnorderedDoubleLinkedList<Integer> l1 = new
UnorderedDoubleLinkedList<Integer>();
System.out.println("\n");
l1.addToRear(5);
l1.addToRear(2);
l1.addToRear(4);
l1.addToRear(7);
l1.addToRear(1);

l1.visualizarNodos();
System.out.println("-Num-elementos:-" + l1.size());

System.out.println("Prueba-merge-(Dada-una-lista-desordenada).....");
l.merge(l1);

System.out.println("\n-Resultado,-num-elementos:-"+ l.size());
```

```
        l.visualizarNodos();

        System.out.println("-----------------------------------------------");
        System.out.println("\n-1.");
        l=new OrderedDoubleLinkedList<Integer>();
        l.add(1);
        l.add(3);
        l.add(6);
        l.add(9);
        l.add(0);
        l.add(20);

        l.visualizarNodos();

        OrderedDoubleLinkedList<Integer> l3 = new
        OrderedDoubleLinkedList<Integer>();
        l3.add(3);
        l3.add(5);
        l3.add(6);
        l3.add(4);
        l3.add(1);
        System.out.println("-2.");
        l3.visualizarNodos();
        System.out.println("-Num-elementos:-" + l3.size());

        System.out.println("Prueba-merge-(Dada-otra-lista-ordenada).....");
        l.merge(l3);

        System.out.println("\n-Resultado,-num-elementos:-"+ l.size());
        l.visualizarNodos();

        System.out.println("-----------------------------------------------");
        System.out.println("\n-1.");
        l2.visualizarNodos();
        OrderedDoubleLinkedList<Persona> l4 = new
        OrderedDoubleLinkedList<Persona>();
        System.out.println("\n");
        l4.add(new Persona("iker", "8888"));
        l4.add(new Persona("urko", "5474"));
        l4.add(new Persona("eneko", "3945"));
        System.out.println("\n-2.");
        l4.visualizarNodos();
        System.out.println("-Num-elementos:-" + l4.size());

        System.out.println("Prueba-merge-(Con-Persona).............");
        l2.merge(l4);
        System.out.println("\n-Resultado,-num-elementos:-"+ l2.size());
        l2.visualizarNodos();
        }
}
```

5.5 Test con Práctica 1

En la práctica se nos pedía implementar las listas doblemente enlazadas en algún momento de la práctica 1. Es por eso, que hemos decidido cambiar el atributo de salientes de la clase Web, que en un principio era un ArrayList de Web, a una UnorderedDoubleLinkedList de Web.

Estos son los métodos que hemos tenido que modificar levemente para la implementación de la estructura enlazada.

```
public class Web {
    private int id;
    private String direccion;
    private UnorderedDoubleLinkedList<Web> salientes;
    private HashSet<String> palabrasClave;

    public Web(int id, String dir) {
        this.id = id;
        this.direccion = dir;
        this.salientes = new UnorderedDoubleLinkedList<Web>();
        this.palabrasClave = new HashSet<>();
    }

    public void anadirSaliente(Web pSaliente) {
        this.salientes.addToRear(pSaliente);
    }

    public ArrayList<Web> getSalientes() {
        ArrayList<Web> al=new ArrayList<Web>();
        Iterator<Web> itr = this.salientes.iterator();
        while(itr.hasNext()) {
            al.add(itr.next());
        }
        return al;
    }

    public void imprimirSalientes() {
        Iterator<Web> itr=this.salientes.iterator();
        while(itr.hasNext()) {
            System.out.println(itr.next().getDireccion()+"-");
        }
    }
}
```

Por lo que una vez implementada la UnorderedDoubleLinkedList en la clase Web, hemos decidido ponerla a prueba mediante el método cargarSalientes() de ListaWebs, que no ha sufrido ninguna modificación. Una vez cargados, se imprimirán por pantalla.

```
ListaWebs lista = new ListaWebs();
lista.cargarLista("datuak-2024-2025/index-2024-25");
lista.cargarSalientes("datuak-2024-2025/pld-arcs-1-N-2024-25");
lista.imprimir();
```

Apartado 6

Conclusiones

Las nuevas estructuras aprendidas e implementadas, son muy útiles a la hora de almacenar datos y poder manejarlos según las necesidades. Estas estructuras, permiten además de almacenar datos, enlazarlos entre ellos para así poder recorrer el conjunto de datos.

En esta práctica, hemos podido implementar las listas doblemente enlazadas, que es uno de los subtipos de las listas enlazadas. Estas estructuras pueden ser simples o doblemente enlazadas, como la utilizada en la práctica, asimismo, pueden ser secuenciales, como la utilizada, o circulares.

Además, con el fin de ver su funcionamiento real, se ha implementado la estructura creada en la práctica 1. Es aquí dónde hemos observado que no siempre es útil esta estructura, ya que en un principio queríamos implementarla en la clase ListaWebs, cambiando su atributo array, por una UnorderedDoubleLinkedList. Al ejecutar métodos como cargarSalientes(), había un notorio empeoramiento del coste, ya que de tardar segundos, pasó a tardar horas en cargar todos los salientes. Este problema viene dado por el método de obtención de un dato en la posición "x", que a diferencia del array, que tiene coste constante O(1), nuestra nueva estructura enlazada tiene coste lineal O(n). Por lo que era ineficiente para el la cantidad de datos que teníamos. Finalmente, decidimos implementarlo en el atributo de salientes de la clase Web, ya que no generaba este problema.