Universidad de Alcalá

Departamamento Ciencias de la Computación

Programación Orientada a Objetos

Tema 6:

Estructuras de Datos

Universidad de Alcalá

Departamento Ciencias de la Computación

Contenidos

- Tema 6: Estructuras de Datos

1. MÉTODOS BÁSICOS DE BÚSQUEDA Y ORDENACIÓN
2. TIPOS ENUMERADOS
3. COLECCIONES
4. ORDENACIÓN DE OBJETOS

Fundamentos de Ordenación y Búsqueda

- Los algoritmos de ordenación y búsqueda son **ampliamente utilizados** en las aplicaciones modernas.
- Existe una gran variedad de algoritmos que intentan **optimizar** distintas situaciones, aplicando **diferentes técnicas.**
- Muchos lenguajes de alto nivel proporcionan funciones que evitan que el usuario deba implementar estos algoritmos por sí mismo.
- Hay que tener en cuenta el coste computacional asociado a la búsqueda y ordenación sobre colecciones de tamaño N.

Búsqueda secuencial

Localizar, en una colección de elementos, el primer elemento que cumple una condición dada.

Ej: buscar aula libre en una determinada fecha y horario que tenga al menos n asientos

Con mucha frecuencia los elementos (objetos-sujetos) que interesa tratar informáticamente son identificables → **clave** =identificador unívoco

Ej: localizar **la** ficha del alumno con un determinado número de expediente

Algoritmo de búsqueda secuencial

La implementación concreta depende de la estructura de datos

PSEUDOCÓDIGO DEL ALGORITMO GENERAL

```
FUNCION posicion( L, clave)
{
   actual←primero_de_L;
   MIENTRAS queden y clave≠actual.clave HAZ actual←siguiente;
   SI clave=actual.clave ENTONCES posicion←pos_de_actual
   EN CASO CONTRARIO posicion←NoEncontrado
}
```


Búsqueda binaria

Aplicabilidad
- La lista está ordenada por la clave de búsqueda
- Se conoce el número de elementos
- Se tiene acceso directo al elemento por posición en la lista

PSEUDOCÓDIGO DEL ALGORITMO GENERAL

```
FUNCION posicion_bin( L, clave)
{
   tam←tamaño_de_la_colección;
   inf←0;                                  //inf: limite inferior del intervalo
   sup←tam-1;                              //sup: limite superior del intervalo
   MIENTRAS inf<=sup HAZ
```

Ordenación por inserción directa

La colección inicial de elementos se descompone en dos partes: Una parte cuyos elementos mantienen un orden y la parte restante.

Inicialmente la parte ordenada consta de un solo elemento (el primero)

En la iteración i, los i primeros elementos están ya ordenados. Se localiza la posición que correspondería en la parte ordenada al primer elemento de la parte desordenada (elemento i+1) y **se inserta** en esa posición.

Ejemplo (En cada iteración se marca la parte ordenada con subrayado):

i=1	44 **55** 12 42 94 18 06 67
2	44 55 **12** 42 94 18 06 67
3	12 44 55 **42** 94 18 06 67
4	12 42 44 55 **94** 18 06 67
5	12 42 44 55 94 **18** 06 67
6	12 18 42 44 55 94 **06** 67
7	06 12 18 42 44 55 94 **67**
9	06 12 18 42 44 55 67 94

Ordenación por selección directa

La colección inicial de elementos se descompone en dos partes: Una parte cuyos elementos mantienen un orden y la parte restante.

En la iteración i se busca dentro de la parte desordenada el elemento cuyo valor clave es menor (o mayor, según el criterio de ordenación) y se coloca en esa posición, de forma que deja de pertenecer a la parte desordenada de la colección.

Ejemplo (En cada iteración se marca la parte ordenada con subrayado):

i=1	44 55 12 42 94 18 **06** 67

Ordenación por intercambio directo

También es conocido como *método de la burbuja*.

Se basa en realizar pasadas sucesivas sobre todos los elementos de la colección. En cada pasada se compara cada uno de los elementos de la colección con su adyacente, y se realiza el intercambio entre ellos en caso de que el criterio de ordenación lo requiera (como si hubiera que ordenar sólo los dos elementos comparados).

PSEUDOCÓDIGO DEL ALGORITMO GENERAL

```
FUNCION ordenarBurbuja (L)
{
    PARA i=0 HASTA tamaño_de_la_colección-1 HAZ
    {
        PARA j=0 HASTA tamaño_de_la_colección-2 HAZ
            SI L[j].clave>L[j+1].clave ENTONCES          // caso de ordenación de menor a mayor
            {
                    // se intercambian los elementos consecutivos
                    aux←L[j];
                    L[j]←L[j+1];
                    L[j+1]←aux;
            }
    }
}
```


Universidad de Alcalá
Departamento Ciencias de la Computación

Ordenación por intercambio directo

Ejemplo *método de la burbuja*.

Inicio:	44 55 12 42 94 18 06 67
i=0	44 12 42 55 18 06 67 94
1	12 42 44 18 06 55 67 94
2	12 42 18 06 44 55 67 94
3	12 18 06 42 44 55 67 94

- **Tipos Enumerados:**
- Los tipos enumerados sirven para restringir la selección de valores a un conjunto previamente definido.
- Un tipo enumerado permite que una variable tenga solo un valor dentro de un conjunto de valores predefinidos, es decir, valores dentro de una lista enumerada.
- Los valores que se definen en un tipo enumerado son constantes y se suelen escribir en mayúsculas.
- La clase que representa los tipos enumerados en Java es java.lang.Enum.
- Posee una serie de métodos útiles como:
  - toString(): permite visualizar el nombre de las constantes de una enumeración.
  - ordinal(): permite saber el orden de declaración de las constantes.
  - values(): genera un vector con los valores de la enumeración.

- **Ejemplos Tipos Enumerados:**

```
public enum ColoresSemaforo {
   VERDE, NARANJA, ROJO
}

public class PruebaEnum {
   public static void main(String[] args) {
      ColoresSemaforo cs = ColoresSemaforo.VERDE;
      switch (cs) {
         case ROJO:
            System.out.println("No puedes pasar.");          break;
         case VERDE:
```

```
public class PruebaEnum2 {

   enum DiasSemana {
      L, M, X, J, V, S, D
   };

   public static void main(String[] args) {
      DiasSemana ds = DiasSemana.L;
      System.out.println(ds);
   }
}
```

- Una colección es un objeto que recopila y organiza otros objetos.
- La colección define las formas específicas en las que se puede acceder y con las que se pueden gestionar esos objetos, los cuales se denominan elementos de la colección.
- Existen muchos tipos específicos de colecciones que permiten resolver distintas problemáticas.
- Las colecciones se pueden clasificar en dos categorías generales: lineales y no lineales (ej.: jerárquica o en red).
- La organización relativa de los elementos de una colección está determinada usualmente por:
  - El orden en que se añaden los elementos a la colección.
  - Alguna relación inherente entre los propios elementos.

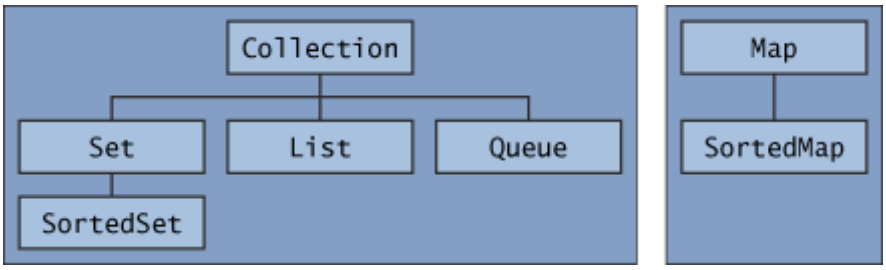
Universidad de Alcalá
COLECCIONES
Departamento Ciencias de la Computación

- Las *Colecciones* en Java ofrecen un mecanismo orientado a objetos para almacenar conjuntos de datos de tipo similar.
- Tienen su propia asignación de memoria y un conjunto de métodos para su iteración y recorrido.
- El **framework de las colecciones** en Java se basa en una arquitectura unificada que contiene:
  - **Interfaces**. Tipos abstractos de datos que representan a las

búsquedas y ordenación. Estos métodos son polimórficos.

- Las clases que representan colecciones en Java se encuentran dentro del paquete java.util.
- Las clases que representan colecciones se basan en una serie de interfaces que definen los métodos necesarios para su gestión y que estas clases implementarán.

- Las interfaces más importantes son las siguientes:



- Las interfaces de las colecciones se basan en **genéricos**. Los genéricos implementan el concepto de *tipos parametrizados*, que permiten crear colecciones que resulten fáciles de utilizar con múltiples tipos.

- El término “genérico” significa “perteneciente o apropiado para grandes grupos de clases”.

COLECCIONES

- La interface **Collection** representa una secuencia de elementos individuales a los que se aplica una o más reglas.

- Una colección **List** debe almacenar los elementos en la forma en que fueron insertados, una colección **Set** no puede tener elementos duplicados y una colección **Queue** produce los elementos en el orden determinado por una disciplina de cola.

- La interface **Map** representa un grupo de parejas de objetos clave-valor, que permite realizar búsquedas de objetos. No se permiten claves duplicadas.

COLECCIONES: LISTAS

- **Listas:**
- Un **ArrayList** es un array de objetos cuyo tamaño puede variar en tiempo de ejecución. Implementa la interface **List**.
- Los objetos se pueden almacenar al final de la colección o en una posición concreta utilizando el método *add()* y borrarlos mediante *remove().*
- También podemos remplazar un elemento de la colección

EJEMPLO ARRAYLIST

```
import java.util.*;
public class PruebaArrayList {

   private static final String colores[] = {"rojo", "verde", "azul"};

   public static void main(String[] args) {
      ArrayList<String> array = new ArrayList <String>();
      imprimeArrayList(array);
      // añadimos elementos al ArrayList
      array.add("amarillo");
      array.addAll(Arrays.asList(colores));
      array.add("blanco");
      imprimeArrayList(array);

      // primer y último elemento
      System.out.println("Primer elemento: " + array.get(0));
      System.out.println("Último elemento: " + array.get(array.size()-1));

      // encontrar un elemento
      if (array.contains("rojo")) {
         System.out.println("\n\"rojo\" encontrado en el índice " +
               array.indexOf("rojo") + "\n");
      } else {
         System.out.println("\n\"rojo\" no encontrado\n");
      }
      // borrar un elemento
      array.remove("rojo");
      imprimeArrayList(array);
   }

   private static void imprimeArrayList(ArrayList array) {
      if (array.isEmpty()) {
         System.out.print("el ArrayList está vacío");
      } else {
         System.out.print("Contenido del ArrayList: ");

         Iterator items = array.iterator();

         while (items.hasNext()) {
            System.out.print(items.next() + " ");
         }
      }

      System.out.println("\n");
   }
}
```


COLECCIONES: CONJUNTOS

- **Conjuntos:**
- **HashSet:**
  - Un **HashSet** se basa en una *tabla hash* para almacenar los objetos y es una implementación de la interface Set y representa un conjunto de valores que no admite duplicados.
  - Para almacenar un objeto utilizaremos el método *add()* y para borrarlo *remove().*

- *Nota: Una tabla hash o mapa hash es una estructura de datos que*

EJEMPLO HASHSET


```
import java.util.*;

public class PruebaHashSet {
   private static final String colores[] = {"rojo", "verde", "azul"};
   public static void main(String[] args) {
      HashSet<String> conjunto = new HashSet<String>();
      // añadimos elementos al conjunto
      conjunto.add("amarillo");
      conjunto.add("amarillo"); //no se añade porque no admite duplicados
      conjunto.addAll(Arrays.asList(colores));
      conjunto.add("blanco");
      // imprimimos el contenido
      System.out.println(conjunto.toString());
      // encontrar un elemento
      if (conjunto.contains("rojo")) {
         System.out.println("\n\"rojo\" encontrado\n");
      } else {
         System.out.println("\n\"rojo\" no encontrado\n");
      }
      // borrar un elemento
      conjunto.remove("rojo");
      // imprimimos el contenido
      System.out.println(conjunto.toString());
   }
```


COLECCIONES: CONJUNTOS

- **Conjuntos:**
- **TreeSet:**
  - Un **TreeSet** es una colección en forma de árbol de objetos cuyos elementos están ordenados. Implementa la interface **SortedSet**.
  - El orden de los objetos puede ser el natural (ej.: los números se encuentran en orden ascendente) o en un orden especificado a través de un objeto comparador. Utiliza los mismos métodos de inserción y borrado que HashSet. La mayor utilidad que tiene esta colección es la de presentar elementos en determinadas

EJEMPLO TREESET

```
import java.util.*;
public class PruebaTreeSet {

   public static void main(String[] args) {
      //Comparador para ordenar las personas por su edad
      Comparator EdadPerComp = new Comparator() {
         public int compare(Object o1, Object o2) {
            Persona per1 = (Persona) o1;
            Persona per2 = (Persona) o2;
            Integer e1 = new Integer(per1.getEdad());
            Integer e2 = new Integer(per2.getEdad());
            return e1.compareTo(e2);
         }
      };
      //Fechas de nacimiento
      Fecha f1 = new Fecha(15, 3, 1965);      Fecha f2 = new Fecha(05, 8, 1975);
      Fecha f3 = new Fecha(25, 11, 1980);     Fecha f4 = new Fecha(27, 10, 1990);

      //Diversos objetos de tipo persona
      Persona obj1 = new Persona("06634246S", "Javier", f1, "calle1");
      Persona obj2 = new Persona("65834916K", "Pedro", f2, "calle2");
      Persona obj3 = new Persona("91635476F", "Laura", f3, "calle3");
      Persona obj4 = new Persona("15664386T", "Carmen", f4, "calle4");

      //Introducimos los objetos en el árbol
      TreeSet<Persona> personas = new TreeSet<Persona>(EdadPerComp);
      personas.add(obj1);
      personas.add(obj2);
      personas.add(obj3);
      personas.add(obj4);

      //intentamos añadir un duplicado
      Persona duplicado = new
        Persona("15664386T", "Carmen", f4, "calle4");
      if (personas.contains(duplicado)) {
         System.out.println("\nPersona ya existe.");
      }
      personas.add(duplicado);

      //Presentamos la información
      System.out.println(personas.toString());
      System.out.println("\nPersonas ordenadas por
        edad:");
      for (Persona per : personas) {
         System.out.println(per.toString());
      }
      //Presentamos la información de las personas
      // menores de 35
      Persona p = new Persona();
      p.setEdad(35);
      SortedSet conjuntoMenor =
        personas.headSet(p);
      System.out.println("\nPersonas menores de
        35:");
      items = conjuntoMenor.iterator();
      while (items.hasNext()) {
         Persona per = (Persona) items.next();
         System.out.println(per.toString());
      }
   }
}
```


COLECCIONES: MAPAS

- **Mapas:**
- **HashMap:**
  - Una **HashMap**, es una *tabla hash*, realiza una implementación de la interface **Map**, por lo tanto, representa una colección para almacenar objetos claves/valores.
  - Para almacenar un objeto utilizaremos el método *put(clave,valor)*. Después se puede utilizar *get(clave)* para recuperar el objeto. Para borrarlo utilizaremos *remove(clave).*
- **TreeMap:**

EJEMPLO HASHMAP

```
import java.util.*;

public class PruebaHashMap {
   public static void main(String[] args) {
      HashMap<String, Persona> personas = new HashMap<String, Persona>();
      //Creamos personas
      Fecha f1 = new Fecha(15, 3, 1965);       Fecha f2 = new Fecha(05, 8, 1975);
      Persona per1 = new Persona("06634246S", "Javier García", f1, "calle1");
      Persona per2 = new Persona("65834916K", "José Sánchez", f2, "calle2");
      //Las introducimos en la HashMap
      personas.put(per1.getDni(), per1);
      personas.put(per2.getDni(), per2);
      //Presentamos la información
      System.out.println(personas.toString());
      //Recuperamos una persona mediante su DNI
      Persona obj = personas.get("06634246S");
      System.out.println("Nombre: " + obj.getNombre());
      //Modificamos datos
      obj.setNombre("Nuevo Nombre");
      //Eliminamos una persona
      personas.remove("65834916K");
      if (personas.containsKey("65834916K")) {  System.out.println("No eliminada"); } else
                { System.out.println("Eliminada"); }
      //Presentamos la información
      System.out.println(personas.toString());
   }
}
```


EJEMPLO TREEMAP

```
import java.util.*;
public class PruebaTreeMap {
   public static void main(String[] args) {
      //Fechas de nacimiento
      Fecha f1 = new Fecha(15, 3, 1965);       Fecha f2 = new Fecha(05, 8, 1975);
      Fecha f3 = new Fecha(25, 11, 1980);      Fecha f4 = new Fecha(27, 10, 1990);
      //Diversos objetos de tipo persona
      Persona obj1 = new Persona("06634246S", "Javier", f1, "calle1");
      Persona obj2 = new Persona("65834916K", "Pedro", f2, "calle2");
      Persona obj3 = new Persona("91635476F", "Laura", f3, "calle3");
      Persona obj4 = new Persona("15664386T", "Carmen", f4, "calle4");

      //Introducimos los objetos en una tabla hash ordenada por DNI
      TreeMap<String, Persona> personas = new TreeMap<String, Persona>();
      personas.put(obj1.getDni(), obj1);
      personas.put(obj2.getDni(), obj2);
```



```
   }
```

COLECCIONES: ITERADORES

- **Iteradores:**
- **Iterator:**
  - **Iterator** es una interfaz simple que permite recorrer todos los elementos de una colección de objetos.
  - Especifica dos métodos para su recorrido *hasNext()* y *next()*.
- **ListIterator:**
  - **ListIterator** es un subtipo de Iterator y es más potente ya que permite realizar un recorrido bidireccional de la colección añadiendo el método *hasPrevious()* y *previus()*.

COLECCIONES: ALGORITMOS

- **Algoritmos:**
- La clase **Collections** contiene una serie de algoritmos de utilidad para aplicarlos a las colecciones, estos algoritmos son polimórficos ya que se pueden aplicar a cualquier tipo de dato. Entre sus métodos destacan los de ordenación y búsqueda. Todos los métodos de la clase son estáticos. La mayoría se aplican sobre las colecciones de tipo lista.
- El método *sort()* nos permite ordenar una lista, bien por su

COLECCIONES: ALGORITMOS

- **Algoritmos:**
- Los métodos min() y max() nos permiten saber cual es el elemento menor y mayor de la lista.
- El método *frecuency()* nos permite saber el número de veces que se repite un elemento de una lista.
- El método *rotate()* nos permite mover todos los elementos de una lista hacia adelante un determinado número de posiciones, extrayéndolos por el extremo y colocándolos al principio.
- El método *suffle()* nos permite permutar una lista de manera aleatoria.
- El método swap() nos permite intercambiar dos elementos de una lista.

ORDENACIÓN DE OBJETOS

- Para ordenar objetos a través de alguno de sus atributos podemos utilizar objetos comparadores o implementar la interfaz **Comparable**.
- Un objeto **Comparator** nos proporcionará la forma en la que debemos ordenar la colección a través del método **compareTo()**.

```
//Comparador para ordenar las personas por su DNI
Comparator DniPerComp = new Comparator() {
    public int compare(Object o1, Object o2) {
        Persona per1 = (Persona) o1;
        Persona per2 = (Persona) o2;
        return per1.getDni().compareTo(per2.getDni());
    }
};
```

```
public class Persona
    implements Comparable<Persona> {
  private String dni;
  private String nombre; ...

  public int compareTo(Persona p) {
      return this.dni.compareTo(p.getDni());
  }
```

ORDENACIÓN DE OBJETOS

- Ordenación de los objetos de un **ArrayList** por el atributo que seleccionemos y búsqueda de un elemento.

```
//Introducimos los objetos en un ArrayList
ArrayList<Persona> personas = new ArrayList<Persona>();
personas.add(obj1); …          personas.add(obj4);
//Ordenamos los objetos del array por el atributo Nombre
Collections.sort(personas, NomPerComp);
//Buscamos una persona por su nombre
BufferedReader entrada=
     new BufferedReader(new InputStreamReader(System.in));
System.out.println("\nIntroduce el nombre de la persona a buscar:");
String nombre = entrada.readLine();
//creamos una persona con el nombre a buscar
Persona p = new Persona();
p.setNombre(nombre);
int pos = Collections.binarySearch(personas, p, NomPerComp);
if (pos>=0) {
          System.out.println("\nDatos de la persona:");
          Persona per = personas.get(pos);
          System.out.println(per.toString());
} else System.out.println("\n Persona no encontrada.");
```


ORDENACIÓN DE OBJETOS

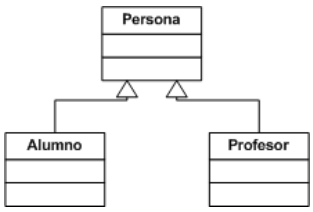
- Ordenación de los objetos de una **HashMap** a través de un ArrayList por el atributo que seleccionemos.

```
//Introducimos los objetos en una tabla hash
HashMap<String, Persona> personas = new HashMap<String, Persona>();
personas.put(obj1.getDni(), obj1); …          personas.put(obj4.getDni(), obj4);

//Convertimos la tabla hash en un ArrayList de objetos
Collection<Persona> colec = personas.values();
ArrayList<Persona> personasA = new ArrayList<Persona>(colec);

//Ordenamos los objetos del array por el atributo nombre
Collections.sort(personasA, NomPerComp);
```

TIPOS DE OBJETOS EN COLECCIONES

- Para saber con que clase se instanció un objeto se debe utilizar el método *getClass* de la clase Object, si lo complementamos con *getSimpleName* de la clase Class, nos devuelve una cadena con el nombre de la clase.
- Ejemplo: Tenemos una jerarquía de clases e introducimos diversos objetos en una tabla hash de tipo Persona:

```
HashMap<String, Persona> personas = new HashMap<String, Persona>();
Persona obj1 = new Profesor(…); …          Persona obj4 = new Alumno(…);
personas.put(obj1.getDni(), obj1); …       personas.put(obj4.getDni(), obj4);
//Buscamos una persona por su DNI
BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));
System.out.println("\nIntroduce el dni de la persona a buscar:");
String dni = entrada.readLine();
Persona paux = personas.get(dni);
//Sacamos la clase para poder aplicarle sus métodos
String clase = paux.getClass().getSimpleName();
System.out.println("Clase: " + clase);
if (clase.equals("Alumno")) {
        Alumno alumno = (Alumno) paux;
        System.out.println("- Titulación: " + alumno.getTitulacion());
        System.out.println("- Universidad: " + alumno.getUniversidad());
} else if (clase.equals("Profesor")) {
        Profesor profe = (Profesor) paux;
        System.out.println("- Departamento: " + profe.getDepartamento());
        System.out.println("- Sueldo: " + profe.getSueldo());
}
```



EJEMPLO USO DE COLECCIONES

- El siguiente ejemplo muestra la utilización de un ArrayList para gestionar objetos de tipo Persona en un censo. También se utiliza un iterador para recorrer sus elementos. Para interactuar con éstas clases se ha realizado una aplicación gráfica en Swing.