Otra posibilidad es utilizar un objeto **Comparator<E>**. Esta es otra interfaz que define el método **compare** al que se le pasan los dos objetos a comparar y cuyo resultado es como el de **compareTo** (0 si son iguales, positivo si el primero es mayor y negativo si el segundo es mayor).

Para definir un comparador de esta forma hay que crear una clase que implemente esta interfaz y definir el método **compare**; después crear un objeto de ese tipo y usarle en la construcción del árbol. Ejemplo (comparable al anterior):

```
public class ComparadorAlumnos implements
Comparator<Alumno>{
   public int compare(Alumno o1, Alumno o2) {
          int comparacion=o1.nombre.compareToIgnoreCase(o2.nombre);
          if(comparacion==0){
                return (int)(o1.nota-o2.nota);
          }
          else return comparacion;
}
```

Después al construir la lista se usa el comparador como parámetro en el constructor:

```
TreeSet<Alumno> tre=
          new TreeSet<Alumno>(new ComparadorAlumnos());
```

Normalmente se usa más la interfaz **Comparable**, sin embargo los comparadores de tipo **Comparator** permite ordenar de diferentes formas, por eso en la práctica se utilizan mucho. En el caso de definir la lista con un objeto **Comparator**, esa será la forma prioritaria para ordenar la lista, por encima del método **compareTo** de la interfaz **Comparable**.

Nota: El método **sort** de la clase **Arrays** también admite indicar un comparador para saber de qué forma deseamos ordenar el array.

(9.8) mapas

(9.8.1) introducción a los mapas

Las colecciones de tipo Set tienen el inconveniente de tener que almacenar una copia exacta del elemento a buscar. Sin embargo en la práctica es habitual que haya datos que se consideran clave, es decir que identifican a cada objeto (el dni de las personas por ejemplo) de tal manera que se buscan los datos en base a esa clave y por otro lado se almacenan los datos normales.

Los mapas permiten definir colecciones de elementos que poseen pares de datos clave-valor. Esto se utiliza para localizar valores en función de la clave que poseen. Son muy interesantes y rápidos.

(9.8.2) interfaz Map<K,V>

Es la raíz de todas las clases capaces de implementar mapas. Hasta la versión 1.5, los mapas eran colecciones de pares clave, valor donde tanto la clave como el valor eran de tipo **Object**. Desde la versión 1.5 esta interfaz tiene dos genéricos: **K** para el tipo de datos de la clave y **V** para el tipo de los valores.

Esta interfaz no deriva de **Collection** por lo que no usa iteradores ni ninguno de los métodos vistos anteriormente. La razón es que la obtención, búsqueda y borrado de elementos se hace de manera muy distinta. Los mapas no permiten insertar objetos nulos (provocan excepciones de tipo **NullPointerException**). **Map** define estos métodos:

Método	uso
V **get**(K clave)	Devuelve el objeto que posee la clave indicada
V **put**(Object clave, V valor)	Coloca el par clave-valor en el mapa (asociando la clave a dicho valor). Si la clave ya existiera, sobrescribe el anterior valor y devuelve el objeto antiguo. Si esa clave no aparecía en la lista, devuelve **null**
V **remove**(Object clave)	Elimina de la lista el valor asociado a esa clave. Devuelve el valor que tuviera asociado esa clave o **null** si esa clave no existe en el mapa.
boolean **containsKey**(Object clave)	Indica si el mapa posee la clave señalada
boolean **containsValue**(Object valor)	Indica si el mapa posee el valor señalado
void **putAll**(Map<?extends K, extends V> mapa)	Añade todo el mapa indicado, al mapa actual
Set<K> **keySet**()	Obtiene un objeto *Set* creado a partir de las claves del mapa
Collection<V> **values**()	Obtiene la colección de valores del mapa, permite utilizar el **HashMap** como si fuera una lista normal al estilo de la clase **Collection** (por lo tanto se permite recorrer cada elemento de la lista con un iterador)
int **size**()	Devuelve el número de pares clave-valor del mapa
Set<Map.Entry <K,V>> **entrySet**()	Devuelve una lista formada por objetos **Map.Entry**
void **clear**()	Elimina todos los objetos del mapa

Las operaciones fundamentales son **get**, **put** y **remove**. El conjunto de claves no puede repetir la clave. Hay que tener en cuenta que las claves se almacenan en una tabla hash (es decir es una estructura de tipo **Set**) por lo que para detectar si una clave está repetida, la clase a la que pertenecen las claves del mapa deben definir (si no lo está ya) adecuadamentelos métodos **hashCode** y **equals**.

(9.8.3) interfaz Map.Entry<K,V>

La interfaz Map.Entry se define de forma interna a la interfaz Map y representa un objeto de par clave/valor. Es decir mediante esta interfaz podemos trabajar con una entrada del mapa. Tiene estos métodos:

método	uso
K getKey()	Obtiene la clave del elemento actual Map.Entry
V getValue()	Obtiene el valor
V setValue(V valor)	Cambia el valor y devuelve el valor anterior del objeto actual
boolean equals(Object obj)	Devuelve verdadero si el objeto es un Map.Entry cuyos pares clave-valor son iguales que los del Map.Entry actual

(9.8.4) clases de mapas

clase HashMap

Es la clase más utilizada para implementar mapas. Todo lo comentado con la interfaz Map funciona en la clase HashMap, no añade ningún método o forma de funcionar particular.
Ejemplo:

```
public static void main(String[] args) {
    Alumno a1=new Alumno("alberto",7);
    Alumno a2=new Alumno("mateo",8.5);
    Alumno a3=new Alumno("julián",7.2);
    Alumno a4=new Alumno("adrian",8);
    Alumno a5=new Alumno("alberto",7);
    Alumno a6=new Alumno("adrian",8);
    HashMap<Integer,Alumno> l=new HashMap<Integer,Alumno>();
    l.put(1,a1);
    l.put(2,a2);
    l.put(3,a3);
    l.put(4,a4);
    l.put(5,a5);
    l.put(6,a6);
    System.out.println(l.get(4));
    l.remove(4);
    System.out.println(l);
}
/*Sale:
    adrian-8.0
    {1=alberto-7.0, 2=mateo-8.5, 3=julián-7.2, 5=alberto-7.0, 6=adrian-8.0}
*/
```

Esta clase usa un mapa de dispersión en la tabla Hash que permite que los tiempos de respuesta de **get** y **put** se mantengan aun con muchísimos datos almacenados en el mapa.

Al igual que ocurre con la clase **HashSet**, el orden de entrada no está asegurado. Constructores

constructor	uso
HashMap()	Crea un mapa con una tabla Hash de tamaño 16 y una carga de 0,75
HashMap(Map<?extends K, ?extends V> m)	Crea un mapa colocando los elementos de otro
HashMap(int capacidad)	Crea un mapa con la capacidad indicada y una carga de 0,75
HashMap(int capacidad, float carga)	Crea un mapa con la capacidad y carga máxima indicada

clase LinkedHashMap<K,V>

Es derivada de la anterior, y la única diferencia es que esta clase sí respeta el orden en el que los objetos del mapa fueron insertados. También permite que el orden de los elementos se refiera al último acceso realizado en ellos. Los más recientemente accedidos aparecerán primero. Constructores:

constructor	uso
LinkedHashMap()	Crea un mapa con una tabla Hash de tamaño 16 y una carga de 0,75
LinkedHashMap(Map<?extends K, ?extends V> m)	Crea un mapa colocando los elementos de otro
LinkedHashMap(int capacidad)	Crea un mapa con la capacidad indicada y una carga de 0,75
LinkedHashMap(int capacidad, float carga)	Crea un mapa con la capacidad y carga máxima indicada
LinkedHashMap(int capacidad, float carga, boolean orden)	Crea un mapa con la capacidad y carga máxima indicada. Además si el último parámetro (*orden*) es verdadero el orden se realiza según se insertaron en la lista, de otro modo el orden es por el último acceso.

mapas ordenados. clase TreeMap

Se trata de una estructura de tipo árbol binario, que permite que los elementos del mapa se ordenan en sentido ascendente según la clave. En ese sentido es una clase muy parecida a **TreeSet**.

TreeMap implementa la interfaz **SortedMap** que, a su vez, es heredera de **Map**, por lo que todo lo dicho sobre los mapas funciona con las colecciones de tipo **TreeMap**.

Lo que aportan de nuevo es que los datos en el mapa se ordenan según la clave. Sólo eso y para ordenarlos (al igual que ocurre con la clase **TreeSet**, página 30) la clase de las claves tiene que implementar la interfaz

Comparable o bien durante la creación del TreeMap indicar un objeto Comparator.

Constructores:

constructor	uso
TreeMap()	Crea un mapa que utiliza el orden natural establecido en las claves
TreeMap(Comparator<? super K> m)	Crea un mapa colocando los elementos de otro
TreeMap(Map<?extends K, ?extends V> m)	Crea un mapa a partir de los elementos del mapa indicado, se usará el orden natural de sus claves
TreeMap(SortedSet<K, ?extends V> m)	Crea un mapa usando los elementos y orden de otro mapa

mapas hash débiles. clase WeakHashMap <V,K>

A veces ocurre que cuando se crea un mapa, hay objetos del mismo que no se usan. Es decir claves que nunca se utilizan. Por ello a veces interesaría que esos objetos se eliminaran.

La clase WeakHashMap realiza esa acción, elimina los elementos que no se usan de un mapa y hace que actúe sobre ellos el recolector de basura. Por eso es un mapa de claves débiles. El funcionamiento consiste en detectar las claves que no se han usado y según van apareciendo más, las que llevan más tiempo sin usarse acaban desapareciendo.

Evidentemente este juego es peligroso ya que muchas veces podemos pasar mucho tiempo sin usar una clave y luego necesitarla, por ello hay que usarla sólo con datos de los que estamos seguros que su falta de uso prolongada requiere su eliminación.

(9.9) la clase Collections

Hay una clase llamada Collections (no confundir con la interfaz Collection) que contiene numerosos métodos estáticos para usar con todo tipo de colecciones.

método	uso
static <T>boolean addAll(Collection<? super T> c, T ...elementos)	Añade la lista de elementos de tipo *T* (que irán separados por comas) a la colección indicada por *c*. Java 1.5
static <T> int binarySearch(List<? extends T> l, T valor, Compararator<? super T> c)	Busca de forma binaria el objeto en la lista que deberá estar ordenada según el comparador indicado.
static <T> int binarySearch(List<? extends Comparable <? super T>> l, T valor)	Busca de forma binaria el objeto en la lista que deberá estar ordenada (por ello los elementos de la lista deben de ser de una clase que implemente la interfaz Comparable)