API Collections en Java

¿Qué son las colecciones?

Una colección representa un grupo de objetos.

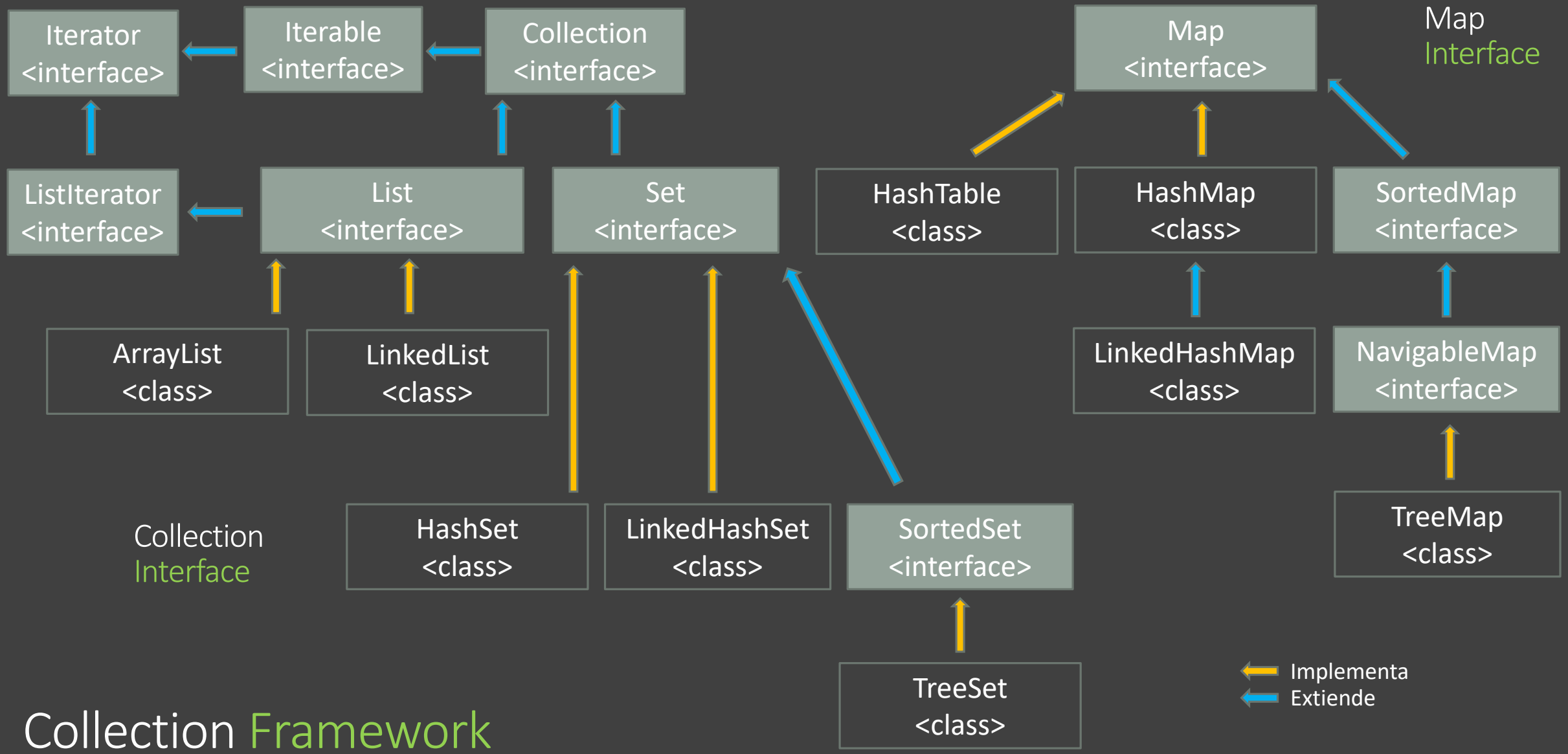
Esto objetos son conocidos como elementos.

Cuando queremos trabajar con un conjunto de elementos de una manera sincronizada, necesitamos un almacén especial donde poder guardarlos.

En Java, se emplean las interfaces Collection y Map para este propósito. El conjunto de ambas interfaces es conocido como Collection Framework que se compone de Collection Interface y de Map Interface.

Gracias a ellas podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección…

Partiendo de las intefaces Collection y Map, extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.



Collection Framework

Joselu Cantero

Collection Framework

Clases de Utilidad

- Collections <class>
- Arrays <class>

Interfaces de Ordenamiento

- Comparator <interface>
- Comparable <interface>

La interface List<E>

La interface List define una sucesión de elementos. A diferencia de la interface Set, la interface List sí admite elementos duplicados.

Aparte de los métodos heredados de Collection, añade métodos que permiten mejorar los siguientes puntos:

1 - Acceso posicional a elementos: manipula elementos en función de su posición en la lista (método get).

2 - Búsqueda de elementos: busca un elemento concreto de la lista y devuelve su posición.

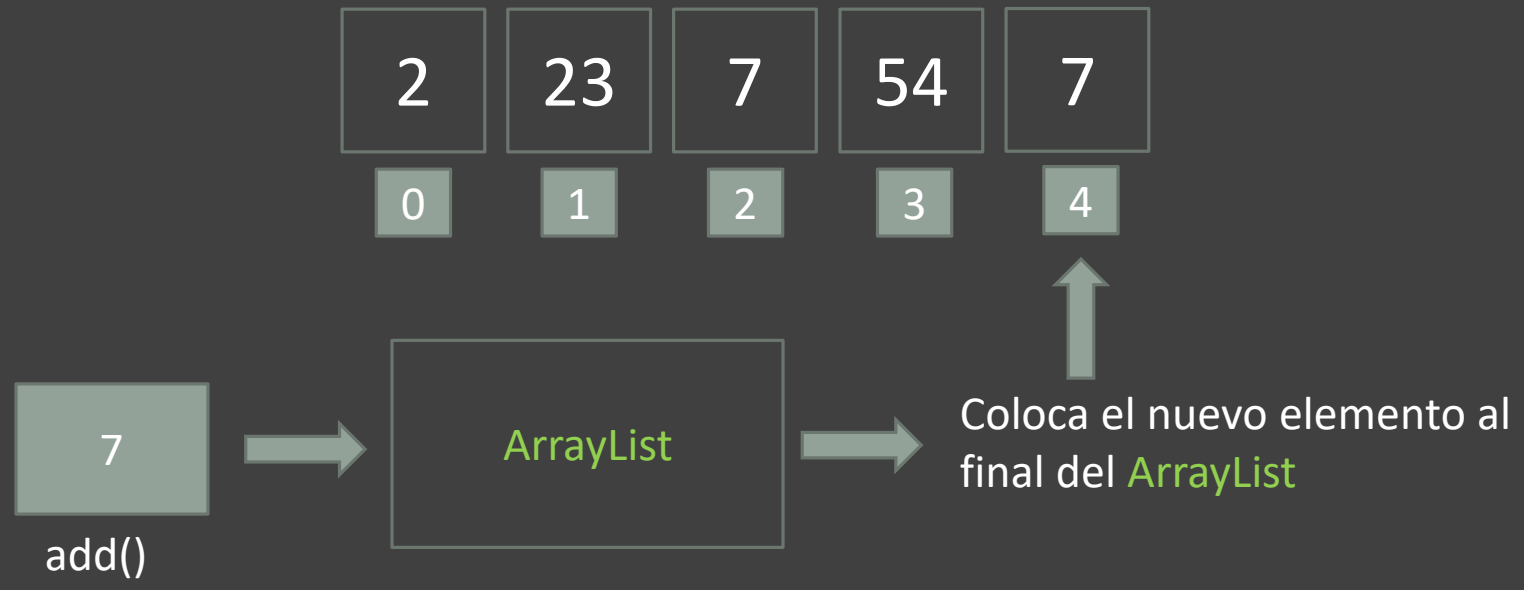
3 - Iteración sobre elementos: mejora el Iterator por defecto con ListIterator.

4 - Rango de operación: permite realizar ciertas operaciones sobre rangos de elementos dentro de la propia lista.

Un List permite elementos duplicados y los almacena de una manera ordenada. Mediante el método sort podemos ordenar con diferentes criterios.

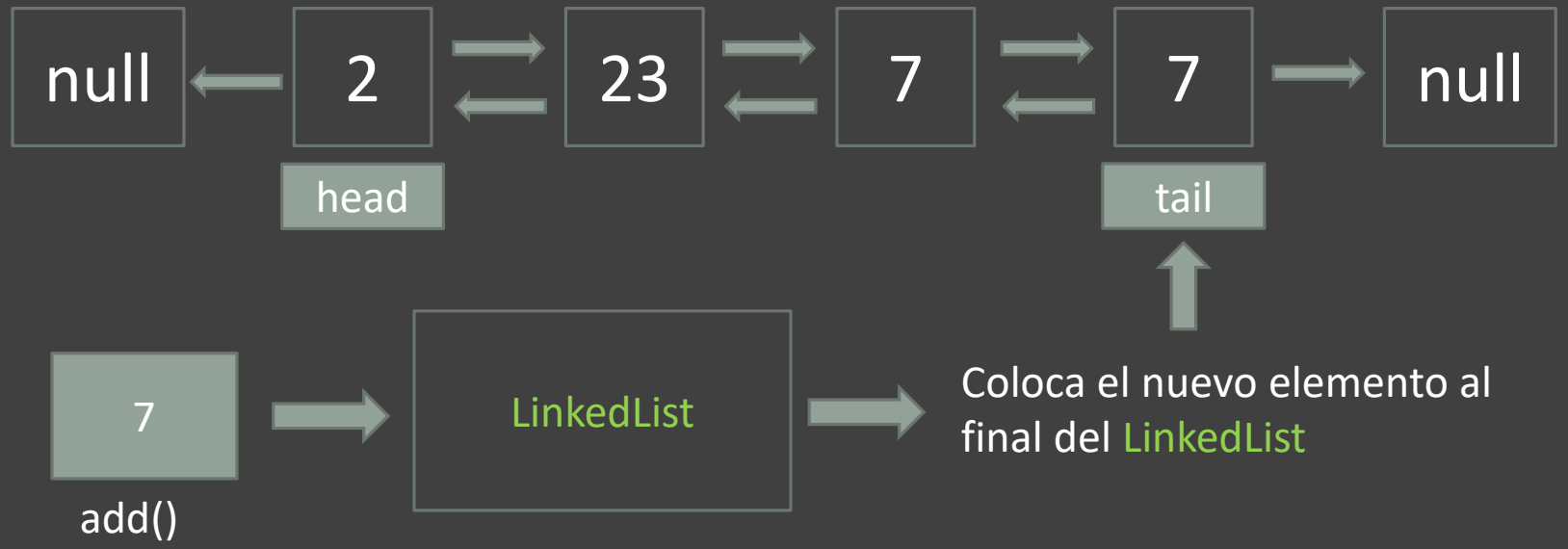
ArrayList<E> (interface List<E>)

Es la implementación típica. 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.



LinkedList<E> (interface List<E>)

Esta implementación permite que, en ciertas ocasiones dependiendo de nuestras necesidades, mejore el rendimiento. Se basa en una lista doblemente enlazada teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.



ListIterator<E>

ListIterator es un iterador para la interface List que permite al programador recorrer la lista en cualquier dirección, modificar la lista durante la iteración y obtener la posición actual del iterador en la lista.

Un ListIterator no tiene elemento actual; su posición de cursor siempre se encuentra entre el elemento que sería devuelto por una llamada a previous() y el elemento que sería devuelto por una llamada a next().

Un iterador para una lista de longitud n tiene n+1 posibles posiciones del cursor:

Posiciones posibles del pointer (cursor) de un ListIterator

Métodos de ListIterator<E>

Método	Descripción	Devuelve
hasNext()	Devuelve true si el ListIterator tiene más elementos al iterar la lista en el sentido de avance	boolean
next()	Devuelve el siguiente elemento de la lista y avanza la posición del cursor	Object
nextIndex()	Devuelve el índice del elemento que sería devuelto por una llamada posterior a next	int
hasPrevious()	Devuelve true si el ListIterator tiene más elementos al iterar la lista en el sentido de retroceso	boolean
previous()	Devuelve el elemento anterior de la lista y mueve la posición del cursor hacia atrás.	Object
previousIndex()	Devuelve el índice del elemento que sería devuelto por una llamada posterior a previous	int
remove()	Este método elimina de la lista el último elemento devuelto por next() o previous()	void
set(Object o)	Este método sustituye el último elemento devuelto por next() o previous() por el nuevo elemento	void
add(Object o)	Este método se utiliza para insertar un nuevo elemento en la lista	void

La interface Set<E>

La interface Set define una colección que no puede contener elementos duplicados.

Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos equals y hashCode.

Para comprobar si dos colecciones Set son iguales, se comprobarán si todos los elementos que los componen son iguales sin importar en el orden que ocupen dichos elementos.

Un ejemplo de un Set sería un HashSet, un LinkedHashSet y un TreeSet que al ser también un SortedSet contiene elementos ordenados.

Un Set no permite elementos duplicados y los almacena de una manera desordenada (excepto en el caso de un TreeSet). No dispone del método sort ni del método get.

La interface SortedSet<E>

Esta interfaz es muy similar a la interface Set.

Tan solo se diferencia en que SortedSet permite que los elementos dentro del conjunto de la colección estén ordenados totalmente, facilitando por tanto su acceso en búsquedas y haciendo más rápido su consulta.

Los elementos son ordenados usando su orden natural, o bien usando un Comparator.

Un ejemplo de un SortedSet sería un TreeSet.

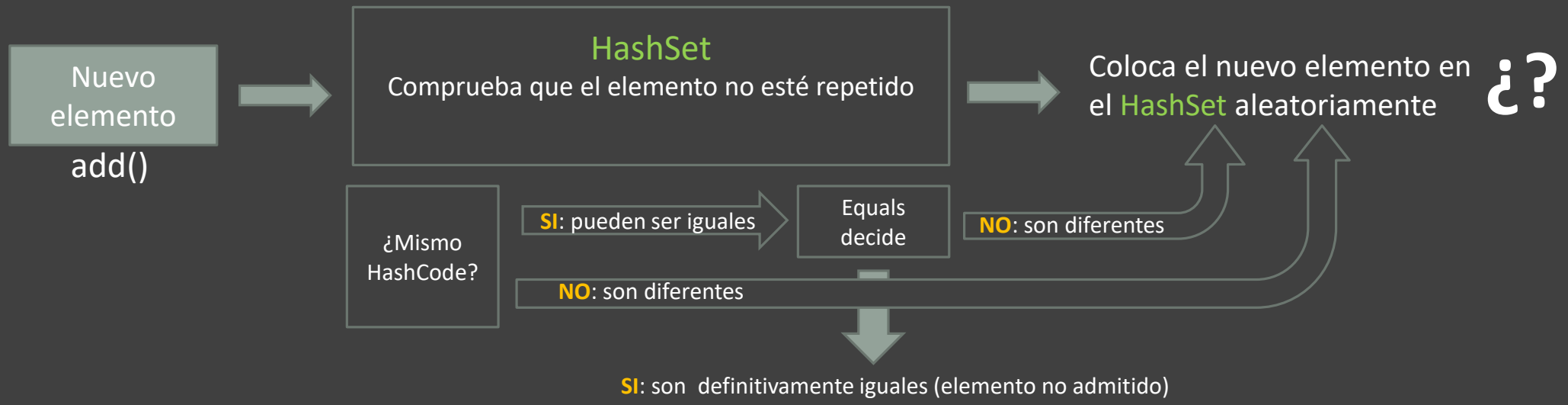
Un SortedSet no permite elementos duplicados (ya que es un Set) y los almacena de una manera ordenada.

HashSet<E> (interface Set<E>)

Esta implementación almacena los elementos en una tabla hash.

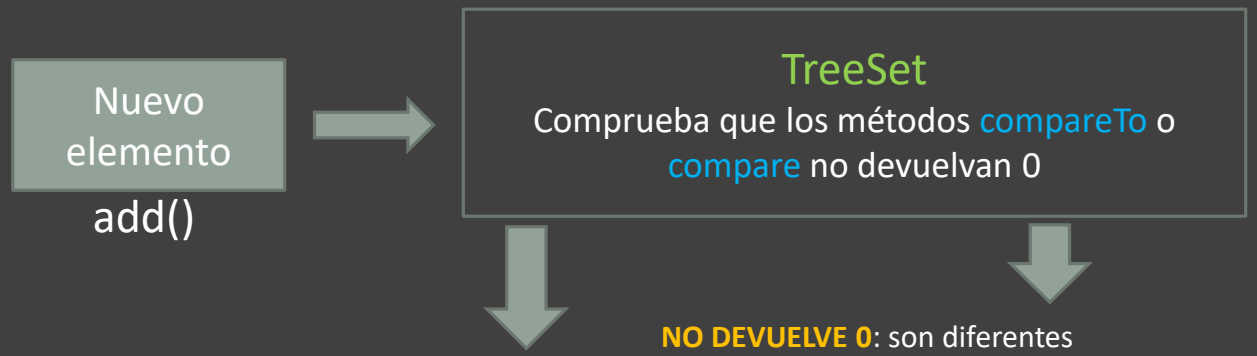
Es el Set con mejor rendimiento pero no garantiza ningún orden a la hora de realizar iteraciones.

Es la implementación Set más empleada debido a su rendimiento y a que, generalmente, no nos importa el orden que ocupen los elementos.



TreeSet<E> (interface Set<E>)

Es bastante más lento que HashSet. Esta implementación almacena los elementos ordenándolos en función de un criterio de comparación. Para ello necesita que su clase genérica tenga implementada la interface Comparable o utilizar un objeto Comparator en su constructor.



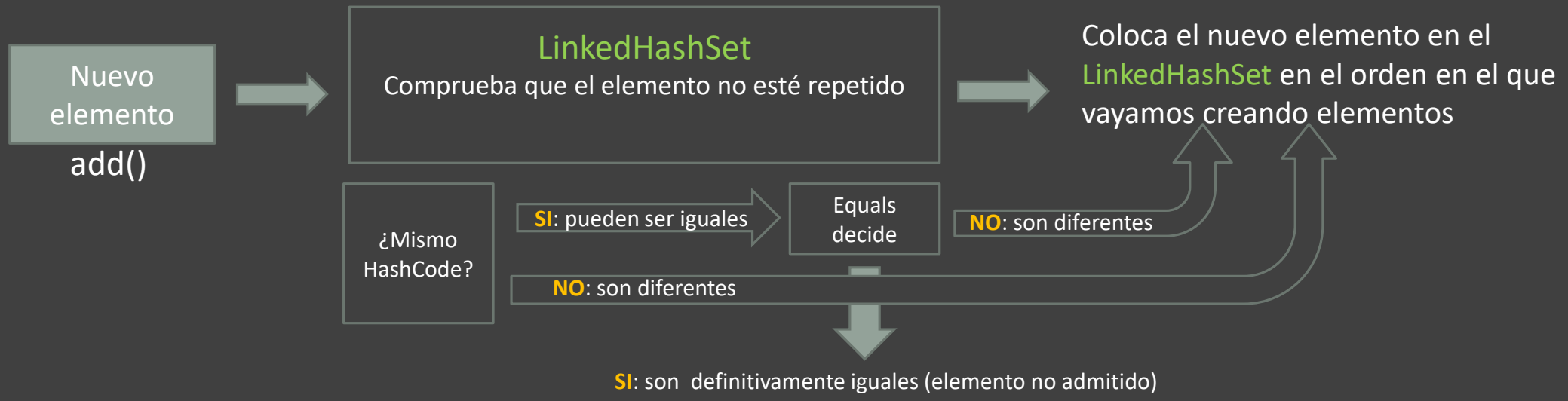
Coloca el nuevo elemento ordenadamente en el TreeSet en función de la implementación en su genérico del método compareTo de la interfaz Comparable o del método compare de un objeto Comparator utilizado en su constructor

En un Set no ordenado (HashSet o LinkedHashSet), el criterio de igualdad para los nuevos elementos candidatos a ingresar en él ser realiza en torno a equals y hashCode, pero en un Set ordenado (TreeSet) éstas comparaciones son realizadas utilizando su método compareTo de la interface Comparable o el método compare de la interface Comparator, por lo que dos elementos que se consideran iguales mediante estos métodos son, desde el punto de vista del Set Ordenado, iguales.

LinkedHashSet<E> (interface Set<E>)

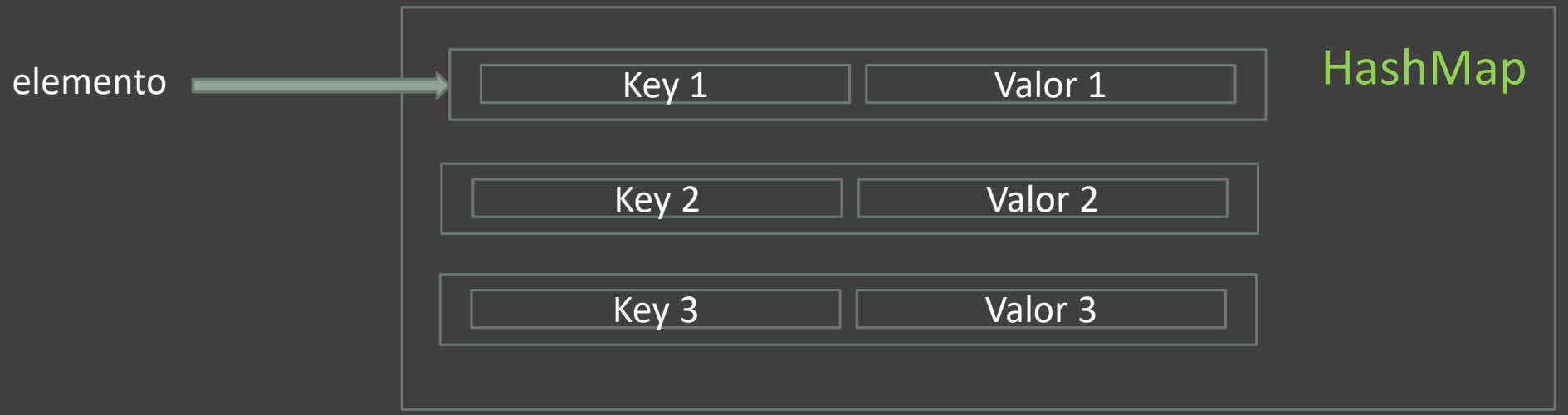
Esta implementación almacena los elementos en función del orden de inserción.

Es, simplemente, un poco más costosa que HashSet.



La interface Map<K,V>

La Interface Map, nos permite representar una estructura de datos para almacenar pares "clave/valor:

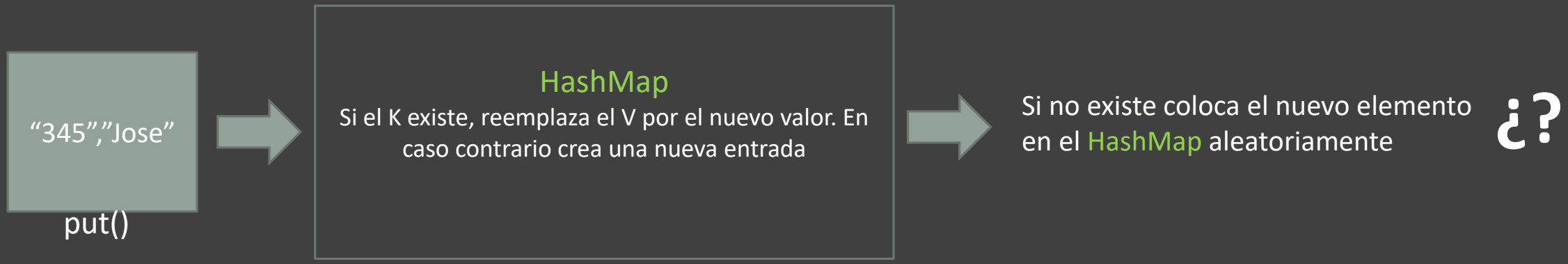


Un Map no permite claves duplicadas y solo puede existir un valor por clave.

HashMap<K,V> (interface Map<K,V>)

Esta implementación almacena las claves en una tabla hash.

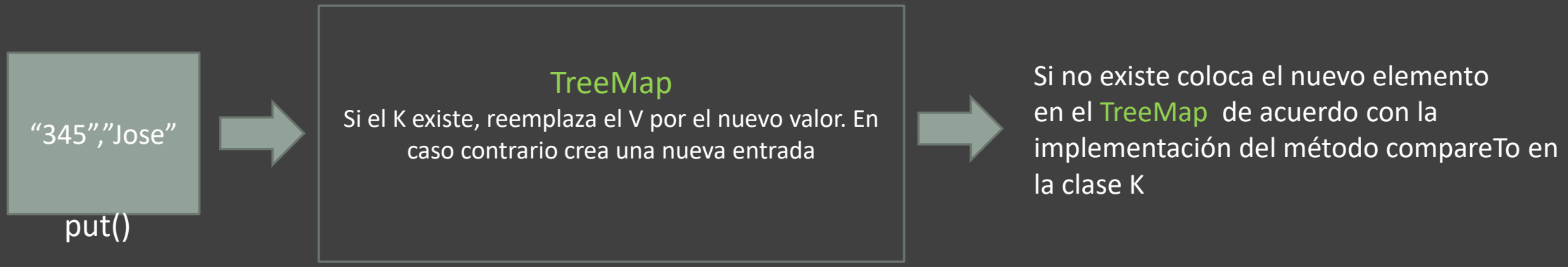
Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones.



TreeMap<K,V> (interface Map<K,V>)

Esta implementación almacena las claves ordenándolas en función de sus valores.

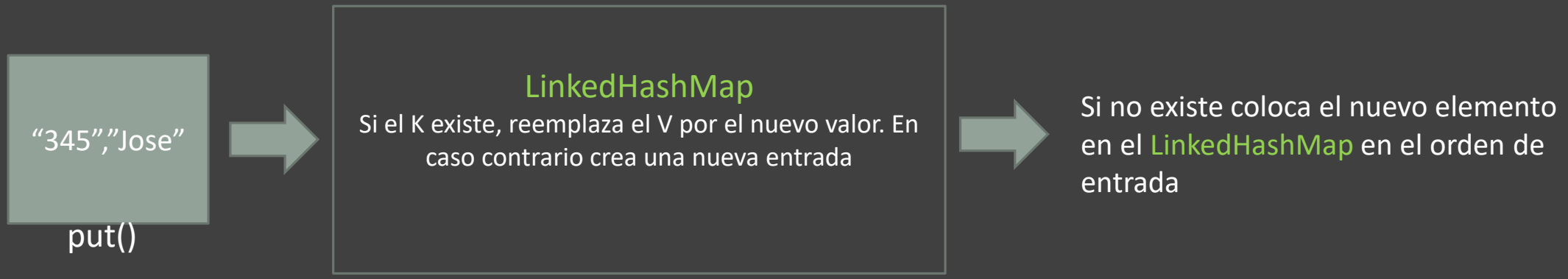
Es bastante más lento que HashMap. Las claves almacenadas (K) deben implementar la interfaz Comparable con su único método compareTo para que se pueda realizar la ordenación.



LinkedHashMap<K,V> (interface Map<K,V>)

Esta implementación almacena las claves en función del orden de inserción.

Es, simplemente, un poco más costosa que HashMap.



Métodos comunes de Collection<E> I

Método	Descripción	Nota	Devuelve
add(E)	Agrega un elemento a la colección	Devuelve false si no se pudo agregar el elemento por cualquier motivo	boolean
addAll(Collection)	Agrega una colección completa		boolean
clear()	Elimina todos los elementos de la colección		void
contains(E)	Comprueba si un objeto existe dentro de la colección	Devuelve true si la colección contiene el objeto que pasamos como parámetro utilizando el método equals	boolean

Métodos comunes de Collection<E> II

Método	Descripción	Nota	Devuelve
isEmpty()	Comprueba si la colección está vacía	Devuelve true si la colección está vacía	boolean
iterator()	Crea un objeto Iterator que nos permitirá movernos a través de los elementos		Iterator
remove(E)	Elimina de la colección el objeto pasado como argumento si este existe	Devuelve true si el elemento se ha eliminado	boolean
removeAll(Collection)	Elimina todos los elementos contenidos en la colección que se pasa como argumento	Devuelve true consigue eliminar al menos un elemento	boolean

Métodos comunes de Collection<E> III

Método	Descripción	Nota	Devuelve
retainAll(Collection)	Mantiene solamente los elementos contenidos en la colección que se pasa como argumento	Devuelve true si se ha conseguido hacer algún cambio	boolean
size()	Nos informa del número de elementos que componen la colección		int
toArray()	Nos permite pasar una colección al tipo Array		Object[]

Métodos adicionales de los LinkedList<E>

Método	Descripción	Devuelve
addFirst(E)	Inserta el elemento especificado al principio de la lista	void
addLast(E)	Inserta el elemento especificado al final de la lista	void
getFirst()	Devuelve el primer elemento de la lista	E
getLast()	Devuelve el último elemento de la lista	E
removeFirst()	Elimina y devuelve el primer elemento de la lista	E
revoveLast()	Elimina y devuelve el último elemento de la lista	E

Métodos comunes de Map<K,V> I

Método	Descripción	Nota	Devuelve
clear()	Elimina todos los elementos del Map		
get(K)	Devuelve el valor al que está asignada la clave especificada.	Devuelve null si el Map no contiene ninguna asignación para la clave.	V
isEmpty()	Devuelve true en caso de que el Map esté vacío		boolean
keySet()	Devuelve un Set con los keys contenidos en el Map		Set<K>
put(K,V)	Si existe la K, sustituye el valor de la V para esa K. En caso contrario, crea una nueva entrada (un nuevo elemento)	Sirve para crear un elemento dentro del Map o para sustituir el valor de un elemento teniendo en cuenta su K.	V

Métodos comunes de Map<K,V> II

Método	Descripción	Nota	Devuelve
remove(K)	En caso de existir, elimina el elemento del Map coincidente con el Key especificado		V
replace(K,V)	Sustituye la entrada de la clave especificada sólo si está asignada actualmente a algún valor.		V
size()	Devuelve el número de elementos contenidos en el Map		int
values()	Devuelve uns Collection con los values contenidos en el Map		Collection<V>

¿Qué es una tabla Hash? I

Las tablas hash son uno de los mecanismos más utilizados en el desarrollo de aplicaciones.

Existen múltiples librerías en casi todos los lenguajes de programación que proporcionan implementaciones muy eficientes de estas tablas (por ejemplo la clase java.util.Hashtable de las librerías del lenguaje Java).

La implementación de una tabla hash está basada en los siguientes elementos:

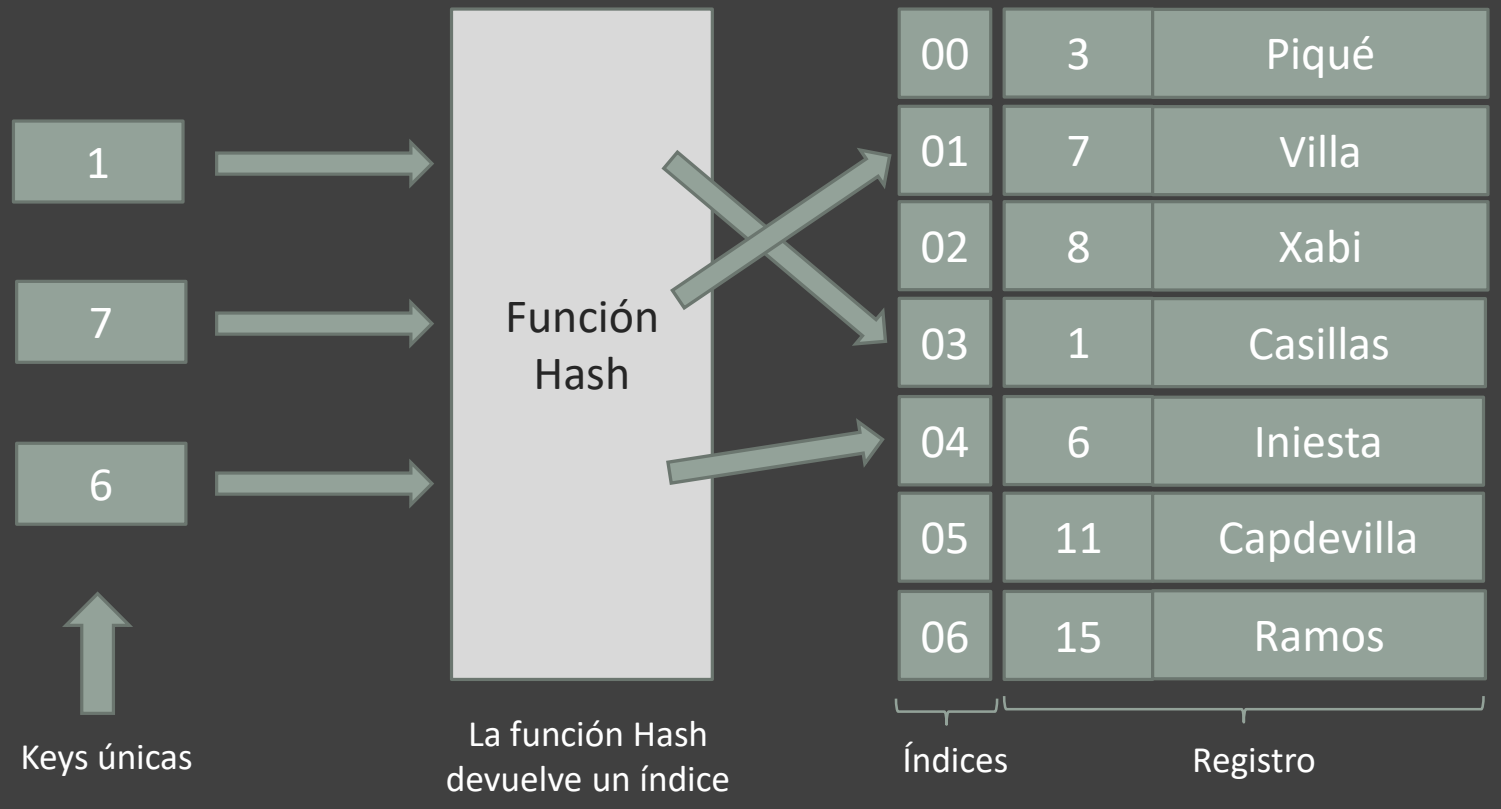
1 - Una tabla para almacenar los pares (clave, valor).

2 - Una función “hash” que recibe la clave y devuelve un índice para acceder a una posición de la tabla.

3 - Un procedimiento para tratar los casos en los que la función anterior devuelve el mismo índice para dos claves distintas. Esta situación se conoce con el nombre de colisión.

Las posibles implementaciones de cada uno de estos tres elementos se traducen en una infinidad de formas de implementar una tabla hash: HashSet, LinkedHashSet, HashMap, LinkedHashMap…

¿Qué es una tabla Hash? II



Índice (Key) de una tabla Hash

La norma es: utiliza siempre como Key objetos INMUTABLES

Si fuera mutable, entonces el valor de hashcode() o la condición de equals() podrían cambiar, y nunca podrías recuperar la clave de tu HashMap.

Más concretamente, los campos de clase que se utilizan para calcular equals() y hashcode() deben ser inmutables.

compareTo (Comparable) y compare (Comparator)

Cómo usarlo

compare(a, b) compara los objetos a y b. Si el valor devuelto es < 0, a es menor que b. Si es 0, ambos objetos son iguales. Si es superior a 0, a es mayor que b.

a.compareTo(b) también compara los objetos a y b. El valor de retorno funciona de la misma manera que para compare. Sin embargo, al llamarlo hay que tener cuidado de que a no sea nulo, de lo contrario lanza una NullPointerException.

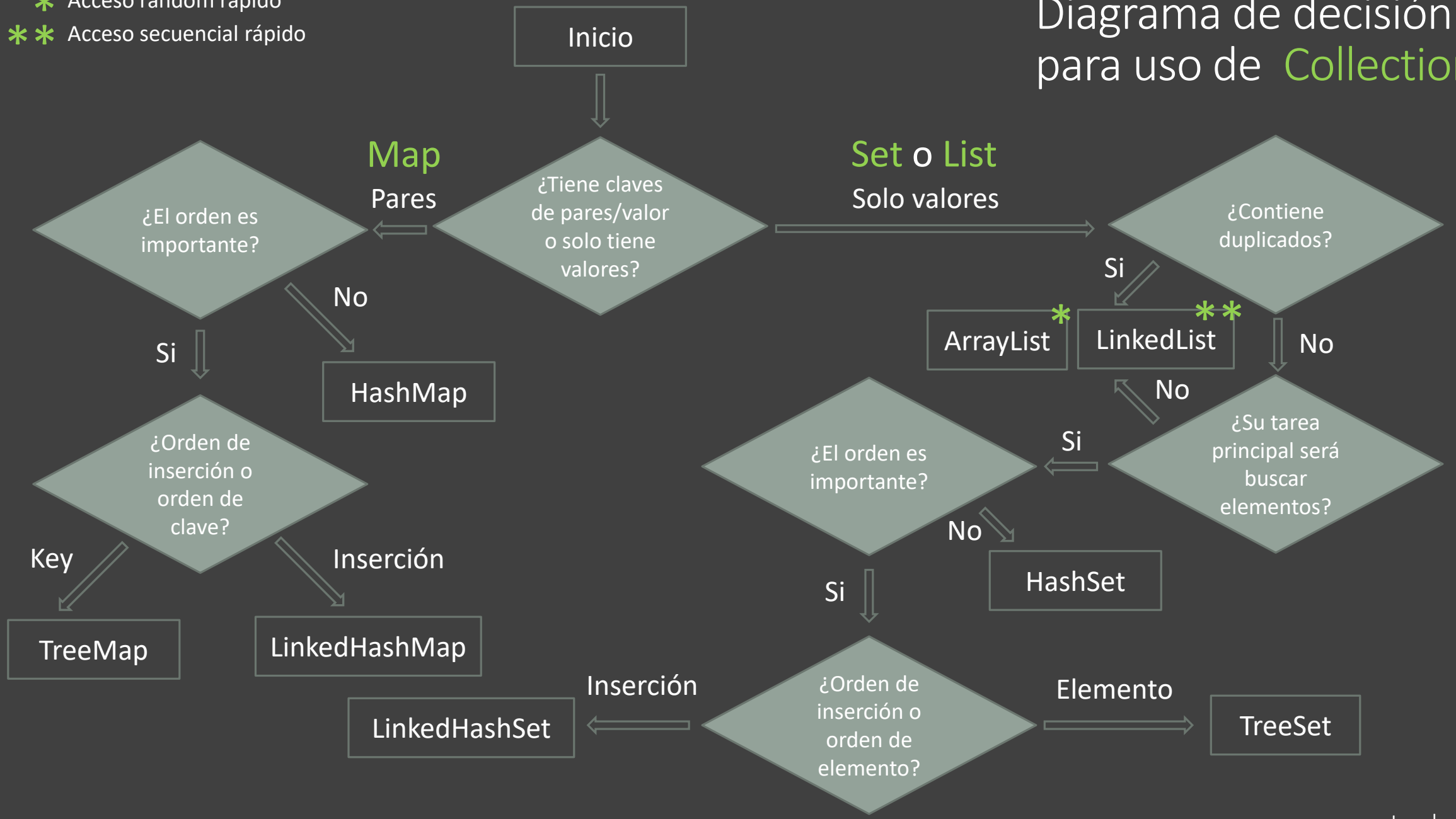
Cúando usarlo

Si la clase sólo necesita un esquema de comparación, usar únicamente compareTo es lo correcto.

Sin embargo, si necesitamos ordenar una clase de varias maneras, por ejemplo ordenarla por edad y en otro por nombre, aquí es donde Comparator se vuelve útil. Podemos crear dos clases que implementen la interface Comparator<LaClase>, uno que ordene por edad y otro por nombre. Cuando utilicemos sort pasaremos como argumento una instancia de una de estas clases para obtener la ordenación deseada.

Diagrama de decisión para uso de Collections

* Acceso random rápido
** Acceso secuencial rápido



Joselu Cantero