Organización lógica de los datos. Estructuras dinámicas

1. Introducción

Tipos de datos simples o **primitivas**: no están compuestos de otras estructuras de datos. Cada variable o constante almacena un único valor. En Java disponemos de:

- int
- short
- long
- float
- double
- char
- boolean

Una **estructura de datos** es una colección de datos que pueden ser caracterizados por su organización y las operaciones que en ellos se definen.

Características:

Datos simples	- cada variable representa un elemento.
Estructura de datos	- cada variable representa un conjunto de ellos pudiendo ser referenciados individualmente.

Distinguimos dos clases de estructuras de datos:

a) ESTRUCTURAS FUNDAMENTALES O ESTÁTICAS

Se llaman fundamentales porque constituyen la base de estructuras más complejas. Y estáticas porque una vez definidas, no se puede alterar la dimensión de las ya existentes. Una de estas estructuras son los Arrays. En Java pueden ser dimensionados tanto en tiempo de compilación como en tiempo de ejecución, pero una vez asignado su tamaño, este no podrá ser alterado.

El primer ejemplo (notas) es un array dimensionado en tiempo de compilación. Una vez ejecutado el programa, su dimensión será siempre de 15 enteros. Mientras que en el segundo (otrasNotas), la dimensión del array es desconocida en tiempo de compilación. Hasta que el programa se ejecute y sea leída la variable tamanho, no se fijará la dimensión del array, pero una vez hecho, no podrá ser cambiada.

```
int[] notas = new int[15];

int tamanho = leeEntero();
int[] otrasNotas = new int[tamanho];
```

b) <u>ESTRUCTURAS DINÁMICAS O AVANZADAS</u>

Su característica fundamental es la de poder variar su tamaño después de la compilación del programa y durante la ejecución de este.

Las estructuras de datos dinámicas permiten saltarse las limitaciones inherentes a las estructuras estáticas, en cuanto al tamaño de memoria ocupada, que en estas últimas es fijo y debe ser predeterminado de antemano. Las dinámicas nos permiten variar el espacio reservado dinámicamente en función de las necesidades en tiempo de ejecución. Vamos a ver a continuación las más utilizadas y posteriormente veremos como implementarlas en Java gracia al framework **Collection**

2. Estructuras lineales

a. Listas

Secuencia de 0 o más elementos de un mismo tipo. Conjunto de valores de un mismo tipo que se encuentran ordenados y pueden variar en número.

a1 a2 ..... an

LISTA VACÍA si no contiene ningún elemento

LISTAS ORDENADAS

a1 a2 ..... an

tal que

a1 <= a2 <= ..... <= an

LISTAS ENLAZADAS. Las posiciones de memoria no tienen por qué ser adyacentes. Cada elemento contiene la posición del siguiente elemento. Cada elemento tiene al menos dos campos:

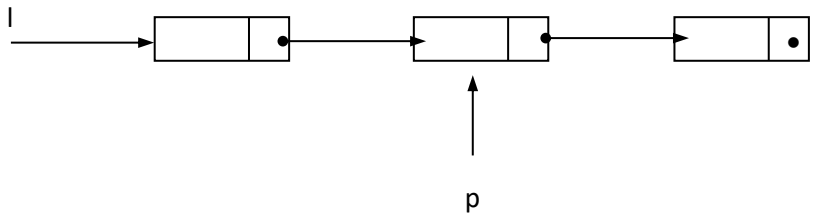
- un **campo de información**
- otro conteniendo un **enlace** al siguiente elemento, que normalmente se realizará mediante un puntero.

Puntero: variable cuyo valor es la dirección o posición de otra variable.

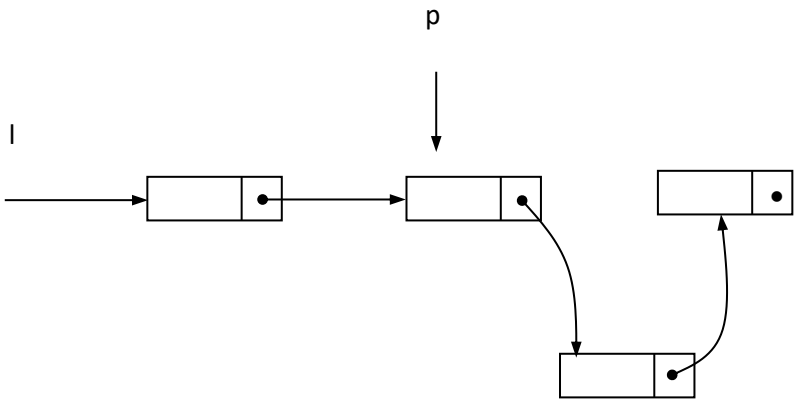
OPERACIONES:

- inserción
- eliminación
- localización
- recorrido
- clasificación
- unión

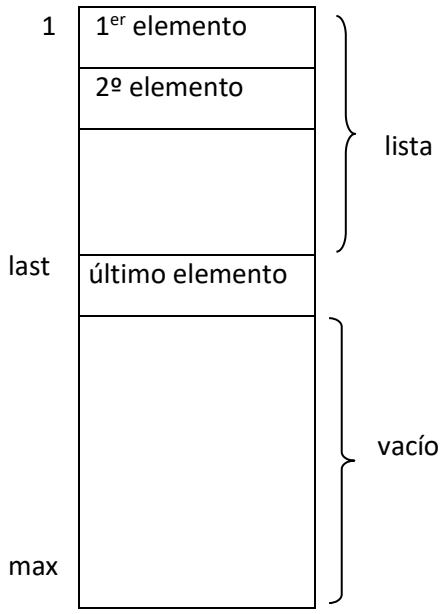
IMPLEMENTACIÓN CON CELDAS (punteros)



Ej. de una operación de inserción:



IMPLEMENTACIÓN CON ARRAYS



- con **arrays** se fija el número máximo de elementos
- con **arrays** operaciones de inserción y borrado son más costosas, implicando el desplazamiento de datos
- más cuidado con **celdas** para no perder información (celda no apuntada)
- **arrays**: se utiliza siempre todo el espacio
- **celdas**: cada elemento ocupa un espacio adicional para el puntero

b. PILAS

Caso especial de listas en que:

- Todas las inserciones y borrados se hacen al final de la lista.
- LIFO (last in first out) El último elemento que entra es el primero en salir

OPERACIONES

- push introducimos un elemento en la pila
- pop sacamos un elemento de la pila

c. COLAS

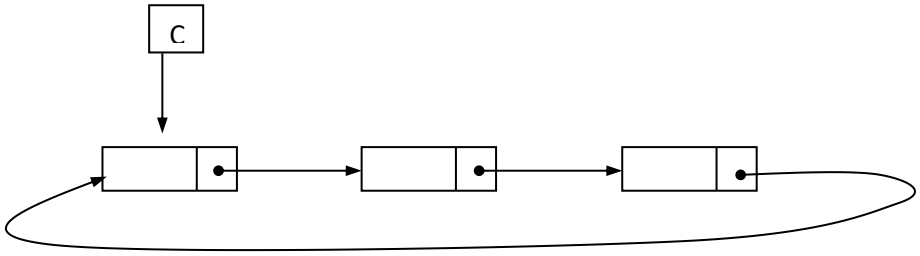
Caso especial de listas.

- Se inserta por un lado y se borra por otro.
- FIFO (first in first out) El primer elemento en entrar es el primero en salir.

d. LISTAS CIRCULARES

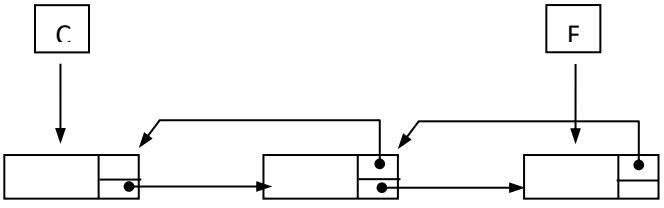
El último elemento apunta al primero.

Uno nodo cabecero: indica el principio de la lista



e. LISTAS DOBLEMENTE ENLAZADAS

Pueden recorrerse en ambas direcciones

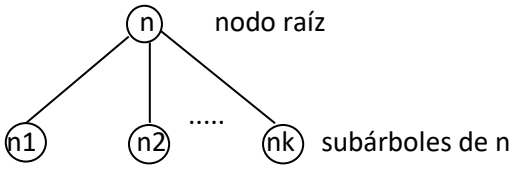


f. ÁRBOLES

Colección de nodos con una relación que jerarquiza su estructura (paternidad).

- Existe un nodo especial denominado raíz.
- los nodos restantes se dividen en m >=0 conjuntos disjuntos (A1 .... Am) cada uno de los cuales es a su vez un árbol (subárboles de la raíz) .

Representación:



ARBOL NULO:

Sin nodos

CAMINO

n1 n2 .... nk secuencia de nodos de un árbol T

ni es el padre de n_{i+1} (1 <= i <= k)

Si en T existe un camino del nodo 'a' al 'b' , se dice que 'b' es **descendiente** de 'a' y que 'a' es **ascendiente** de 'b'

HOJA

Nodo que no tiene descendientes

RAÍZ

Nodo que no tiene ascendientes

LONGITUD DEL CAMINO

nº de nodos de este (en este caso K)

ALTURA DE UN NODO

Longitud del camino más largo desde él a una hoja

ALTURA DE UN ARBOL

Altura de la raíz

PROFUNDIDAD DE UN NODO

Longitud del camino desde la raíz hasta él

FORMAS DE RECORRIDO

PREORDEN (**n** es la raíz)

n, Pre(T1), Pre(T2).....Pre(Tk)

POSTORDEN

Post(T1), Post(T2).....n

INORDEN

In(T1), n , In(T2),.... In(Tk)

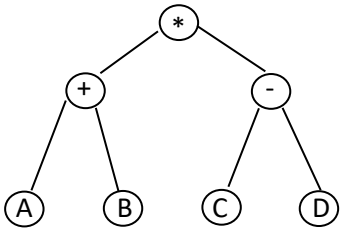
i. ÁRBOLES ETIQUETADOS

Los nodos tienen un valor o etiqueta

Ej . • hojas operandos

• nodos no hojas --> operadores

(A + B) * (C – D)



Preorden: * + A B - C D Notación prefijo

Postorden A B + C D - * Notación sufijo

Inorden A + B * C - D Notación infijo (sin ())

ii. ÁRBOLES N-ARIOS

Cada nodo puede tener de **0** a **n** hojas

iii. ÁRBOLES BINARIOS

Cada nodo a lo sumo **2** descendientes

Def recursiva: Un árbol binario puede ser:

1 un árbol vacío

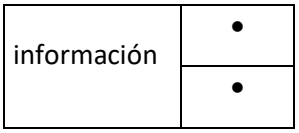
2 un nodo (la raíz)

3 un nodo y un árbol binario llamado izquierdo

4 un nodo y un árbol binario llamado derecho

5 un nodo y dos árboles binarios llamados izquierdo y derecho

Implementación con punteros. Cada elemento está compuesto por tres campos. Uno con la información, y otros para apuntar a los subárboles izquierdo y derecho respectivamente.

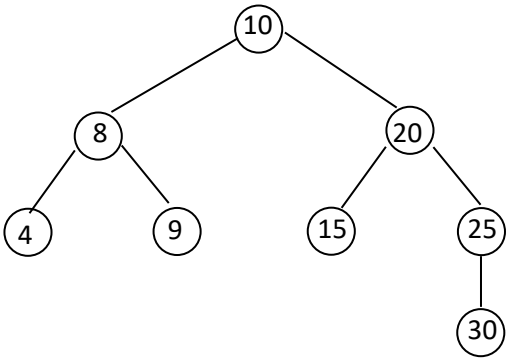


iv. ÁRBOLES BINARIOS DE BÚSQUEDA

Permiten representar conjuntos de elementos sobre los que está definido un orden lineal.

Todas las operaciones se pueden realizar en un tiempo medio de O (log n)

Tiene que cumplir que, para cada nodo, todos los elementos de subárbol izquierdo son menores que el contenido en ese nodo y todos los del derecho mayores



3. Estructuras no lineales

a. GRAFOS

Estructura **no lineal**. Conjunto de nodos (vértices) y conjunto de aristas (arcos) líneas que los unen.

SENCILLO

si no tiene lazos:
- no existe arco a sí mismo
- no existe más de un arco para unir dos nodos

CAMINO

Secuencia de 1 o más nodos que conectan 2 nodos.

LONGITUD DEL CAMINO

número de arcos

VÉRTICES ADYACENTES

Si los une un arco.

DIRIGIDOS O DIGRAFOS.

formados por un conjunto de vértices y un conjunto ordenado de pares de vértices

importa la dirección, que se representa gráficamente con puntas de flecha

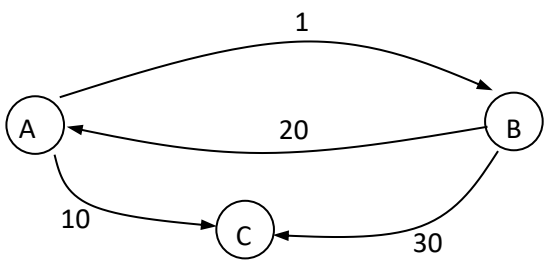
CONECTADOS

Siempre existe un camino entre dos nodos

ETIQUETADO

Los arcos tienen asociado un valor.

Ejemplo de un grafo dirigido, etiquetado y conectado. Podría representar puntos geográficos, caminos que los unen (en una o ambas direcciones) y el coste asociado a tomar cada uno de ellos (en cuanto a distancia, gasto en gasolina, peajes, etc.)



4. Utilización práctica de estructuras de datos

Veremos ahora, con ejemplos concretos, la utilización de estructuras de datos para la resolución de problemas en Java mediante la utilización de la Interfaces del framework Collection y clases que las implementan.

a. Listas

Disponemos de la Interface **List** con un par de implementaciones: **ArrayList**, basada en arrays, y **LinkedList**, basada en celdas. Cualquier aplicación en que necesitemos almacenar varios elementos de un mismo tipo es susceptible de implementarse mediante listas. Lo que hagamos con listas podríamos hacerlo igualmente con arrays, de hecho, ya hemos visto que una de las formas de implementar las listas es mediante arrays, pero con las ventajas de que todo el manejo será transparente para el programador. Con arrays si nos hemos quedado cortos, tendremos que crear uno nuevo de mayor tamaño y copiar a él todo el elemento del primero. Si eliminamos o insertamos un elemento, implicará el desplazamiento a derecha o izquierda respectivamente de todos los elementos a partir de la posición de inserción o borrado. Al hacerlo mediante listas, aunque la implantación subyacente sean arrays (ArrayList), todo ese proceso es transparente para el programador.

Vemos con un ejemplo como trabajar con una lista (ArrayList o LinkedList). Almacenaremos en la lista información sobre alumnos.

```
package ejemplo.arraylist;

import java.util.LinkedList;
import java.util.List;
import java.util.Objects;

public class EjemploArrayList {

// Clase alumno simplificada. Solo dos atributos: el nombre y la nota
// media incluimos un constructor, un toString y un equals para poder
// comprobar la igualdad entre elementos. En este caso consideramos que
// dos alumnos son iguales si tienen el mismo nombre.

  static private class Alumno {

    String nombre;
    int notaMedia;

    public Alumno(String nombre, int notaMedia) {
      this.nombre = nombre;
      this.notaMedia = notaMedia;
    }

    public Alumno() {
    }
```

```
    @Override
    public String toString() {
      return "Alumno{" + "nombre=" + nombre + ", notaMedia=" + notaMedia
+ '}';
    }

    @Override
    public int hashCode() {
      int hash = 3;
      hash = 67 * hash + Objects.hashCode(this.nombre);
      return hash;
    }

    public Alumno(String nombre) {
      this.nombre = nombre;
    }

    @Override
    public boolean equals(Object obj) {
      if (this == obj) {
        return true;
      }
      if (obj == null) {
        return false;
      }
      if (getClass() != obj.getClass()) {
        return false;
      }
      final Alumno other = (Alumno) obj;
      if (!Objects.equals(this.nombre, other.nombre)) {
        return false;
      }
      return true;
    }

  }

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {

// Creamos la lista. La siguiente es una forma de crear la lista vacía.
// Podemos crearla ya con datos utilizando:
//     List<Alumno> alumnos = List.of(--lista de elementos---);
// o   List<Alumno> alumnos = Arrays.asList(---lista de elementos ---);

    List<Alumno> alumnos = new LinkedList();

// añadimos elementos a la lista. El método add(elemento) añade el
// elemento al final de la lista para insertar en una posición
// determinada utilizaríamos: add(posición, elemento)

    alumnos.add(new Alumno("José", 7));
    alumnos.add(new Alumno("Belén", 5));
    alumnos.add(new Alumno("Adrián", 3));
    alumnos.add(new Alumno("Elena", 9));

// Vamos a recorrer la lista imprimiendo sus valores, de tres modos
// diferentes. El primero con un bucle for utilizando índices.
```

```
// Este método NO se debe utilizar nunca con LinkedList, ya que para
// cada índice necesita volver al comienzo de la lista y realizar una
// recorrido secuencial hasta el mismo. Con ArrayList el acceso es
// directo.

    System.out.println("1ª forma");
    for (int i = 0; i < alumnos.size(); i++) {
      System.out.println(alumnos.get(i));
    }

// El método indexOf() nos devuelve la posición de un objeto en la lista
// o -1 si no está en ella

    int pos = alumnos.indexOf(new Alumno("Adrián"));

// El método contains() nos devuelte true o false en función de que el
// objeto esté o no en la lista

    if (alumnos.contains(new Alumno("Adrián"))) {
      System.out.println("SI que está");
    }

    System.out.println("Posición de Adrián = " + pos);

// Para eliminar un objeto de la lista se utiliza el método remove(). Se
// puede hacer por posición o bien pasándole un objeto. En este último
// caso es necesario haber sobreescrito el método equals() de la clase
// del objeto, de modo que pueda comparar la igualdad.

    alumnos.remove(2);
    System.out.println("Borramos a Adrián");

// 2ª forma de recorrer la lista. Con un bucle For-each. Recordar que
// tiene limitaciones, como el que siempre se realiza el recorrido
// completo comenzando desde el comienzo y avanzando de uno en uno
// (con for con índices esto no es necesario), ni permite operaciones
// concurrentes, por ej, no podemos eliminar un elemento mientras
// hacemos el recorrido

    for (Alumno i : alumnos) {
      System.out.println(i);
    }


// borrado por objeto

    alumnos.remove(new Alumno("Elena"));
    System.out.println("Borramos a Elena");
    for (Alumno i : alumnos) {
      System.out.println(i);
    }

// insertando en una posición determinada

    alumnos.add(1, new Alumno("Manuel", 8));
    System.out.println("Insertamos Manuel");
    for (Alumno i : alumnos) {
      System.out.println(i);
    }
```

```
// Recorrido con Iteradores. Similar al for-each, pero permite
// operaciones concurrentes. Por ej. Podemos eliminar al mismo tiempo
// que hacemos el recorrido.
// Necesitamos un objeto de la clase Iterator al que asignamos el
// iterador devuelto por el método iterator() de la lista. El método
// hasNext() del iterador nos devuelve true si quedan elementos por
// recorrer. El método next() nos devuelte el siguiente elemento
// (el iterador se posiciona al comienzo antes del primer elemento).

    System.out.println("Insertamos Manuel");
    for (Iterator<Alumno> it = alumnos.iterator(); it.hasNext(); ) {
      System.out.println(it.next());
    }


// El método clear() elimina todos los elementos de la lista

    alumnos.clear();
    System.out.println("Borramos todo");


  }

}
```

b. Conjuntos

Un conjunto es una colección que almacena elementos únicos, no admite duplicados. Disponemos de dos Interfaces **AbstractSet**, **Set y OrderedSet.** Trabajaremos con las implementaciones **HashSet** y **TreeSet** , esta última representa un conjunto ordenado siguiendo el orden natural de los datos (ej, números de menor a mayor), o bien pasándole un **Comparator** al crear el conjunto. Si utilizamos alguna clase diferente a las de envoltorio de las primitivas o String, debemos proporcionar el código para el método **equals**() e implementar **Comparable** o **Comparator**

Vamos a ver con un ejemplo las principales operaciones que se suelen realizar con conjuntos. Creamos dos conjuntos, uno para los alumnos que están matriculados en algún módulo de 1º de DAW y otro para los alumnos que están matriculados en algún módulo de 2º de DAW. Un alumno puede tener módulos de ambos cursos, con lo que aparecerá en ambos conjuntos.

```
package set.union.interseccion.resta;

import java.io.FileNotFoundException;
import java.io.IOException;
import static java.lang.System.in;
import java.util.Objects;
import java.util.Set;
import java.util.TreeSet;
```

```
// Vamos a crear un conjunto de Alumnos. Implementamos la Interface
// Comparable para que se pueda lleva a cabo la ordenación, y los
// métodos equals() y hashCode() para las comparaciones de igualdad

class Alumno implements Comparable<Alumno> {

  String nombre, apellidos, dni;
  int notaFinal;
  boolean accedeFCT;

  public Alumno(String nombre, String apellidos, String dni, int
notaFinal, boolean accedeFCT) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.dni = dni;
    this.notaFinal = notaFinal;
    this.accedeFCT = accedeFCT;
  }

// El criterio de ordenación va a ser el número de DNI

  @Override
  public int compareTo(Alumno o) {
    return dni.compareTo(o.dni);
  }

  @Override
  public int hashCode() {
    int hash = 7;
    hash = 31 * hash + Objects.hashCode(this.dni);
    return hash;
  }

// Consideramos dos alumnos iguales si tienen el mismo DNI

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final Alumno other = (Alumno) obj;
    return Objects.equals(this.dni, other.dni);
  }

  @Override
  public String toString() {
    return "Alumno{" + "nombre=" + nombre + ", apellidos=" + apellidos +
", dni=" + dni + ", notaFinal=" + notaFinal + ", accedeFCT=" + accedeFCT
+ '}';
  }

}

/**
```

```
 *
 * @author Amador Abelleira Gómez
 */
public class SetUnionInterseccionResta {

  /**
   * @param args the command line argument
   */
  public static void main(String[] args) {

// Creamos los dos conjuntos utilizando la clase TreeSeet, esto
// implica que los alumnos estarán ordenados, en este caso por DNI

    Set<Alumno> daw1 = new TreeSet<>();
    Set<Alumno> daw2 = new TreeSet<>();

// Para añadir elementos utilizamos el método add() Si un elemento
// ya existe en el conjunto, no será añadido.

    daw1.add(new Alumno("César", "Algo más", "445456", 8, true));
    daw1.add(new Alumno("Cristina", "Otro apellido", "121213", 8,
true));
    daw1.add(new Alumno("David", "D segundo", "78978979", 8, false));

    daw2.add(new Alumno("David", "D segundo", "78978979", 8, false));
    daw2.add(new Alumno("Uxía", "No lo Se", "69455855", 8, true));

// Vamos a sacar un listado de todos los alumnos de DAW. Es la UNION de
// los dos conjuntos (daw1 y daw2), y se realiza con el método addAll()

    System.out.println("Todos los alumnos de DAW");

// Creamos un conjunto axuliar a partir de los datos de daw1

    Set<Alumno> todos = new TreeSet<>(daw1);

// Y le añadimos los elementos de daw2

    todos.addAll(daw2);

// Sacamos el listado

    for (Alumno it : todos) {
      System.out.println(it);
    }

// Vamos a sacar un listado de los alumnos de 2º DAW con alguna materia
// pendiente. Es la INTERSECIÓN de los dos conjuntos (daw1 y daw2), y se
// realiza con el método retainAll()

    System.out.println("Alunnos de 2º con materias pendientes");

// Creamos un conjunto axuliar a partir de los datos de daw1

    Set<Alumno> pendientes = new TreeSet<>(daw1);

// Y calculamos su intersección con daw1

    pendientes.retainAll(daw2);
    for (Alumno it : pendientes) {
      System.out.println(it);
```

```
    }

// Vamos a sacar un listado de los alumnos de 1° DAW que NO están
// matriculados en ningún módulo de 2° DAW. Es una RESTA de dos
// conjuntos (daw1 y daw2), y se realiza con el método removeAll()

    System.out.println("Alunnos solo de primero");
    Set<Alumno> solo1 = new TreeSet<>(daw1);
    solo1.removeAll(daw2);
    for (Alumno it : solo1) {
      System.out.println(it);
    }

  }
```

c. Mapas

Un mapa es una colección donde asociamos claves a valores. Tanto la clave como el valor pueden ser de cualquier tipo (menos primitivas). Vamos a trabajar con dos implementaciones de mapas, **HashMap** y **TreeMap**, donde la segunda representa un mapa ordenado por el orden natural de la clave o bien mediante un **Comparator** que se le pase como parámetro en la construcción del mapa.

Un ejemplo de un mapa ordenado a modo de diccionario de abreviaturas.

```
package mapaabreviaturas;

import java.util.Iterator;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;
import java.util.TreeMap;

/**
 *
 * @author Amador Abelleira Gómez
 */
public class MapaAbreviaturas {

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    Scanner tec = new Scanner(System.in);

// Creamos el mapa, indicando entre los símbolos < y > el tipo de la
// clave y el tipo del valor asociado, en este caso ambos son String

    Map<String, String> ab = new TreeMap<>();

// Añadimos parejas clave, valor mediante el método put()

    ab.put("p", "programación");
    ab.put("cd", "contornos de desenvolvemento");
    ab.put("lm", "linguaxes de marcas");
```

```
    ab.put("fol", "formación e orientación laboral");
    ab.put("bd", "bases de datos");
    ab.put("si", "sistemas informáticos");

// Ahora realizamos una búsqueda en el mapa. Pedimos una abreviatura
// (clave) y mostramos el nombre con el que se corresponde (valor)
// mediante el método get()

    System.out.print("Introduce siglas módulo: ");
    String modulo = tec.nextLine();
    System.out.println(ab.get(modulo));

// Para recorrer le mapa tenemos varias opciones, entre las que NO
// se encuentra el for con índices, ya que los elementos de un Map,
// al igual que los de un Set, no están indexados.


// Con for-each iterando sobre el conjunto de claves que obtenemos con
// keySet(). No es la mejor opción, teniendo un rendimiento pobre

    System.out.println("Con For Each");
    for (String clave : ab.keySet()) {
      String valor = ab.get(clave);
      System.out.println("FE- " + clave + " - " + valor);
    }

// En su lugar, sería más óptimo el uso de iteradores, al igual que
// antes, sobre el conjunto de claves que obtenemos con keySet()

    System.out.println("Con Iteradores");
    Set claves = ab.keySet();
    for (Iterator<String> it = claves.iterator();
            it.hasNext();) {
      String clave = it.next();
      String valor = ab.get(clave);
      System.out.println("It- " + clave + " - "
              + valor);
    }

// La opción preferida en la mayoría de ocasiones: Mediante un
// objeto Map.Entry que recoge el conjunto devuelto por el método
// entrySet()

    System.out.println("Con Map.Entry");
    for (Map.Entry<String, String> it : ab.entrySet()) {
      String clave = it.getKey();
      String valor = it.getValue();
      System.out.println("ME- " + clave + " - " + valor);
    }

  }

}
```

d.Ordenar colecciones

Algunas colecciones como TreeMap o TreeSet, está permanentemente ordenadas; cada vez que introducimos un nuevo elemento, se sitúa en el lugar correspondiente. Pero disponemos de métodos para ordenar otras colecciones , por ejemplo, una Lista. Si los elementos d la colección son envoltorios de tipos primitivos o String, no hace falta ningún paso más. En otro caso, es responsabilidad del programador proporcionar el método de comparación, indicando cuando un elemento es mayor, menor o igual a otro. Vamos a ver varias formas de hacerlo.

```
package ordenarlista;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;
import java.util.Objects;
import java.util.Scanner;

// Vamos a trabajar con una lista de Proyecto. Tenemos que proporcionar
// la manera de ver si dos objetos son iguales:  métodos hashCode() y
// equals()
// y además, para poder ordenarlos, cuando uno es mayor, menor o igual
// a otro. Para esto implementamos la Interfaz Comparable y proporcionar
// el código adecuado al método compareTo() que tendremos que
// sobreescribir.

class Proyecto implements Comparable<Proyecto> {

  String nombre;
  String lenguaProgramacion;
  String fechaInicio;
  Double coste;

  public Proyecto(String nombre, String lenguaProgramacion, String
fechaInicio, Double coste) {
    this.nombre = nombre;
    this.lenguaProgramacion = lenguaProgramacion;
    this.fechaInicio = fechaInicio;
    this.coste = coste;
  }

//
  @Override
  public int hashCode() {
    int hash = 3;
    return hash;
  }

// Consideramos dos proyectos iguales si tienen el mimo nombre

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
```

```
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final Proyecto other = (Proyecto) obj;
    return Objects.equals(this.nombre, other.nombre);
  }

// El método compareTo() recibe como parámetro el objeto (o) con el
// que queremos comparar al objeto actual (this). Debe devolver
// un entero <0 si this es menor que o, >0 si this es mayor que o
// o 0 si son iguales
// El método compareTo() ya está correctamente implementado para
// la clase String (lo utilizamos aquí para comparar los nombres)
// y las clases envoltorio.

  @Override
  public int compareTo(Proyecto o) {
    int v = this.getLenguaProgramacion().
            compareTo(o.getLenguaProgramacion());
    if (v != 0) {
      return v;
    }
    return this.getNombre().compareTo(o.getNombre());
  }

  @Override
  public String toString() {
    return "\nProyecto{" + "\n  nombre=" + getNombre()
            + "\n lenguaProgramacion=" + getLenguaProgramacion()
            + "\n fechaInicio=" + getFechaInicio()
            + "\n coste=" + getCoste() + "\n}";
  }



}

public class OrdenarLista {

  static Scanner tec = new Scanner(System.in);

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {

// Creamos una Lista de Proyecto y le añadimos unos cuantos.

    List<Proyecto> lista = new ArrayList<>();
    lista.add(new Proyecto("p4", "java", "10/10/2022", 2000d));
    lista.add(new Proyecto("p1", "java", "03/07/2022", 10000.10));
    lista.add(new Proyecto("p3", "php", "15/12/2022", 3000.50));
    lista.add(new Proyecto("p2", "lisp", "01/02/2022", 2500.10));

// Para ordenarla llamamos al método sort() de la clase Collections
// pasándole como parámetro la list
// Si la queremos en el orden inverso podemos hacer:
```

```
// Collections.sort(lista, Collections.reverseOrder);

    Collections.sort(lista);
    System.out.println("Ordenado por lenguaje y nombre");
    for (Iterator<Proyecto> it = lista.iterator();
            it.hasNext();) {
      Proyecto x = it.next();
      System.out.println(x.getLenguaProgramacion()
              + " - " + x.getNombre());
    }

// Si queremos disponer de más de un criterio de ordenación
// podemos implementar la Interface Comparator. En este caso la hemos
// implementado en una clase nueva llamada PorCoste. Esta Interface
// no pide implementar el método:
// public int compare(Proyecto o1, Proyecto o2)
// que devolverá <0, >0 o 0 en función de si o1 es menor, mayor o igual
// a o2
// En cada clase solo podemos tener un método compare(), pero podemos
// crear nuevas clases para obtener nuevos criterios de ordenación
// Ordenamos por coste:
// En la clase PorCost tendríamos:
//public class PorCoste implements Comparator<Proyecto> {
//
//  @Override
//  public int compare(Proyecto o1, Proyecto o2) {
//    return o1.getCoste().compareTo(o2.getCoste());
//  }
//
//}

    System.out.println("Ordenado por coste");
    Collections.sort(lista, new PorCoste());
    for (int i = 0; i < lista.size(); i++) {
      System.out.println(lista.get(i).getNombre() + " -"
              + lista.get(i).getCoste());
    }

// Otra forma de ordenar. En este caso utilizamos el método sort()
// de la colección, que necesita un parámetro que proporciones el
// comparador. Aquí en lugar de implementar Comparator en una clase
// Creamos una clase anónima con la implementación de método
// directamente como parámetro
// Ordenamos por fecha de inicio:

    System.out.println("Ordenado por fecha");
    lista.sort(new Comparator<Proyecto>() {
      @Override
      public int compare(Proyecto o1, Proyecto o2) {
        return o1.getFechaInicio().
                compareTo(o2.getFechaInicio());
      }
    });
    for (Proyecto it : lista) {
      System.out.println(it.getNombre() + " - "
              + it.getFechaInicio());
    }

  }

}
```