

C.F.G.S. DESARROLLO DE APLICACIONES WEB

## MODULO DE PROGRAMACIÓN

UNIDAD 8 – PARTE II  
ESTRUCTURAS DE DATOS DINÁMICAS  
COLECCIONES Y MAPAS

# Estructuras de Datos Dinámicas

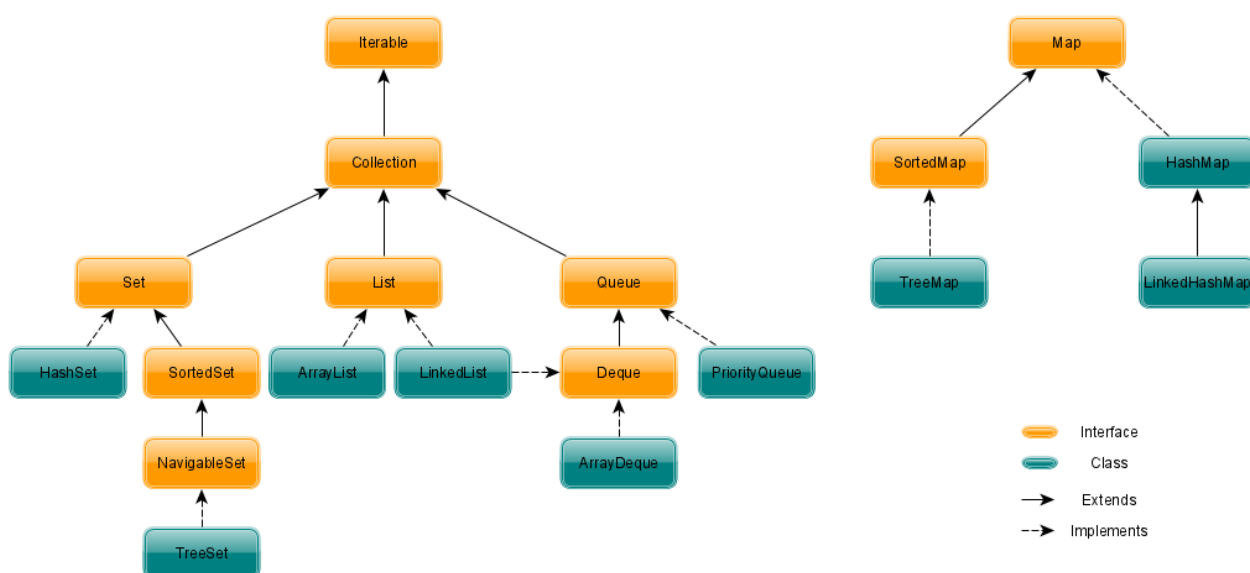
## Colecciones y Mapas

1. INTRODUCCIÓN .....	2
2. COLECCIONES .....	3
2.1. COLECCIONES DE DATOS Y TIPOS DE DATOS PRIMITIVOS .....	4
2.2. ITERAR SOBRE UNA COLECCIÓN .....	4
2.2.1. <i>Iteradores</i> .....	4
2.2.2. <i>Bucle for mejorado</i> .....	5
2.2.3. <i>Método forEach</i> .....	5
2.3. LISTAS .....	6
2.3.1. <i>ArrayList</i> .....	7
2.3.2. <i>LinkedList</i> .....	7
2.3.3. <i>Iterar sobre listas</i> .....	8
2.3.4. <i>Ejemplo de un problema resueltos con listas</i> .....	8
2.4. COLAS Y PILAS .....	10
2.4.1. <i>Colas</i> .....	10
2.4.2. <i>Pilas</i> .....	11
2.4.3. <i>Ejemplos de problemas resueltos con colas y pilas</i> .....	12
2.5. CONJUNTOS .....	15
2.5.1. <i>Ejemplos de problemas resueltos con conjuntos</i> .....	15
3. MAPAS .....	18
3.1. ITERAR SOBRE MAPAS .....	19
3.2. EJEMPLOS DE PROBLEMAS RESUELTOS CON MAPAS .....	20

## 1. Introducción

Cuando se usan de estructuras de datos estáticas, su tamaño se establece en el momento de su creación y no puede variar en tiempo de ejecución. Esto hace que no resulten adecuadas cuando se desconoce a priori la cantidad máxima de elementos que van a almacenar. En ese caso es preferible el uso de estructuras de datos dinámicas, ya que su tamaño varía en tiempo de ejecución a medida que se van realizando inserciones y borrados. A lo largo de la historia de la computación se han estudiado y desarrollado distintos tipos de estructuras de datos dinámicas, como listas, pilas, colas, tablas hash o diferentes tipos de árboles, todas ellas de gran utilidad a la hora de solucionar de forma eficiente una gran variedad de problemas.

En la actualidad, los lenguajes de programación incluyen en sus APIs la implementación de varias estructuras de datos dinámicas que facilitan el desarrollo de aplicaciones de calidad. Dentro del API de Java se encuentra el [Java Collections Framework](#), que define una amplia variedad de clases para ser usadas como contenedores de objetos, implementando cada una de ellas una determinada estructura de datos dinámica. En este *framework* se definen un conjunto de interfaces y clases en dos jerarquías independientes para representar dos tipos de contenedores de objetos, las colecciones y los mapas<sup>1</sup>:



A partir de la versión 5 de Java, las interfaces y clases de estas dos jerarquías se definen mediante tipos genéricos. Por tanto, cualquier referencia a una colección se declara mediante la clase correspondiente, parametrizada con el tipo de elementos que van a almacenar en ella. Por ejemplo, la sentencia siguiente declara una referencia a una lista enlazada en la que se van a almacenar cadenas de caracteres:

```
LinkedList<String> lista;
```

Diagrama de anotación: Una línea horizontal divide el código. Una flecha azul apunta desde el texto "Parámetro tipo" al tipo genérico <String>. Otra flecha azul apunta desde el texto "Tipo parametrizado" al nombre de la clase LinkedList.

<sup>1</sup> Aunque los mapas no están en la misma jerarquía que las colecciones (no implementan la interfaz `Collection`), pertenecen al mismo framework debido a que desempeñan una función similar a éstas como contenedores de objetos. En el apartado de mapas se explica la razón por la cual se definen en su propia jerarquía de clases.

## 2. Colecciones

En la raíz de la jerarquía de clases del *Java Collections Framework* está la interface **Collection**, que define las operaciones básicas para cualquier colección con independencia de los detalles de su implementación. La tabla siguiente recoge algunos de los métodos que en ella se declaran:

Método	Descripción
<code>add</code>	Añade un objeto a la colección.
<code>remove</code>	Elimina un objeto de la colección.
<code>size</code>	Retorna el número de objetos que almacena la colección.
<code>clear</code>	Elimina todos los objetos de la colección.
<code>contains</code>	Comprueba si un objeto está en la colección.
<code>isEmpty</code>	Comprueba si una colección está vacía.

Dado que todas las colecciones implementan la interface **Collection**, es posible manejar cualquiera de ellas mediante estas operaciones básicas. En el ejemplo siguiente se muestra cómo se manipulan dos tipos distintos de colecciones con una única referencia polimórfica, realizando con cada una de ellas las mismas operaciones básicas:

```
Collection<String> c;

c = new ArrayList<String>();
c.add("Carmen");
c.add("Fernando");
System.out.println("La lista contiene a Roberto: " + c.contains("Roberto"));
c.remove("Fernando");
System.out.println("Número de nombres en la lista: " + c.size());
c.clear();

c = new HashSet<String>();
c.add("Elisa");
c.add("Roberto");
System.out.println("El conjunto contiene a Roberto: " +
c.contains("Roberto"));
c.remove("Roberto");
System.out.println("Numero de nombres en el conjunto: " + c.size());
c.clear();
```

Sin embargo, si fuese necesario acceder desde una referencia polimórfica a la funcionalidad definida en alguna subclase, habría que convertir dicha referencia al tipo adecuado. Por ejemplo, en la interface **List** se sobrecarga el método **get** para poder obtener cualquier elemento a partir de un índice. Así pues, para invocar a este método con la referencia polimórfica declarada en el ejemplo anterior, tendríamos que hacerlo de la forma siguiente:

```
Collection<String> l = new ArrayList<String>();
l.add("Carmen");
l.add("Fernando");
System.out.println(((List<String>) c).get(1));
```

Obviamente, si declaramos las referencias usando tipos más específicos, no será necesaria la conversión:

```
List<String> c = new ArrayList<String>();
c.add("Carmen");
c.add("Fernando");
System.out.println(c.get(1));
```

## 2.1. Colecciones de datos y tipos de datos primitivos

Las colecciones son contenedores de objetos. Por tanto, para poder almacenar en ellas datos de tipo primitivo es necesario recurrir a clases envoltorio como `Integer`, `Float`, etc. En el ejemplo siguiente se crea una lista para almacenar números enteros y se añade un número a la lista:

```
List<Integer> lista = new ArrayList<Integer>();
lista.add(100);
```

El compilador sustituye la expresión `lista.add(100)` por la expresión `lista.add(Integer.valueOf(100))` en un proceso denominado *boxing* que se realiza de forma transparente. Esto ocurre de forma automática en cualquier contexto en el que se espera un objeto de una clase envoltorio, pero en su lugar se especifica un dato de tipo primitivo.

El proceso inverso se denomina *unboxing* y ocurre en cualquier contexto en el que se espera un tipo de dato primitivo, pero en su lugar se especifica una referencia a un objeto del tipo envoltorio correspondiente. Por ejemplo, el compilador sustituye de forma transparente la expresión `int valor = lista.get(0)` por la expresión `int valor = lista.get(0).intValue()`.

## 2.2. Iterar sobre una colección

Al iterar sobre una colección se recorren en un orden determinado todos los objetos almacenados en ella. El orden de iteración depende del tipo de colección, por ejemplo, en una lista los elementos se recorren en el mismo orden en que fueron insertados. En cualquier caso, el proceso de iteración se puede hacer de varias formas:

- Con iteradores, disponibles a partir de la versión 1.2 de Java en respuesta a las limitaciones que presenta el bucle `for` clásico a la hora de recorrer cualquier tipo de colección.
- Utilizando la versión mejorada del bucle `for`, disponible a partir de la versión 5 de Java. Utiliza una sintaxis específica para iterar tanto sobre colecciones como sobre arrays. De hecho, puede iterar sobre cualquier objeto que implemente la interface `Iterable`.
- Las dos anteriores representan formas de iteración externa. Con la llegada de la versión 8 de Java y la incorporación al lenguaje de elementos de la programación funcional, se hace posible una nueva forma de iteración denominada iteración interna. Se basa en la utilización de expresiones lambda junto con el nuevo método `forEach` declarado en la interface `Iterable` implementada por todas las colecciones.

### 2.2.1. Iteradores

La interface `Collection` declara el método `iterator` para que cualquier tipo de colección retorne un objeto `Iterator` con el que iterar sobre los elementos que contiene. `Iterator` es una interface que declara los tres métodos siguientes:

Método	Descripción
<code>boolean hasNext()</code>	Retorna <code>true</code> si quedan elementos sobre los que aún no se ha iterado y <code>false</code> en caso contrario.
<code>E next()</code>	Retorna el siguiente objeto de tipo <code>E</code> en el orden de iteración, siendo <code>E</code> el tipo de los objetos almacenados en la colección subyacente.
<code>void remove()</code>	Elimina de la colección subyacente el objeto retornado en la última llamada al método <code>next</code> .

En el ejemplo siguiente muestra como iterar sobre una colección con un iterador:

```
List<String> alumnos = new ArrayList<String>();
alumnos.add("Carmen");
alumnos.add("Fernando");
alumnos.add("Elisa");
alumnos.add("Roberto");
Iterator<String> i = alumnos.iterator();
while (i.hasNext()) {
    String nombre = i.next();
    System.out.println(nombre);
}
```

Cualquier modificación del contenido de una colección mientras se está iterando provoca el lanzamiento de la excepción [ConcurrentModificationException](#). Ejemplo:

```
Iterator<String> i = alumnos.iterator();
while (i.hasNext()) {
    String nombre = i.next();
    if (nombre.equals("Elisa")) {
        // la ejecución de la sentencia siguiente provoca que se lance
        // la excepción ConcurrentModificationException
        alumnos.add("Felix");
    }
}
```

La excepción a esta regla está en la posibilidad de eliminar objetos de la colección con el método [remove](#) del iterador. El ejemplo siguiente muestra cómo eliminar de la colección anterior todos los alumnos cuyo nombre empieza por 'F' utilizando un iterador:

```
Iterator<String> i = alumnos.iterator();
while (i.hasNext())
    if (i.next().startsWith("F"))
        i.remove();
}
```

### 2.2.2. Bucle for mejorado

Este bucle está diseñado para iterar sobre arrays y en general sobre cualquier instancia de una clase que implemente la interface [Iterable](#). En las colecciones se exige que la referencia correspondiente se declare con un tipo parametrizado. El ejemplo siguiente muestra cómo iterar sobre una colección:

```
List<String> alumnos = new ArrayList<String>();
alumnos.add("Carmen");
alumnos.add("Fernando");
alumnos.add("Elisa");
for (String nombre: alumnos)
    System.out.println(nombre);
```

Con este método de iteración tampoco es posible añadir o eliminar objetos de la colección mientras se itera. Hacerlo también provocará el lanzamiento de la excepción [ConcurrentModificationException](#).

### 2.2.3. Método forEach

Esta forma de iterar reduce el número de sentencias debido a que se implementa en el método [forEach](#) toda la lógica necesaria (iteración interna). Lo único que se necesita es una expresión lambda que determina qué hacer con cada elemento de la colección. Por ejemplo:

```
List<String> alumnos = new ArrayList<String>();
alumnos.add("Carmen");
alumnos.add("Fernando");
alumnos.add("Elisa");
alumnos.forEach(nombre -> System.out.println(nombre));
```

## 2.3. Listas

Las colecciones que implementan la interface `List` representan una lista de elementos, es decir, una secuencia ordenada en la que:

- El orden de iteración se corresponde con el orden de inserción de los elementos.
- Se admiten tanto valores duplicados como el valor `null`.
- El método `add(objeto)` añade cada objeto a continuación del almacenado en la última posición.

La interface `List` extiende a la interface `Collection` con métodos que basan su funcionalidad en la utilización de índices. Los índices son números enteros que identifican las posiciones dentro de la colección de forma similar a como lo hacen en los arrays: el índice 0 corresponde a la primera posición, el 1 a la segunda, y así sucesivamente hasta la posición `size() - 1`, en la que se almacena el último elemento. Además, al igual que ocurre con los arrays, la utilización de índices fuera de rango también provoca el lanzamiento de la excepción `IndexOutOfBoundsException`.

En el ejemplo siguiente se añaden varios objetos a una lista y después se muestra por pantalla. Al mostrar el contenido, se puede ver que los objetos están ordenados según el orden de inserción:

```
List<String> alumnos = new ArrayList<String>();
alumnos.add("Carmen");
alumnos.add("Fernando");
alumnos.add("Elisa");
System.out.println(alumnos);
```

Salida:

```
[Carmen, Fernando, Elisa]
```

El método `add(índice, objeto)` inserta un objeto en una posición dada, desplazando los objetos que se encuentran a partir de dicha posición. Por ejemplo, si insertamos `"Manuel"` en la posición 1 de la lista anterior, `"Fernando"` y `"Elisa"` se desplazan a la posición 2 y 3 respectivamente:

```
alumnos.add(1, "Manuel");
System.out.println(alumnos);
```

Salida:

```
[Carmen, Manuel, Fernando, Elisa]
```

Además de inserciones, en una lista también es posible realizar operaciones de consulta, modificación y borrado basadas en índices con los métodos siguientes:

- `get(índice)`: retorna el objeto almacenado en la posición especificada.
- `remove(índice)`: elimina de la colección el elemento almacenado en la posición especificada.
- `set(índice, objeto)`: reemplaza el objeto almacenado en la posición especificada.

En este documento se describen dos tipos de listas, `ArrayList` y `LinkedList`. La diferencia entre ambas radica en la estructura de datos subyacente que implementa cada una de ellas y, por tanto, también en el rendimiento de las operaciones de consulta, inserción, modificación y borrado. En general, las operaciones de inserción `-add-` y borrado `-remove-` son más eficientes en una `LinkedList` que en un `ArrayList`. Sin embargo, con las operaciones de consulta `-get-` y modificación `-set-` ocurre lo contrario.

### 2.3.1. `ArrayList`

La estructura de datos subyacente en este tipo de lista es un array. La naturaleza estática de los arrays obliga a implementar toda la lógica necesaria para proporcionar la funcionalidad de una estructura de datos dinámica. Así pues, esta colección se crea un array con una determinada capacidad inicial. Cuando esta se agota, se crea un array nuevo con un 50% más de capacidad y se le transfirieren los elementos del array original, dejando este último a merced del recolector de basura.

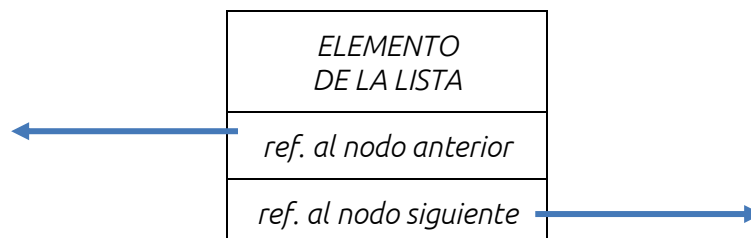
Las operaciones de inserción y borrado de elementos en posiciones intermedias son costosas incluso aunque no impliquen redimensionar el array, ya que obligan a mover una determinada cantidad de elementos hacia nuevas posiciones dentro del array.

La clase `ArrayList` proporciona tres constructores:

- `ArrayList()`: crea una lista con un tamaño inicial predeterminado.
- `ArrayList(int initialCapacity)`: crea una lista con el tamaño inicial especificado.
- `ArrayList(Collection<? extends E> c)`: crea una copia de la lista especificada.

### 2.3.2. `LinkedList`

La clase `LinkedList` utiliza como estructura de datos subyacente una lista doblemente enlazada. Esta estructura de datos está basada en la utilización de subestructuras denominadas nodos que tienen una doble función: almacenar cada elemento de la lista a la vez que apuntan al nodo anterior y al nodo siguiente:



En una lista doblemente enlazada se guardan dos referencias especiales: una que apunta al primer nodo y otra que apunta al último nodo. En una lista vacía estas dos referencias tendrán el valor `null` y en una lista con un solo elemento harán referencia al mismo nodo.

En el primer nodo de la lista, la referencia que apunta al anterior tendrá el valor nulo. En el último nodo de la lista, la referencia que apunta al siguiente tendrá el valor nulo.

La representación gráfica de una lista que almacena varios elementos podría ser algo así:





Las diferencias más notables con `ArrayList` son:

- La lista crece y decrece de forma natural con las inserciones y borrados. Por tanto, es innecesario un constructor que especifique un tamaño inicial.
- Define nuevos métodos que realizan de forma más eficiente las operaciones con el primer y último elemento de la lista: `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst` y `removeLast`. Estos métodos se definen por conveniencia, pero no incrementan la funcionalidad de la interface `List`. Por ejemplo, en una lista no vacía las expresiones `lista.getFirst()` y `lista.get(0)` son equivalentes.

### 2.3.3. Iterar sobre listas

Para las listas se ha definido un iterador avanzado que permite recorrer una lista en ambas direcciones, modificarla durante el proceso de iteración y obtener la posición actual. Este iterador se obtiene invocando al método `listIterator` de cualquier lista. Este retorna un objeto de tipo `ListIterator`, una interface que extiende a la interface `Iterator` con los métodos siguientes:

Método	Método y descripción
<code>void add(E e)</code>	Inserta el elemento especificado en la lista.
<code>boolean hasPrevious()</code>	Retorna <code>true</code> si este iterador tiene más elementos que preceden al elemento actual.
<code>E previous()</code>	Se sitúa en el elemento previo y lo retorna
<code>int nextIndex()</code>	Retorna el índice del elemento que retornará la siguiente llamada a <code>next</code> .
<code>int previousIndex()</code>	Retorna el índice del elemento que retornará la siguiente llamada a <code>previous</code> .
<code>void set(E e)</code>	Reemplaza el último elemento retornado por <code>next</code> o <code>previous</code> con el elemento especificado.

### 2.3.4. Ejemplo de un problema resueltos con listas

Se trata de desarrollar un programa que dadas varias secuencias de números enteros elimine de las mismas una determinada cantidad máxima de elementos, cada uno de ellos elegido con el algoritmo siguiente: se recorre desde el principio la secuencia y si se encuentra un número menor que el siguiente, ese número se elimina y finaliza el proceso. Si se llega al final sin encontrar un número menor que el siguiente, se elimina el último.

La entrada de datos se hará por teclado y se procesarán de la forma siguiente:

1. Se lee una línea con el número `S` de secuencias que se van a procesar.
2. Se obtienen los datos de cada secuencia de la forma siguiente:
  - Se lee una línea con el número `N` de elementos totales y el número `K` de elementos que hay que borrar, ambos separados por espacios en blanco.
  - Se lee una línea con la secuencia de elementos, cada uno de ellos separados por espacios en blanco
3. Se procesa la secuencia, se muestra el resultado y se pasa a la siguiente

Para simplificar el problema se asume que la entrada siempre es correcta, es decir, el número de elementos a borrar nunca será mayor que el número de elementos de la secuencia y todos los valores de entrada son números enteros.

Ejemplo de datos de entrada y datos de salida que se deberían de obtener:

ENTRADA	SALIDA
3	
3 1	
3 100 1	
	[100, 1]
5 2	
19 12 3 4 17	
	[19, 12, 17]
5 3	
23 45 11 77 18	
	[77, 18]

Solución:

```
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.Scanner;

public class Main4 {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        List<Integer> lista = new LinkedList<Integer>();
        int s = in.nextInt();
        for (int i=0; i<s; i++) {
            int n = in.nextInt();
            int k = in.nextInt();
            for (int j=0; j<n; j++)
                lista.add(in.nextInt());
            for (int j=0; j<k; j++)
                borrarElemento(lista);
            System.out.println(lista);
            lista.clear();
        }
        in.close();
    }

    static void borrarElemento(List<Integer> lista) {
        ListIterator<Integer> i = lista.listIterator();
        if (i.hasNext()) {
            int a = i.next();
            while (i.hasNext()) {
                int b = i.next();
                if (a < b) {
                    i.previous();
                    i.previous();
                    i.remove();
                    return;
                }
            }
            else
                a = b;
        }
        i.remove();
    }
}
```



## 2.4. Colas y Pilas

En el *Collections Framework* se definen las clases e interfaces siguientes para el manejo de colas y pilas:

C/I	NOMBRE	DESCRIPCIÓN
Interface	<code>Queue</code>	Cola convencional.
Interface	<code>Deque</code>	Cola doblemente terminada.
Clase	<code>PriorityQueue</code>	Implementa la interface <code>Queue</code> para definir una cola de prioridad.
Clase	<code>LinkedList</code>	Implementa la interface <code>Deque</code> para definir tanto una cola convencional como una cola doblemente terminada.
Clase	<code>ArrayDeque</code>	Implementa la interface <code>Deque</code> para definir una cola doblemente terminada.

Las colas se crean instanciando las clases `PriorityQueue`, `LinkedList` o `ArrayDeque`. Las pilas se crean instanciando cualquiera de las dos últimas.

Tanto pilas como colas admiten valores duplicados, sin embargo, solamente aquellas colas o pilas que se creen instanciando la clase `LinkedList` admiten valores nulos.

### 2.4.1. Colas

Una **cola**, **queue** en inglés, es una lista de elementos sujeta a las restricciones siguientes:

- Los elementos se añaden por un extremo y se extraen por el otro. Por tanto, siempre se extrae el elemento de mayor antigüedad en la cola, por lo que también se conocen como estructuras FIFO (First In First Out, el primero en entrar el primero en salir).
- El elemento que se encuentra al principio de la cola se denomina **head** y el que se encuentra al final de la cola se denomina **tail**.
- Los elementos se insertan en el final de la cola mediante una operación denominada **enqueue**.
- Los elementos se retiran del principio de la cola mediante una operación denominada **dequeue**.
- El elemento head se puede consultar mediante una operación denominada **front** o **element**.

Además de las colas convencionales, existen otros dos tipos de colas:

- Cola doblemente terminada, en inglés **deque** (Double Ended Queue), son colas en las que se pueden añadir y retirar los elementos por ambos extremos. Aúnan en una única estructura la funcionalidad de una cola y de una pila.
- Colas de prioridad, en las que los elementos tienen asociada una prioridad que determina el orden en el que se retiran de la cola. Siempre se retira el elemento que tenga asociada la mayor prioridad. En caso de que varios elementos compartan la mayor prioridad la extracción se realiza como se haría en una cola convencional.

La sentencia siguiente crea una **cola convencional** que se maneja a través de una referencia a un objeto de tipo `Queue`:

```
Queue<String> cola = new LinkedList<String>();
```

La tabla siguiente resume los métodos que se declaran en la interface [Queue](#) para realizar las operaciones propias de una cola convencional. Se distinguen dos tipos de métodos para cada operación, aquellos que lanzan una excepción si la operación no es posible por alguna razón, y aquellos que retornan un valor especial en la misma situación:

Operaciones	Métodos	
	Lanzan excepción	Valor de retorno especial
Enqueue	<code>add(e)</code>	<code>offer(e)</code>
Dequeue	<code>remove()</code>	<code>poll()</code>
Front	<code>element()</code>	<code>peek()</code>

Las sentencias siguientes crean **colas doblemente terminadas** instanciando las clases [LinkedList](#) y [ArrayDeque](#) respectivamente para manejarlas a través de una referencia a un objeto [Deque](#):

```
Deque<String> cola1 = new LinkedList<String>();
Deque<String> cola2 = new ArrayDeque<String>();
```

La tabla siguiente resume los métodos que se declaran en la interface [Deque](#) para realizar las operaciones propias de una cola doblemente terminada. Se distinguen dos tipos de métodos para cada operación, aquellos que lanzan una excepción si la operación no es posible por alguna razón, y aquellos que retornan un valor especial en la misma situación:

Operaciones	Métodos			
	Primer elemento (head)		Último elemento (tail)	
	Lanzan excepción	Valor de retorno especial	Lanza excepción	Valor de retorno especial
Enqueue	<code>addFirst(e)</code>	<code>offerFirst(e)</code>	<code>addLast(e)</code>	<code>offerLast(e)</code>
Dequeue	<code>removeFirst()</code>	<code>pollFirst()</code>	<code>removeLast()</code>	<code>pollLast()</code>
Front	<code>getFirst()</code>	<code>peekFirst()</code>	<code>getLast()</code>	<code>peekLast()</code>

La sentencia siguiente crea una **cola de prioridad** que se maneja a través de una referencia a un objeto de tipo [PriorityQueue](#):

```
PriorityQueue<String> cola = new PriorityQueue<String>();
```

La clase [PriorityQueue](#) ofrece varios constructores que determinan cómo se establece la prioridad de los elementos. Así, la prioridad de cada elemento puede estar basada en el orden natural definido en la clase correspondiente o en un objeto [Comparator](#) que se ha de especificar en el constructor y que determina el criterio de ordenación para los elementos que se van a insertar. En el primer caso no se permite la inserción de objetos no comparables (recordemos que un objeto es comparable si la clase a la que pertenece implementa la interface [Comparable](#)).

La clase [PriorityQueue](#) implementa la interface [Queue](#). Por tanto, una vez creada, la cola de prioridad se maneja utilizando los mismos métodos que se utilizan para manejar una cola convencional.

### 2.4.2. Pilas

Una **pila**, **stack** en inglés, es una lista de elementos sujeta a las restricciones siguientes:

- Los elementos se añaden y se eliminan por el mismo extremo, denominado **tope** de la pila. Por tanto, siempre se extrae el elemento que se ha añadido más recientemente, por lo que también se conocen como estructuras LIFO (Last In First Out, último en entrar primero en salir).

- Los elementos se insertan en el tope de la pila mediante una operación denominada *push*.
- Los elementos se retiran del tope de la pila mediante una operación denominada *pop*.
- El elemento del tope de la pila se puede obtener sin retirarlo mediante una operación denominada *peek*.

En el *Collection Framework* se define la clase [Stack](#) que representa una pila. Sin embargo, en la documentación del API de java se aconseja instanciar las clases [LinkedList](#) o [ArrayDeque](#) para crear pilas, ya que implementan la interface [Deque](#) que representa una pila más completa y de mayor consistencia que la clase [Stack](#).

Las sentencias siguientes crean **pilas** instanciando las clases [LinkedList](#) y [ArrayDeque](#) respectivamente para manejarlas a través de una referencia a un objeto de tipo [Deque](#):

```
Deque<String> pila1 = new LinkedList<String>();
Deque<String> pila2 = new ArrayDeque<String>();
```

La tabla siguiente resume los métodos que se utilizan para realizar las operaciones propias de una pila cuando se maneja mediante una referencia a un objeto [Deque](#):

Operaciones	Métodos
Push	<code>addFirst(e)</code> / <code>offerFirst(e)</code> / <code>push(e)</code>
Pop	<code>removeFirst()</code> / <code>pollFirst()</code> / <code>pop()</code>
Peek	<code>getFirst()</code> / <code>peekFirst()</code> / <code>element()</code> / <code>peek()</code>

### 2.4.3. Ejemplos de problemas resueltos con colas y pilas

**Ejemplo 1:** Programa que simule la ejecución organizada de una serie de procesos. Los procesos se identifican con un número entero de 1 a N, siendo N el número de procesos. Los datos de entrada al programa son:

- El número N de procesos.
- Los N pares (*nombre\_proceso*, *id\_proceso*) especificados en un orden arbitrario.
- Los N identificadores de cada proceso en el orden ideal en que deberían ser ejecutados.

Cada proceso se representa mediante la clase siguiente, que simula su ejecución mostrando su nombre en la consola:

```
public class Proceso {

    private String nombre;
    private int id;

    public Proceso(String nombre, int id) {
        this.nombre = nombre;
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public int getId() {
        return id;
    }

    public void ejecutar() {
        System.out.println("Ejecutando " + nombre);
    }
}
```

```

    }
}

```

Ejemplo de datos de entrada y datos de salida que se deberían de obtener:

**Entrada:**

```

3
P3 3 P2 2 P1 1
1 3 2

```

**Salida:**

```

Ejecutando P1
Ejecutando P3
Ejecutando P2

```

Este problema se puede resolver con dos **colas**, una en la que se insertan los procesos que se crean con los datos introducidos en la segunda línea, y otra en la que se van almacenando los N identificadores de proceso introducidos en la tercera línea. Para simplificar la solución del problema se asume que los datos introducidos siempre cumplen las especificaciones dadas:

```

import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt();
        Queue<Proceso> procesos = new LinkedList<Proceso>();
        for (int i=0; i<n; i++)
            procesos.offer(new Proceso(s.next(), s.nextInt()));
        Queue<Integer> ideal = new LinkedList<Integer>();
        for (int i=0; i<n; i++)
            ideal.offer(s.nextInt());
        int esperado;
        Proceso proceso;
        while(!procesos.isEmpty()) {
            proceso = procesos.poll();
            esperado = ideal.poll();
            while (proceso.getId() != esperado) {
                procesos.offer(proceso);
                proceso = procesos.poll();
            }
            proceso.ejecutar();
        }
    }
}

```

**Ejemplo 2:** Programa que lee por teclado una línea con varios números enteros separados por espacios y los muestra en orden inverso.

La solución con una **pila** consiste en introducir en ella todos los números según se van leyendo. Cuando se hayan añadido todos, se van extrayendo y mostrando en la pantalla uno a uno.

```

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.ArrayDeque;
import java.util.Deque;

```

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) throws IOException {
        Deque<Integer> pila = new ArrayDeque<Integer>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Números: ");
        Scanner in = new Scanner(br.readLine());
        while (in.hasNextInt())
            pila.push(in.nextInt());
        while (!pila.isEmpty())
            System.out.printf("%d ", pila.pop());
        in.close();
    }
}
```

**Ejemplo 3:** Programa que determina si los paréntesis, corchetes y llaves utilizados en una expresión introducida por teclado están balanceados. Este problema se puede resolver con una **pila** en la que se insertan los elementos de apertura y se extraen cuando se lee uno de cierre. Si el elemento de apertura extraído no casa con el elemento de cierre o la pila está vacía cuando se lee un elemento de cierre o la pila no está vacía cuando se termina de procesar la expresión, se puede concluir que estos elementos no están balanceados.

Ejemplos de expresiones balanceadas	Ejemplos de expresiones no balanceadas
([]{})	([])
((([[](){}))[]){})	((([[](){}))[]){}

Solución:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {

    public static void main(String[] args) throws IOException {
        Deque<Character> pila = new ArrayDeque<Character>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Expresión: ");
        String linea = br.readLine();
        char c;
        int i = 0;
        boolean balanceados = true;
        while (i < linea.length() && balanceados) {
            c = linea.charAt(i++);
            if (c == '(' || c == '[' || c == '{')
                pila.push(c);
            else if (c == ')' || c == ']' || c == '}')
                balanceados = casan(pila.pop(), c);
        }
        if (balanceados && !pila.isEmpty()) {
            balanceados = false;
            pila.clear();
        }
        if (!balanceados)
            System.out.print("no ");
        System.out.println("están balanceados");
    }

    static boolean casan(char apertura, char cierre) {
        return (apertura == '(' && cierre == ')') ||
            (apertura == '[' && cierre == ']') ||
            (apertura == '{' && cierre == '}');
    }
}
```

| }

## 2.5. Conjuntos

La interface `Set`, implementada por las clases `HashSet` y `TreeSet`, es una abstracción del concepto matemático de conjunto. Por tanto, las colecciones que implementan esta interface no van a admitir que se almacene más de una vez un mismo objeto y las operaciones que cobran mayor relevancia van a ser las operaciones propias de conjuntos:

- Pertenencia al conjunto: métodos `contains`.
- Subconjunto: `containsAll`.
- Unión de conjuntos: método `addAll`.
- Intersección de conjuntos: método `retainAll`.
- Diferencia de conjuntos: método `removeAll`.

Las diferencias entre `HashSet` y `TreeSet` se resumen en la tabla siguiente:

	HashSet	TreeSet
Estructura de datos	Tabla hash	Árbol AVL
Admite valores nulos	Si	No
Ordena los elementos	No	Si
Ejemplo	<pre>Set&lt;String&gt; s = new HashSet&lt;String&gt;(); s.add("Fernando"); s.add("Elisa"); s.add("Mateo"); s.add(null); System.out.println(s);</pre>	<pre>Set&lt;String&gt; s = new TreeSet&lt;String&gt;(); s.add("Fernando"); s.add("Elisa"); s.add("Mateo"); System.out.println(s);</pre>
Salida	[Mateo, null, Fernando, Elisa]	[Elisa, Fernando, Mateo]

### 2.5.1. Ejemplos de problemas resueltos con conjuntos

**Ejemplo 1:** Programa que lee una línea de texto de la entrada estándar y muestra qué caracteres están repetidos y qué caracteres aparecen una sola vez, ignorando los espacios en blanco.

Ejemplo de datos de entrada y datos de salida que se deberían de obtener:

ENTRADA	SALIDA
colecciones y mapas	Caracteres sin repetir: [p, i, y, l, m, n] Caracteres repetidos: [a, c, s, e, o]

Este problema se puede resolver con dos conjuntos, uno que vaya guardando los caracteres no repetidos y otro que vaya guardando los repetidos. Cada carácter se procesa de la forma siguiente:

- Si no pertenece a ninguno de los conjuntos, se añade al conjunto de los no repetidos.
- Si pertenece al conjunto de los no repetidos, se saca de ese conjunto y se añade al de los repetidos.
- Si pertenece al conjunto de los repetidos, no se hace nada.

**Solución 1:**

```
import java.io.BufferedReader;
```



```
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashSet;
import java.util.Set;

public class Main {

    public static void main(String[] args) throws IOException {
        Set<Character> repetidos = new HashSet<Character>();
        Set<Character> sinRepetir = new HashSet<Character>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String linea = br.readLine();
        for (Character c: linea.toCharArray())
            if (!Character.isSpaceChar(c)) {
                if (sinRepetir.contains(c)) {
                    repetidos.add(c);
                    sinRepetir.remove(c);
                }
                else if (!repetidos.contains(c))
                    sinRepetir.add(c);
            }
        System.out.println("Caracteres sin repetir: " + sinRepetir);
        System.out.println("Caracteres repetidos: " + repetidos);
    }
}
```

## Solución 2:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashSet;
import java.util.Set;

public class Main {

    public static void main(String[] args) throws IOException {
        Set<Character> repetidos = new HashSet<Character>();
        Set<Character> sinRepetir = new HashSet<Character>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String linea = br.readLine();
        for (Character c: linea.toCharArray())
            if (!Character.isSpaceChar(c) && !repetidos.contains(c) &&
                !sinRepetir.add(c)) {
                repetidos.add(c);
                sinRepetir.remove(c);
            }
        System.out.println("Caracteres sin repetir: " + sinRepetir);
        System.out.println("Caracteres repetidos: " + repetidos);
    }
}
```

## Solución 3:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashSet;
import java.util.Set;

public class Main {

    public static void main(String[] args) throws IOException {
        Set<Character> repetidos = new HashSet<Character>();
        Set<Character> sinRepetir = new HashSet<Character>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String linea = br.readLine();
        for (Character c: linea.toCharArray())
            if (!Character.isSpaceChar(c) && !sinRepetir.add(c))
                repetidos.add(c);
    }
}
```

```

        repetidos.add(c);
        sinRepetir.removeAll(repetidos);
        System.out.println("Caracteres sin repetir: " + sinRepetir);
        System.out.println("Caracteres repetidos: " + repetidos);
    }
}

```

En este punto conviene resaltar que en la primera solución se podrían haber utilizado otro tipo de colecciones, como por ejemplo listas. Sin embargo, la segunda y la tercera solución solo son posibles usando conjuntos.

**Ejemplo 2:** Programa que lee de la entrada estándar varios compuestos químicos y, después de leerlos todos, muestra en la salida estándar todos los elementos químicos que forman dichos compuestos sin que se repita ninguno y en orden alfabético. La primera línea de texto contiene el número de compuestos. El resto de las líneas contienen los compuestos a razón de uno por línea. Cada compuesto se escribe como una secuencia de elementos químicos separados por espacios en blanco.

Ejemplo de datos de entrada y datos de salida que se deberían de obtener:

ENTRADA	SALIDA
4	[Ce, Mo, Ne, O]
Ce O	
Mo O Ce	
Ne	
Mo	

**Solución:**

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Set;
import java.util.TreeSet;

public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Set<String> elementos = new TreeSet<String>();
        List<List<String>> compuestos = new ArrayList<List<String>>();
        int n = Integer.valueOf(br.readLine());
        for (int i=0; i<n; i++) {
            Scanner s = new Scanner(br.readLine());
            List<String> compuesto = new ArrayList<String>();
            while (s.hasNext())
                compuesto.add(s.next());
            compuestos.add(compuesto);
        }
        for (List<String> compuesto: compuestos)
            elementos.addAll(compuesto);
        System.out.println(elementos);
    }
}

```

### 3. Mapas

Un mapa es una colección en la que cada elemento almacenado tiene dos componentes: una clave y un valor. En el *Collections Framework* la interface `Map` está en la raíz de la jerarquía de clases en la que se definen los mapas. Esta interface no extiende a la interface `Collection` debido que sus métodos están orientados a las relaciones clave-valor.

Tanto claves como valores han de ser objetos y cada dupla clave-valor recibe el nombre de entrada. Las características básicas que definen la funcionalidad de un mapa son:

- Las claves tienen una función similar a las claves primarias de las tablas en el modelo relacional de datos, es decir, se define sobre ellas una restricción de unicidad.
- Un mismo valor puede estar asociado a más de una clave en varias entradas.
- No se puede utilizar `null` como clave, pero si como valor.

Por ejemplo, podemos almacenar nombres de personas en un mapa usando como clave su NIF. De esta forma no habrá problema a la hora de almacenar los datos de dos personas que tengan el mismo nombre:

CLAVE	VALOR	
11111111	Elena Fernández	*
11222333	Carlos González	
11444555	Rodrigo Rodríguez	
11321123	Elena Fernández	*
11777999	Pilar Ramos	

Se pueden crear tres tipos de mapas:

- `HashMap`: implementa la interface `Map` y se caracteriza por utilizar una tabla hash como estructura de datos subyacente y por no mantener ningún orden específico en las entradas.
- `TreeMap`: Implementa la interface `SortedMap` que extiende a su vez a la interface `Map`. Por tanto, una de sus características básicas es que mantiene ordenadas las entradas atendiendo al orden natural de las claves. Utiliza como estructura de datos subyacente un árbol AVL.
- `LinkedHashMap` que extiende a la clase `HashMap`. Además de la tabla hash, emplea una lista enlazada para proveer de un orden de iteración basado en dos posibles criterios: el orden de inserción o el orden en el que se accedió a los elementos por última vez. Por defecto se utiliza el primer criterio de ordenación, pero esta clase define un constructor que permite elegir cualquiera de los dos.

Las operaciones más relevantes que se declaran en la interface `Map` se resumen en la tabla siguiente:

MÉTODO	DESCRIPCIÓN
<code>void clear()</code>	Elimina todas las entradas del mapa.
<code>boolean containsKey(Object key)</code>	Retorna <code>true</code> si el mapa contiene un elemento asociado a la clave especificada.
<code>boolean containsValue(Object value)</code>	Retorna <code>true</code> si el mapa contiene el valor especificado asociado a una o más claves.
<code>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</code>	Retorna el conjunto de entradas del mapa en forma de objetos <code>Map.Entry</code> , una clase interna definida en la clase <code>Map</code> .
<code>default void forEach(BiConsumer&lt;? super K,? super V&gt; action)</code>	Iteración interna sobre las entradas del mapa, procesando cada una con la expresión lambda



	especificada.
<code>V get(Object key)</code>	Retorna el valor asociado a la clave especificada o <code>null</code> si en el mapa no se asocia ningún valor con dicha clave.
<code>V getOrDefault(Object key, V defaultValue)</code>	Retorna el valor asociado a la clave especificada o el valor por defecto especificado si en el mapa no se asocia ningún valor con dicha clave.
<code>boolean isEmpty()</code>	Retorna <code>true</code> si el mapa está vacío.
<code>Set&lt;K&gt; keySet()</code>	Retorna el conjunto de claves contenidas en el mapa.
<code>V put(K key, V value)</code>	Añade una nueva entrada asociando la clave especificada al valor especificado y retorna <code>null</code> . Si la clave ya está asociada a otro valor, se sustituye el valor antiguo por el valor especificado retornando el valor antiguo.
<code>void putAll(Map&lt;? extends K, ? extends V&gt; m)</code>	Añade a este mapa todas las entradas del mapa especificado. Cada entrada se añade tal y como lo haría el método <code>put</code> .
<code>V putIfAbsent(K key, V value)</code>	Si la clave especificada no se encuentra aún asociada a un valor (o está asociada al valor <code>null</code> ), se asocia al valor especificado y retorna <code>null</code> . Si la clave especificada ya tiene un valor asociado, no se modifica la entrada correspondiente y se retorna dicho valor.
<code>V remove(Object key)</code>	Elimina la entrada clave/valor correspondiente a la clave especificada.
<code>boolean remove(Object key, Object value)</code>	Elimina la entrada correspondiente a la clave especificada sólo si está asociada al valor especificado y retorna <code>true</code> . Retorna <code>false</code> en caso contrario.
<code>V replace(K key, V value)</code>	Reemplaza el valor asociado a la clave especificada con el valor especificado sólo si dicha clave está asociada con algún valor. Retorna el valor antiguo se produce el reemplazo, o <code>null</code> en caso contrario.
<code>boolean replace(K key, V oldValue, V newValue)</code>	Reemplaza el valor asociado a la clave especificada con el valor nuevo especificado sólo si dicha clave está asociada con el valor antiguo especificado. Retorna el valor antiguo se produce el reemplazo, o <code>null</code> en caso contrario.
<code>int size()</code>	Retorna el número de entrada en el mapa.
<code>Collection&lt;V&gt; values()</code>	Retorna una colección con todos los valores que contiene el mapa.

### 3.1. Iterar sobre mapas

Al contrario que en las colecciones, en los mapas no existe un método que retorne un iterador ni la posibilidad de utilizar el bucle `for` mejorado para iterar sobre las entradas. Sin embargo, la clase `Map` permite llevar a cabo una iteración interna sobre las entradas a través del método `forEach`.

También es posible obtener la colección de entradas, la de claves o la de valores, retornadas por los métodos `entrySet`, `keySet` y `values` respectivamente, para iterar sobre ellas. A continuación, se muestran varios ejemplos de iteración sobre el mapa siguiente:

```
Map<Integer, String> mapa = new TreeMap<Integer, String>();
m.put(11111111, "Elena Fernández");
m.put(11222333, "Carlos González");
m.put(11444555, "Rodrigo Rodríguez");
```

```
m.put(11321123, "Elena Fernández");  
m.put(11777999, "Pilar Ramos");
```

Iteración sobre las entradas con un iterador:

```
Iterator<Entry<Integer, String>> i = mapa.entrySet().iterator();  
while (i.hasNext()) {  
    Entry<Integer, String> e = i.next();  
    System.out.printf("NIF: %d - Nombre: %s\n", e.getKey(), e.getValue());  
}
```

Iteración sobre las entradas con bucle `for` mejorado:

```
for (Entry<Integer, String> e: mapa.entrySet())  
    System.out.printf("NIF: %d - Nombre: %s\n", e.getKey(), e.getValue());
```

Iteración interna sobre las entradas:

```
mapa.forEach((nif, nombre) -> System.out.printf("NIF: %d - Nombre: %s\n", nif, nombre));
```

Iteración sobre las claves con un iterador:

```
Iterator<Integer> i = mapa.keySet().iterator();  
while (i.hasNext())  
    System.out.printf("NIF: %d\n", i.next());
```

Iteración sobre las claves con bucle `for` mejorado:

```
for (Integer nif: mapa.keySet())  
    System.out.printf("NIF: %d\n", nif);
```

Iteración interna sobre las claves:

```
mapa.keySet().forEach(nif -> System.out.printf("NIF: %d\n", nif));
```

Iteración sobre los valores con un iterador:

```
Iterator<String> i = mapa.values().iterator();  
while (i.hasNext())  
    System.out.printf("Nombre: %s\n", i.next());
```

Iteración sobre los valores con bucle `for` mejorado:

```
for (String nombre: mapa.values())  
    System.out.printf("Nombre: %s\n", nombre);
```

Iteración interna sobre los valores:

```
mapa.values().forEach(nombre -> System.out.printf("Nombre: %s\n", nombre));
```

## 3.2. Ejemplos de problemas resueltos con mapas

**Ejemplo 1:** Programa que lee una línea de texto de la entrada estándar y contabiliza el número de veces que se repite cada carácter que no es un espacio en blanco. Muestra el resultado en la salida estándar.

Este problema se puede resolver mediante un mapa en el que las claves son los caracteres leídos y los valores son el número de veces que se repiten.

```
import java.io.BufferedReader;  
import java.io.IOException;
```

```
import java.io.InputStreamReader;
import java.util.Map;
import java.util.TreeMap;

public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Map<Character, Integer> mapa = new TreeMap<Character, Integer>();
        System.out.println("Texto:");
        String linea = br.readLine().toLowerCase();
        for (char c: linea.toCharArray()) {
            if (!Character.isSpaceChar(c)) {
                if (mapa.containsKey(c))
                    mapa.put(c, mapa.get(c) + 1);
                else
                    mapa.put(c, 1);
            }
        }
        System.out.println(mapa);
    }
}
```

**Ejemplo 2:** Programa que lea de la entrada estándar dos o más líneas de texto. La primera línea ha de contener el número de líneas que le siguen. A medida que va leyendo el resto de las líneas tendrá que ir almacenando, sin que se repitan, las palabras que contiene cada una de ellas en diferentes colecciones. Cada colección ha de contener palabras de una longitud determinada y se creará únicamente cuando se necesite. Las colecciones de palabras se han de almacenar a su vez en una estructura de datos que permita recuperarlas fácilmente. Para poner a prueba la estructura de datos creada, el programa leerá de la entrada estándar longitudes y mostrará la lista de palabras de cada una o `null` si no hay palabras de dicha longitud. Finalizará cuando se introduzca una longitud menor que 1.

Este problema se puede resolver con un mapa en el que las claves son una longitud de palabra y los elementos una colección de palabras de dicha longitud. En concreto, el mapa puede ser un `HashMap` (no se considera necesario que las entradas estén ordenadas por la clave) y la colección un `TreeSet` (para que se muestre la lista de palabras en orden alfabético).

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import java.util.TreeSet;

public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Map<Integer, TreeSet<String>> mapa = new HashMap<Integer, TreeSet<String>>();
        System.out.println("Lineas:");
        int n = Integer.valueOf(br.readLine());
        for (int i=0; i<n; i++) {
            Scanner s = new Scanner(br.readLine());
            while (s.hasNext())
                agregarPalabra(mapa, s.next());
        }
        int longitud;
        do {
            System.out.println("Longitud:");
            longitud = Integer.valueOf(br.readLine());
            if (longitud > 0)
                System.out.println(mapa.get(longitud));
        } while (longitud > 0);
    }

    static void agregarPalabra(Map<Integer, TreeSet<String>> mapa, String palabra) {
```



```
int longitud = palabra.length();
TreeSet<String> set = mapa.get(longitud);
if (set == null) {
    set = new TreeSet<String>();
    mapa.put(longitud, set);
}
set.add(palabra);
}
```