

# UNIDAD 11

---

# COLECCIONES EN JAVA

1. Colecciones en Java
2. List
3. Map
4. Set
5. Stack
6. Queue
7. PriorityQueue
8. Prioridad en PriorityQueue: Comparable
9. ¿Cómo elegir la colección adecuada?
10. Iterando sobre colecciones
11. Colecciones dentro de colecciones



Las colecciones son simplemente grupos de objetos, que se almacenan y agrupan de una determinada manera. La interfaz genérica **Collection** sirve para esto en Java.

Las colecciones de Java se encuentran en el paquete `java.util`, y son:

- Listas: `ArrayList`, `LinkedList`, `Vector`
- Conjuntos: `HashSet`, `TreeSet`, `LinkedHashSet`
- Mapas: `HashMap`, `TreeMap`, `LinkedHashMap`
- Pilas: `Stack`
- Colas: `Queue`, `PriorityQueue`





Las listas en Java heredan de la interfaz **List**. Pueden declararse como ArrayList, LinkedList o Vector, con o sin parámetros <T>. Estos tres tipos de listas tienen una implementación diferente y están pensadas para cosas diferentes:

ArrayList y Vector tienen en realidad la misma implementación interna, pero Vector está preparada para el trabajo concurrente, además de ser por lo general más eficiente que ArrayList. Su idea se basa en un conjunto de posiciones contiguas en memoria en las que guardar objetos:



A diferencia de los Arrays de Java, a los que se parecen mucho, no indicamos de forma predefinida el tamaño obligatoriamente, y este además es dinámico: Cambia conforme nosotros insertamos elementos.

- Si al insertar no tiene espacio, se reserva dinámicamente un 50% más.
- Al borrar un elemento, el tamaño nunca cambia.

--	--	--	--	--	--

### Ventajas:

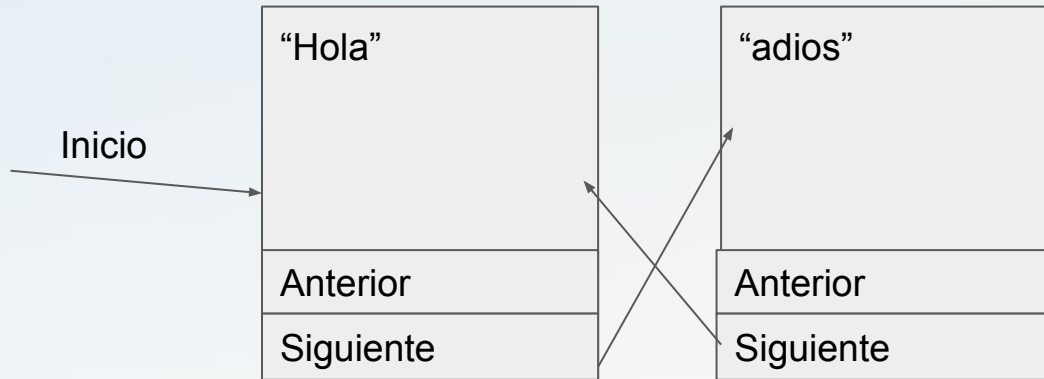
- Acceso aleatorio muy rápido:  
 $\text{PosicionBuscada} * \text{TamañoDeCadaElemento}$
- Inserción al final muy rápida
- Borrado al final muy rápido
- Recorrido en orden de índice rápido

### Desventajas:

- Inserción y borrado al principio o en el medio muy lentos
- Ordenación muy lenta
- Búsqueda muy lenta

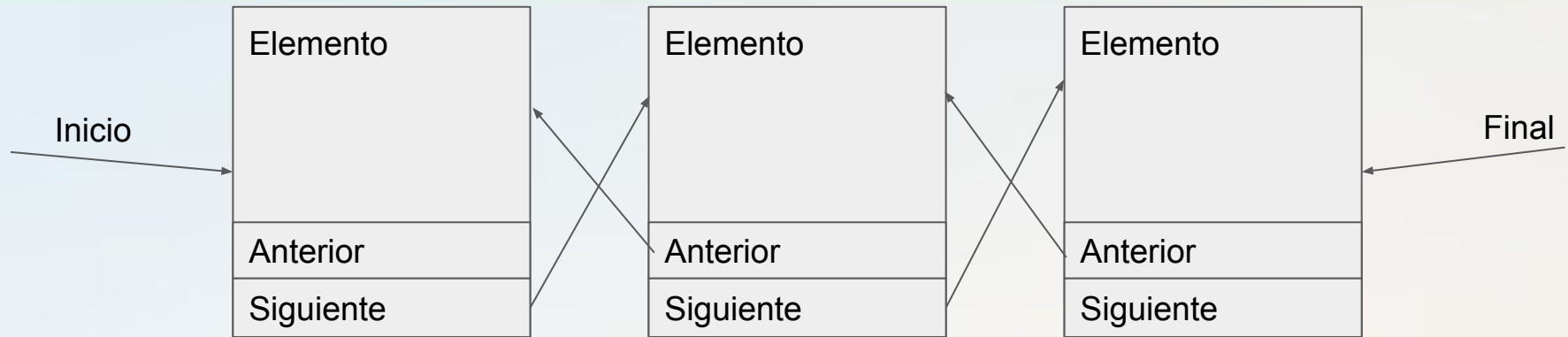


LinkedList tiene otro funcionamiento, como lista doblemente enlazada. En esto intervienen punteros. Su estructura es esta:



En este caso, los objetos no tienen que estar consecutivamente en memoria, y se ocupa algo más de espacio para los punteros que apuntan al anterior y al siguiente.

En esta implementación el acceso aleatorio mediante un cálculo no es posible, pero la inserción y el borrado en medio son más baratos que en ArrayList.



### Ventajas:

- Inserción en cualquier posición muy rápida, sobre todo al principio.
- Borrado en cualquier posición es rápido, sobre todo al principio.

### Desventajas:

- Recorrido muy lento, lo que lastra a casi todas sus ventajas, salvo las que afectan al inicio de la lista.

Ejemplo: <https://visualgo.net/en/list>



**I'm the map,**

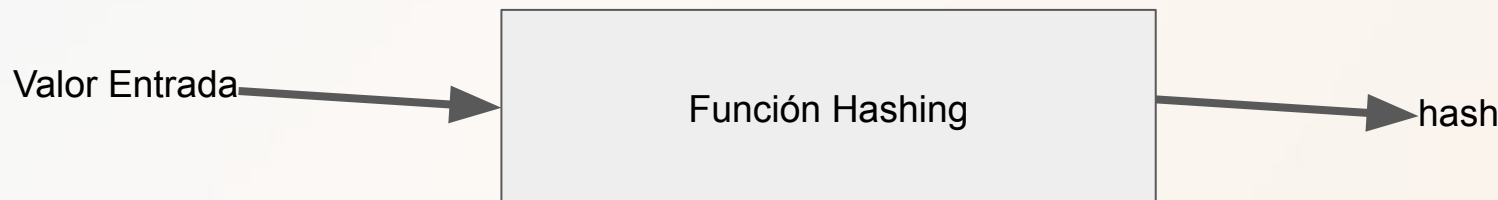
Los Mapas en Java heredan de la interfaz **Map**. Pueden declararse como **HashMap**, **TreeMap** o **LinkedHashMap**, con o sin parámetros  $\langle K, V \rangle$ .

Esta colección es distinta a todas las demás, porque tiene dos parámetros. El primero K, representa una clave, y el segundo V, un valor.

Los mapas sirven para encontrar objetos complejos a partir de identificadores simples, para mantener una ordenación y lograr una eficiencia en la búsqueda mayor que si buscamos un valor determinado en un `arrayList`.

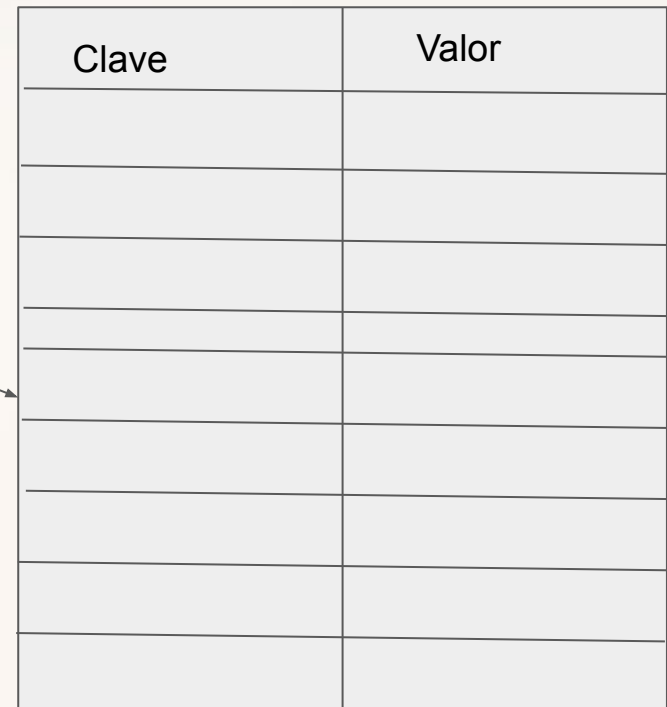
La primera de las implementaciones, Hashmap, se puede representar conceptualmente como una tabla con dos columnas: Clave y valor. En ella, los elementos no se insertan en orden, sino que se insertan en la posición que indica el método `hashCode()` en Java.

El **Hashing** es la operación de generar valores únicos (aunque puede haber colisiones) llamados **hash** para cada dato de entrada, a través de operaciones matemáticas más o menos complejas. Siempre se devuelve el mismo hash para el mismo valor de entrada, y está implementado en la clase `Object`, por lo que está disponible en todas las clases, aunque en los tipos básicos, como `int` o `char`, es necesario declarar las variables como `Integer` o `Character` para poder usarlas.

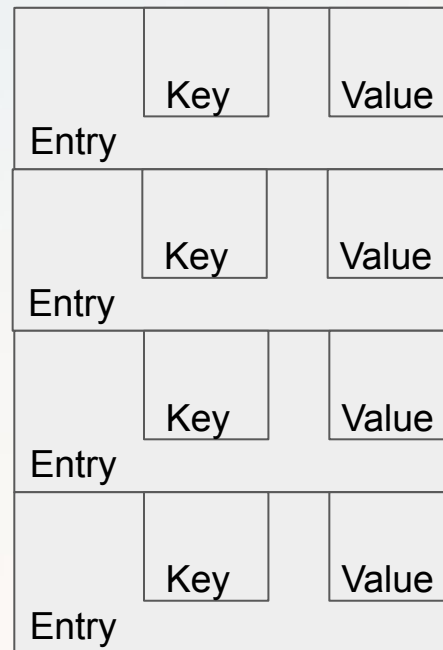




A partir del resultado de esa función (que no tenemos que llamar explícitamente, se llama automáticamente al insertar/buscar/borrar) sobre la **clave**, el HashMap sabe en qué posición tiene que insertar el par clave-valor, de forma que al pedirle al hashmap recuperar el valor, solo tiene que rehacer el hash para la clave pedida, y conoce la posición donde estará el elemento.



**HashMap** implementa internamente un HashTable. Mete la clave en el espacio de memoria para Key, y el valor en el de Value. Este par clave-valor para cada objeto, se guarda dentro de un objeto de tipo `java.util.Entry`.



De este modo, buscamos rápidamente por una clave simple a través de su hash, y recuperamos un valor complejo en un tiempo muy rápido.

#### Ventajas:

- Comprobar existencia es muy rápido
- Inserción y borrado rápidos

#### Desventajas:

- No permite ordenación por orden de inserción
- No se garantiza el orden de los elementos

Ejemplo: <https://visualgo.net/en/hashtable>

Entry	Key	Value
Entry	Key	Value
Entry	Key	Value
Entry	Key	Value

**LinkedHashMap** tiene el mismo funcionamiento que HashMap, pero cada elemento cuenta con un puntero a siguiente y otro a anterior, para mantener el orden de inserción



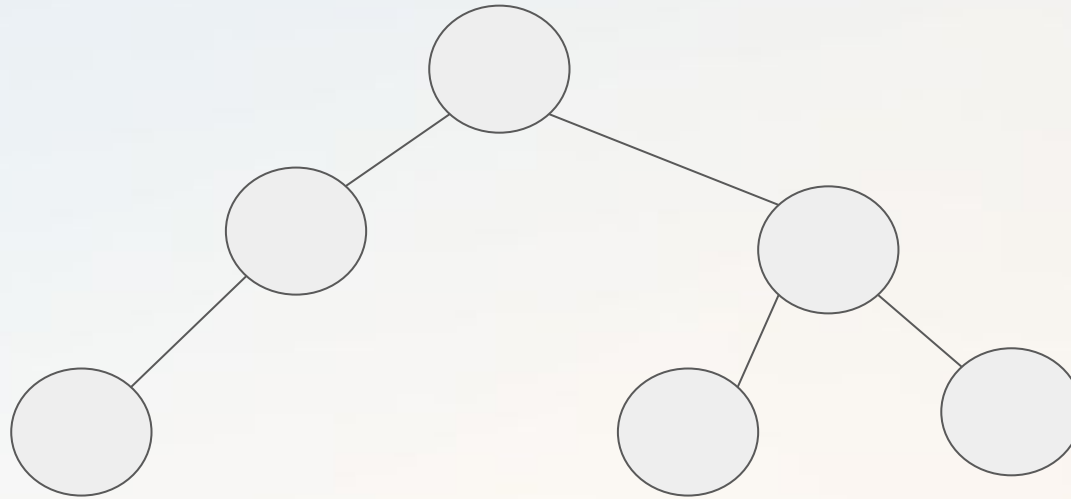
**Ventajas:**

- Comprobar existencia es muy rápido
- Permite recorrer la colección por orden de inserción

**Desventajas:**

- Inserción y borrado algo más lentos que en el hashmap

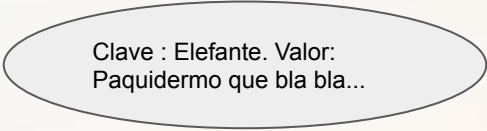
**TreeMap** funciona de forma diferente. En lugar de guardar los datos en forma de tabla, los guarda en una estructura llamada árbol binario. Tiene una forma como esta:



A esta estructura se le llama **árbol** porque tiene forma de árbol (si lo miramos boca abajo). Los círculos son lugares donde se colocan los datos, y se llaman **nodos**. El nodo que está arriba del todo se llama el **nodo raíz**, porque de él cuelgan todas las **ramas**, que son las conexiones entre un nodo y otro. A este árbol se le llama **binario**, porque cada nodo solo puede tener dos ramas: Un hijo izquierda, y un hijo derecha.



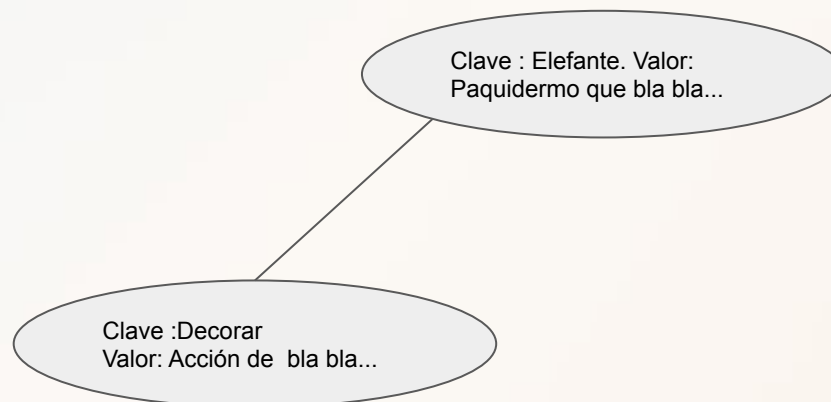
Conceptualmente (la implementación es más compleja), un treemap siempre empieza vacío, y si se mete el primer Entry (con clave y valor), siempre se introduce en la raíz. Por ejemplo, en un `TreeMap<String,String>` que representa un diccionario, donde la clave es la palabra, y el valor contiene toda su definición, insertar como primera palabra la palabra “Elefante”, nos dejaría un solo nodo raíz en la colección.

A light blue oval representing a root node in a treemap.

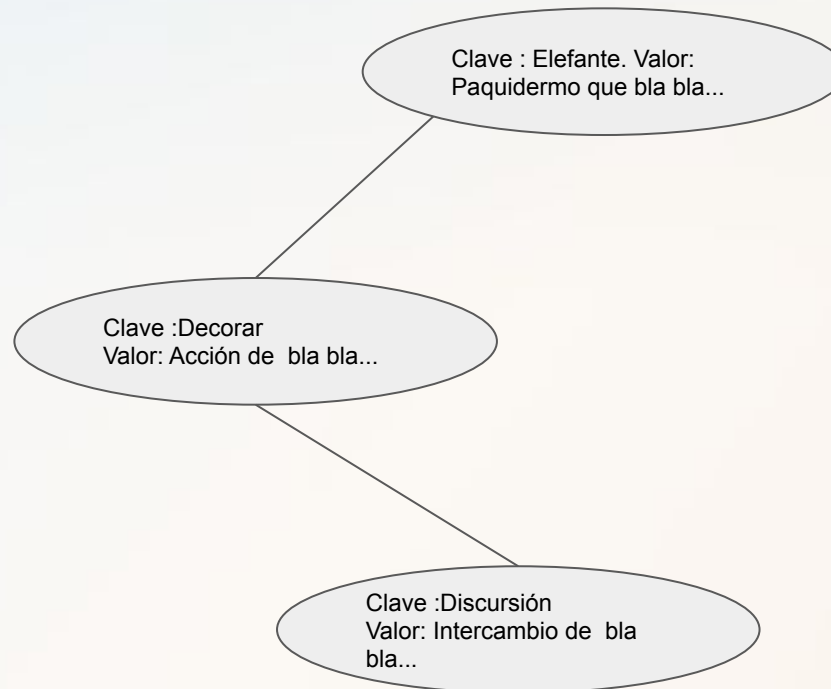
Clave : Elefante. Valor:  
Paquidermo que bla bla...

Los datos que se inserten después se insertarán siempre como nodos hijos. Depende del valor de la clave el lugar en que se inserten. Siempre se empieza por la raíz, y se inserta en el primer hueco que quede libre: como hijo izquierda si el valor de la clave es más pequeño que el del nodo actual que se evalúa, y como hijo derecha si la clave es más grande.

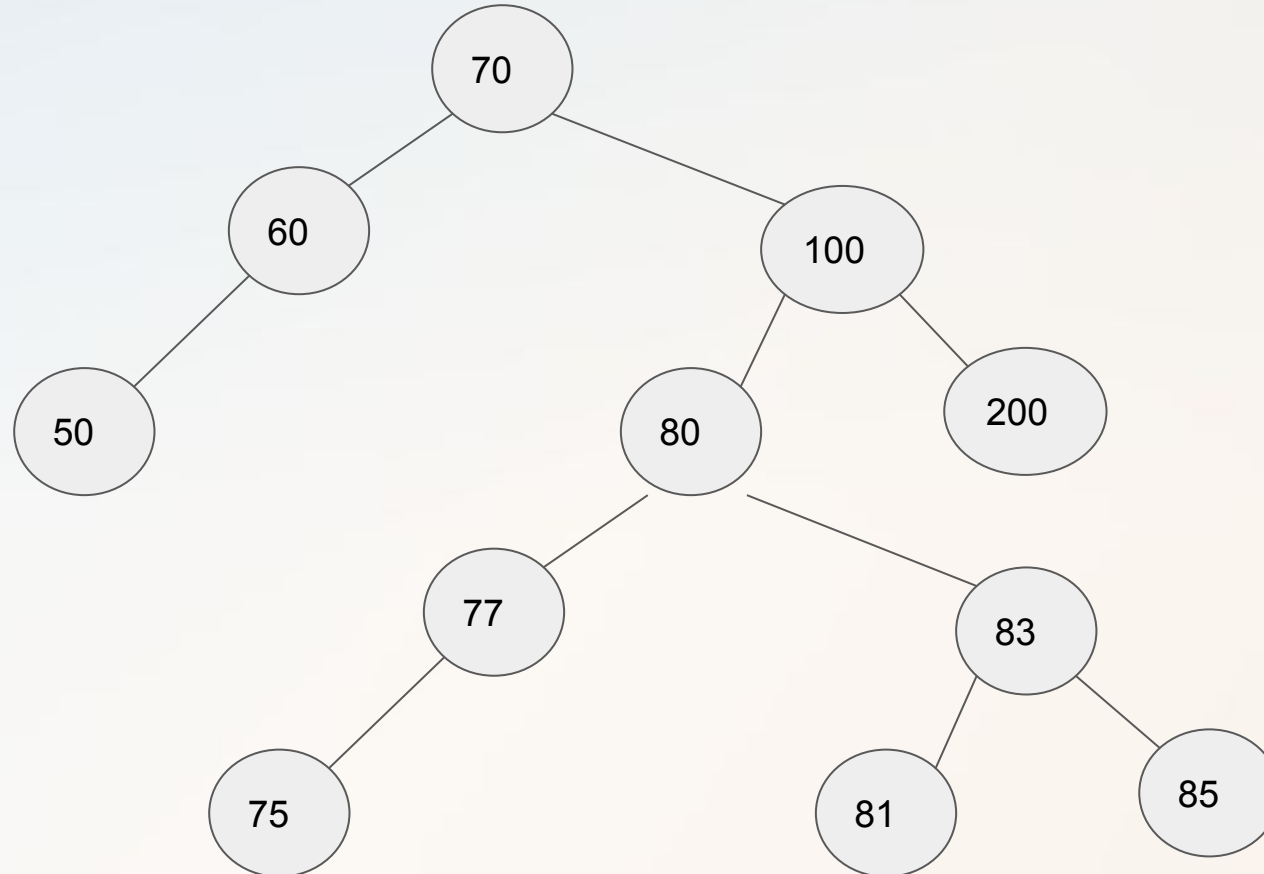
Por ejemplo, para insertar la palabra “decorar”, nos vamos al nodo raíz, evaluamos que decorar es más pequeño alfabéticamente que elefante, y nos vamos a la rama izquierda, como no tiene hijos, insertamos ahí.



Si ahora queremos insertar discursión, desde la raíz evaluamos que es más pequeño que elefante. Por tanto va a la izquierda. Como a la izquierda de elefante ya hay un nodo (decorar), evaluamos con este. Discursión es más grande alfabéticamente que decorar, por lo que debe ir a su derecha. Como es un hueco libre, ahí lo insertamos.



Vamos a cambiar de ejemplo a un árbol de enteros, fijándonos solo en la clave, para entender cómo se hace el borrado:



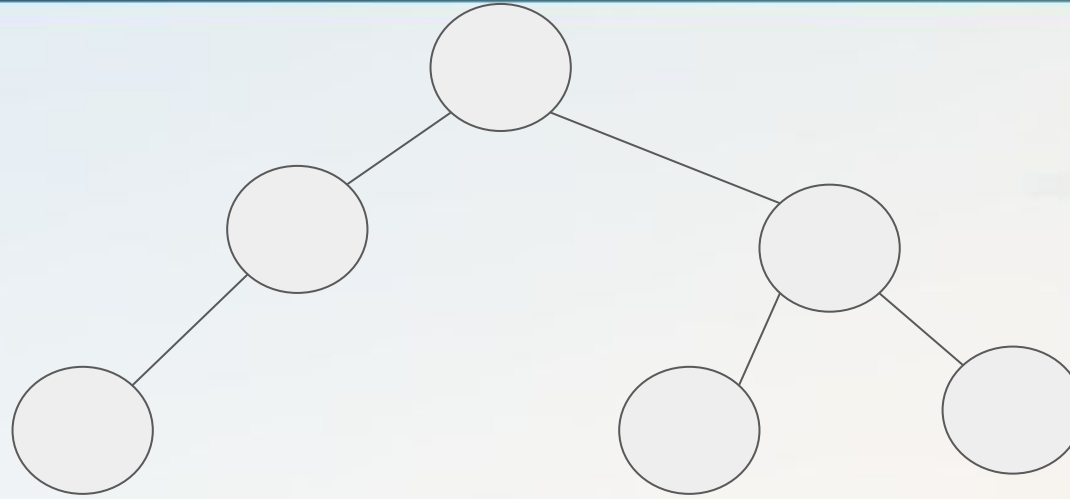
En los árboles binarios, lo normal es que al borrar:

- si es un nodo **hoja** (no tiene hijos) se borre directamente
- Si no es hoja:
  - Si tiene hijos a la derecha, sustituir el que se borra por el descendiente (hoja) más ala izquierda de su hijo derecha.
  - Si no tiene hijos a la derecha, se sustituye el que se borra por su hijo directamente a la izquierda.



- Si queremos borrar el 80, se cambiará por el 81, que es el descendiente más a la izda de su hijo dcha.
- Si queremos borrar el 200, se borraré directamente, al ser una hoja.
- Si queremos borrar el 60, al tener solo hijos a la izquierda, se sustituirá por su hijo inmediato a la izquierda, el 50.

Podéis practicar todo esto en vivo en: <https://visualgo.net/en/bst>



### Ventajas:

- La ordenación por clave es automática: No tenemos más que recorrerlo y ya estará ordenado

### Desventajas:

- Búsqueda, inserción y borrado más lentos que en el resto de Map.

Ejemplo: <https://visualgo.net/en/bst>



**SQUAD GOALS**

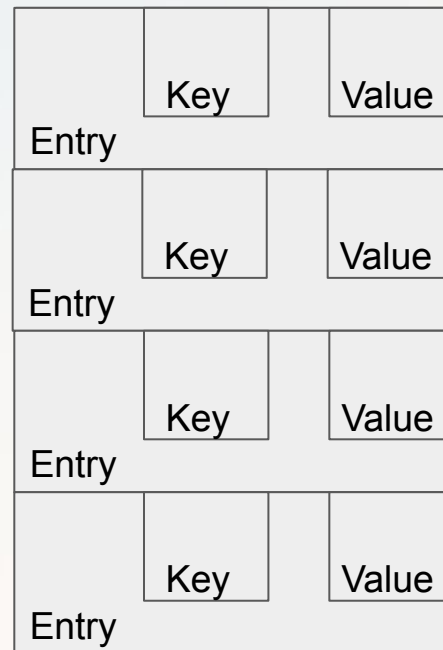
Los conjuntos en Java heredan de la interfaz **Set**. Pueden declararse como HashSet, TreeSet o LinkedHashSet, con o sin parámetros <T>.

Aunque la implementación en las tres es diferente, la idea subyacente es la misma: **un contenedor donde pueden almacenarse elementos no repetidos:**



En java hay un comportamiento no estándar: Machaca los elementos repetidos en los conjuntos, en lugar de dar error.

**HashSet** implementa internamente un HashTable. La diferencia es que la clave es el objeto a insertar, y el valor es una constante llamada PRESENT.



La desventaja de esto es que la clave no es un objeto simple, y se “Desperdicia” la variable value de cada Entry, pero a cambio comprobar existencia es muy rápido.



## 4 - Conjuntos

### Ventajas:

- Comprobar existencia es muy rápido
- Inserción y borrado rápidos

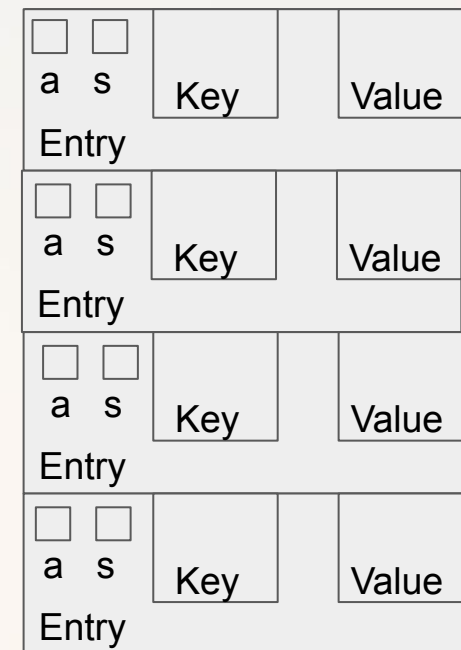
### Desventajas:

- No permite ordenación por orden de inserción
- No se garantiza el orden de los elementos
- No devuelve el objeto con una función get, solo indica si existe o no existe. Para recuperar objetos hay que iterar.

	Key		Value
Entry			
	Key		Value
Entry			
	Key		Value
Entry			
	Key		Value
Entry			

Ejemplo: <https://visualgo.net/en/hashtable>

**LinkedHashSet** tiene el mismo funcionamiento que **HashSet**, pero cada elemento cuenta con un puntero a siguiente y otro a anterior, para mantener el **orden de inserción**



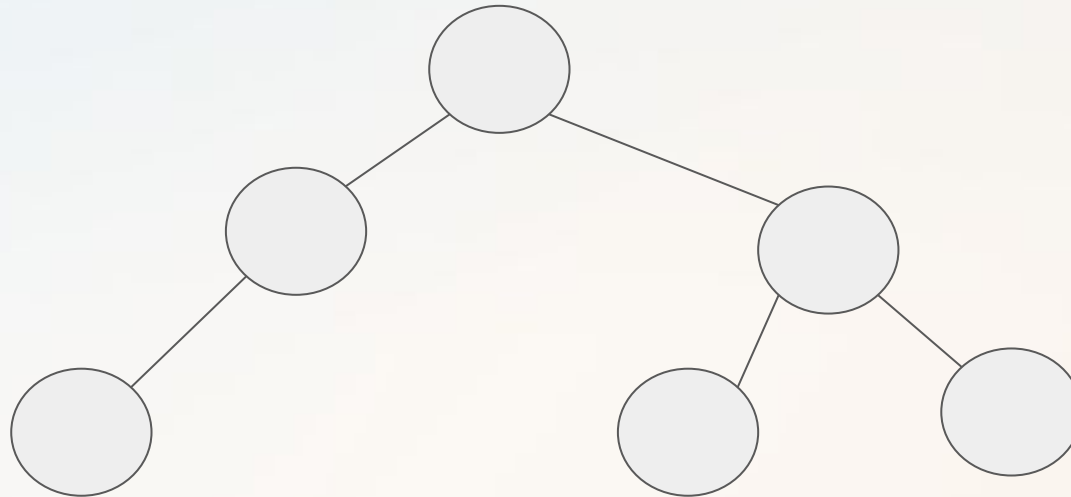
### Ventajas:

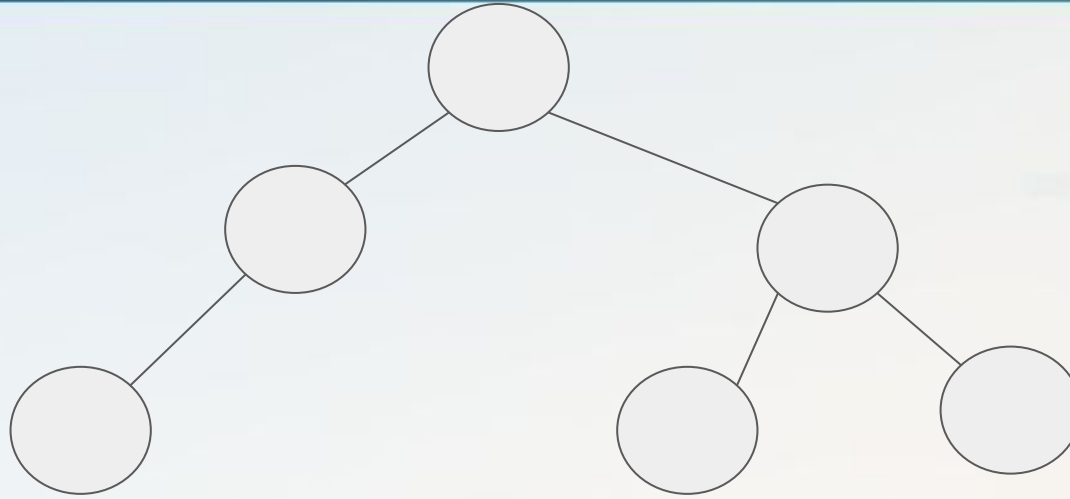
- Comprobar existencia es muy rápido
- Inserción y borrado rápidos
- Recorrido por orden de inserción

### Desventajas:

- No devuelve el objeto con una función **get**, solo indica si existe o no existe. Para recuperar objetos hay que iterar.

**TreeSet** tiene el mismo funcionamiento que TreeMap. De hecho, internamente es un TreeMap : implementa un árbol binario de búsqueda. La clave es el objeto a insertar, y el valor una constante `PRESENT`. Cada uno de los nodos es un `Entry` de Map.





### Ventajas:

- Permite recorrer el Set ordenadamente
- Comprobación de existencia razonablemente rápida

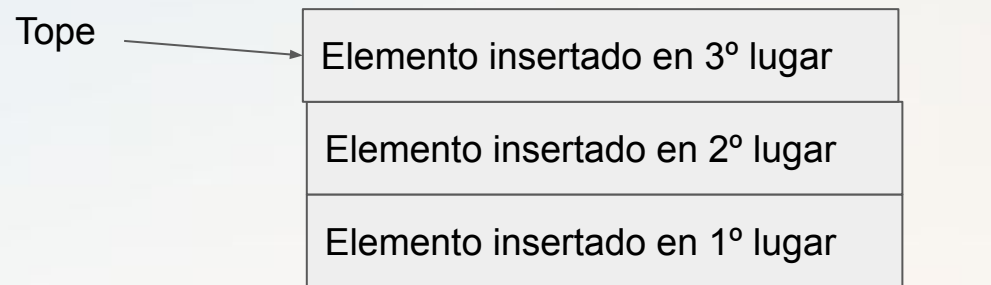
### Desventajas:

- No permite ordenación por orden de inserción
- Inserción, borrado, y comprobación de existencia algo más lentos que en HashSet.

Ejemplo: <https://visualgo.net/en/bst>

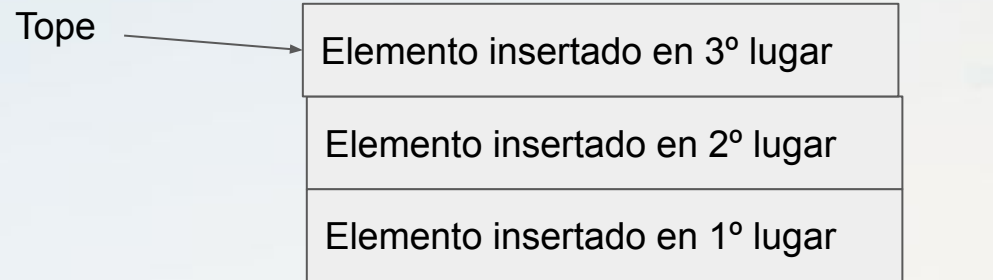


Las pilas o **Stack** son estructuras LIFO (Last In First Out), que no se pueden recorrer, y que solo permiten insertar y borrar del tope. Internamente implementan un LinkedList, por lo que las pocas operaciones que realiza, las realiza muy rápido.



Sus principales métodos son:

- `add(Object o)` Añade un elemento al tope
- `peek()` devuelve el elemento en el tope sin sacarlo
- `pop()` devuelve el elemento en el tope y lo saca



### Ventajas:

- Muy rápida en lo que hace

### Desventajas:

- Solo sirve para lo que sirve



Este es un ejemplo sencillo de uso de Stack y sus principales funciones:

```
Stack<Integer> pila=new Stack<Integer>();  
    pila.add(64);  
    pila.add(53);  
    pila.add(89);  
    pila.add(-12);  
    pila.add(0);  
  
    //pop devuelve el último elemento y lo saca  
    System.out.println(pila.pop());  
    //peek devuelve el último elemento pero no lo saca  
    System.out.println(pila.peek());
```

```
//Recorrido con iterator (La recorre al revés).  
//Si solo la vamos a recorrer así, mejor usar ArrayList.  
Iterator it=pila.iterator();  
while(it.hasNext()) {  
    int actual=(int)it.next();  
    System.out.print(actual+"\t");  
}
```

```
System.out.println();
```

recoremos

```
//estilo LIFO que es como está pensada  
while(!pila.isEmpty()) {  
    int tope=pila.pop();  
    System.out.print(tope+"\t");  
}
```



Las colas o **Queue** son estructuras FIFO (First In First Out), que no se pueden recorrer, y que solo permiten insertar y borrar del tope, que en este caso es el inicio de la cola. Internamente implementan un LinkedList, por lo que las pocas operaciones que realiza, las realiza muy rápido.



De hecho, no tiene implementación propia: Hay que instanciarla con un linkedList, y la responsabilidad de usarla como una estructura FIFO es nuestra.

**Ventajas:**

- Muy rápida en lo que hace

**Desventajas:**

- Solo sirve para lo que sirve
- Insertar (o borrar) al final de la cola es costoso (no pasa en todos los lenguajes)
- Podemos darle un uso no adecuado a LinkedList.

Este es un ejemplo sencillo de uso de Queue y sus principales funciones:

```
Queue<Integer> cola=new LinkedList<Integer>();  
    cola.add(64);  
    cola.add(53);  
    cola.add(89);  
    cola.add(-12);  
    cola.add(0);  
  
    //poll devuelve el primer elemento y lo saca  
    System.out.println(cola.poll());  
    //peek devuelve el primer elemento pero no lo saca  
    System.out.println(cola.peek());
```

```
//Recorrido con iterator
    Iterator it=cola.iterator();
    while(it.hasNext()) {
        int actual=(int)it.next();
        System.out.print(actual+"\t");
    }

    System.out.println();
    //Recorrido al estilo cola pura: Sacando elementos conforme
recorremos
    //estilo FIFO que es como está pensada
    while(!cola.isEmpty()) {
        int tope=cola.poll();
        System.out.print(tope+"\t");
    }
```





Las colas con prioridad o **PriorityQueue** son lo mismo que las colas, pero se diferencian en dos cosas: Java sí trae implementación propia, y la inserción no es en orden de llegada: se insertan antes o después en la cola según su prioridad.



### Ventajas:

- Muy rápida en lo que hace
- Permite insertar con prioridad

### Desventajas:

- Solo sirve para lo que sirve
- La inserción es menos eficiente que en las Stack

Este es un ejemplo sencillo de uso de Queue y sus principales funciones:

```
PriorityQueue<Integer> cola=new PriorityQueue<Integer>();  
    cola.add(64);  
    cola.add(53);  
    cola.add(89);  
    cola.add(-12);  
    cola.add(0);  
  
    //poll devuelve el primer elemento y lo saca  
    System.out.println(cola.poll());  
    //peek devuelve el primer elemento pero no lo saca  
    System.out.println(cola.peek());
```

```
//Recorrido con iterator
    Iterator it=cola.iterator();
    while(it.hasNext()) {
        int actual=(int)it.next();
        System.out.print(actual+"\t");
    }

    System.out.println();
    //Recorrido al estilo cola pura: Sacando elementos conforme
recorremos
    //estilo FIFO que es como está pensada
    while(!cola.isEmpty()) {
        int tope=cola.poll();
        System.out.print(tope+"\t");
    }
```



Las interfaz **Comparable** y la clase **Comparator** sirven para ordenar colecciones:

- Comparable define un orden natural para las clases que la implementan. Pertenece a java.lang
- Comparator se emplea para especificar la relación de orden o anular el orden natural. Pertenece a java.util

Dato: Podemos ordenar cualquier lista de elementos con el método **Collections.sort**;

**Comparable** permite definir el orden natural de los objetos de cualquier clase. Se pueden ordenar la colecciones que contienen objetos de clases que implementan la interfaz Comparable. Algunos ejemplos de clases que ya lo hacen son Long, String, Date, Float.

Para escribir tipos Comparable personalizados, tenemos que implementar el método `compareTo` de la interfaz Comparable. Ejemplo:

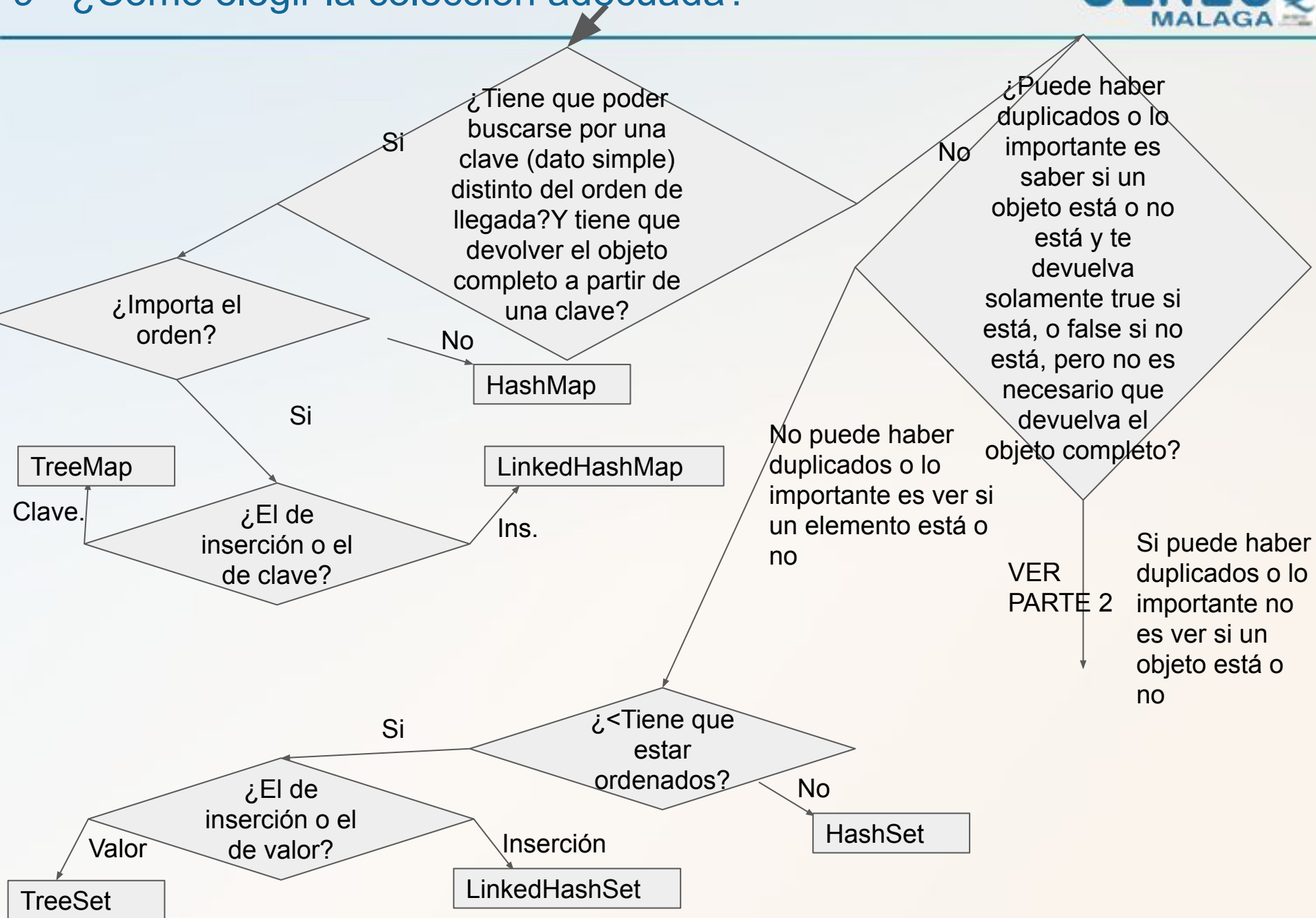
```
class Usuario implements Comparable{
    String nombre;
    String dni;
    ...
    public int compareTo(Object o){
        if(dni < ((Usuario)o).dni){
            return -1; //Si el objeto this es menor.
        }
        if(dni > ((Usuario)o).dni){
            return 1; //Si el objeto this es mayor.
        }
        return 0; //Si son iguales.
    }
}
```



## 9 - ¿Cómo elegir la colección adecuada?

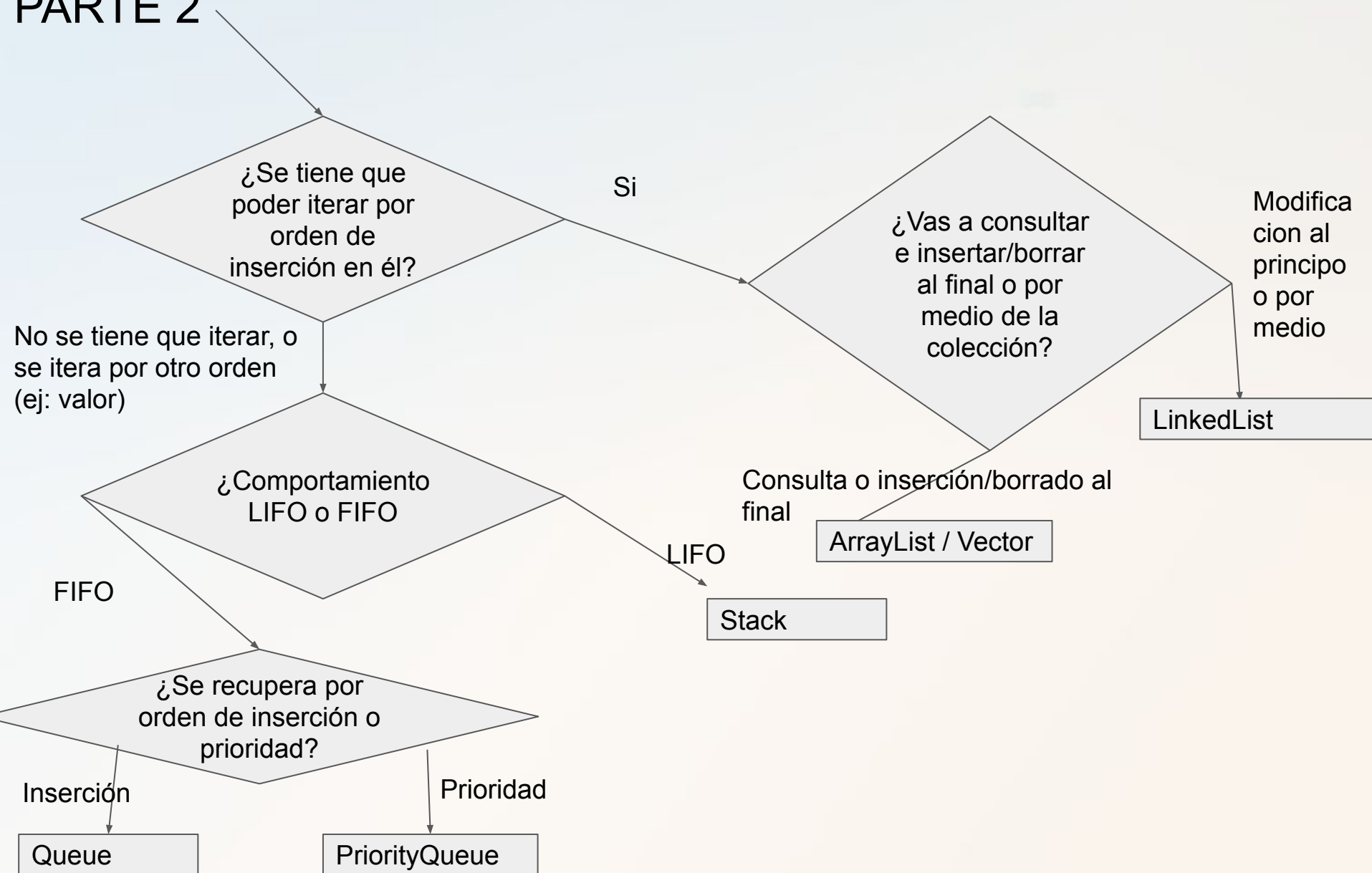


# 9 - ¿Cómo elegir la colección adecuada?



# 9 - ¿Cómo elegir la colección adecuada?

## PARTE 2





Recorrer las tres clases que implementan List es muy fácil: Se recorren exactamente igual que los arrays, recordando que en lugar de usar [], usamos el operador get, y en lugar de .length, .size().

```
for(int i=0;i<miArray.size();i++){  
    System.out.println(miArray.get(i));  
}
```

Pero en el resto de colecciones no tenemos esa facilidad. Tanto para las listas como para el resto de colecciones, si que tenemos disponible la clase `java.util.Iterator`. OJO: ¡Si importas otra clase Iterator de otro paquete, no va a funcionar la iteración!

Ejemplo para recorrer los set:

```
HashSet<Integer> hs=new HashSet<Integer>();  
    Iterator it=hs.iterator();  
    while(it.hasNext()) {  
        int actual=(Integer)it.next();  
        System.out.println(actual);  
    }
```

Hay que convertir el valor que llega de `it.next()` al valor del que hemos declarado el Set. En este caso a `integer`, porque hemos declarado un `new HashSet<Integer>()`.



Ejemplo para recorrer los mapas por clave:

```
HashMap<Integer,String> map=new  
HashMap<Integer,String>();  
    Iterator it=map.keySet().iterator();  
    while(it.hasNext()) {  
        int actual=(Integer)it.next();  
        System.out.println(actual+" "+map.get(actual));  
    }
```

De nuevo, convertimos lo que viene de next() a Integer, porque es la clave, lo que estamos recorriendo al usar keySet();



Ejemplo para recorrer los mapas por valor:

```
HashMap<Integer,String> map=new HashMap<Integer,String>();  
    Iterator it=map.values().iterator();  
    while(it.hasNext()) {  
        String actual=(String)it.next();  
        System.out.println(actual);  
    }
```

En este caso, de `it.next()` nos viene un `String` porque es el tipo indicado para el valor, y estamos recorriendo `values()`.

Por último, la función `entrySet` nos devuelve objetos de la clase `java.util.map.Entry`, que tiene una función para obtener la clave, y otra para obtener el valor, cada una de su respectivo tipo de dato.

```
HashMap<Integer,String> map=new HashMap<Integer,String>();  
    Iterator it=map.entrySet().iterator();  
    while(it.hasNext()) {  
        Entry actual=(Entry)it.next();  
        System.out.println((Integer)actual.getKey()+"  
"+(String)actual.getValue());  
    }
```



A veces, en algunas colecciones, sobre todo en los map, queremos que una misma clave apunte a varios elementos. Pero el comportamiento por defecto de los map es que si introduzco una clave que ya existía, el nuevo valor que tiene asociado, machaca al valor anterior.

Ej:

```
mapa.put("12345678A",lolaMento); //lolaMento es de tipo Persona  
mapa.put("12345678A",hugoFirst); //hugoFirst es de tipo Persona
```

Aquí, después de ejecutar ambas líneas, solo queda hugoFirst, lolaMento ha sido machacada.

La solución es que cada entrada del mapa no sea un elemento, sino una colección de elementos. Lo más sencillo es ArrayList, pero podríamos tomar otra decisión si por ejemplo necesitamos hacer búsquedas entre esos elementos con la misma clave, o que no se repitan, u ordenarlos...

```
HashMap<LocalDate,ArrayList<Persona>>();
```

Con esto tendremos un HashMap donde se puedan meter a varias personas que tengan la misma fecha de nacimiento.

La forma de inicializar cambia, porque al ser los valores una colección, también es posible que de un nullpointer. Entonces cada vez que insertemos una clave, la primera vez que la insertemos, tenemos que inicializar la colección subyacente.

Lo más ordenado y lo que queda mejor, es programarse una clase para esto. Y si le ponemos genéricos, mejor :D

```
public class HashMapArray<K,V> {  
    private HashMap<K, ArrayList<V>> mapaArray;  
  
    /**  
     * El constructor inicializa el HashMap para evitar NullPointers  
     */  
    public HashMapArray() {  
        this.mapaArray = new HashMap<K, ArrayList<V>>();  
    }  
  
    public ArrayList<V> get(K fNacimiento) {  
        return mapaArray.get(fNacimiento);  
    }  
}
```



# 11 - Colecciones dentro de colecciones

```
/**
```

```
* Intenta borrar una V, devuelve false si la V no existe en la
* colección
*
* @param p V a eliminar
* @return true si la V se ha encontrado y eliminado, false en caso
* contrario
*/
```

```
public boolean delete(K c, V p) {
    ArrayList<V> arr = mapaArray.get(c);
    for (int i = 0; i < arr.size(); i++) {
        if (arr.get(i).equals(p)) {
            mapaArray.get(c).remove(i);
            //POSIBLE MEJORA: SI LA V QUE HEMOS BORRADO ERA LA
            //ÚLTIMA DEL
            //ARRAYLIST, PODEMOS BORRAR EL ARRAYLIST.
            return true;
        }
    }
    return false;
}
```

```
public boolean delete(K fNacimiento) {  
    return mapaArray.remove(fNacimiento) != null;  
}
```

```
public void insert(K c,V p) {  
    // la Clave del mapa (fNacimiento) no existía antes  
    if (mapaArray.get(c) == null) {  
        mapaArray.put(c, new ArrayList<V>());  
    }  
    // ya había antes otra V insertada con esa clave  
    // POSIBLE MEJORA: SI QUEREMOS QUE LAS VS NO PUEDAN  
    REPETIRSE...  
    // QUE SE PUEDA REPETIR LA K PERO NO LA V  
    // EN VEZ DE ARRAYLIST USO HASHSET  
    mapaArray.get(c).add(p);  
}  
}
```

Con esto, podemos hacer un uso de HashMapArray con cualquier clave y cualquier valor. El main podría tener algo como:

```
HashMapArray<LocalDate,Persona> mapa=new  
HashMapArray<LocalDate,Persona>();
```

```
    mapa.insert(LocalDate.of(2001,11,1),new  
Persona("12345778C","Lola","Mento",LocalDate.of(2001,11,1)));  
    mapa.insert(LocalDate.of(2001,11,1),new  
Persona("56545778D","Hugo","First",LocalDate.of(2001,11,1)));  
    mapa.insert(LocalDate.of(1991,1,6),new  
Persona("22312464Q","Paco","Mer",LocalDate.of(1991,1,6)));
```

```
System.out.println(mapa.get(LocalDate.of(2001,11,1)));  
System.out.println(mapa.get(LocalDate.of(1991,1,6)));
```

```
mapa.delete(LocalDate.of(2001,11,1));
```