

# Laboratorio 8: Estructuras de Datos Lineales

## Programación 2

Ángel Herranz  
aherranz@fi.upm.es

Universidad Politécnica de Madrid

2023-2024

Este laboratorio va a girar alrededor del mundo de las **estructuras de datos lineales** y de las **pruebas unitarias**.

**Se pide:** Tendrás que entregar tres ficheros:


- `NodoStr.java`: Una clase para crear cadenas simplemente enlazadas de *strings*.
- `LinkedListStr.java`: Una implementación del tipo abstracto de datos de listas utilizando cadenas simplemente enlazadas.
- `ArrayListStr.java`: Una implementación del tipo abstracto de datos de listas utilizando *arrays* redimensionables.


Para poder probar tus implementaciones tienes a tu disposición dos clases: `NodoStr` y `TestListStr`. En concreto para probar cada implementación tienes que modificar el código fuente de `TestListStr.java` para usar tu clase `LinkedListStr` o la clase `ArrayListStr`.


```
C:\Users\UPM\lab02> javac TestListStr.java
```

```
C:\Users\UPM\lab02> java -ea TestListStr
```


**Nota:** No olvides utilizar el *flag* `-ea` al ejecutar los tests.

 **Ejercicio 1.** Ten a mano las transparencias de las sesiones 11 (TADs)), 14, 15 (estructura de datos de cadenas simplemente enlazadas), 16, 17 (tipo abstracto de datos de listas) y 22 (estructura de datos de *arrays redimensionables*). Antes de empezar el laboratorio el profesor hará una mini-presentación de las mismas.

 **Ejercicio 2.** Ya hemos trabajado con el concepto de TAD (Tipo Abstracto de Datos), *ADT* (*Abstract Data Type*) en inglés. Básicamente, un TAD es un nombre, el nombre del tipo, un API, su conjunto de operaciones públicas (métodos públicos en Java) y, muy importante, su semántica, es decir, qué hace cada uno de los métodos que forman parte de su API.

 **Ejercicio 3.** Para poder implementar un TAD es necesario decidir cómo lo vamos a hacer internamente, es decir, qué atributos y métodos extra internos, privados, hacen falta para que las operaciones públicas del tipo se comporten de acuerdo a su semántica.

Esa *implementación interna* se conoce como la *estructura de datos* que implementa el TAD. Por ejemplo, en el caso del TAD Punto2D, puedes optar por usar como estructura de datos interna las coordenadas cartesianas, o las coordenadas polares. La implementación cambia pero el TAD es el mismo y los métodos públicos se comportan (semántica) igual.

 **Ejercicio 4.** En este laboratorio vamos a enfrentarnos a un TAD que todo el mundo llama **listas**. El API que vamos a dar a dicho TAD es similar al propio del lenguaje de programación Java:

```
/**
 * Inserta un elemento en la lista en una posición determinada.
 *
 * @param index el lugar donde se va a insertar
 * @param elem el nuevo elemento que se va a insertar
 * @pre. {@code 0 <= index && index <= size()} y la lista no puede estar llena
 * @post. coloca {@code element} en {@code index} y desplaza desde
 * esa posición el resto de los elementos una posición a la derecha
 * @throws IndexOutOfBoundsException si el índice no está entre los
 * límites indicados (depende de la implementación)
 */
void add (int index, String elem);

/**
 * Inserta un elemento al final de la lista.
 *
 * @param elem el nuevo elemento que se va a insertar
 * @pre. la lista no puede estar llena
 * @post. coloca {@code element} como último elemento de la lista
 */
void add (String elem);

/**
 * Devuelve el elemento que está en la posición {@code index}.
 *
 * @param index el índice del elemento que se va a devolver (empezando en cero)
```

```

    * @pre. {@code 0 <= index && index < size()}
    * @post. no modifica la lista
    * @return el elemento en la posición dada por index
    * @throws IndexOutOfBoundsException si el índice no está entre los
    * límites indicados (depende de la implementación)
    */
    String get (int index);

    /**
     * Devuelve la longitud de la lista.
     *
     * @post. no modifica la lista
     * @return el número de elementos actualmente en la lista
     */
    int size ();

    /**
     * Pone {@code elem} en la posición {@code index} de la lista,
     * "machacando" el elemento que había en esa posición.
     *
     * @param index el lugar donde se va a colocar el número elemento
     * @param elem el nuevo elemento de la lista
     * @pre. {@code 0 <= index && index < size()}
     * @post. {@code get(index) == elem} y el resto de los índices
     * quedan como estaban
     * @throws IndexOutOfBoundsException si el índice no está entre los
     * límites indicados (depende de la implementación)
     */
    void set (int index, String elem);

    /**
     * Busca un elemento en la lista.
     *
     * @param elem el elemento a buscar en la lista
     * @post. no modifica la lista
     * @return el índice que ocupa (empezando en cero) la primera
     * ocurrencia de {@code elem} en la lista, -1 si no existe
     */
    int indexOf (String elem);

    /**
     * Quita de la lista el elemento que está en la posición {@code index}.
     *
     * @param index el índice del elemento que se va a eliminar
     * @pre. {@code 0 <= index && index < size()}
     * @post. el tamaño de la lista se ha decrementado en uno
     * @throws IndexOutOfBoundsException si el índice no está entre los

```

```

    * límites indicados (depende de la implementación)
    */
    void remove (int index);

    /**
     * Quita la primera ocurrencia de {@code elem} de la lista. Si no existe,
     * no hace nada.
     *
     * @post. Si existe el tamaño de la lista lista se decrementa en uno
     * @param elem el elemento a eliminar
     */
    void remove (String elem);

    /**
     * Devuelve una representación de la lista. En el string devuelto,
     * <b>el elemento de la lista, el que ocupa la posición 0, debería
     * ser el primer elemento en aparecer</b>.
     */
    String toString();

    /**
     * Devuelve true si <code>o</code> y esta lista son iguales, false en otro
     * caso.
     */
    boolean equals(Object o);

```

- 📖 **Ejercicio 5.** Te toca implementar el TAD de listas utilizando como estructura de datos una cadena simplemente enlazada. El primer paso es crear una primera versión vacía pero que compile de la clase `LinkedList.java`:

```

public class LinkedListStr {
    private NodoStr cadena;

    ...
}

```

- 📖 **Ejercicio 6.** Aplica TDD (*Test Driven Design*) para ir implementando la clase `LinkedList` a medida que vas resolviendo los errores que revelan las pruebas `TestListStr`. Tu objetivo es que tu implementación pase todas las pruebas.

- 📖 **Ejercicio 7.** El TAD de listas es un TAD. Eso significa que podemos implementar el mismo TAD con estructuras de datos diferentes. En nuestro caso, podemos implementar el TAD de las listas utilizando algo diferente a una cadena simplemente enlazada. La propuesta es utilizar *arrays*.

Te toca implementar la clase `ArrayListStr` que es el mismo TAD que `LinkedListStr`, es decir, tiene el mismo API y la misma semántica pero cuya estructura de datos es un **array redimensionable**.

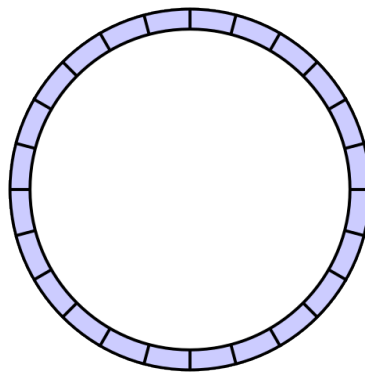
La estructura de datos de *arrays* redimensionables permite representar una colección usando un *array*,

1. Cuando se añade un elemento es necesario crear un nuevo *array* de tamaño uno más, mover los datos al nuevo *array* y añadir el nuevo elemento.
2. Cuando se borra un elemento es necesario crear un nuevo *array* de tamaño uno menos, y mover los datos no borrados al nuevo *array*.

📄 **Ejercicio 8.** Aplica TDD (*Test Driven Design*) para ir implementando la clase `ArrayListStr` a medida que vas resolviendo los errores que revelan las pruebas `TestListStr`<sup>1</sup>. Tu objetivo es que tu implementación pase todas las pruebas.

📖 **Ejercicio 9.** La estructura de datos de *arrays* redimensionables tal y como se describe en el ejercicio 7 es extraordinariamente ineficiente.

Para tener una implementación eficiente te sugiero que uses las ideas de una nueva estructura de datos: **buffer circular**. Para ello puedes visitar el artículo en Wikipedia: [https://es.wikipedia.org/wiki/Buffer\\_circular](https://es.wikipedia.org/wiki/Buffer_circular).



📄 **Ejercicio 10.** Modifica la estructura de datos usada en `ArrayListStr` para que la implementación con *arrays* redimensionables utilizando las ideas de un *buffer* circular.

---

<sup>1</sup>Tendrás que cambiar las líneas que referencian a `LinkedListStr` para que referencien a `ArrayListStr`