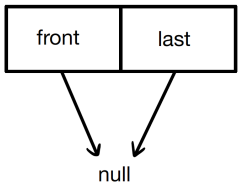
PRÁCTICA 3.- LISTA DOBLEMENTE ENLAZADA Y LISTA CIRCULAR DOBLEMENTE ENLAZADA CON NODO CABECERA

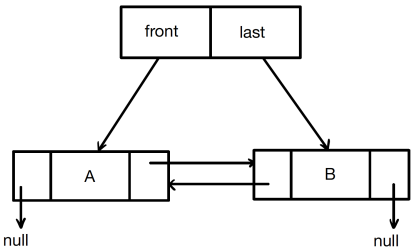
En esta práctica se implementará el interface IDoubleNotOrderedList mediante dos estructuras de datos diferentes:

-**Lista doblemente enlazada** : viene dada por dos referencias front (referencia al primer nodo de la lista) y last (referencia al último nodo de la lista). La referencia prev del primer nodo apunta a null y la referencia next del último nodo apunta a null. La lista vacía tiene front y last apuntando a null.

Lista vacía

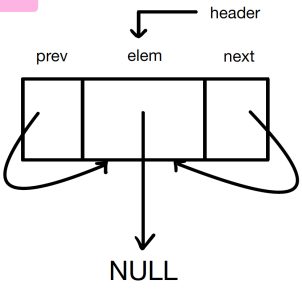


Lista con elementos

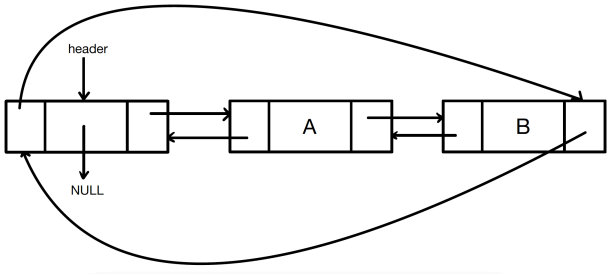


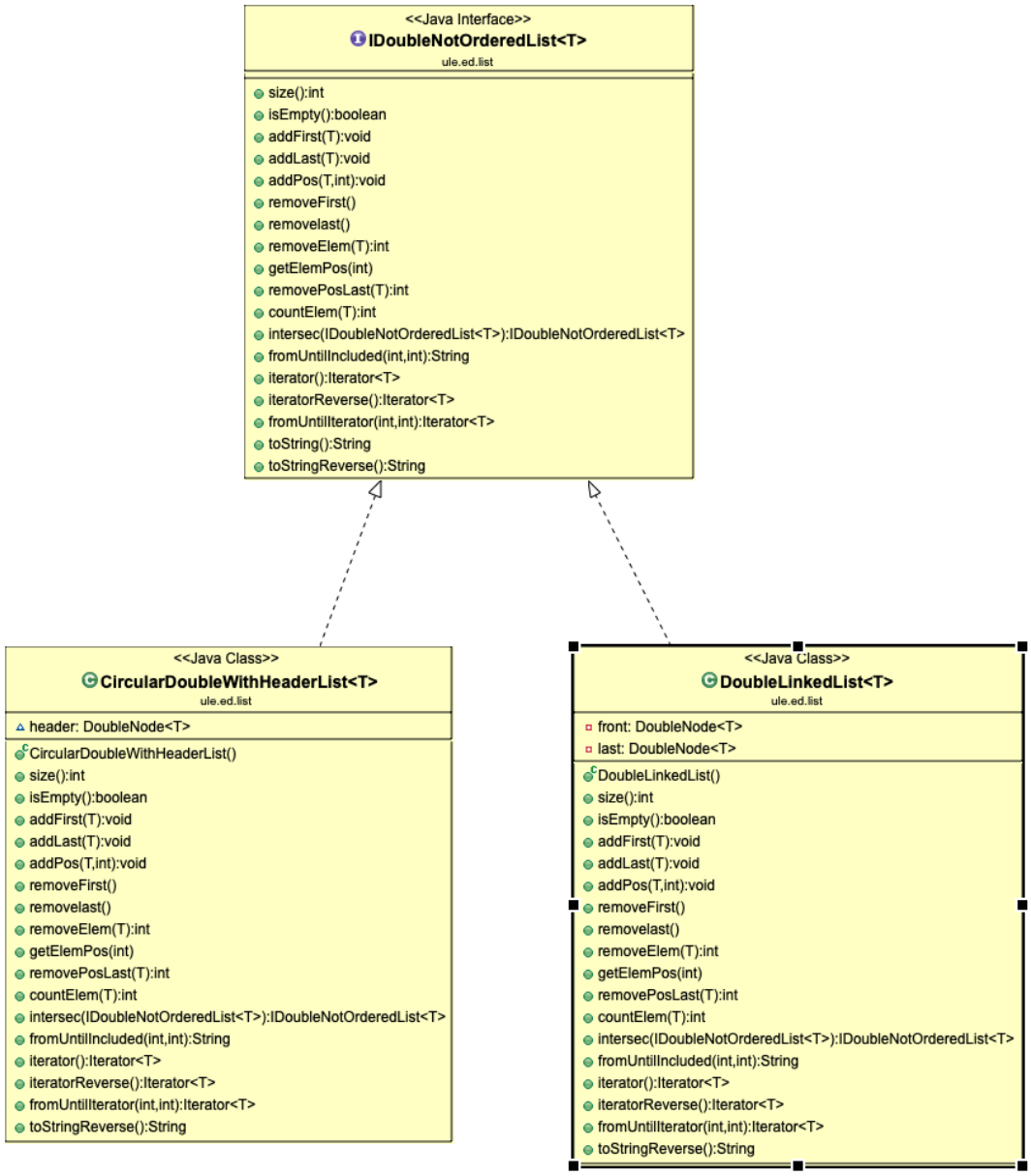
-**Lista doblemente enlazada circular con nodo cabecera**: viene dada por una referencia header a un nodo vacío (su atributo elem es null) que siempre existe en la lista (se creará en el constructor de la lista y nunca se modifica la referencia header). El constructor crea un nodo doble vacío (con elem apuntando a null y prev y next apuntando a sí mismo).

Lista vacía



Lista con elementos




<<Java Interface>>
IDoubleNotOrderedList<T>
ule.ed.list
size():int
isEmpty():boolean
addFirst(T):void
addLast(T):void
addPos(T,int):void
removeFirst()
removelast()
removeElem(T):int
getElemPos(int)
removePosLast(T):int
countElem(T):int
intersec(IDoubleNotOrderedList<T>):IDoubleNotOrderedList<T>
fromUntilIncluded(int,int):String
iterator():Iterator<T>
iteratorReverse():Iterator<T>
fromUntilIterator(int,int):Iterator<T>
toString():String
toStringReverse():String
<<Java Class>>
CircularDoubleWithHeaderList<T>
ule.ed.list
header: DoubleNode<T>
CircularDoubleWithHeaderList()
size():int
isEmpty():boolean
addFirst(T):void
addLast(T):void
addPos(T,int):void
removeFirst()
removelast()
removeElem(T):int
getElemPos(int)
removePosLast(T):int
countElem(T):int
intersec(IDoubleNotOrderedList<T>):IDoubleNotOrderedList<T>
fromUntilIncluded(int,int):String
iterator():Iterator<T>
iteratorReverse():Iterator<T>
fromUntilIterator(int,int):Iterator<T>
toStringReverse():String
<<Java Class>>
DoubleLinkedList<T>
ule.ed.list
front: DoubleNode<T>
last: DoubleNode<T>
DoubleLinkedList()
size():int
isEmpty():boolean
addFirst(T):void
addLast(T):void
addPos(T,int):void
removeFirst()
removelast()
removeElem(T):int
getElemPos(int)
removePosLast(T):int
countElem(T):int
intersec(IDoubleNotOrderedList<T>):IDoubleNotOrderedList<T>
fromUntilIncluded(int,int):String
iterator():Iterator<T>
iteratorReverse():Iterator<T>
fromUntilIterator(int,int):Iterator<T>
toStringReverse():String

PASOS PARA LA REALIZACIÓN DE LA PRÁCTICA:

1. **Descargar el proyecto** de la página de la asignatura en agora.unileon.es.

2. **Importar el proyecto en Eclipse** : Import… General…..Existing proyects into Workspace… Select archive file…

   (indicar el archivo ZIP descargado)

3. El proyecto tiene un paquete **ule.ed.list** que contiene las siguientes clases:

   a. Un interface `IDoubleNotOrderedList` **que NO DEBE MODIFICARSE**.

      Contiene la descripción de los métodos que tienen que implementar las clases DoubleLinkedList y CircularDoubleWithHeaderList (ya que implementan el interface IDoubleNotOrderedList.

```
public interface IDoubleNotOrderedList<T> {
    /**
     * TAD 'IDoubleNotOrderedList'
     *
     * Almacena una colección de objetos de tipo T, permitiendo
     * elementos repetidos.
     *
     * Ejemplo: (A B C A B D )
     *
     * Excepciones
     *
     * No se permiten elementos null. Si a cualquier método que recibe
     * un elemento se le pasa el valor null, lanzará una excepción
     * NullPointerException.
     *
     * Los valores de parámetros position deben ser mayores que cero. Si se
    * recibe un valor cero se lanzará {@link IllegalArgumentException}.
     *
     * Constructores
     *
     * Se definirá un constructor por defecto que inicialice la instancia como
     *  lista vacía.
     *
     *
     * Método {@link Object#toString()}
     *
     * El formato será mostrar el toString de los elementos separados por
     *  espacios (A B C D D D B )
     *
     *
     * @author profesor
     *
     * @param <T> tipo de elementos en la lista
     */

    /**
     * Devuelve el número total de elementos en esta lista. <br>
     *
     * Ejemplo:<br>
     * Si una lista l contiene (A B C B D A B ): <br>
     * l.size() -> 7
     *
     * @return número total de elementos en esta lista
```

```
     */
    public int size();

    /**
     * Indica si esta lista está vacía
     *
     * @return <code>true</code> si no contiene elementos
     */
    public boolean isEmpty();


    /**
     * Añade un elemento como primer elemento de la lista.
     * <p>
     * Si una lista l contiene (A B C ) y hacemos l.addFirst("C")
     *  la lista quedará. (C A B C )
     *
     * @param elem el elemento a añadir
     *
     * @throws NullPointerException si elem es <code>null</code>
     */
    public void addFirst(T elem);

    /**
     * Añade un elemento como último elemento de la lista
     * <p>
     * Si una lista l contiene (A B C ) y hacemos l.addLast("C") la lista
     *  quedará (A B C C )
     *
     * @param elem el elemento a añadir
     *
     * @throws NullPointerException si elem es <code>null</code>
     */
    public void addLast(T elem);


    /**
     * Añade un elemento en la posición pasada como parámetro desplazando los
     * elementos que estén a partir de esa posición.
     * <p>
     * Si una lista l contiene (A B C ) y hacemos l.addPos("Z", 2) la lista
     * quedará. (A Z B C ).
     * <p>
     * Si position>size() se insertará como último elemento.
     *
     * @param elem     el elemento a añadir
     * @param position la posición en la que añadirá el elemento
     *
     * @throws NullPointerException     si elem es <code>null</code>
     * @throws IllegalArgumentException si position <= 0
     *
     */
    public void addPos(T elem, int position);

    /**
     * Elimina y devuelve el primer elemento de la lista.
     * <p>
     * Si una lista l contiene (A B C ) y hacemos l.removeFirst() la lista
     *  quedará (B C ) y devolverá A
     *
     * @throws EmptyCollectionException si la lista es vacía
     *
     */
    public T removeFirst() throws EmptyCollectionException;
```

```
/**
 * Elimina y devuelve el último elemento de la lista.
 * <p>
 * Si una lista l contiene (A B C ) y hacemos l.removeLast() la lista
 *  quedará (A B ) y devolverá C
 *
 * @throws EmptyCollectionException si la lista es vacía
 *
 */
public T removelast() throws EmptyCollectionException;


/**
 * Elimina la primera aparición del elemento y devuelve la posición en la
 *   que estaba.<p>
 * Si una lista l contiene (A B C B ) y hacemos l.removeElem("B") la lista
 * quedará (A C B ) y devolverá 2
 *
 * @param elem el elemento a eliminar
 * @return posicion donde estaba el elemento
 *
 * @throws EmptyCollectionException si la lista es vacía
 * @throws NullPointerException   si elem es <code>null</code
 * @throws NoSuchElementException   si la lista no contiene el elemento
 *
 */
public int removeElem(T elem) throws EmptyCollectionException;

/**
 * Devuelve el elemento que está en position.
 * Position puede ser negativa, en ese caso se empezaría a contar por el
 *  final. <p>
 * Si una lista l contiene (A B C D E ): <br>
 * l.getElemPos(1) -> A <br>
 * l.getElemPos(3) -> C <br>
 * l.getElemPos(10) -> IllegalArgumentException
 * l.getElemPos(-1)  -> E
 * l.getElemPos(-2)  -> D
 *
 * @param position: posición a comprobar para devolver el elemento.
 *
 * @throws IllegalArgumentException si position es cero
 *           o es menor que -size() o mayor que size()
 *
 */
public T getElemPos(int position);


/**
 * Devuelve la posición de la última aparición del elemento
 *     y elimina dicha aparición.
 *<p>
 * Si una lista l contiene (A B C B D A ): <br>
 * l.removePosLast("A") -> 6 <br>
 * l.removePosLast("B") -> 4 <br>
 * l.removePosLast("Z") -> NoSuchElementException
 *
 * @param elem elemento a encontrar.
 *
 * @throws EmptyCollectionException si la lista es vacía
 * @throws NullPointerException   si elem es <code>null</code>
 * @throws NoSuchElementException si elem no está en la lista.
 *
```

```
     */
    public int removePosLast(T elem) throws EmptyCollectionException;

    /**
     * Devuelve el número de repeticiones del elemento que hay en la lista.
     * <p>
     * Si una lista l contiene (A B A C B D A ): <br>
     * l.countElem("A") -> 3 <br>
     * l.countElem("B") -> 2 <br>
     * l.countElem("Z") -> 0 <br>
     *
     * @param elem elemento a contar.
     * @return n° de apariciones del elemento en la lista

     * @throws NullPointerException   si elem es <code>null</code>
     *
     */
    public int countElem(T elem);

    /**
     * Crea una nueva lista con todos los elementos de la lista this que
     *  también estén en la lista other.
     * Aparecerán en el mismo orden en el que están colocados en this.
     * El número de instancias de cada elemento será el número de instancias
    *.  que haya en la lista que menos tenga.
     * Las listas this y other debe quedar en el mismo estado.
     * <p>
     * Si una lista lista1 contiene (A B C B D A B ) y
     * otra lista  lista2 contiene (Z B A B A K A ): <br>
     * lista1.instersec(lista2) devolverá la lista -> (A A B B ).
     *
     * Si una lista lista1 contiene (A B C B D A B ) y
     * otra lista  lista2 contiene (Z L K ): <br>
     * lista1.instersec(lista2) devolverá la lista vacía ya que no tienen
    *.   ningún elemento en común.
     *
     * Si alguna de las listas es vacía, devolverá una lista vacía.
     *
     * @param other -> lista para hacer la intersección con la lista this.
     * @return lista intersección de ambas listas
    *
     */
    public IDoubleNotOrderedList<T> intersec(IDoubleNotOrderedList<T> other);


    /**
     * Devuelve una cadena con los elementos comprendidos entre las posiciones
     *   from y until INCLUIDAS.
     * La lista actual tiene que quedar en el mismo estado.
     * from y until no pueden tomar valores 0 y los dos parámetros tienen que
    *   ser del mismo signo.
     * Dispara la excepción IllegalArgumentExceotion si :
     *    - from o until toman valores 0
     *    - from y until tienen distinto signo.
     * Si son positivos se considera el recorrido desde el primer elemento
    *   hacia el siguiente:
     *   si from >until -> el iterador no devolverá ningún elemento
     *   si until>size() -> devuelve hasta el final de la lista
     * Si son negativos se considera el recorrido desde el último elemento
    *   hacia el anterior, por lo que:
     *  - si from <until -> devuelve la cadena vacía
     *  - si until< -size() -> devuelve hasta el final de la lista
     *
     * <br>
```

```
     * Ejemplos:<br>
     * l1=(A B C D E ) ; l1.fromUntilIncluded(1,3) -> (A B C ) <br>
     * l1=(A B C D E ) ; l1.fromUntilIncluded(-2,-4) -> (D C B ) <br>
     * l1=(A B C D E ) ; l1.fromUntilIncluded(-1,-10) -> (E D C B A ) <br>
     * l1=(A B C D E ) ; l1.fromUntilIncluded(20,10) -> () <br>
     * l1=(A B C D E ) ; l1.fromUntilIncluded(-20,-10) -> () <br>
     * l1=(A B C D E ) ; l1.fromUntilIncluded(4,10) -> (D E ) <br>
     * l1=(A B C D E ) ; l1.fromUntilIncluded(-4,-10) -> (B A ) <br>
     *
     *
     * @param from  posición desde la que se empieza a considerar la lista
     *              (incluida)
     * @param until posición hasta la que se incluyen elementos (incluida)
     *
     * @return cadena formada por el toString() de los  elementos en el rango
     *          establecido por los dos parámetros (AMBOS INCLUIDOS).
     *
     * @throws IllegalArgumentException   si from o until toman valores 0
     *    o si toman distinto signo
     *
     */
    public String fromUntilIncluded(int from, int until);

    /**
     * Devuelve un iterador que recorre la lista desde el primero hasta el
     *  último elemento. <br>
     *
     * Por ejemplo, para una lista x con elementos (A B C D E)
     *
     * el iterador creado con x.iterator() devuelve en sucesivas llamadas a
     *    next():   A, B, C, D y E.
     *
     * @return iterador .
     */
    public Iterator<T> iterator();

    /**
     * Devuelve un iterador que recorre la lista desde el último hasta el
     *  primero elemento. <br>
     *
     * Por ejemplo, para una lista x con elementos (A B C D E)
     *
     * el iterador creado con x.iteratorReverse() devuelve en sucesivas
     *  llamadas a next():  E, D, C, B y A.
     *
     * @return iterador .
     */
    public Iterator<T> iteratorReverse();


    /**
     * Devuelve un iterador que recorre los elementos que ocupan posiciones
     *  desde from hasta until (ambas incluidas) en la lista.
     *
     * Dispara la excepción IllegalArgumentExceotion si :
     *    - from o until toman valores 0
     *    - from y until tienen distinto signo.
     *
     * Por ejemplo, para una lista x con elementos (1 2 3 4 5 6 7 8 9 )
     *
     * el iterador creado con x.fromUntilIterator(3,6) devuelve en sucesivas
     *  llamadas a next(): 3, 4, 5, 6
     *
     * el iterador creado con y.fromUntilIterator(-3,-6) devuelve en sucesivas
```

```
     * llamadas a next(): 7, 6, 5, 4
     * @param from  posición desde la que se empieza a considerar la lista
     *      (incluida)

     * @param until posición hasta la que se incluyen elementos (incluida)
     * @return iterador para recorrer los elementos de la lista comprendidos
     *         entre las posiciones from y until.
     *
     * @throws IllegalArgumentException   si from o until toman valores 0
     *    o si toman distinto signo
     */
    public Iterator<T> fromUntilIterator(int from, int until);

    /**
     * Devuelve una cadena de texto con el contenido de la lista
     * desde el primero hasta el último
     *
     * Por ejemplo, para una lista x con elementos (1 2 3 4 5 6 7 8 9 )
     * x.toString() devolverá la cadena"(1 2 3 4 5 6 7 8 9 )"
     *
     * @return una cadena de texto con el contenido de la lista
     * desde el primero hasta el último
     *
     */
    public String toString();

    /**
     * Devuelve una cadena de texto con el contenido de la lista
     * desde el último hasta el primero
     *
     * Por ejemplo, para una lista x con elementos (1 2 3 4 )
     * x.toStringReverse() devolverá la cadena"(4 3 2 1 )"
     *
     * @return una cadena de texto con el contenido de la lista
     * desde el primero hasta el último
     *
     */
    public String toStringReverse();


}
```

b. **DoubleLinkedList.java** : implementará el interface mediante una lista doblemente enlazada con dos referencias **front y last.**

c. **CircularDoubleWithHeaderList.java**: donde se implementará el interface mediante una lista circular doblemente enlazada con nodo cabecera. La lista viene definida por una referencia al nodo cabecera. Si la lista tiene elementos header.next apuntará al nodo que contiene el primer elemento; header.prev apuntará al nodo que contiene el último elemento de la lista.

```
private DoubleNode<T> header;
```

Se deben implementar para estas dos estructuras de datos los métodos definidos en el interface (incluyendo los 3 iteradores). Se creará una clase interna por cada iterador en cada una de las implementaciones (similar a lo realizado en la práctica 2).

d. CircularDoubleLinkedListTest.java y DoubleLinkedList.java

A la vez que se van desarrollando los métodos de las clases anteriores se deben crear los correspondientes métodos de **prueba JUnit 4** para ir comprobando su correcto

funcionamiento. Serán los mismos tests para las dos implementaciones ya que hay que probar la misma funcionalidad (solamente cambiará la clase elegida para crear las variables de tipo lista). Se valorará la cobertura del código implementado.

4. Una vez desarrollada la práctica se procederá a probarla tantas veces como se necesite en la plataforma de prueba automática VPL.

5. **Además se deberá entregar en agora.unileon.es la versión final de la práctica** (proyecto exportado como zip) que deberá coincidir con la última evaluación realizada en la plataforma de evaluación (VPL).

<u>NOTA IMPORTANTE</u>: NO SE PUEDEN UTILIZAR LAS ESTRUCTURAS DE DATOS DEL API COLLECTIONS DE JAVA (HAY QUE UTILIZAR LAS ESTRUCTURAS DE DATOS INDICADAS EN LAS CLASES DEL PROYECTO ED_Pract2_2021)

FECHA LIMITE de entrega de la práctica ED_Pract3_2023:16 de Abril de 2023 a las 23:59