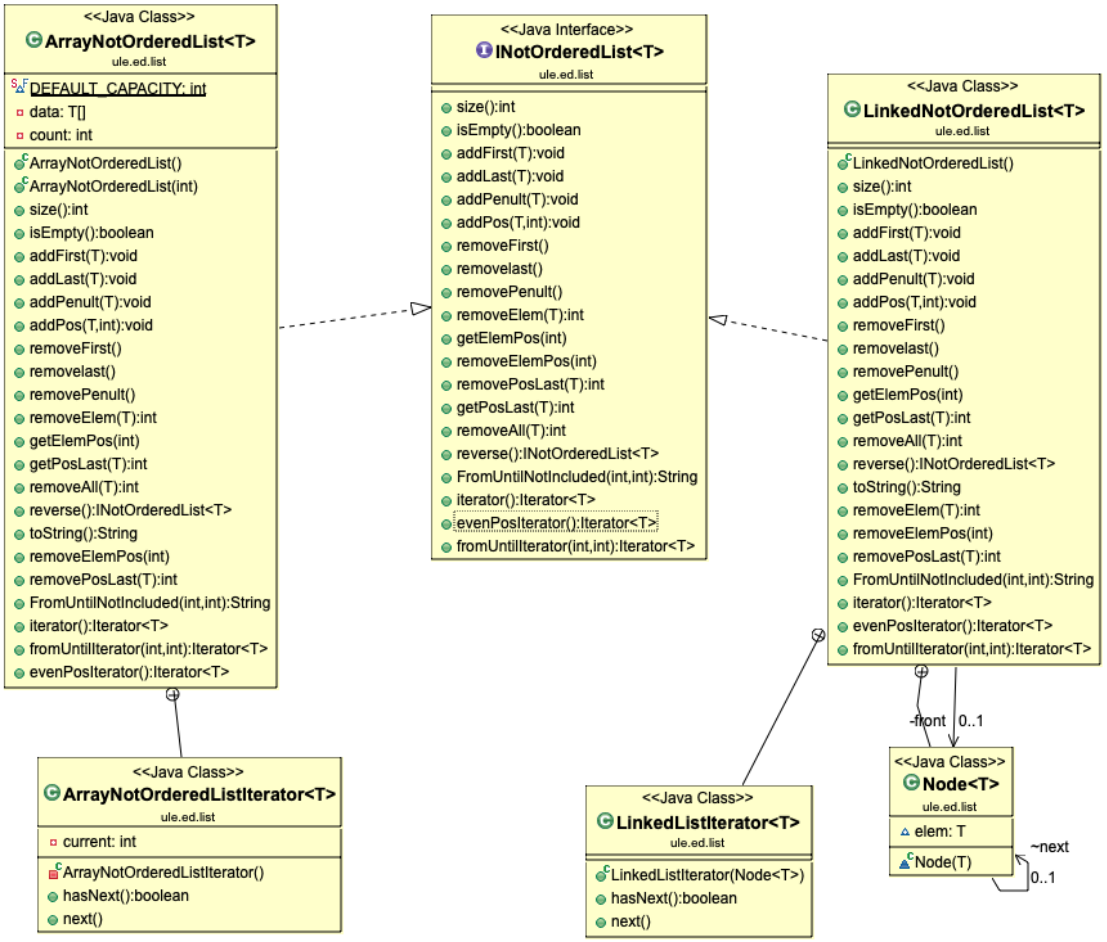
PRÁCTICA 2.- LISTAS NO ORDENADAS (Arrays y estructuras enlazadas)



PASOS PARA LA REALIZACIÓN DE LA PRÁCTICA:

1. **Descargar el proyecto** de la página de la asignatura en agora.unileon.es.
2. **<u>Importar el proyecto en Eclipse</u>** : Import… General…..Existing projects into Workspace… Select archive file…

   (indicar el archivo ZIP descargado)

Los proyectos normalmente contendrán la estructura a respetar ..

3. El proyecto tiene un paquete **ule.ed.list** que contiene las siguientes clases:
   a. Un interface INotOrderedList **que <u>NO DEBE MODIFICARSE</u>**.

      Contiene la descripción de los métodos que tienen que implementar las clases LinkedNotOrderINotOrderedList y ArrayNotOrderINotOrderedList (ya que implementan el interface INotOrderedList

```
package ule.ed.list;
import java.util.Iterator;

public interface INotOrderedList<T> {
```

```
/**
 * TAD 'INotOrderedList'
 *
 * Almacena una colección de objetos de tipo <code>T</code>, permitiendo elementos repetidos.
 * Ejemplo: (A B C A B D )
 *
 * Excepciones
 *
 * No se permiten elementos <code>null</code>. Si a cualquier método que recibe un elemento
 * se le pasa el valor <code>null</code>, lanzará una excepción
 * {@link NullPointerException}.
 *
 * Los valores de parámetros <code>position</code> deben ser mayores que cero y
 * nunca negativos. Si se recibe un valor negativo o 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:  Si una lista l contiene (A B C B D A  B ): <br>
 *
 * @return número total de elementos en esta lista  l.size() -> 7
 */
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 como penúltimo elemento de la lista
     * <p>
     * Si una lista l contiene (A B C ) y hacemos l.addPenult("D") la lista quedará (A B D C )
     *
     * Si la lista está vacía lo inserta normal
     *
     * @param elem el elemento a añadir
     *
     * @throws NullPointerException si elem es <code>null</code>
     */
    public void addPenult(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 y devuelve el penúltimo elemento de la lista.
         * <p>
         * Si una lista l contiene (A B C ) y hacemos l.removePenult() la lista quedará
         * (A C ) y devolverá B
         *
         * @throws EmptyCollectionException si la lista es vacía
         * @throws NoSuchElementException   si la lista solo tiene un elemento
         *
         */
        public T removePenult() 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 posicioon donde estaba el elemento
         *
         * @throws EmptyCollectionException si la lista es vacía
         * @throws NoSuchElementException   si la lista no contiene el elemento
         *
         */
        public int removeElem(T elem) throws EmptyCollectionException;;

        /**
         * Devuelve el elemento que está en position.
         * <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
         *
         * @param position posición a comprobar para devolver el elemento
         *
         * @throws IllegalArgumentException si position no está entre 1 y size()
         *
         */
        public T getElemPos(int position);

        /**
         * Devuelve y elimina el elemento que está en position.
         * <p>
         * Si una lista l contiene (A B C D E ): <br>
         * l.removeElemPos(1) -> A , l queda (B C D E ) <br>
         * Dada l=(A B C D E ); l.removeElemPos(3) -> C , l queda (A B D E ) <br>
         * Dada l=(A B C D E ); l.removeElemPos(10) -> IllegalArgumentException
         *
         * @param position posición a comprobar para devolver y eliminar el elemento
         *
         * @throws IllegalArgumentException si position no está entre 1 y size()
         *
         */
        public T removeElemPos(int position);


        /**
```

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

    /**
     * Devuelve la posición de la última aparición del elemento.
     * <p>
     * Si una lista l contiene (A B C B D A ):  <br>
     * l.removePosLast("A") -> 6 ; l=(A B C B D ) <br>
     * Dada l=(A B C B D A ); l.removePosLast("B")  -> 4; l=(A B C D A ) <br>
     * l.removePosLast("Z") -> NoSuchElementException
     *
     * @param elem elemento a encontrar y eliminar.
     *
     * @throws NullPointerException   si elem es <code>null</code>
     * @throws NoSuchElementException si elem no está en la lista.
     *
     */
    public int getPosLast(T elem);

    /**
     * Elimina todas las apariciones del elemento y devuelve el número de instancias eliminadas.
     * <p>
     * Si una lista l contiene (A B C B D A B ):  <br>
     * l.removeAll("A") -> 2, dejando la lista (B C B D B ): <br>
     * Dada l=(A B C B D A B ); l.removeAll("B")  -> 3, l= (A C D A )  <br>
     * l.removeAll("Z") -> NoSuchElementException
     *
     * @param elem elemento a eliminar.
     * @throws EmptyCollectionException
     *
     * @throws NullPointerException   si elem es <code>null</code>
     * @throws NoSuchElementException si elem no está en la lista.
     *
     */
    public int removeAll(T elem) throws EmptyCollectionException;

    /**
     * Crea una nueva lista inversa de esta lista. Si esta lista es vacía devuelve la lista vacía.
     * <br>
     *
     * Ejemplo:<br>
     * Si una lista l contiene (A B C ):    l.reverse().toString() -> (C B A )
     *
     * @return lista inversa de esta lista
     */
    public INotOrderedList<T> reverse();
```

```
    /**
     * Devuelve una cadena con los elementos comprendidos entre las posiciones from
     * hasta until NO INCLUIDAS.
     * La lista actual tiene que quedar en el mismo estado.
     * <br>
     * Si until > size() se inserta en la lista hasta el final de la lista. <br>
     * Si from > size() se devuelve la lista vacía
     *
     * <br>
     * Ejemplos:<br>
     * l1=(A B C D E ) ; l1.FromUntilNotIncluded(1,3) -> (B ) <br>
     * l1=(A B C D E ) ; l1.FromUntilNotIncluded(3,10) -> (D E ) <br>
     * l1=(A B C D E ) ;l1.FromUntilNotIncluded(10,20) -> () <br>
     *
     * @param from  posición desde la que se empieza a considerar la lista (no incluida)
     * @param until posición hasta la que se incluyen elementos (no incluida)
     *
     * @return cadena formada por el tosString() de los  elementos en el rango
     *         establecido por los dos parámetros (NO INCLUIDOS).
     *
     * @throws IllegalArgumentException si from o until son <=0 ; o si until < from
     *
     */
    public String FromUntilNotIncluded(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 los elementos con posición par de la lista.
     *
     * Por ejemplo, para una lista x con elementos (A B C D E )
     *
     * el iterador creado con x.evenPosIterator() devuelve en sucesivas llamadas a next(): B y D.
     *
     * @return iterador para recorrer elementos en posiciones pares.
     */
    public Iterator<T> evenPosIterator();


    /**
     * Devuelve un iterador que recorre los elementos que ocupan posiciones desde from
     * hasta until (ambas incluidas) en la lista.
     *
     * 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
     *
     * @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.
             */
            public Iterator<T> fromUntilIterator(int from, int until);
}
```

a. **ArrayNotOrderINotOrderedList.java**: donde se implementará el interface INotOrderedList mediante un array de elementos (data) un contador de elementos (count) que indica el número de elementos y la primera posición libre del array.

```
private T[] data;
private int count;
```

Se deben implementar para estas estructuras de datos los métodos definidos en el interface INotOrderedList (incluyendo los iteradores). Se creará una clase interna por cada iterador (similar a la clase ArrayNotOrderedListIterator<T> que aparece en la clase)

En las operaciones de insertar elementos en la lista (addFirst, addLast, addPenult y addPos) si no hay espacio en el array **se debe expandir capacidad a un tamaño igual al doble del actual.**

b. **LinkedNotOrderINotOrderedList.java**: donde se implementará el interface INotOrderedList mediante estructuras enlazadas (una lista estará definida por una referencia al primer nodo de la lista o a null si es vacía). **No existe el atributo count en esta implementación**.

```
private Node<T> front;
```

Se deben implementar para esta estructura de datos los métodos definidos en el interface INotOrderedList <T> (incluyendo los iteradores). Se creará una clase interna por cada iterador (similar a la clase LinkedNotOrderedListIterator<T> que aparece en la clase)

c. ArrayNotOrderedListTest.java y LinkedNotOrderedList Test.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 y los métodos de tests para probar la expansión de la capacidad que solo estarán en la del array). Se valorará la cobertura del código implementado.

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

3. **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).

NOTA IMPORTANTE: 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_2023)

FECHA LIMITE de entrega de la práctica ED_Pract2_2023: 23 de Marzo de 2023 a las 23:59