

Java Standard Web Programming

Módulo 5



Colecciones


Introducción

Java permite, a través de los arreglos, almacenar “agrupar” muchos elementos bajo un mismo identificador. El problema es que se debe conocer de antemano el tamaño que tendrá el arreglo y este no varía. No es posible añadir elementos nuevos ni eliminarlos. En el mejor de los casos, se puede dejar el valor por defecto del dato primitivo y de ser un Objeto dejarlo en **null**.

Además de los arreglos convencionales que ya conocemos, Java proporciona un conjunto de **interfaces y clases** (*Collections Framework*) **para agrupar elementos de forma dinámica**.

Se pueden agregar más Objetos sin indicar un tamaño único o inicial; también se reduce el esfuerzo de programación al tiempo que aumenta el rendimiento.

Las colecciones son Interfaces y Clases que tienen **parámetros (Genéricas)** por lo que debemos indicarles el **tipo de Objeto a almacenar**.



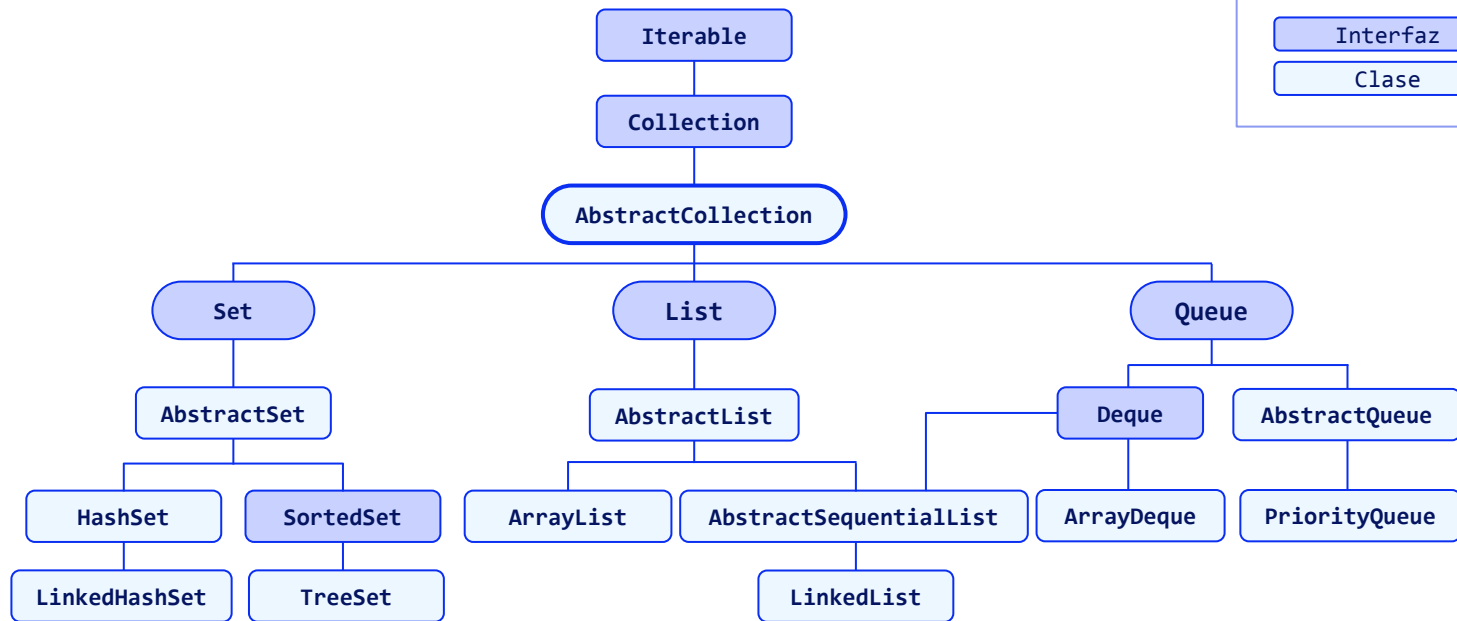
Jerarquía

Estas clases e interfaces están estructuradas en una jerarquía.

A medida que se va **descendiendo a niveles más específicos aumentan los requerimientos y comportamientos**.



Colecciones



Referencias

Interfaz

Clase

Métodos Collection

Tipo	Método	Descripción
boolean	<code>add(E e)</code>	Asegura que esta colección contiene el elemento especificado (operación opcional).
boolean	<code>addAll(Collection<? extends E> c)</code>	Agrega todos los elementos de la colección especificada a esta colección.
void	<code>clear()</code>	Elimina todos los elementos de esta colección (operación opcional).
boolean	<code>contains(Object o)</code>	Devuelve verdadero si esta colección contiene el elemento especificado.
boolean	<code>containsAll(Collection<?> c)</code>	Devuelve verdadero si esta colección contiene todos los elementos de la colección especificada.
boolean	<code>equals(Object o)</code>	Compara el objeto especificado con esta colección para la igualdad.
int	<code>hashCode()</code>	Devuelve el valor del código hash para esta colección.

Más métodos en la siguiente pantalla.

Métodos Collection

Tipo	Método	Descripción
boolean	<code>isEmpty()</code>	Devuelve verdadero si esta colección no contiene elementos.
Iterator<E>	<code>iterator()</code>	Devuelve un iterador sobre los elementos de esta colección.
boolean	<code>remove(Object o)</code>	Elimina una sola instancia del elemento especificado de esta colección, si está existe
boolean	<code>removeAll(Collection<?> c)</code>	Elimina de este conjunto todos sus elementos que están contenidos en la colección especificada.
Int	<code>size()</code>	Devuelve el número de elementos en esta colección.
Object[]	<code>toArray()</code>	Devuelve una matriz que contiene todos los elementos de este conjunto.
<T> T[]	<code>toArray(T[] a)</code>	Devuelve una matriz que contiene todos los elementos de este conjunto; el tipo de tiempo de ejecución de la matriz devuelta es el de la matriz especificada.

Ejemplos

```
E elemento1, elemento2, elemento3, elemento4, elemento5;

Coleccion<E> coleccionPrincipal = new Implementacion<E>();
Coleccion<E> coleccionA = new Implementacion<E>();
Coleccion<E> coleccionB = new Implementacion<E>();

// Agregar elementos
coleccionA.add(elemento1);
coleccionA.add(elemento2);
coleccionB.add(elemento3);
coleccionB.add(elemento4);

// Agregar una Coleccion en Otra
coleccionPrincipal.addAll(coleccionA);
coleccionPrincipal.addAll(coleccionB);

// verificar que contenga el elemento
System.out.println(coleccionPrincipal.contains(elemento1));

// verificar la coleccion contenga todos los elementos de otra coleccion
System.out.println(coleccionPrincipal.containsAll(coleccionA));

// verificar si son iguales dos colecciones
System.out.println(coleccionA.equals(coleccionB));
```

```
// remover un elemento de la coleccion
coleccionPrincipal.remove(elemento4);

// remover los elementos de una coleccion que existan en otra
coleccionPrincipal.removeAll(coleccionA);

// convertir una coleccion en un arreglo convencional
Object[] objetos = coleccionPrincipal.toArray();

E[] elementos;
coleccionPrincipal.toArray(elementos);

//conocer el tamaño de la coleccion
System.out.println(coleccionPrincipal.size());

// recorrer una coleccion
for (E elemento : coleccionPrincipal) {
    System.out.println(elemento);
}

// limpiar la coleccion
coleccionPrincipal.clear();
```


Iteradores

Algo particular de las colecciones es que **no se pueden recorrer y operar al mismo tiempo** (*Eliminar o Actualizar la colección*), para eso la interfaz `Collection` nos proporciona un método que devuelve un **Iterator**.



Tipo	Método	Descripción
boolean	<code>hasNext()</code>	Devuelve true si la iteración tiene más elementos.
E	<code>next()</code>	Devuelve el siguiente elemento en la iteración.
void	<code>remove()</code>	Elimina de la colección subyacente el último elemento devuelto por este iterador.

```
Iterator<E> iterador = coleccionPrincipal.iterator();

while(iterador.hasNext()) {
    E elementoAuxiliar = iterador.next();
    if (condicion) {
        iterador.remove();
    }
}
```

Interfaz Set

Interfaz Set

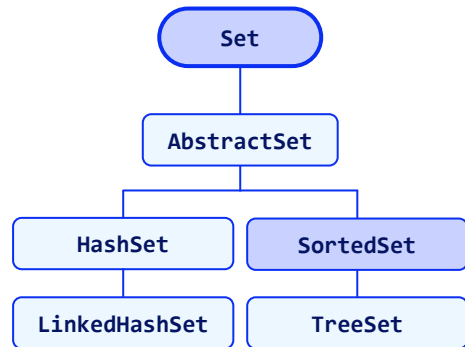
La interfaz **Set** tiene una peculiaridad, y es que **no admite duplicados** y es especialmente útil para ir **almacenando datos sin la preocupación de que alguno se repita**. Estos elementos no se mostrarán de forma ordenada.

La clase AbstractSet

Proporciona una implementación esquelética de la interfaz Set y simplemente agrega implementaciones para los métodos **equals** y **hashCode**.

La interfaz SortedSet

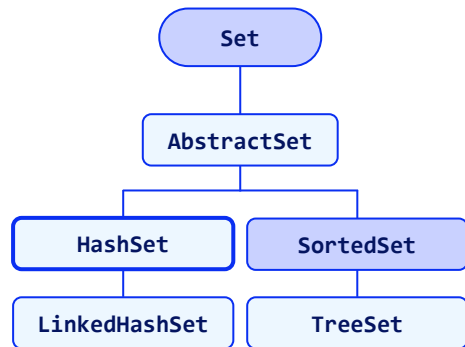
Permite que las clases que la implementen tengan los **elementos ordenados**.



HashSet

Esta implementación **almacena los elementos en una tabla hash** (es un contenedor asociativo “tipo Diccionario” que permite un almacenamiento y posterior recuperación eficiente de elementos). Este acceso hace que la clase sea ideal para **búsqueda, inserción y borrado de elementos**.

Representa un conjunto de **valores únicos** (no puede tener valores duplicados sin ordenar, por ejemplo, si hacemos un recorrido de los objetos dentro de un HashSet no siempre los obtendremos en igual orden) y tiene una **iteración más rápida** que otras colecciones.



```
AbstractSet<String> nombres = new HashSet<>();  
Set<String> nombresA = new HashSet<>();  
HashSet<String> nombresB = new HashSet<>();
```

Equals y hashCode

```
public boolean equals(Object obj)
```

Hemos visto que es posible comparar con el método **equals** una cadena de caracteres. Este método devuelve *true* o *false* según sea el caso.

Ejemplo

```
"Octavio".equals("octavio");
```

Devuelve *false* ya que el método es sensible a las mayúsculas y minúsculas.



Pero, ¿cómo sabe Java que un objeto propio es igual a otro?

Primero que nada, debemos saber que este es un método que heredamos de la clase **Object** y que las clases de Java sobrescriben para que los podamos utilizar. Nosotros debemos hacer lo mismo e indicarle al método cuáles son los atributos que debe comparar para que un objeto sea igual a otro.



```
public int hashCode()
```

También debemos sobrescribir otro método, el **hashCode**, ya que si dos objetos son iguales se debe generar el mismo número hash para poder localizar el elemento en la tabla hash.

Por ejemplo

Podemos decir que *un Auto es igual a otro si poseen la misma Patente*, en otros casos los objetos son iguales si poseen todos los atributos iguales; siempre dependerá del caso de uso.

Los **IDE** nos proporcionan alguna manera de generar de forma automática estos métodos.

Por ejemplo

En *Eclipse* podemos ir a:

Source > Generate hashCode() and equals()

En el panel seleccionamos los atributos que queremos que se tengan en cuenta en estos métodos.

Es importante aclarar que si los atributos a comparar también son objetos propios debemos sobrescribir en esos objetos dichos métodos, de lo contrario la interfaz no detectará que son iguales.

Además, podemos utilizar el método **equals** a partir de ahora cuando lo creamos conveniente.



Ejemplo

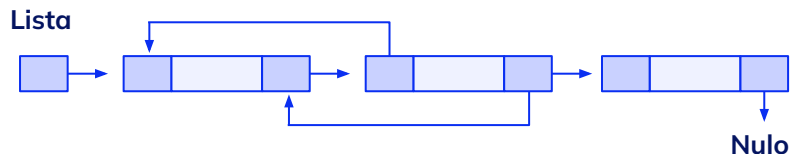
```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((patente == null) ? 0 : patente.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Auto other = (Auto) obj;
    if (patente == null) {
        if (other.patente != null)
            return false;
    } else if (!patente.equals(other.patente))
        return false;
    return true;
}
```

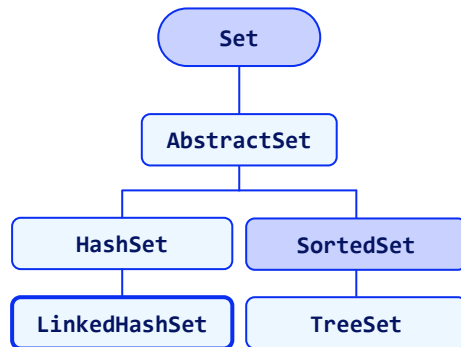


LinkedHashSet

Esta implementación almacena los elementos en función del orden de inserción lo que la hace un poco más costosa que **HashSet**.



Define el concepto de elementos añadiendo una **lista doblemente enlazada** en la ecuación que nos asegura que los elementos siempre se recorren de la misma forma.



```
AbstractSet<String> nombres = new LinkedHashSet<>();  
Set<String> nombresA = new LinkedHashSet<>();  
HashSet<String> nombresB = new LinkedHashSet<>();
```

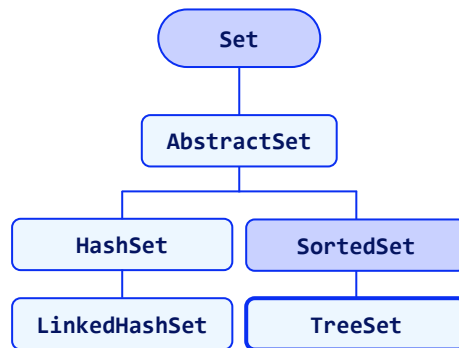
TreeSet

Esta implementación almacena los elementos ordenándolos en función de sus **valores**.

Implementa el **algoritmo del árbol rojo – negro**: árbol de búsqueda binaria, también llamado *árbol binario ordenado* o *árbol binario ordenado*. Por eso, es bastante más lento que **HashSet**.

Este orden podrá ser **Natural** o **Alternativo**.

```
Set<Auto> autos = new TreeSet<>(); // Orden Natural  
Set<Auto> autos = new TreeSet<>(new OrdenAlternativo());
```



```
AbstractSet<String> nombres = new TreeSet<>();  
Set<String> nombresA = new TreeSet<>();  
TreeSet<String> nombresB = new TreeSet<>();
```

Orden

Java proporciona dos interfaces (**Comparable<T>** y **Comparator<T>**) para dar un **orden a los objetos** de un tipo creado por el usuario. El lenguaje ya proporciona un orden natural a los tipos envoltorio como **Integer** y **Double** o al tipo inmutable **String**.

Comparator (orden alternativo)

Esta interfaz debe ser implementada en una nueva clase comparando los dos objetos que toma como parámetros el método **compare**.

Comparable (orden natural)

Esta interfaz debe ser implementada por la clase y compara el objeto **this** con otro que toma como parámetro el método **compareTo**.

El orden natural definido debe ser coherente con la definición de igualdad. Si **equals** devuelve **true** **compareTo** debe devolver cero.

Los métodos **compareTo** y **compare** comparan dos objetos **o1** y **o2** (en **compareTo** **o1** es **this**) y devuelve un entero que es:

- *Negativo* si **o1** es menor que **o2**.
- *Cero* si **o1** es igual a **o2**.
- *Positivo* si **o1** es mayor que **o2**.

Si el atributo a comparar es un **String** podemos apoyarnos en el método **compareTo**, de lo contrario con una resta entre atributos numéricos basta.



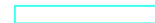
Comparable

```
public class OrdenAutoMarca implements Comparator<Auto> {  
  
    @Override  
    public int compare(Auto auto1, Auto auto2) {  
        return auto1.getMarca().compareTo(auto2.getMarca());  
        // return auto1.puestos - auto2.puestos;  
    }  
  
}
```



Comparator

```
public abstract class Auto implements MantenimientoMecanico, Archivo, Comparable<Auto>{  
  
    @Override  
    public int compareTo(Auto auto) {  
        return this.patente.getNumero().compareTo(auto.getPatente().getNumero());  
        // return this.puestos - auto.puestos;  
    }  
  
}
```

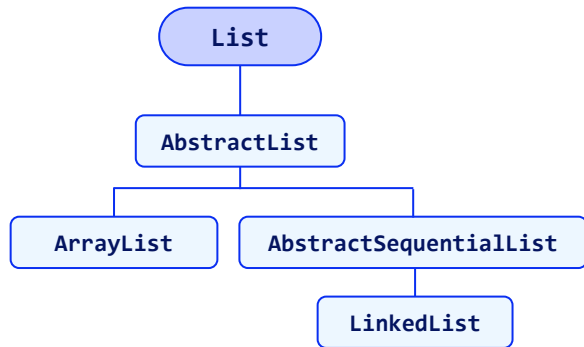


Interfaz List

List

La interfaz **List**, a diferencia de **Set**, sí permite **elementos duplicados** y es la interfaz más usada por nosotros ya que implementa una serie de métodos que la hacen más comprensible y fácil de manejar:

- **Acceso posicional a elementos:** manipula elementos en función de su posición en la lista.
- **Búsqueda de elementos:** busca un elemento concreto de la lista y devuelve su posición.
- **Iteración sobre elementos:** mejora el Iterador por defecto.

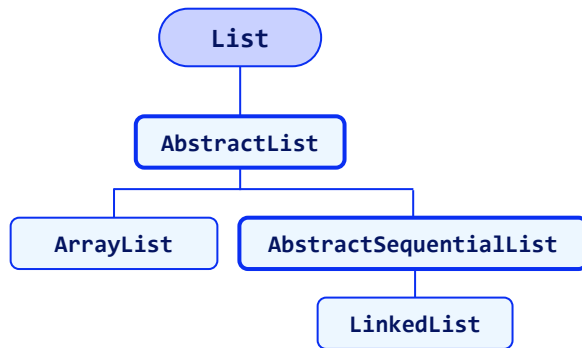


La clase `AbstractList`

Proporciona una implementación esquelética de la interfaz dándole **comportamiento a los nuevos métodos** que utilizaremos.

La clase `AbstractSequentialList`

Proporciona una implementación esquelética de la interfaz dándole **acceso secuencial a la colección**.



Métodos de las listas



```
// se agrega elemento en los indice
coleccion.add(1, elemento1);

// recorrido de lista con for comun
for (int i = 0; i < coleccion.size(); i++) {
    System.out.println(coleccion.get(i));
}

// se reemplaza elemento de la lista
coleccion.set(0, elemento2);

// se remueve elemento en la posicion indicada, devuelve el elemento eliminado
System.out.println(coleccion.remove(2));

// devuelve el indice de la primera aparicion de Ariel
System.out.println(coleccion.indexOf(elemento4));

// devuelve el indice de la ultima aparicion de Ariel
System.out.println(coleccion.lastIndexOf(elemento5));

// creamos una sublista a partir de un rango
List<E> subLista = coleccion.subList(1, 3);

// ordenamos
coleccion.sort(new Comparador());
```

Iteradores de lista

Es un **iterador** Java que nos permite **recorrer una lista de elementos en varias direcciones**, bien como fueron insertados los elementos o en reversa (hacia adelante o hacia atrás), además, de proporcionarnos nuevos métodos sin eliminar los que ya conocemos de los iteradores.

```
// creamos un iterador mejorado
ListIterator<E> iteradorLista = subLista.listIterator();

// recorrer el iterador
while (iteradorLista.hasNext()) {
    E elementoAuxiliar = iteradorLista.next();

    if (elementoAuxiliar.equals(elemento3)) {
        // eliminamos el elemento
        iteradorLista.remove();
    }

    if (elementoAuxiliar.equals(elemento4)) {
        // sustituimos el elemento
        iteradorLista.set(elemento5);
    }
}

// recorrer el iterador en reversa
while (iteradorLista.hasPrevious()) {
    E elementoAuxiliar = iteradorLista.previous();

    if (elementoAuxiliar.equals(elemento5)) {
        // eliminamos el elemento
        elementoAuxiliar.add(elemento4);
    }
}
```

Iteradores de lista

Tipo	Método	Descripción
void	<code>add(E e)</code>	Inserta el elemento especificado en la lista
boolean	<code>hasPrevious()</code>	Devuelve <code>true</code> si este iterador de lista tiene más elementos al recorrer la lista en la dirección inversa.
Int	<code>nextIndex()</code>	Devuelve el índice del elemento que sería devuelto por una llamada posterior a <code>next()</code> .
E	<code>previous()</code>	Devuelve el elemento anterior de la lista y mueve la posición del cursor hacia atrás.
Int	<code>previousIndex()</code>	Devuelve el índice del elemento que sería devuelto por una llamada posterior a <code>previous()</code> .
void	<code>set(E e)</code>	Reemplaza el último elemento devuelto por <code>next()</code> o <code>previous()</code> con el elemento especificado

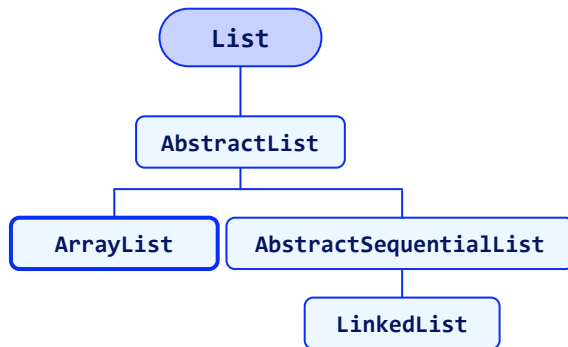
ArrayList

Esta es la implementación más típica de la interfaz.

Se basa en un **array** redimensionable que **aumenta su tamaño según crece la colección de elementos**.

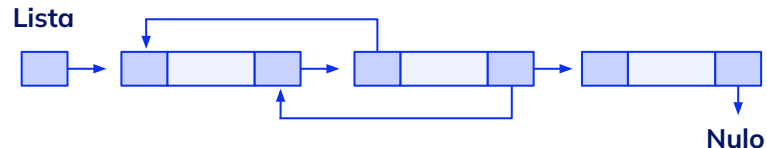
Es la que mejor rendimiento tiene sobre la mayoría de situaciones y como podemos observar se parece mucho a la forma que trabajamos los arreglos comunes con acceso por índice.

```
AbstractList<String> nombres = new ArrayList<>();  
List<String> nombresA = new ArrayList<>();  
ArrayList<String> nombresB = new ArrayList<>();
```

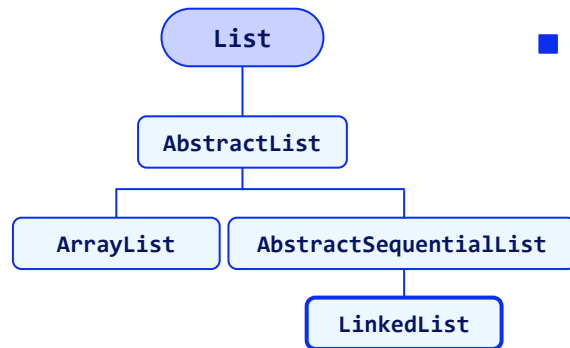


LinkedList

Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.



```
AbstractList<String> nombres = new LinkedList<>();  
List<String> nombresA = new LinkedList<>();  
LinkedList<String> nombresB = new LinkedList<>();
```



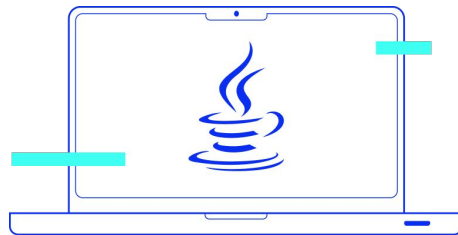
Si estas dos implementaciones recorren la colección como fueron insertados **¿Cuál debo usar?**

LinkedList

Permite **inserciones o eliminaciones en mejor tiempo** utilizando los iteradores, pero solo acceso secuencial de elementos.

ArrayList

Permite un **acceso de lectura aleatorio rápido**, para que puedas obtener cualquier elemento en un tiempo más óptimo. Pero agregar o eliminar desde cualquier lugar menos el final requiere desplazar todos los últimos elementos.



**¡Sigamos
trabajando!**