Tema 3 – Listado de Ejercicios

IMPORTANTE: *Algunos ejercicios de esta lista corresponden a exámenes de años anteriores y en ellos, como veréis, se utiliza una implementación ligeramente diferente de LinkedList<E> (usando nodos fantasma), que es que existía en el OpenJDK en ese momento. Como dicha implementación está ilustrada en el enunciado, no hay problema alguno en resolverlos siguiéndola (y, de hecho, pueden re-resolverse usando la implementación actual).*

Ejercicio 1

Define cada una de las siguientes estructuras de datos: pila, lista y cola. Las definiciones deben permitir diferenciar claramente cada una de las otras dos.

Ejercicio 2

ArrayList: indica si les següents afirmacions són correctes o incorrectes, i raona la teva resposta:

1) (0.15p) Afegir elements al principi de la llista és una operació d'ordre lineal, O(N).
2) (0.15p) Afegir elements al final de la llista és una operació d'ordre O(1).
3) (0.15p) Esborrar el primer element és una operació d'ordre O(1).

Ejercicio 3

Durant la primera part del curs, concretament al Tema 2 i Tema 3, hem vist iteradors sobre les llistes. Hem comentat dos tipus, els Iterators, i els ListIterators, i que aquests últims ens resulten més interessants, ja que ens permeten recórrer les llistes cap endavant i endarrere, i fer-ne operacions sobre elles (p. ex. afegir, eliminar).

Us demanem que implementeu la següent operació utilitzant ListIterator i en un ordre d'execució lineal, O(N).

```
public static <E> void reverse (List<T> list)
```

Capgira l'ordre dels elements de la llista. Per exemple, converteix la llista {A, B, C, D} en la llista {D, C, B, A}. Per fer aquest exercici, a més d'utilitzar next i previous, necessiteu utilitzar el mètode set definit a ListIterator. Aquest mètode reemplaça el valor de l'atribut element de lastReturned per un de nou:

```
public void set(E e) {
```

```
    if (lastReturned == header)
        throw new IllegalStateException();
    lastReturned.element = e;
```

Observació: Aquest problema admet diferents solucions. Nosaltres valorarem únicament aquelles que treballin amb *ListIterator*. Podeu treballar amb més d'un *ListIterator*.

Ejercicio 4

Implementa el següent mètode genèric. Fes-ho en temps lineal i amb un *ListIterator*. Us recordem que els principals mètodes dels *ListIterator* són: *hasNext()*, *hasPrevious()*, *next*, *previous* i *set*.

```
public static <E> void fill (List<? super E> list, E obj)
```

El mètode omple (totes les posicions de) la llista *list* amb l'element *obj*.

Ejercicio 5

- Tenemos un *ListIterator*<*E*> sobre una lista no vacía, el cual se encuentra en su última posición posible. Si realizamos *previous()*, ¿qué ocurre? ¿Y si llamamos a *next()*?

- Tenemos la clase Fish que tiene un método `public int getWeight()` para obtener el peso de un pescado. Se pide implementar un método que reciba una lista de pescados y un peso mínimo, y elimine de la lista los pescados que no lleguen al peso mínimo. La cabecera de dicha función será: `public static void removeSmallFish(List<Fish> fishes, int minimumWeight)`.

    - El coste de la función debe ser O(N), donde N es el número de elementos de la lista pasada como parámetro.

Ejercicio 6

Tenemos un *ListIterator*<*E*> sobre una lista no vacía, el cual se encuentra en su última posición posible. Si realizamos *previous()*, ¿qué ocurre? ¿Y si llamamos a *next()*?

Ejercicio 7

a) Tenemos *listIt*, un *ListIterator<E>* sobre una *LinkedList<E>* no vacía, el cual se encuentra en una posición intermedia, es decir, en una posición que no es ni la primera ni la última. Si realizamos la secuencia de operaciones "1º *listIt.previous()* -> 2º *listIt.remove()* -> 3º *listIt.remove()*" ¿qué ocurrirá?

b) En términos de coste, ¿hay diferencia a la hora de hacer una eliminación en una cola circular implementada utilizando un array con respecto a una cola implementada con una *LinkedList<E>*? ¿Por qué?

c) ¿Cuál es la diferencia, en términos de coste, entre implementar con *ArrayList<E>* o con *LinkedList<E>* un método que realice inserciones al final de una lista?

Ejercicio 8

Queremos realizar una implementación de la interfaz Stack<E>, definida como:

```
public interface Stack<E> {
    void push(E elem);
    void pop();
    E top();
    boolean isEmpty();
}
```

La implementación que se pide se basará en el uso de un array para representar la pila, existiendo una constante que defina su tamaño inicial por defecto (*DEFAULT_CAPACITY*). En este sentido, la clase *MyStack<E>* tendrá dos constructores, uno sin parámetros (haciendo uso del tamaño por defecto) y otro con un parámetro que será el tamaño inicial del array que represente la pila.

Es importante tener en cuenta que, si es necesario redimensionar, se debe duplicar el tamaño del array en cada redimensión.

Las operaciones *top* y *pop* lanzan *NoSuchElementException* e *IllegalStateException* respectivamente si la pila está vacía.

El esquema sobre el que trabajar podría ser el siguiente:

```
public MyStack<E> implements Stack<E> {
    private static final int DEFAULT_CAPACITY = 10;
    private int size;
    private Object[] theStack;

    public MyStack(){
```

```
        . . .
    }

    public MyStack(int initialCapacity){
        . . .
    }
```

Nota: Investigueu el mètodes estàtics de còpila de la classe d'utilitat `Arrays`.

Ejercicio 9

Implementa el método *addFirst(E elem)* que inserta el elemento *elem* como primer elemento de un *ArrayList<E>*. Esta implementación se hará dentro de la clase *ArrayList<E>*, en la cual, los elementos se encuentran almacenados en un array de Objects (siendo el primer elemento de la lista el que ocupa la posición 0, el segundo el que ocupa la 1, etc.). El número de elementos en la lista viene indicado por *size*.

Es importante tener en cuenta que, si no cabe el elemento que queremos insertar con *addFirst(E e),* el array que almacena los elementos debe ser redimensionado (doblad el tamaño del array en cada redimensión).

Si se hace uso de algún método auxiliar (cosa recomendada porque hacerlo todo en el mismo método no es nada aconsejable), debe ser implementado junto a *addFirst(E e)*.

```
public class ArrayList<E>{

    private Object[] theArray;
    private int size;
    . . .
    public void addFirst(E elem){
        . . .
    }
    . . .
}
```

Ejercicio 10

Implementa el método *addInPosition(Collection<? extends E> elements, int position)* que inserta los elementos de la colección *elements* a partir de la posición *position* de un *ArrayList<E>*. El primer elemento de *elements* quedará en la posición *position,* el segundo de *elements* en *position + 1*, etc. Evidentemente, los elementos que había en el *ArrayList<E>* desde *position* quedarán desplazados, no eliminados.

Para acceder a los elementos de *elements* se exige el uso de un iterador.

La implementación se hará dentro de la clase *ArrayList<E>*, en la cual, los elementos se encuentran almacenados en un array de *Objects* (siendo el primer elemento de la lista el que ocupa la posición 0, el segundo el que ocupa la 1, etc.). El número de elementos en la lista viene indicado por *size*.

```
public class ArrayList<E>{
    private Object[] theArray;
    private int size;
    //. . .

    public void addInPosition(Collection<? extends E> elements, int
position){
        //. . .
    }
    //. . .
}
```

Es importante tener en cuenta que, si no caben los elementos que queremos insertar con *addInPosition,* el array que almacena los elementos debe ser redimensionado para que quepan. Además, si *position* representa una posición no válida (*position < 0 || position > size*), el método *addInPosition* devolverá la excepción *IndexOutOfBoundsException*.

Si se hace uso de algún método auxiliar debe ser implementado junto a *addInPosition*.

Ejercicio 11

Implementa el método *addLast* que añade un elemento pasado como parámetro al final de un *ArrayList<E>*. Si fuera necesario redimensionar, se hará al doble del tamaño existente en el momento de la redimensión.

Si se hace uso de algún método auxiliar debe ser implementado junto a *addLast*.

```
public class ArrayList<E> {

    private Object[] theArray;
    private int size;

    //. . .

    public void addLast(E element){

        //. . .

    }

    //. . .

}
```

Ejercicio 12

Implementa el método *rightBulkDelete* que dada una posición (parámetro *position*) de un *ArrayList<E>* elimine un número concreto de elementos (parámetro *length*) hacia la derecha (desde la posición indicada por *position*). La figura muestra un ejemplo de uso para un *ArrayList<E>* de enteros.

En función del valor de *position y length,* el método actuará de la siguiente manera:

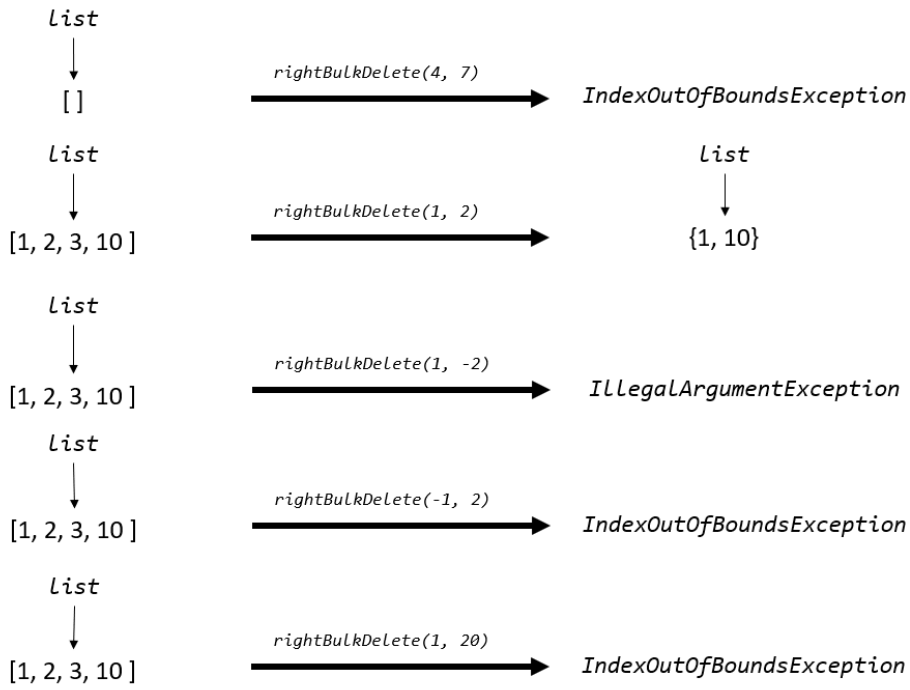
- La variable *position* deberá ser una posición válida. En caso de no serlo *(position < 0 || position > size)*, el método *rightBulkDelete* devolverá la excepción *IndexOutOfBoundsException*.
- Si *length* es negativo, lanza la excepción *IllegalArgumentException*.
- Todos los elementos [position, position+ length) han de estar en el rango del *ArrayList<E>*. Si no es así, se lanzará la excepción *IndexOutOfBoundsException*.
- Si los parámetros son correctos, borra *length* elementos del *ArrayList<E>* desde la posición *position* hacia la derecha, es decir, borrará [position, position+ length):
    - Como se puede comprobar, el valor en la posición *position* será uno de los elementos eliminados, siempre que *length* no sea cero.
    - Recuerda que, en matemáticas, "[" representa un extremo cerrado y ")" un extremo abierto.

¿Cuál es el orden para realizar las comprobaciones? Primero se comprueba *position* y después, *length*.

El método será implementado dentro de la clase *ArrayList<E>*, en la cual, los elementos se encuentran almacenados en un array de Objects (siendo el primero elemento de la lista el que ocupa la posición 0, el segundo el que ocupa la 1, etc.). El número de elementos en la lista viene indicado por size. Con respecto a la variable *modcount* existente en *ArrayList<E>* no deberás preocuparte.

```
public class ArrayList<E> {
    private Object[] theArray;
    private int size;
    //. . .

    public void rightBulkDelete(int position, int length){
    //. . .
    }
}
```

Ten en cuenta en la respuesta a la hora de implementar *rightBulkDelete,* evitar mantener referencias innecesarias que impidan al Garbage Collector de Java recuperar memoria inaccesible.



Ejemplos sobre el funcionamiento del método *rightBulkDelete*

Ejercicio 13

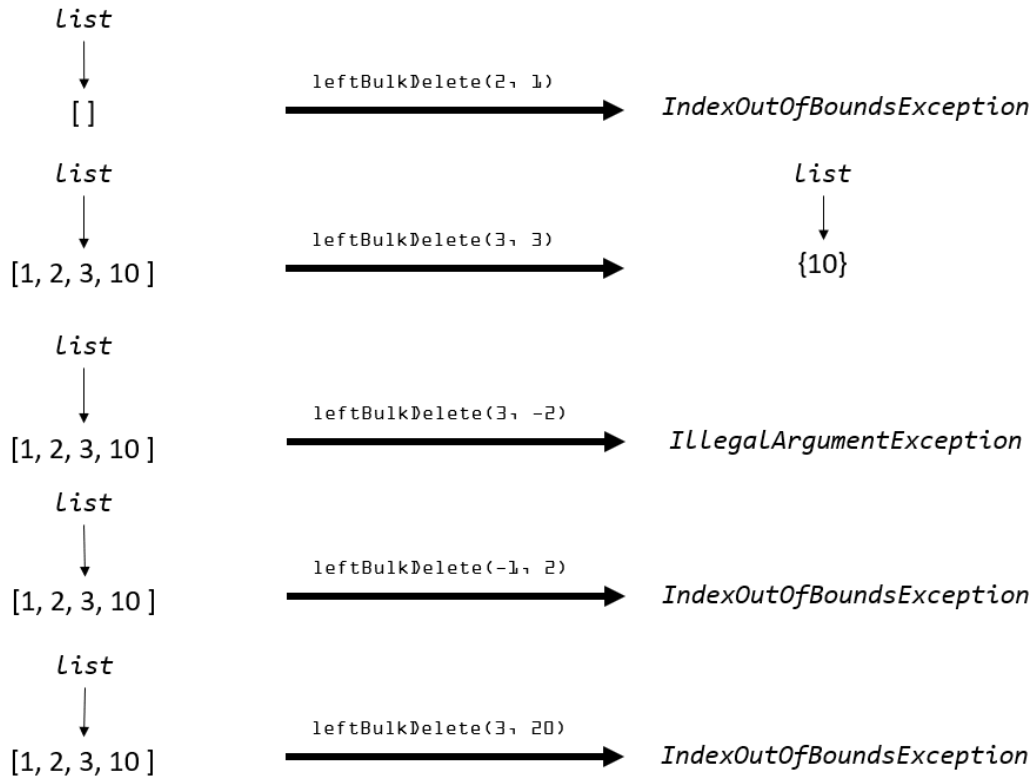
Implementa el método *leftBulkDelete* que dada una posición (parámetro *position*) de un *ArrayList<E>* elimine un número concreto de elementos (parámetro *length*) hacia la izquierda (hasta la posición indicada por *position*). La figura muestra un ejemplo de uso para un *ArrayList<E>* de enteros.

En función del valor de *position y length,* el método actuará de la siguiente manera:

- La variable *position* deberá definir un límite derecho válido, por lo tanto, debe estar en el rango [0, *size*]
- Si la variable *position* no define un límite válido, *(position < 0 || position > size)*, el método *leftBulkDelete* devolverá la excepción *IndexOutOfBoundsException*.
- Si *length* es negativo, lanza la excepción *IllegalArgumentException*.
- Todos los elementos [*position* - *length*, *position*) han de estar en el rango del *ArrayList<E>*. Si no es así, se lanzará la excepción *IndexOutOfBoundsException*.
- Si los parámetros son correctos, borra *length* elementos del *ArrayList<E>* desde la posición *position* hacia la izquierda, es decir, borrará los elementos definidos por el rango [position - length, position):

- Como se puede comprobar, el valor en la posición *position* no será uno de los elementos eliminados. En este sentido, en el caso de que *position* sea 0 hay que tener en cuenta que será un caso válido si también *length* es 0.
- Recuerda que, en matemáticas, "[" representa un extremo cerrado, que incluye el límite; y ")" un extremo abierto, que no lo incluye

¿Cuál es el orden para realizar las comprobaciones? Primero se comprueba *position* y después, *length*.



Ejemplos sobre el funcionamiento del método *leftBulkDelete*

El método será implementado dentro de la clase *ArrayList<E>*, en la cual, los elementos se encuentran almacenados en un array de Objects (siendo el primero elemento de la lista el que ocupa la posición 0, el segundo el que ocupa la 1, etc.). El número de elementos en la lista viene indicado por size. Con respecto a la variable *modcount* existente en *ArrayList<E>* no deberás preocuparte.

```
public class ArrayList<E> {
    private Object[] theArray;
    private int size;
    //. . .
```

```
    public void leftBulkDelete(int position, int length){
    //. . .
    }
}
```

Ten en cuenta en la respuesta a la hora de implementar *leftBulkDelete,* evitar mantener referencias innecesarias que impidan al Garbage Collector de Java recuperar memoria inaccesible.

Ejercicio 14

Implementa el método *addAllBeforePosition(Collection<? extends E> elements, int position)* que inserta los elementos de la colección *elements* antes de la posición position de un *ArrayList<E>*. Los elementos de elements mantendrán dentro del *ArrayList<E>* el mismo orden que tenían en elements. Además, los elementos que había en el *ArrayList<E>* no son eliminados.

Para acceder a los elementos de *elements* se exige el uso de un iterador.
La implementación se hará dentro de la clase *ArrayList<E>,* en la cual, los elementos se encuentran almacenados en un array de *Objects* (siendo el primer elemento de la lista el que ocupa la posición 0, el segundo el que ocupa la 1, etc.). El número de elementos en la lista viene indicado por *size*.

```
public class ArrayList<E>{
    private Object[] theArray;
    private int size;
    //. . .

    public void addAllBeforePosition(
            Collection<? extends E> elements,
            int position){
        //. . .
    }

    //. . .

}
```

Es importante tener en cuenta que, si no caben los elementos que queremos insertar con *addAllBeforePosition*, el array que almacena los elementos debe ser redimensionado para que quepan. Además, si position representa una posición no válida *(position < 0 || position > size)*, el método *addAllBeforePosition* devolverá la excepción *IndexOutOfBoundsException*.

Si se hace uso de algún método auxiliar debe ser implementado junto a *addAllBeforePosition*.

Importante: El coste de la solución ha de ser lineal.

Ejercicio 15

Implementa el método *addAtFirstPosition(Collection<? extends E> elements)* que **inserta** los elementos de la **colección** *elements* al inicio de un ArrayList<E>. Los elementos de *elements* mantendrán dentro del *ArrayList<E>* el mismo orden que tenían en *elements*. Además, los elementos que había en el *ArrayList<E>* no son eliminados.

Para acceder a los elementos de *elements* se exige el uso de un iterador.

La implementación se hará **dentro** de la clase *ArrayList<E>*, en la cual, los elementos se encuentran almacenados en un array de *Objects* (siendo el primer elemento de la lista el que ocupa la posición 0, el segundo el que ocupa la 1, etc.). Además, el número de elementos en la lista viene indicado por *size*.

```
public class ArrayList<E>{
    private Object[] theArray;
    private int size;
    //. . .

    public void addAtFirstPosition(
            Collection<? extends E> elements){
        //. . .
    }
    //. . .
}
```

Es **importante** tener en cuenta que, si no caben los elementos que queremos insertar con *addAtFirstPosition*, el array que almacena los elementos debe ser redimensionado para que quepan.

Si se hace uso de algún método auxiliar debe ser implementado junto a *addAtFirstPosition*.

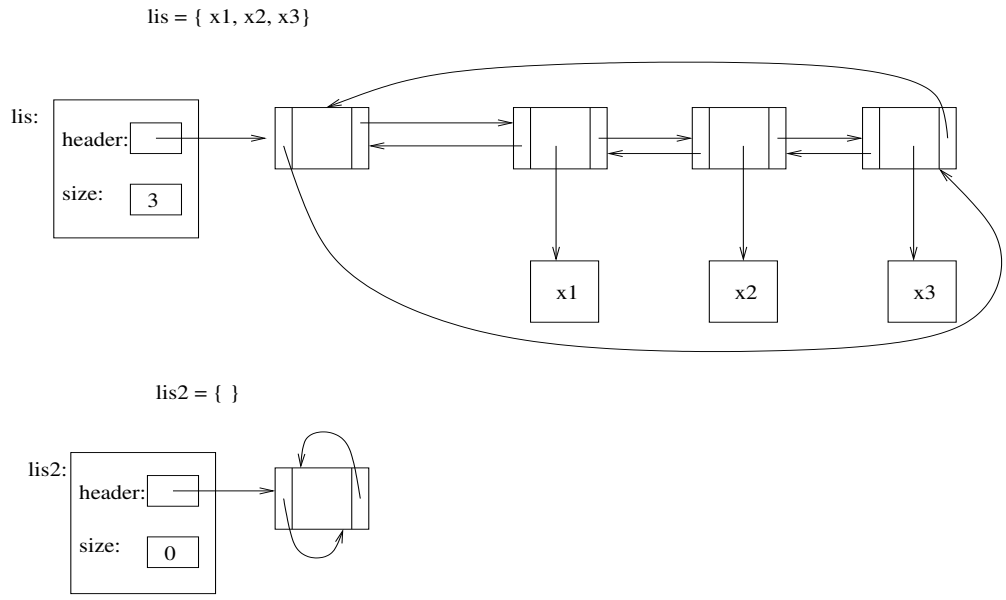
Importante: El coste de la solución ha de ser lineal.

Ejercicio 16

Al Tema 3, hem vist la LinkedList<E>, que està definida a la versió JAVA SE 6 de la següent manera:

```
public class LinkedList<E> extends AbstractSequentialList<E>
implements List<E>...{
    private Node<E> header = new Node<E>(null, null, null);
    private int size = 0;
```

```
    public LinkedList(){
        header.next = header.previous = header;
    }

    private static class Node<E>{
        E element;
        Node<E> next;
        Node<E> previous;
        Node (E e, Node<E> next, Node<E> previous){
            element = e;
            this.next = next;
            this.previous = previous;
        }
    }
}
```



Us demanem que implementeu les següents operacions:

- (0.5p) **`public E get First()`**

Retorna el primer element de la llista.

- (0.5p) **`public E getLast()`**

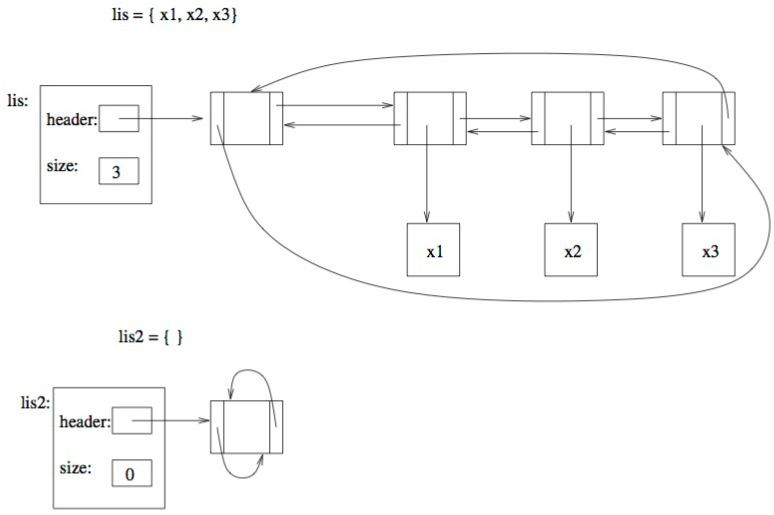
Retorna l'últim element de la llista.

- (1p) **`public void addFirst(E element)`**

Inserta element al principi de la llista. És a dir, el node amb E element passarà a ser el primer node de la llista. Podeu fer ús del mètode que hem vist a teoria: `addBefore (E e, Node<E> entry)`.

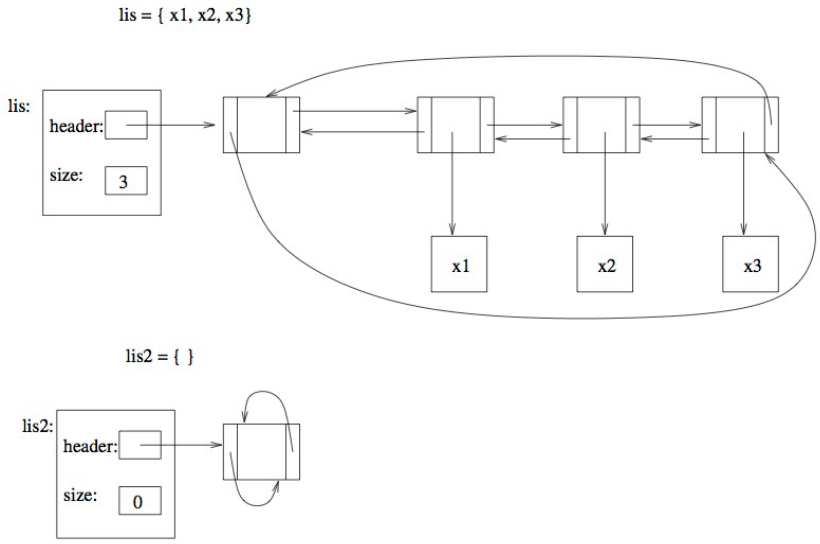
Ejercicio 17

En base a la implementación vista en clase de *LinkedList*<E>, implementad las operaciones *removeFirst()* y *removeLast()*. Ambas operaciones devuelven el elemento eliminado y lanzan *NoSuchElementException* si éste no existe.



Ejercicio 18

En base a la implementación vista en clase de *LinkedList*<*E*>, implementa las operaciones *insertFirst()* y *removeLast()*, esta última devuelve el elemento eliminado y lanza *NoSuchElementException* si éste no existe.



Ejercicio 19

Queremos realizar una implementación de la interfaz *Stack*<*E*>, definida como:

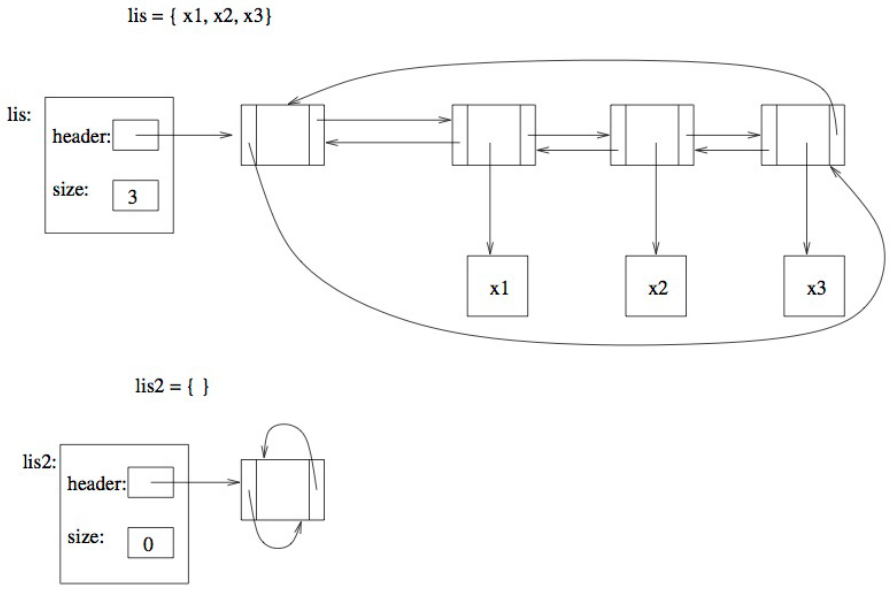
```
public Interface Stack<E> {
      void push(E elem);
      void pop();
```

```
        E top();
        boolean isEmpty();
}
```

La implementación que se os pide se basará en la que se ha presentado en clase para las *LinkedList*, es decir, con una estructura de nodos enlazados. Sin embargo, en este caso, cada nodo solamente tendrá un apuntador al nodo siguiente. Las operaciones *top* y *pop* lanzan *NoSuchElementException* y *IllegalStateExcepction* si la pila está vacía, respectivamente.

Ejercicio 20

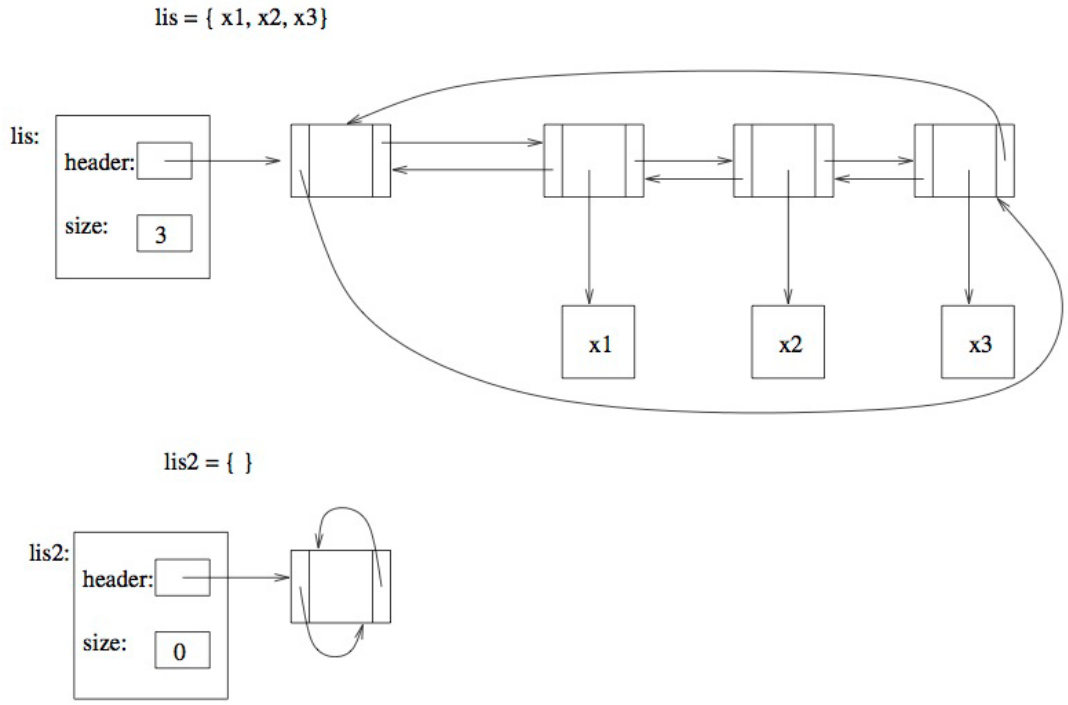
En base a la implementación vista en clase de *LinkedList*<E> (ver Figura), implementa la operación *addAfter(E e, Entry<E> entry)*. Esta operación inserta el elemento *e* después del Nodo referenciado por *entry*.



Ejemplo de la representación de una LinkedList<E>

Ejercicio 21

En base a la implementación vista en clase de *LinkedList<E>* (ver figura), implementa (dentro de la clase *LinkedList<E>*) las operaciones *removeFirst* y *addLast*, que eliminan el primer elemento del *LinkedList<E>* y añaden un elemento al final del *LinkedList<E>*, respectivamente. En el caso de *removeFirst,* esta operación devuelve el elemento eliminado y lanza *NoSuchElementException* si éste no existe.



LinkedList<E> para la Pregunta 13

Ejercicio 22

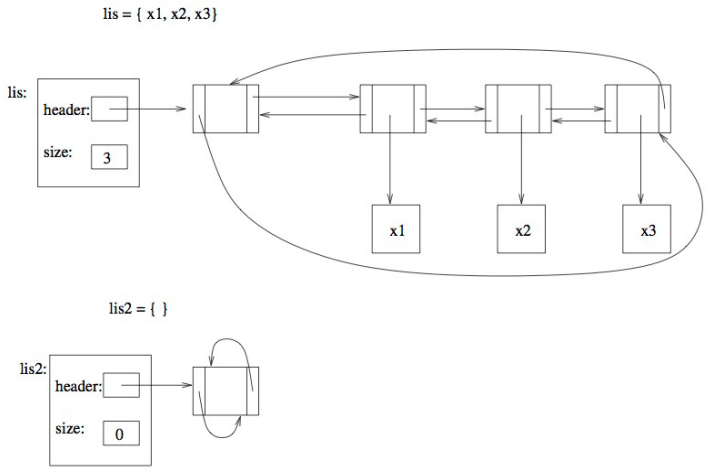
En base a la implementación vista en clase de *LinkedList*<E> (ver figura), implementa las siguientes operaciones:

- *addAfter(E e, Entry<E> entry):* operación que inserta el elemento *e* después del Nodo referenciado por *entry*.
- *addBefore(E e, Entry<E> entry*): operación que inserta el elemento *e* antes del Nodo referenciado por *entry*.

```
public class LinkedList<E> {

    private Entry<E> header;
    private int size;
    . . .
    private static class Entry<E> {
        Entry<E> previous;
        E element;
        Entry<E> next;
        Entry(Entry<E> previous, E element, Entry<E> next) {
            this.previous = previous;
            this.element = element;
            this.next = next;
        }
    }
    . . .

    public void addAfter(E element, Entry<E> reference) {
        . . .
    }
```

```
    public void addBefore(E element, Entry<E> reference) {
        . . .
    }
    . . .
}
```

Si es necesario hacer uso de algún método auxiliar, deberá ser implementado junto a *addAfter y addBefore*.



Ejemplo de la representación de una LinkedList<E>

Ejercicio 23

En base a la implementación vista en clase de *LinkedList*<E>, implementa las siguientes operaciones:

- *void linkLast(E e):* operación que inserta el elemento *e* como el último elemento del *LinkedList*<*E*>.
- *E unlinkFirst()*: operación que elimina el primer elemento del *LinkedList*<*E*> el cual será devuelto. Si la lista está vacía lanza la excepción *NoSuchElementException*.

Si fuera necesario hacer uso de algún método auxiliar, deberá ser implementado junto a *linkLast* y *unlinkFirst*. Además, deberá considerarse la variable *modcount* como se ha hecho en los apuntes.

```
public class LinkedList<E> {

    private Node<E> first = null;
    private Node<E> last  = null;
    private int size      = 0;
    private int modCount  = 0;

    // . . .


    private static class Node<E> {
        E item;
        Node<E> next;
```

```
        Node<E> prev;
        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }

    // . . .

    public void linkLast(E e) {

        // . . .

    }



    public E unlinkFirst() {

       // . . .

    }

    // . . .

}
```

Ejercicio 24

En base a la implementación vista en clase de *LinkedList*<E>, implementa las siguientes operaciones:

- *void linkBeforeLast(E e):* operación que inserta el elemento *e* antes del último elemento del *LinkedList<E>*.
- *void linkAfterFirst(E e):* operación que inserta el elemento *e* después del primer elemento del *LinkedList<E>*.

Si fuera necesario hacer uso de algún método auxiliar, deberá ser implementado junto a *linkBeforeLast* y *linkAfterFirst*. Además, deberá considerarse la variable *modcount* como se ha hecho en los apuntes.

En caso de que la lista esté vacía, ambas operaciones lanzan *IllegalStateException*.

```
public class LinkedList<E> {

    private Node<E> first = null;
    private Node<E> last  = null;
    private int size      = 0;
    private int modCount  = 0;

    // . . .
```

```
    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;
        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }

    // . . .

    public void linkBeforeLast(E e) {

        // . . .

    }

    public void linkAfterFirst(E e) {

       // . . .

    }

    // . . .

}
```

Ejercicio 25

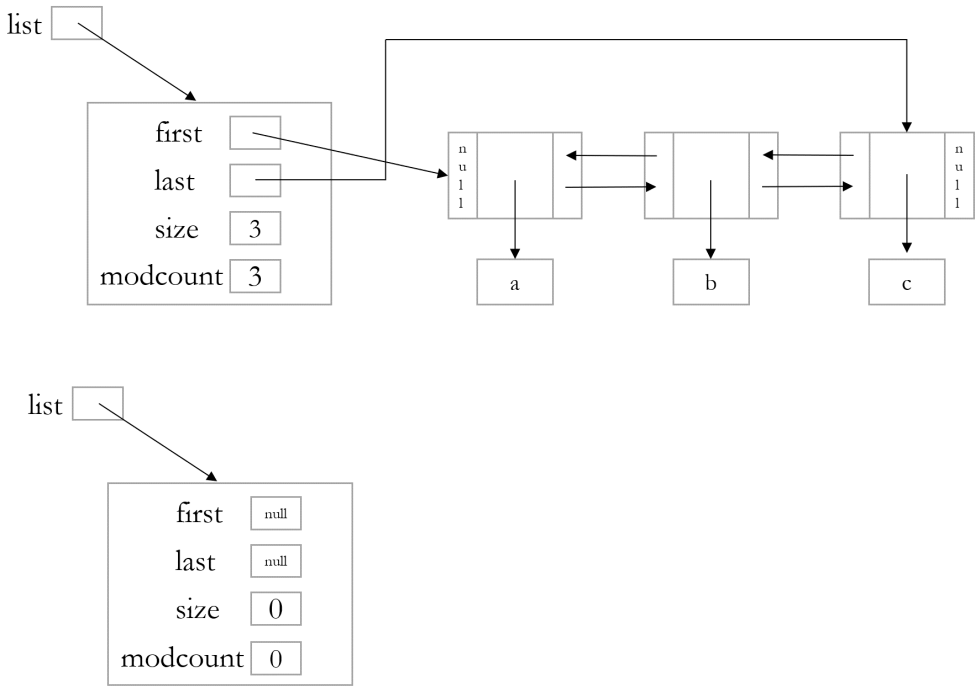
En base a la implementación vista en clase de *LinkedList*<E> (ver diagrama adjunto), implementa la siguiente operación:

- *void linkAfter(E e, Node<E> pred):* operación que inserta el elemento *e* después del nodo no *null* del *LinkedList<E>* llamado *pred.*

Si fuera necesario hacer uso de algún método auxiliar, deberá ser implementado junto a *linkAfter*. Además, deberá considerarse la variable *modCount* como se ha hecho en los apuntes.

```
public class LinkedList<E> {

    private Node<E> first = null;
    private Node<E> last  = null;
    private int size      = 0;
    private int modCount  = 0;

    // . . .

    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;
        Node(Node<E> prev, E element, Node<E> next) {
```

```
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }

    // . . .

    private void linkAfter(E e, Node<E> pred) {

       // . . .

    }

    // . . .

}
```



Ejemplo de la representación de una LinkedList<E>

Ejercicio 26

Implementa el método *addAfter* que inserta el elemento pasado como parámetro dentro de una lista en la posición posterior a otra indicada. La posición también debe ser pasada como parámetro. La posición pasada como parámetro siempre será válida.

En concreto, se piden dos implementaciones. Una dentro de la clase *LinkedList<E>* y otra dentro de *ArrayList<E>*. En aquel caso en el que sea necesario redimensionar, se hará al doble del tamaño existente en el momento de la redimensión.

En *ArrayList<E>*, las posiciones válidas que puede recibir el método *addAfter* podrán ser desde *-1* hasta *size – 1* (índice del último elemento) ambas incluídas. ¿Por qué *-1*? Porque servirá para colocar un elemento en la primera posición. En el caso de un *LinkedList<E>*, para colocar un elemento en la primera posición el *Entry<E>* que se recibirá como parámetro será *header*.

Si se hace uso de algún método auxiliar debe ser implementado junto a *addAfter*.

```
public class ArrayList<E>{

    private Object[] theArray;
    private int size;
    . . .
    public void addAfter(E element, int position){
        . . .
    }
    . . .
}

public class LinkedList<E>{

    private Entry<E> header;
    private int size;
    . . .
    private static class Entry<E> {
        Entry<E> previous;
        E element;
        Entry<E> next;
        Entry(Entry<E> previous, E element, Entry<E> next){
            this.previous = previous;
            this.element = element;
            this.next = next;
        }
    }
    . . .
    public void addAfter(E element, Entry<E> pred){
        . . .
    }
    . . .
}
```