

## Tutorial 11

### Temas

- Búsqueda
- Clasificación
- Estructura de Java Collection
- Problemas del tutorial
- Ejercicios de diseño

### Búsqueda

El objetivo de la búsqueda es localizar un documento específico identificado a través de una o más claves. Existen diferentes enfoques de búsqueda muy distintos entre sí en relación al tiempo de acceso en el caso de bases de datos muy extensas.

#### Búsqueda lineal

La búsqueda lineal es el proceso secuencial que consiste en examinar los registros, comenzando por el primero, hasta que se encuentra un equivalente o hasta que se completa la búsqueda sin éxito. La búsqueda lineal es la opción adecuada cuando los registros están ordenados de forma **aleatoria** o cuando se almacenan los datos en una lista enlazada linealmente. Este enfoque podría ser razonable si el tamaño de los datos es pequeño y si el contenido de la base de datos es **altamente dinámico** (por ej., supresiones frecuentes).

#### Búsqueda binaria

La búsqueda binaria es un procedimiento directo de localización de un elemento en un array ya **ordenado**. Dicha búsqueda es directamente análoga al método de bisección para la búsqueda de raíces. Funciona dividiendo el conjunto de datos en dos, determinando en qué mitad se encuentra el elemento deseado (en el caso de que exista) y posteriormente cortándolo de nuevo en dos, etc. Es decir, la búsqueda binaria es una estrategia del tipo "divide y vencerás". Esta búsqueda es mucho más eficaz que la lineal puesto que, en el peor de los casos, cuando no existe un equivalente, son necesarias interacciones  $O(\log n)$ . No obstante, es necesario pre-procesar/pre-clasificar los datos.

#### Búsqueda con ayuda de un árbol de búsqueda binario

Un árbol de búsqueda binario es aquel que posee la propiedad por la cual el descendente izquierdo de cualquier nodo es "menor" que el nodo de la raíz y el descendente derecho "mayor". Esta propiedad se aplica de forma recursiva a cada subárbol del árbol principal. Para buscar un elemento en un árbol binario seguimos los siguientes pasos:

- Comparamos la clave de búsqueda con la clave de la raíz.
- Si es la misma, la búsqueda habrá finalizado.
- Si no es la misma, determinamos si esta es más pequeña que la clave de la raíz.
- De ser ese el caso, debe ubicarse en el subárbol izquierdo.
- Si es mayor, debe ubicarse en el subárbol derecho.

Por consiguiente, en cada momento, el subárbol derecho o el izquierdo es eliminado por consideración. Finalmente, el procedimiento encuentra la clave en el árbol o viene a ser un nodo NULL y concluye que la clave no se encuentra almacenada en el árbol. El resultado del algoritmo depende de la forma del árbol binario.

### Hashing (Dispersión)

La búsqueda lineal, la búsqueda binaria y la búsqueda con un árbol de búsqueda binario suponen el examen de los registros en una secuencia sistemática hasta que el registro es visto y hallado o hasta que la búsqueda se considera infructuosa. Además, la eficacia de estos enfoques depende directamente del tamaño de los datos. En el caso de conjuntos de datos muy extensos o diversos, es posible que dichos métodos de búsqueda requieran el examen de diversos registros. Para minimizar el tiempo de búsqueda necesario en el caso de conjuntos de datos extensos, se puede utilizar un enfoque alternativo denominado *hashing*. El dilema es que este enfoque requiere más memoria que los métodos anteriores. La memoria adicional utilizada por el *hashing* ayuda a reducir el número de registros examinados en cada búsqueda. A su vez, facilita la adición y supresión de registros.

### Tabla Hash

Las tablas hash son una manera rápida de catalogar y buscar datos relacionados con tipos de valores de clave arbitraria, por ejemplo, nombres que corresponden a números de teléfono. La solución que ofrecen estas tablas es casi similar a un hack para acelerar el proceso de búsqueda. Generalmente, la búsqueda de una base de datos con claves complejas, como por ejemplo cadenas (*strings*), requiere mucho tiempo, ya que la comparación es lenta, a diferencia de la comparación de enteros. Además, comprobar listas enlazadas es un proceso lento si se compara con el acceso a arrays. Por tanto, el truco de las tablas hash está en el valor clave complejo, trácelo exclusivamente en un valor entero y, a continuación, utilice ese valor para indexar en una tabla donde haya almacenado los datos. De esta forma se minimiza el tiempo de cálculo a lo que se tarda en trazar la clave compleja en un valor entero, ya que el acceso a los arrays es rápido.

### Terminología

A continuación se indican los términos más comunes relacionados con las tablas hash:

Función hash	Función que toma la clave y establece una correspondencia con un entero. Todas las funciones hash son parecidas a esta: <code>int HashFunction(Key key).</code>
Colisión	Cuando hay más de una clave que corresponde o se identifica con el valor entero.
Código hash	El valor devuelto por la función hash.
Resolución de colisión	Cuando se halla algún modo para diferenciar las claves en el momento que sucede la colisión. Por ejemplo, encadenado y doble hashing.

### Encadenado

El encadenado es una manera sencilla de resolver las colisiones. Básicamente, cuando más de un conjunto de pares de valores clave produce el mismo código hash, se ubican en el mismo hueco. A la hora de buscarlos, sólo tiene que buscar a través de esa lista el correcto.

### Inconvenientes

A pesar de que las tablas hash son rápidas, a menudo resulta difícil hallar una buena función hash que distribuya los datos uniformemente a través de la tabla y evite la colisión. Otro problema de las tablas hash es que se puede desperdiciar potencialmente mucha memoria. Por ejemplo, si su función hash produce valores de 1 a 5000, pero usted sólo cuenta con 1000 elementos, entonces habrá desperdiciado

al menos 4000 elementos de la tabla. Por consiguiente, lo más importante es seleccionar cuidadosamente la función hash.

#### Ejemplo de función hash

A continuación, le proporcionamos un operador módulo sencillo como función hash que funcionará únicamente como tal si los objetos a los que se realiza un hash son enteros y, en ese caso, generalmente no es una elección adecuada. La función hash general es operador módulo(%). Para una clave de entero, la función:

$$h(\text{key\_value}) = \text{key\_value} \% M$$

proporcionará un índice en el rango deseado.

#### Eficacia

Consideremos la eficacia de estos métodos de búsqueda. Por ejemplo, en la oficina del secretario general del MIT se guardan los registros de 9000 estudiantes. Se accede a cada uno de ellos mediante números de identificación (ID) del MIT. Suponga que el secretario almacenó los registros en un array de 9000 elementos sin ningún orden particular, y que realizaba una búsqueda lineal de los mismos cuando necesitaba información sobre un estudiante. En el caso de que la búsqueda hubiese sido infructuosa, se habrían tomado 9000 comparaciones. Suponga que los registros se almacenaron en orden ascendente en relación al número de identificación del MIT. Una búsqueda binaria infructuosa requeriría 14 comparaciones ( $\log_2(9000)$  es aproximadamente 14). Ahora, si se utiliza el hashing, una búsqueda infructuosa podría tomar solamente 1 ó 2 comparaciones. Suponga que se utiliza una tabla hash con espacio para 20000 entradas. La función hash podría ser (número ID del MIT)%20000. Dado que serían pocos los números ID que se asignarían a la misma posición, se requerirían muchas menos comparaciones.

#### Ejemplo

Observaremos una pequeña muestra para ver como funciona el hashing y, a continuación, discutiremos algunos esquemas de resolución de colisiones. Digamos que tenemos 5 estudiantes en una clase y que cada uno de ellos tiene un número de identificación especial de 4 dígitos. Podemos almacenar esta información en una tabla hash de la siguiente forma. Cree una tabla hash de tamaño 20 y utilice la función hash  $ID\%20$  para almacenar y buscar elementos. Digamos que la lista de estudiantes es:

Nombre	Número de identificación	ID%20
Bob	1235	15
Jane	1472	12
Gary	9209	9
Cindy	7427	7
Paula	8328	8

Nuestra tabla hash sería la siguiente:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
							7427	8328	9209			1472			1235				

Cindy Paula Gary

Jane

Bob

Suponga que estamos buscando a un estudiante con número de identificación de 2003 y cuya búsqueda produce 3 cuando se realiza un hash. Observamos que no existe ninguna entrada en la posición 3, por lo que inmediatamente podemos determinar que la búsqueda es infructuosa.

A continuación, ¿cómo tratamos las colisiones si dos registros tienen el mismo código hash? (el resultado de una función hash se denomina código hash). Por ejemplo, digamos que Tom, con número de identificación 5567, se une a la clase. La función hash indicaría que su registro pertenece a la posición 7. Sin embargo, esa misma posición la ocupa ya el registro de Cindy. Por lo tanto, debemos tratar esta colisión. Existen tres esquemas de resolución de colisiones distintos: sondeo lineal, encadenamiento y doble hashing.

### Sondeo lineal

Cuando sucede una colisión, el nuevo registro se inserta simplemente en la siguiente posición vacía de la tabla hash. De este modo, en el ejemplo anterior, el registro de Tom se insertará en la posición 10.

### Encadenamiento

Cada posición de la tabla hash almacena una lista enlazada. Siempre que se añade un registro en una posición de la tabla ocupada por otro, el nuevo registro se añade simplemente a la lista enlazada. Para el ejemplo anterior, el registro de Tom se añadiría a la lista enlazada en la posición 7.

### Doble hashing

Cuando sucede una colisión, se llama a una segunda función hash con la clave de búsqueda del nuevo elemento. Digamos que esto devuelve el valor  $x$ . A continuación, comenzando desde la posición indicada por la primera función hash, se comprueba cada posición  $x$  de la tabla hash hasta hallar una posición libre, en la cual se ubicará el nuevo registro. Observe que la función hash secundaria no debería devolver cero, o de lo contrario este esquema fracasaría.

Aquí, construimos la función hash secundaria como:

$$h'(k) = M - 2 - k\%(M-2)$$

donde  $M$  es el tamaño de la tabla hash. Por tanto, el valor de la segunda función hash se añadiría al resultado de la primera función hash para producir el nuevo índice de la tabla hash. Para el ejemplo anterior,  $M=20$ , la clave es el número de identificación, que en el caso de Tom es  $k=5567$ ,  $h'(k)=13$ . Por lo tanto, el registro de Tom se insertará en la posición  $(13+7)\%20 = 0$ .

### Clasificación

Generalmente, se utiliza la clasificación para ordenar volúmenes extensos de datos, como por ejemplo datos de facturación, donde resulta que la ordenación de datos facilita la búsqueda y manipulación de los mismos. Existen varios algoritmos de clasificación.

### Ordenación por inserción

Esta es una técnica muy similar a la selección de un conjunto de cartas. Primero, se coge una carta, a continuación, se coge otra y se coloca antes o después de la primera, de manera que las cartas estén ordenadas en la mano. Posteriormente, coja una tercera carta y colóquela en una posición tal que las cartas sigan ordenadas. Este proceso continua hasta que todas las cartas se han colocado correctamente en orden. Por ejemplo, para ordenar el array

7 4 8 2 1 3,

la ordenación por inserción funciona de la forma siguiente: comience por 7. A continuación, coja el siguiente elemento, 4,  $4 < 7$ , de manera que el 4 debería estar colocado antes del 7:

4 7 8 2 1 3

A continuación, coloque el 8:

4 7 8 2 1 3

luego el 2:

2 4 7 8 1 3

y así sucesivamente:

1 2 4 7 8 3

1 2 3 4 7 8

hasta que el array de los números esté ordenado.

Una posible implementación de ordenación por inserción es:

```
public void sort( Object [] d, int start, int end )
{
    Object key;
    int i, j;
    // j indexa el elemento que se va a insertar
    // i indexa las posibles posiciones donde podría insertarse el elemento j
    para ( j = start + 1; j <= end; j++ )
    {
        key = d[ j ];
        for ( i = j - 1; i >= 0 && compare( key, d[ i ] ) < 0; i-- )
            d[ i + 1 ] = d[ i ];
        d[ i + 1 ] = key;
    }
}
```

## Mergesort

*Mergesort* es un algoritmo de ordenación que utiliza la técnica de diseño "divide y vencerás". La idea básica consiste en dividir los datos en dos conjuntos de datos iguales o prácticamente iguales. En cada nivel, ordene los subconjuntos y, a continuación, fusione el resultado. Este algoritmo se implementa fácilmente utilizando el método recursivo. El caso base es aquel en el que únicamente es necesario ordenar un elemento. Cuando fusiona el resultado, es necesario comparar los elementos correspondientes en dos subconjuntos y utilizar memoria adicional para almacenar el resultado temporal.

Seguimos utilizando el ejemplo anterior. Ahora queremos ordenar el siguiente array:

7 4 8 2 1 3

dividir en dos grupos:

7 4 8 ^ 2 1 3

dividir otra vez:

7 4 ^^ 8 ^ 2 1 ^^ 3

dividir de nuevo:

7 ^^^ 4 ^^ 8 ^ 2 ^^^ 1 ^^ 3

no se puede dividir de nuevo, por lo que se fusiona un nivel:

4 7 ^^ 8 ^ 1 2 ^^ 3  
 fusionar otra vez:  
 4 7 8 ^ 1 2 3  
 y fusionar de nuevo:  
 1 2 3 4 7 8

### Quicksort (ordenación rápida)

Es una buena técnica que requiere un tiempo de ejecución  $O(n \log_2 n)$  por promedio y, en el peor de los casos, un tiempo de ejecución  $O(n^2)$  cuando el número de datos es  $n$ .

*Quicksort* es también un método de ordenación basado en la táctica "divide y vencerás". La idea básica es dividir el conjunto de datos en dos y, a continuación, ordenar estos por separado. La siguiente implementación *quicksort* es más sencilla, aunque tiene defectos que conducen al peor caso de funcionamiento en situaciones a las que probablemente se enfrentará el algoritmo. (Rogamos tenga en cuenta que esta implementación es diferente a la que aprendimos en clase).

```

public void sort( Object [] d, int start, int end )
{
    Object o;
    if ( start >= end )
        return;
    int p = partition( d, start, end );
    sort( d, start, p );
    sort( d, p+1, end );
}

private int partition( Object [] d, int start, int end )
{
    //En un principio se indica el pivote como valor del elemento situado más a la izquierda del
    subarray.
    Object pivot = d[ start ];
    Object o;
    int low = start - 1;
    int high = end + 1;
    while ( true )
    {
        //Escanea la izquierda desde el extremo derecho del subarray en busca de un elemento
        inferior o igual al pivote.
        //Observe que el cuerpo de while está vacío. Toda la acción se realiza en
        --high.
        //Observe que, puesto que estamos utilizando un operador de predecremento, high
        se inicializa end+1.
        while ( compare( pivot, d[ --high ] ) < 0 );
        //igualmente, escanea la derecha desde el extremo izquierdo en busca de un elemento
        superior o igual al pivote.
        while ( compare( pivot, d[ ++low ] ) > 0 );
        //En este punto, si low y high no se han cruzado, entonces tenemos un elemento indexado
    }
}

```

por **low** que

```
// pertenece al área por encima del pivote y un elemento indexado por high que pertenece al área por debajo del pivote.
```

```
if ( low < high )
```

```
{ o = d[ low ]; d[ low ] = d[ high ]; d[ high ] = o; } //Los intercambiamos.
```

```
//Si low y high se han cruzado, ya hemos terminado, y la actual posición de high es devuelta como pivote, es decir, como el extremo superior de la división inferior.
```

```
de otro modo,
```

```
return high;
```

```
}
```

```
}
```

Trate este proceso siguiendo el método **partition()**. Observe que la división superior no se queda completamente ordenada, y en esta versión de *quicksort*, el valor de división no termina por separar las divisiones.

Para evitar el peor caso de *quicksort*, podemos seleccionar la mediana de los elementos primero, medio, y último como pivote. Coloque el más pequeño de los tres en primer lugar (start), el mayor en último lugar (end) y la mediana en segundo lugar. A continuación, solamente necesitamos el elemento de división[start+1] para el elemento[end-1].

Introducción del marco de Java Collection

El siguiente tutorial de *Collection* se ha compilado de los tutoriales en línea de [Sun Microsystems](#)

## Definiciones

Una *collection* es simplemente un objeto que agrupa múltiples elementos en una única unidad. Las *collections* se utilizan para almacenar, recuperar y manipular datos, así como para transmitir datos de un método a otro. Generalmente, representan elementos de datos que forman un grupo natural, como una mano de póker (un grupo de cartas de juego), una carpeta de correo (un grupo de cartas), o una agenda de teléfonos (un grupo de nombres relacionados con unos números de teléfono).

El *framework* de las *collections* es una arquitectura unificada de representación y manipulación de *collections*. Todos estos marcos contienen tres cosas:

**Interfaces:** tipos de datos abstractos que representan las *collections*. Las interfaces permiten la manipulación de las *collections* independientemente de los detalles de su representación. En los lenguajes como Java, orientados a los objetos, estas interfaces forman generalmente una jerarquía.

**Implementaciones:** implementaciones concretas de las interfaces de la *collection*. En lo esencial, estas son consideradas *estructuras de datos reusables*.

**Algoritmos:** métodos que realizan computaciones útiles, como por ejemplo la búsqueda y la ordenación de objetos que implementan interfaces de la *collection*. Se dice que estos algoritmos son *polimórficos* porque se puede utilizar el mismo método en muchas implementaciones diferentes de la interfaz adecuada de las *collections*. En lo esencial, los algoritmos son de *funcionalidad reusable*.

## Interfaces

### La interfaz de la collection

Una *collection* representa un grupo de objetos, conocido como sus *elementos*. El principal uso de la interfaz de la *collection* es pasar colecciones y manipularlas cuando se desea máxima generalidad. Por ejemplo, por convencionalismo, todas las implementaciones de *collection* de uso general (que

generalmente implementan alguna subinterfaz de *collection* como *Set* o *Lista*) tienen un constructor que obtiene un argumento de *collection*. Este constructor inicializa la nueva *collection* para que contenga todos los elementos en la *collection* especificada.

Este constructor permite al cliente crear una *collection* a partir de un tipo de implementación deseada que inicialmente contiene todos los elementos de cualquier *collection*, cualquiera que sea la subinterfaz o el tipo de implementación. Suponga que tiene una *collection*, *c*, que puede ser una Lista, un Set, o cualquier otro tipo de *collection*. La siguiente agrupación crea una nueva lista de array (una implementación de la interfaz List), que en principio contiene todos los elementos de *c*:

```
List l = new ArrayList(c);
```

A continuación se muestra la interfaz de la *collection*:

```
public interface Collection {
    // Operaciones básicas
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element); // Opcional
    boolean remove(Object element); // Opcional
    Iterator iterator();

    // Operaciones bulk
    boolean containsAll(Collection c);
    boolean addAll(Collection c); // Opcional
    boolean removeAll(Collection c); // Opcional
    boolean retainAll(Collection c); // Opcional
    void clear(); // Opcional

    // Operaciones de array
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

La interfaz realiza lo esperado, teniendo en cuenta que la *collection* representa un grupo de objetos. Posee métodos que nos indican el número de elementos de una *collection* (*size*, *isEmpty*), que comprueban si un determinado objeto se encuentra en la *collection* (*contains*), que añaden un elemento a la *collection* o lo eliminan (*add*, *remove*), y que proporcionan un iterador sobre la *collection* (*iterator*).

El método de adición se define generalmente lo suficiente como para que tenga sentido en el caso de las *collections* que permiten duplicados y de las que no. Garantiza que la *collection* contendrá el elemento especificado una vez completada la llamada, y devuelve true si la *collection* cambia como resultado de la llamada. Igualmente, el método de supresión se define para eliminar de la *collection* un *único ejemplo* del elemento especificado, suponiendo que la *collection* contenga ese elemento, y devolver true si como resultado se modificó la *collection*.

## La interfaz Set

Un *set* es una *collection* en el que no contiene elementos duplicados. Un *set* modela la bastracción matemática del conjunto. La interfaz *set* extiende *collection* y no contiene más métodos que los heredados de la *collection*. Agrega la restricción de prohibir elementos duplicados. También agrega un contrato más sólido sobre el comportamiento de las operaciones equals y hashCode, permitiendo que objetos *set* con distintos tipos de implementaciones se comparen de manera significativa. Dos objetos *set* son iguales si contienen los mismos elementos.

A continuación se muestra la interfaz *set*:

```
public interface Set {
    // Operaciones básicas
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element); // Opcional
    boolean remove(Object element); // Opcional
    Iterator iterator();

    // Operaciones bulk
    boolean containsAll(Collection c);
    boolean addAll(Collection c); // Opcional
    boolean removeAll(Collection c); // Opcional
    boolean retainAll(Collection c); // Opcional
    void clear(); // Opcional

    // Operaciones de array
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

#### La interfaz List (lista)

Una list es una *collection* ordenada (en ocasiones denominada *secuencia*). Las lists pueden contener elementos duplicados. Además de las operaciones heredadas de la *collection*, la intefaz lista incluye operaciones para:

Acceso posicional: manipula elementos basándose en su posición numérica en la lista.

Búsqueda: busca un objeto concreto de la lista y le devuelve su posición numérica.

Iteración Lista: amplía la semántica del iterador para aprovecharse de la naturaleza secuencial de la lista.

*Range-view*: realiza *operaciones de rango* arbitrarias en la lista.

A continuación se muestra la interfaz List:

```
public interface List extends Collection {
    // Acceso posicional
    Object get(int index);
    Object set(int index, Object element); // Opcional
    void add(int index, Object element); // Opcional
    Object remove(int index); // Opcional
}
```

```

abstract boolean addAll(int index, Collection c); // Opcional

// Búsqueda
int indexOf(Object o);
int lastIndexOf(Object o);

// Iteración
ListIterator listIterator();
ListIterator listIterator(int index);

// Range-view
List subList(int from, int to);
}

```

## La interfaz Map

Un Map es un objeto que asigna claves a valores. Un map no puede contener claves duplicadas. Cada clave puede asignarse a un valor como máximo. A continuación se muestra la interfaz Map:

```

public interface Map {
    // Operaciones básicas
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Operaciones bulk
    void putAll(Map t);
    void clear();

    // Vistas de la collection
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interfaz para elementos entrySet
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}

```

## Implementaciones

### Set

Las dos implementaciones generales de [Set](#) son [HashSet](#) y [TreeSet](#). La decisión acerca de cuál de ellas utilizar es sencilla. [HashSet](#) es mucho más rápido (tiempo constante frente a tiempo prolongado para la mayoría de las operaciones), pero no ofrece garantías de ordenación. Si necesita utilizar las operaciones de [SortedSet](#), o si para usted es importante la iteración en orden, utilice [TreeSet](#). De lo contrario utilice [HashSet](#). Muy probablemente terminará utilizando [HashSet](#) la mayoría de las veces.

## Lista

Las dos implementaciones generales de [Lista](#) son [ArrayList](#) y [LinkedList](#). La mayoría de las veces utilizará probablemente [ArrayList](#). Esta ofrece acceso posicional de tiempo constante, y es sencillamente más rápida, ya que no tiene que asignar un objeto de nodo a cada elemento de la lista y puede aprovecharse del método nativo `System.arraycopy` cuando tiene que desplazar múltiples elementos al mismo tiempo. Piense en [ArrayList](#) como [Vector](#) sin el overhead de sincronización.

## Map (mapa)

Las dos implementaciones generales de [Map](#) son [HashMap](#) y [TreeMap](#). La situación para [Map](#) es *exactamente* análoga a la de [Set](#). Si necesita operaciones [SortedMap](#) o una iteración en orden de vista de la collection, vaya a [TreeMap](#); de lo contrario, elija [HashMap](#). Todo lo demás que aparece en la [sección Set](#) se aplica también a [Map](#). Lea esta sección e infórmese.

## Algoritmos

### Ordenación

El algoritmo `sort` reordena una lista, de manera que sus elementos estén en orden ascendente, según cierta relación de orden. Se ofrecen dos formas de funcionamiento: la más sencilla toma simplemente una `list` y la ordena según el *orden natural* de sus elementos.

Clase	Orden natural
Byte	numérico con signo
Character	numérico sin signo
Long	numérico con signo
Integer	numérico con signo
Short	numérico con signo
Double	numérico con signo
Float	numérico con signo
BigInteger	numérico con signo
BigDecimal	numérico con signo
File	lexicográfico dependiente del sistema en pathname

String	lexicográfica
Date	fecha cronológica
CollationKey	lexicográfica específica del lugar

La operación `sort` utiliza un algoritmo *mergesort* ligeramente optimizado.

A continuación, le indicamos un pequeño programa que imprime sus argumentos en orden lexicográfico o alfabético.

```
import java.util.*;
public class Sort {
    public static void main(String args[]) {
        List l = Arrays.asList(args);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

Ejecutemos el programa con el argumento "i walk the line" y veamos como produce lo siguiente:

```
i, line, the, walk
```

La segunda forma de ordenación toma un [Comparador](#), además de una lista, y ordena los elementos con el comparador. Los detalles se pueden consultar en [Collections Tutorial](#) en la página web de Sun.

## Búsqueda

El algoritmo de búsqueda binaria (`binarySearch`) busca un elemento determinado en una lista ordenada utilizando el algoritmo de *búsqueda binaria*. Este algoritmo tiene dos formas. La primera toma una lista y un elemento para buscar (la clave de búsqueda o "search key"). Esta forma supone que la lista está ordenada en orden ascendente según el orden natural de sus elementos. La segunda forma de la llamada toma un comparador, además de la lista y la clave de búsqueda, y supone que la lista está ordenada en orden ascendente según el comparador especificado. El algoritmo `sort`, descrito anteriormente, puede utilizarse para ordenar la lista antes de llamar a la búsqueda binaria.

El valor devuelto es el mismo para las dos formas: si la lista contiene la clave de búsqueda, se devuelve su índice. De no ser así, el valor devuelto es  $-(\textit{insertion point}) - 1$ , donde el punto de inserción (*insertion point*) se define como el punto en el que el valor se insertaría en la lista: el índice del primer elemento superior al valor, o `list.size()` si todos los elementos de la lista son inferiores al valor determinado. Esta fórmula tan poco grata se seleccionó con el fin de garantizar que el valor devuelto sea  $\geq 0$  si, y sólo si, se halla la clave de búsqueda.

Básicamente, es un hack para combinar una booleana ("found") y un entero ("index") en un único valor `int` devuelto.

El siguiente lenguaje, que puede utilizarse con las dos formas de la operación de búsqueda binaria, busca una clave de búsqueda específica y la inserta en la posición adecuada en el caso de no existir con anterioridad:

```
int pos = Collections.binarySearch(l, key);
if (pos < 0)
    l.add(-pos-1, key);
```

## Problemas del tutorial

### Problema 1

Indique si las siguientes afirmaciones son verdaderas o falsas. En el caso de ser falsas, razone su respuesta.

- En una ordenación por inserción (*insertion sort*) se seleccionan los elementos de uno en uno desde una lista ordenada y se insertan en un conjunto desordenado.
- Mergesort y quicksort son algoritmos de ordenación del tipo divide y vencerás.
- La idea básica de quicksort es dividir el conjunto de datos en dos y, a continuación, ordenar cada uno de ellos con quicksort de manera recursiva.
- Una *Collection* representa un grupo de objetos. Todas las implementaciones de la *collection* están desordenadas.
- Un *Set* es una *collection* que no puede contener elementos duplicados.

### Problema 2

#### Parte A

Dada la siguiente lista

10 20 30 40 25 35

Considere el algoritmo de ordenación por inserción tratado en clase:

```
public void sort( Object [] d, int start, int end )
{
    Object key;
    int i, j;
    // j indexa el elemento que ha de insertarse
    // i indexa las posibles posiciones en las que podría insertarse el elemento j
    for ( j = start + 1; j <= end; j++ )
    {
        key = d[ j ];
        for ( i = j - 1; i >= 0 && compare( key, d[ i ] ) < 0; i-- )
            d[ i + 1 ] = d[ i ];
        d[ i + 1 ] = key;
    }
}
```

(1) ¿Cuántas comparaciones requeriría este array a una ordenación por inserción para producir una ordenación completa?

(2) Muestre el estado del array después de cada comparación.

(3) En este método de ordenación por inserción tenemos dos bucles. En su respuesta al apartado (2), marque el estado del array al final de cada iteración del bucle más alejado.

### Parte B

Simule la ejecución de quicksort en el siguiente array. Utilice el algoritmo más sencillo de quicksort tratado en clase para obtener los dos primeros subarrays.

6 12 18 9 15 3

Los dos subarrays son: \_\_\_\_\_ y \_\_\_\_\_

### Problema 3

Dado el método toString() de la clase TreeMap. La entrada se define de la siguiente forma, public class TreeMap extends AbstractMap implements SortedMap, ....{

... ..

```
private static class Entry implements Map.Entry {
    Object key;
    Object value;
    ... ..

    public String toString() {return key + "=" + value;} }
}
```

Consulte la documentación en línea de Java y escriba el resultado del siguiente código.

```
import java.util.*;
```

```
public class MapExample {
    public static void main(String args[]) {
        Map map = new HashMap();

        String names[] = {"Ben", "Julia", "Steve", "Alice", "Jennifer", "Steve",
            "Andrew", "Melissa", "Alice", "Steve"};
        for (int i=0, n=names.length; i<n; i++) {

            String key = name[i];
            Integer frequency = (Integer)map.get(key);
            if (frequency == null) {
                frequency = new Integer(1);
            } else {
                int value = frequency.intValue();
                frequency = new Integer(value + 1);
            }
            map.put(key, frequency);
        }
        Map sortedMap = new TreeMap(map);
    }
}
```

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

Escriba el resultado a continuación:

### Ejercicios de diseño

Los profesores adjuntos del curso 1.00 deciden construir un sistema de información para los estudiantes. El objetivo principal del sistema es permitir la consulta rápida de las calificaciones de los estudiantes. Los profesores están considerando utilizar las siguientes estructuras de datos para que los estudiantes consulten el sistema. La lista enlazada (Linked List), que almacena a cada estudiante como un nodo. Las tablas Hash

El árbol de búsqueda binario

Responda a las siguientes preguntas sobre las estructuras de datos anteriores.

**Parte 1.** Si los árboles de búsqueda binaria se utilizasen para almacenar las calificaciones de los estudiantes, responda a lo siguiente:

A. De los siguientes puntos, ¿cuál se utilizaría como clave si el principal objetivo fuese permitir una consulta rápida de la calificación de un estudiante en particular?

(a) Calificación del estudiante.

(b) Identificación del estudiante.

B. Justifique su respuesta anterior comparando la eficacia del proceso de consulta del estudiante utilizando las dos posibilidades anteriores como clave.

**Parte 2.** ¿Qué estructura de datos recomienda que utilicen los profesores adjuntos y por qué? Razone su respuesta comparando el funcionamiento del tiempo de búsqueda utilizando las tres estructuras de datos cuando el registro almacenado del número de estudiantes es extenso. En sus respuestas, especifique claramente qué claves se utilizarían para las tablas hash y los árboles de búsqueda binaria, respectivamente.

**Parte 3.** ¿Cuál de las siguientes afirmaciones es verdadera o falsa si se comparan las tres estructuras de datos anteriores?:

a) La lista enlazada proporciona el tiempo de búsqueda más rápido (tiempo que se tarda en consultar la calificación de un estudiante en particular) sin importar el número de estudiantes que componen la lista.

b) En el peor caso, el tiempo de búsqueda de los árboles de búsqueda binaria (es decir, el tiempo que se tarda en consultar un elemento en particular) es el mismo que el de las listas enlazadas.

c) Si se desea obtener los elementos de datos ordenados, las tablas hash son mejor opción que los árboles de búsqueda binaria.

d) Para ubicar un elemento con una clave determinada en una tabla hash, se debe comparar la clave con las claves de todos los buckets.

**Parte 4.** Proporcione el ejemplo de una secuencia de entrada de 5 valores enteros que dará como resultado un árbol de búsqueda binaria con el peor caso de tiempo de búsqueda. Utilice 5 valores enteros para su ejemplo de secuencia de entrada en los siguientes espacios en blanco.

1. \_\_\_\_\_ 2. \_\_\_\_\_ 3. \_\_\_\_\_ 4. \_\_\_\_\_ 5. \_\_\_\_\_

Dibuje un diagrama del árbol de búsqueda binaria construido utilizando su ejemplo de secuencia de entrada:

**Parte 5.** Los profesores adjuntos han escrito la siguiente clase Student para ser almacenada como clave en la tabla hash.

A. Ayude a los profesores a terminar la clase Student que se indica más adelante escribiendo el código para iguales, de forma que los estudiantes con el mismo identificador sean considerados iguales.

```
public class Student {
    String name;
    int id;
    int grade;

    public int hashCode() {
        return id;
    }

    public boolean equals (Object s) {

        //inserte su código abajo

    }
}
```

B. Suponga que se utiliza el encadenamiento como estrategia de resolución de colisiones mediante las tablas hash. Suponga también que las tablas hash utilizan la siguiente función hash para asignar elementos a un bucket:

$$h \% m,$$

donde h es el código hash del elemento (es decir, el valor clave), y m el tamaño de la tabla. Facilite una secuencia ejemplo de 5 valores de identificación que darán como resultado una tabla hash con el peor caso de comportamiento de búsqueda utilizando la clase Student anterior como clave de la tabla hash cuando m tiene un valor de 100.