Tema 6:

Estructuras de datos

***Objetivos**: La resolución de un problema puede complicarse o simplificarse notablemente dependiendo de la representación elegida para los datos. Los tipos de datos básicos que hemos visto hasta ahora a menudo resulta una representación demasiado simple para los datos de los problemas del mundo real. En este tema veremos conjuntos de datos más avanzados que pueden simplificar la resolución de problemas complejos.*

INDICE:

1 Introducción a las estructuras de datos

Las estructuras de datos son colecciones de variables **homogéneas** o **heterogéneas** con algún tipo de relación entre ellas. Estas estructuras de datos pueden clasificarse según sus propiedades:

- En función de su almacenamiento, en datos internos y datos externos. Los datos **internos** son los que residen en la memoria principal del ordenador, mientras que los **externos** residen en los dispositivos de almacenamiento masivo.
- Dependiendo si el tamaño de los datos puede o no cambiar durante la ejecución del programa los datos se clasifican en **estáticos** (su tamaño no puede variar durante la ejecución del programa) y **dinámicos** (su tamaño sí varía durante la ejecución del programa).
- Dependiendo de las relaciones que guardan los datos entre si se pueden clasificar en datos **lineales**, aquellos que están enlazados a un elemento anterior y a uno posterior, y datos **no lineales**, que pueden estar enlazados a múltiples elementos.
- Por último, si cada dato individual está formado a su vez por un conjunto de datos simples o primitivos (carácter, real, entero o lógico) se dice que los datos son **compuestos**.

Según vayamos introduciendo distintas estructuras de datos iremos mostrando sus implementaciones dentro del lenguaje Java. Para ello, necesitamos primero conocer un nuevo aspecto de la sintaxis del lenguaje Java que todavía no hemos abordado: su soporte para el polimorfismo paramétrico o plantillas. Esta característica se incorporó al lenguaje en Java 5, por lo que no está disponible en versiones anteriores.

2 Polimorfismo paramétrico en Java

El polimorfismo paramétrico, plantillas o (empleando un término inglés) "Generics" permite que un objeto pueda usarse uniformemente como parámetro en distintos contextos o situaciones. Las clases genéricas o paramétricas permiten definir clases que no se quiere o puede especificar completamente hasta determinar un tipo de dato. En estas clases el tipo concreto de datos que manipulan se especifica como un parámetro en el momento en que se crean los objetos de la clase.

Como todo tipo de polimorfismo, el polimorfismo paramétrico trata de abstraernos de algún detalle y permitirnos trabajar a un mayor nivel de abstracción. El "detalle" del que este polimorfismo trata de abstraernos es el tipo de dato concreto sobre el cual vamos a operar. Por ejemplo, supongamos que queremos construir una lista en Java; esto es, una estructura que puede crecer de modo dinámico y que

contiene un conjunto de datos, cada uno de los cuales está asociado con un dato anterior y un dato siguiente. Dado el primer dato de la lista, yo puedo recorrer todos los demás datos siguiendo las asociaciones de cada dato con su dato siguiente; también puedo realizar operaciones de borrado o inserción de datos en cualquier punto de la lista.

Para construir una estructura de este tipo, necesitaré algún tipo de contenedor que me permita almacenar tanto el dato que quiero contener, como algún tipo de referencia que me permita localizar cuál es el dato siguiente, y cuál es el dato anterior. ¿Cuál sería la diferencia entre construir una lista de coches, una lista de lápices, float, int o una lista de java.lang.Object? La única diferencia sería el tipo de dato que contiene la lista; la lógica de todas las operaciones de borrado, de inserción, de recorrido de la lista... sería idéntica. Si un lenguaje de programación me proporciona algún mecanismo para "abstraerme" del tipo de dato concreto que está manipulando mi lista, yo podría programar una lista que contenga cualquier dato.

Para esto es, precisamente, para lo que vale el polimorfismo paramétrico o generics: me permite especificar uno o varios datos que parametrizan una clase Java. La sintaxis para definir una clase Java genérica es la siguiente:

```
public class NombreClaseGenerica<E>{
   E dato;
   E getDato () {return dato; }
}
```

Para crear una clase genérica, simplemente después del nombre de la clase tenemos que poner el parámetro (o lista de parámetros: NombreClaseGenerica<A,B,C,...>) que parametrizan la clase. E puede ser cualquier clase Java. No puede, sin embargo, ser un tipo de dato primitivo (float, int ,...); en caso de necesitar que una clase sea para métrica respecto a un tipo de dato primitivo deberemos emplear no el tipo de dato primitivo, sino la clase wrapper correspondiente (Float, Integer ,...). Al construir un objeto de esta clase será necesario indicar cuál es el tipo de dato con el cual vamos a parametrizar esa instancia concreta de ese objeto; por ejemplo:

```
NombreClaseGenerica<Float> instancia1 = new
       NombreClaseGenerica<Float> ();
NombreClaseGenerica< MiClase > instancia3 = new
       NombreClaseGenerica< MiClase > ();
```

La primera sentencia construirá un objeto que pertenecerá a una clase que, a todos los efectos, estará definida como sigue:

```
public class NombreClaseGenerica{
```

```
   Float dato;
   Float getDato () {return dato; }
}
```

mientras que el objeto creado en la segunda sentencia pertenecerá a una clase de la forma:

```
public class NombreClaseGenerica{
   MiClase dato;
   MiClase getDato () {return dato; }
}
```

Una forma de ver el polimorfismo paramétrico es como un mecanismo para crear un conjunto ilimitado de clases donde la única diferencia entre ellas son los tipos de los datos sobre los que operan. Para aquellos que conozcan C++, mencionar que en Java el soporte para polimorfismo paramétrico es sólo a nivel de compilador: una vez el código ha sido compilado toda la información relativa a tipos genéricos se pierde, y sólo se obtiene una única clase compilada (a diferencia de C++, donde se obtiene una clase por cada tipo de parámetro diferente que hayamos empleado para parametrizar una plantilla). Con este esquema, el compilador puede realizar los chequeos de tipo de dato al trabajar con clases genéricas, y a la vez se mantiene compatibilidad con versiones de Java anteriores a Java 5, ya que en las clases compiladas no hay ningún tipo de información paramétrica.

En el listado 1 mostramos un código de ejemplo donde hay una clase que se parametriza con un tipo de dato. La clase tiene dos atributos del tipo que se emplea para parametrizarla, y permite modificarlos mediante dos métodos. La clase también tiene un método que concatena ambos datos en una cadena de caracteres. Desde otra clase que contiene el método *main* se crean tres instancias diferentes de la primera empleando distintos tipos de dato como parámetro.

```
package estructurasdedatos;

class Concatenador<E> {

    private E dato1;
    private E dato2;

    public String toString() {
        return "El primer dato es: " + getDato1()
                + " y el segundo es: " + getDato2();
    }

    public E getDato1() {
        return dato1;
    }

    public void setDato1(E dato1) {
```

```
        this.dato1 = dato1;
    }

    public E getDato2() {
        return dato2;
    }

    public void setDato2(E dato2) {
        this.dato2 = dato2;
    }
}

public class Ejemplo1 {

    public static void main(String[] args) {
        Concatenador<Float> concatenador1 = new
Concatenador<Float>();
        concatenador1.setDato1(42.35F);
        concatenador1.setDato2(2.365F);
        Concatenador<String> concatenador2 = new
Concatenador<String>();
        concatenador2.setDato1("Hola");
        concatenador2.setDato2("Adiós");
        Concatenador<Integer> concatenador3 = new
Concatenador<Integer>();
        concatenador3.setDato1(23);
        concatenador3.setDato2(246);
        System.out.print(concatenador1 + "\n" + concatenador2
+ "\n" + concatenador3);
    }
}
```

3 Arrays

Son estructuras estáticas de datos homogéneos que están constituidas por un número fijo de elementos situados en direcciones contiguas de memoria. Dependiendo de su dimensionalidad se clasifican en:

- Unidimensionales, son los llamados vectores o arrays.
- Bidimensionales, también llamados matrices.
- Multidimensionales, o poliedros.

Se suelen emplear para representar colecciones de datos homogéneas relacionadas lógicamente ellas y de carácter estático, como puede ser los días de la semana (array) o un listado de las notas que cada alumno de una clase ha obtenido en las diferentes asignaturas (matriz).

Posición	0	1	2	3	4	5	6

Lunes	Martes	Miércoles	Jueves	Viernes	Sábado	Domingo

Alumno/ asignatura	Alumno 1	Alumno 2	Alumno 3	Alumno 4
Asignatura 1	8	6	9	3
Asignatura 2	6	4	9	5
Asignatura 3	4	7	6	5

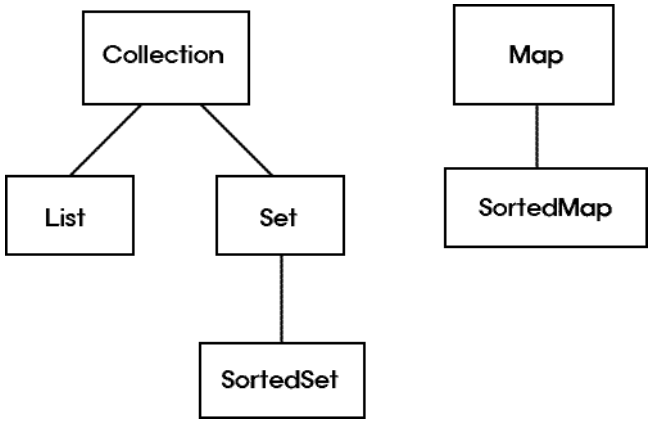
Con esta estructura de datos ya hemos trabajado tanto en C como en Java. Es interesante reflexionar sobre varias de sus propiedades:

- **Tiempo de acceso a los datos**: el tiempo de acceso a los datos en un array es constante. Todos los datos almacenados en un array se encuentran en posiciones contiguas de memoria RAM. Por otro lado, se trata de una colección de datos homogénea; esto es, todos los datos son del mismo tipo y por lo tanto del mismo tamaño. Por tanto, conociendo donde comienza el array y sabiendo el índice correspondiente con el dato al cual queremos acceder podemos saltar directamente a la posición de memoria RAM donde se encuentra. Esto se traduce en que acceder a cualquier dato de un array tiene una complejidad O(1).
- **Son estructuras estáticas**: no podemos por tanto añadir posiciones o eliminar posiciones del array. Cuando se quiere cambiar el tamaño de esta estructura a menudo hay que reservar espacio en otra dirección de memoria RAM para una nueva estructura tipo array y eliminar el espacio correspondiente con la anterior. La única excepción es el escenario en el cual queramos hacer crecer el array y "tengamos suerte" y las posiciones contiguas inmediatamente posteriores al fin del array estén libres y sean suficientes para extenderlo.
- **Operaciones de borrado e inserción**: Si queremos borrar un elemento del array, o insertar un elemento en él, esta operación puede requerir desplazar "hacia atrás" todos los elementos posteriores al elemento borrado, o "hacia adelante" todos los elementos posteriores al elemento que vamos a insertar para hacer hueco para éste. El peor de los escenarios sería insertar un elemento al principio del array, o borrar un elemento al principio del array, que puede requerir desplazar todos los elementos del array. Por tanto, una operación de borrado o insertado en el array tiene una complejidad en el peor de los casos O(n), donde n es el número de datos del array. Por norma general, no es una buena idea eliminar o añadir elementos en medio de un array (si es eficiente eliminar o añadir elementos al final del array). Si necesitamos soportar estas operaciones deberemos evaluar emplear otra estructura de datos.

- **Operaciones de búsqueda**: en el caso general en el cual los elementos del array no se encuentren ordenados, tendremos que emplear una búsqueda secuencial y las operaciones de búsqueda tendrán una complejidad O(n). Si los elementos se encuentran ordenados es trivial emplear una búsqueda dicotómica donde la complejidad es O(log(n)).

4 El framework de Collections

El framework de Collections fue incorporado a Java en la versión 1.2 por el arquitecto Joshua Bloch, uno de los principales gurús de Java. En Java 5 este framework sufrió una revisión bastante fuerte, el último trabajo que Bloch realizó en Sun Microsystems antes de irse a trabajar para Google, que consistió en soportar tipos de datos genéricos. Este framework se construye sobre las seis interfaces que, junto con sus relaciones jerárquicas, se muestran en la figura. En esta sección describiremos brevemente cada una de esas interfaces, así como sus principales implementaciones.



4.1 Collection

Se trata de una de las dos clases raíz de la jerarquía. Representa cualquier colección de objetos que pertenezcan a un mismo tipo y captura el comportamiento común a cualquier colección de objetos. No hay ninguna implementación directa de esta interfaz, ya que esta interfaz define una funcionalidad demasiado abstracta para tener sentido como estructura de datos. Esta interfaz proporciona operaciones para añadir y eliminar objetos de la colección, recorrer la colección, conocer el tamaño de la colección, etcétera. Más concretamente, los métodos que proporciona esta interfaz pueden clasificarse en tres categorías. Dentro de la primera, están los métodos para agregar y eliminar elementos de la lista:

- boolean add(E element): añade a la colección el elemento que se le pasa como argumento. Devuelve true si la operación se realiza con éxito, false en caso contrario.
- boolean addAll(Collection<E> collection): añade a la colección todos los elementos de la colección que se le pasa como argumento. El tipo de dato con el que se ha parametrizado la colección argumento, y el tipo de dato con el que se ha parametrizado la colección sobre la que

se invoca el método debe ser el mismo. Devuelve true si la operación se realiza con éxito, false en caso contrario.

- boolean remove(E element): elimina de la colección el elemento que se le pasa como argumento. Devuelve true si la operación se realiza con éxito, false en caso contrario.
- void clear(): elimina todos los elementos de la colección.
- void removeAll(Collection<E> collection): elimina todos los elementos de la colección sobre la que se invocó el método que estén contenidos en la colección que se le pasa como argumento.
- void retainAll(Collection<E> collection): elimina todos los elementos de la colección sobre la que se invocó el método que no estén contenidos en la colección que se le pasa como argumento.

El segundo tipo de métodos que contiene esta interfaz son métodos para realizar consultas:

- int size(): devuelve el número de elementos de la colección.
- boolean isEmpty(): devuelve true si la colección está vacía, false en caso contrario.
- boolean contains(E element): devuelve true si el elemento que se le pasa como argumento está contenido en la colección.
- boolean containsAll(Collection<E> collection): devuelve true si todos los elementos de la colección que se pasa como argumento están contenidos en la colección sobre la que se invocó el método, y false en caso contrario.

Todavía hay una media docena más de métodos definidos en esta interfaz (el lector puede conocerlos consultando su javadoc), pero nosotros sólo vamos a ver otro más, uno de los métodos más importantes: Iterator<E> iterator(). Este método nos devuelve un iterador. Un iterador es una interfaz que nos permite recorrer cualquier colección de elementos, y que tiene definidos tres métodos:

- boolean hasNext(): devuelve true si todavía quedan más elementos por visitar en la colección y false en caso contrario.
- E next(): devuelve el elemento al cual está apuntando el iterador y desplaza el iterador una posición hacia adelante dentro del contenedor.
- void remove(): elimina el elemento al cual está apuntando el iterador.

Una forma de recorrer una colección de elementos en Java es empleando un iterador:

```
Collection <E> collection=...
Iterator<E> it = collection.iterator();
while (it.hasNext()){//mientras queden elementos por recorrer
   E elemento = it.next ();
   ...//realizar algún procesamiento sobre el elemento
```

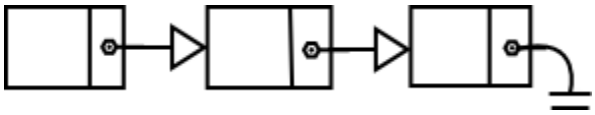
```
}
```

4.2 Listas

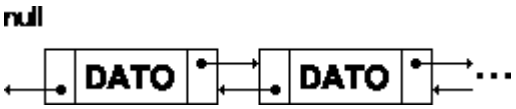
Es una estructura dinámica de datos constituida por elementos (que habitualmente son del mismo tipo) denominados nodos. Esta estructura se corresponde con el concepto de "lista" de la vida cotidiana: una lista de la compra, la lista de invitados a una boda o la lista de las películas más taquilleras se corresponden con esta estructura. En una lista es posible acceder a cualquier elemento, sin importar en la posición en la que esté, siendo posible eliminarlo o modificarlo y añadir elementos en cualquier posición.

Dependiendo de cómo se relacionan entre sí los distintos elementos de una lista, éstas se pueden clasificar en:

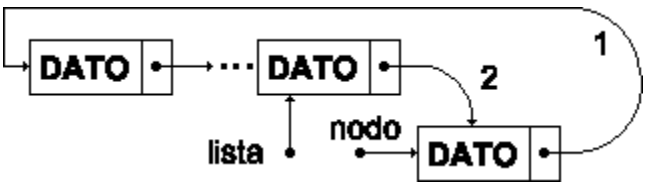
- **Simplemente enlazadas**: cada elemento de la lista está enlazado con el siguiente elemento.



- **Doblemente enlazadas**: cada elemento de la lista está enlazado con los elementos siguiente y anterior.



- **Listas circulares**: son aquellas listas en las cuales el último elemento está enlazado con el primero.



Las operaciones típicas que se pueden realizar sobre una lista son:

- Insertar, eliminar o buscar un nodo.
- Vaciar la lista de elementos.
- Extraer un subconjunto de elementos de la lista.
- Concatenar dos listas.
- Comprobar si la lista está vacía o no.

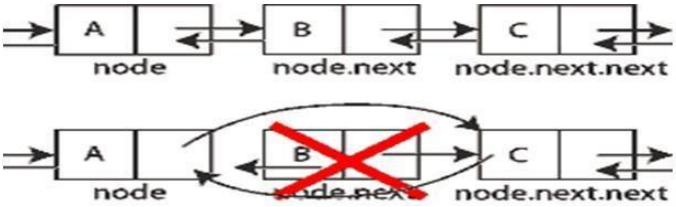
Respecto a la eficiencia de estas estructuras debemos tener en cuenta lo siguiente:

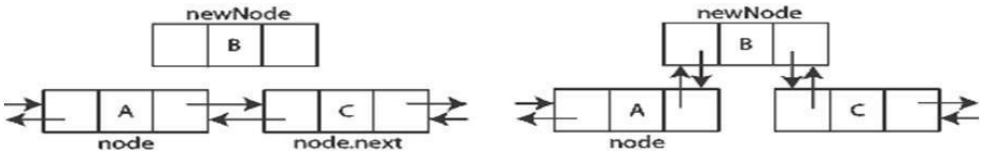
- **Tiempo de acceso a los datos**: cada uno de los nodos de una lista se puede encontrar en cualquier posición de memoria RAM. Dentro de una lista, cada nodo apunta al siguiente, o al

anterior y al siguiente si se trata de una lista doblemente enlazada (esto es, de algún modo contiene sus direcciones de memoria). No tenemos por tanto ninguna forma eficiente de saltar a un elemento determinado de la lista. Para llegar hasta un elemento de la lista, tenemos que comenzar por el primer nodo, saltar desde éste al segundo, desde éste al tercero… y así sucesivamente hasta que lleguemos al nodo que nos interesa. En el peor de los casos, si tenemos que acceder al último nodo de la lista, tendríamos que recorrer todos los nodos. Por tanto, acceder a un dato de una lista tiene en el peor de los casos una complejidad O(n) donde n es el tamaño de la lista.

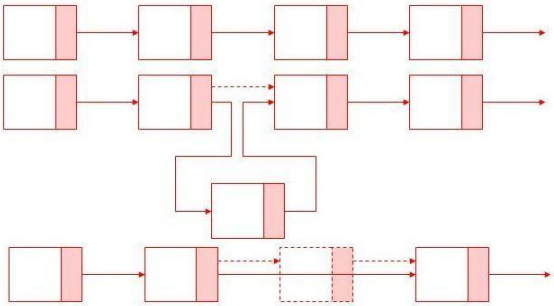
- **Son estructuras dinámicas**: al no tener la restricción de que los distintos elementos de una lista deben situarse en posiciones contiguas de memoria, es posible añadir nuevos elementos a la lista mientras nos quede memoria RAM disponible. Para ello, bastará con reservar la memoria para el nuevo nodo en cualquier región de RAM.

- **Operaciones de borrado e inserción**: Si queremos borrar un elemento de una lista basta con eliminar la memoria asignada a dicho nodo y reajustar los enlaces de la lista, de tal modo que el nodo anterior al nodo eliminado apunte ahora al nodo siguiente al nodo eliminado, y (si se trata de una lista doblemente enlazada) el nodo siguiente al eliminado apunte al anterior, como se muestra en la imagen. De un modo similar, si queremos insertar un elemento en la lista en una posición cualquiera basta con hacer que el elemento anterior a la posición donde queremos insertar apunte al nuevo nodo y (si se trata de una lista doblemente enlazada) que el nuevo nodo apunte al anterior; que el nuevo nodo apunte al siguiente y (si se trata de una lista doblemente enlazada) que el siguiente nodo apunte también al nuevo. Las operaciones de borrado e inserción en una lista tienen por tanto una complejidad O(1). La siguiente figura muestra la operación de insertado de un nodo en una lista doblemente enlazada.





La siguiente figura muestra las operaciones de insertado de un nodo en una lista simplemente enlazada, y de borrado de un nodo de una lista simplemente enlazada.

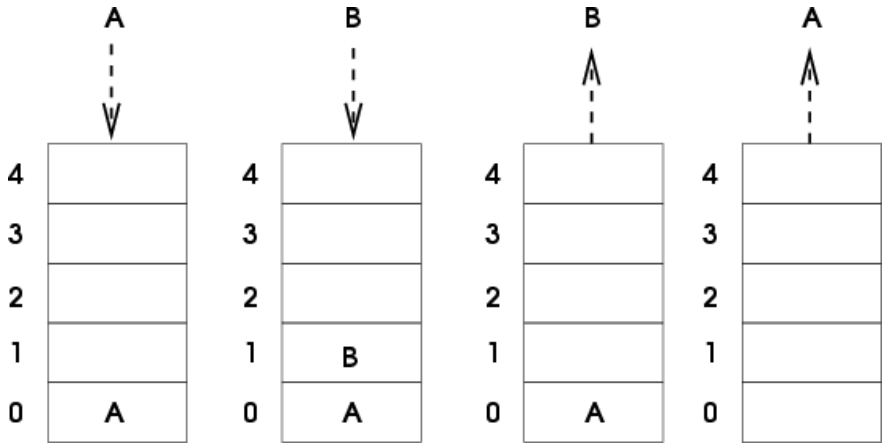


- **Operaciones de búsqueda**: al ser una estructura de datos que carece de acceso aleatorio, no es posible implementar un algoritmo tipo búsqueda dicotómica directamente sobre una lista; siempre habrá que emplear una búsqueda tipo secuencial. Por tanto, las operaciones de buscar un elemento sobre una lista tienen una complejidad O(n).

Existen dos tipos de listas muy empleadas en computación que son casos particulares de las listas generales: las colas y las pilas.

Pilas

Una pila es una lista basada en el criterio LIFO (**last in-first out, el último que entra es el primero que sale**). Los elementos se añaden apilándose unos sobre otros y sólo se tiene acceso al que está en la parte superior.



Las pilas también son estructuras de uso común en la vida real: una pila de libros que está sobre una mesa, una pila de latas de coca cola en un supermercado o la pila de papeles que usa la impresora para imprimir. En computación, son empleadas tanto por los compiladores como por los intérpretes para ejecutar programas.

Las operaciones típicas que se pueden realizar sobre una pila son:

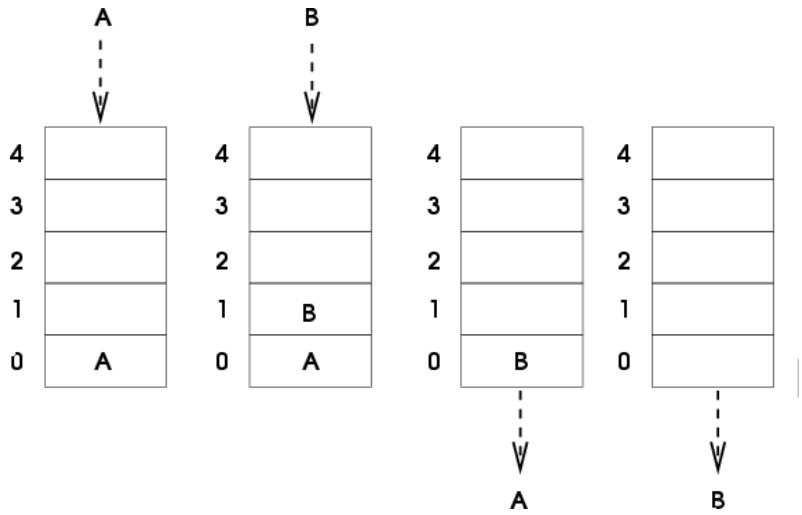
- Apilar un elemento en la parte superior (**push**).
- Extraer el elemento que se encuentra en la parte superior (**pop**).
- Comprobar si la pila está vacía.

- Vaciar la pila.
- Obtener el elemento que está encima de la pila sin extraerlo (**top**).

Dada la restricción de que sólo es posible eliminar y retirar elementos de la cima de la pila, las pilas pueden implementarse de modo eficiente empleando un array, en cuyo caso tendrían las propiedades de eficiencia de los arrays. En caso de que se implementen a través de una lista, tendrán las propiedades de eficiencia de las listas.

Colas

Una cola es una lista basada en el criterio FIFO (**first in-first out, el primero que entra es el primero que sale**). Los nuevos elementos se añaden en la parte de atrás y la extracción de elementos se realiza por la de delante.



Nuevamente, esta estructura informática se inspira en soluciones de la vida real: la cola del supermercado, la cola de trabajos en una impresora, o la cola del comedor universitario funcionan de modo similar a esta estructura.

Las operaciones típicas que se puede realizar sobre una cola son:

- Añadir un elemento a la cola.
- Extraer el primer elemento de la cola.
- Comprobar si la cola está vacía.
- Vaciar todos los elementos de la cola.
- Obtener el primer elemento de la cola sin extraerlo.

Dada la restricción de que sólo es posible añadir elementos al final de la cola, y retirar elementos del principio de la cola, las colas pueden implementarse de modo eficiente empleando un array, en cuyo caso tendrían las propiedades de eficiencia de los arrays. En caso de que se implementen a través de una lista, tendrán las propiedades de eficiencia de las listas.

4.3 La interfaz List de Java y sus implementaciones más comunes

Esta interfaz representa una colección de datos ordenada en la cual se admiten datos duplicados y sobre la cual el usuario tiene control respecto a en qué posición de la lista se encuentra cada elemento. Podríamos pensar en este contenedor como en una lista doblemente enlazada, donde cada elemento tiene un elemento anterior y un elemento siguiente y donde, por tanto, la posición de cada elemento dentro de la lista está bien definida. Para dar soporte a las operaciones que permiten añadir un elemento en una posición determinada de la lista, borrar un elemento que se encuentre en una posición determinada, añadir una colección de elementos después de una posición determinada... esta interfaz añade los siguientes métodos respecto a interfaz Collection: void add(int index, E element), boolean addAll(int index, Collection<E> collection), E get(int index), int indexOf(E element), int lastIndexOf(E element), E remove(int index) y E set(int index, E element). Los métodos int indexOf(E element) e int lastIndexOf(E element) devuelven la posición de la lista donde se encuentra la primera ocurrencia y la última ocurrencia, respectivamente, del objeto element. La funcionalidad de estos métodos debería ser obvia para el lector (en caso contrario, siempre puedes consultar su javadoc).

Otros métodos importantes de esta interfaz son ListIterator<E> listIterator() y ListIterator<E> listIterator(int comienzo). Ambos devuelven un objeto iterador adaptado a las características especiales de una lista; en el primer caso el iterador apunta al primer elemento de la lista, y en el segundo apunta al elemento que se corresponda con la posición que le pasamos como argumento. En el caso de las colecciones de elementos en general, no se puede asumir que dicha colección tenga un orden predefinido para acceder a sus elementos, por lo que el método iterator () de la clase Collection puede devolvernos los elementos de la colección en cualquier orden. Un ListIterator nos permite recorrer una lista en ambas direcciones (hacia adelante y hacia atrás) y partiendo de cualquier elemento de la lista. Para ello esta interfaz, además de con los métodos definidos en Iterator, cuenta con los métodos boolean hasPrevious() y E previous(), que permiten recorrer una lista "hacia atrás". Por ejemplo, para recorrer una lista desde el último elemento al primer elemento podríamos emplear el siguiente código:

```
List<E> lista = ...
ListIterator<E> iterador = lista.listIterator(lista.size());
while (iterator.hasPrevious()) {//mientras haya anterior
E elemento = iterator.previous();
...//hacer algo con elemento
}
```

Las dos implementaciones más comunes de la interfaz List son **ArrayList** y **LinkedList**. Aunque ambas clases añaden algunos métodos respecto a los definidos en la interfaz List (y, por tanto, respecto a los definidos en la interfaz padre de List: Collection) estos métodos no son demasiado relevantes. El comportamiento básico de ambas clases está definido por los métodos de la interfaz List y Collection. La

diferencia entre ambas es la implementación interna: ***ArrayList esta implementado internamente como un array, mientras que LinkedList está implementado como una lista doblemente enlazada***. Si bien esto no tiene ninguna implicación respecto a cómo se pueden usar las clases desde el código fuente Java, sí tiene muchas implicaciones respecto a su rendimiento.

ArrayList es la clase ideal si vamos a realizar muchas operaciones de acceso aleatorio sobre la lista, ya que al estar implementado mediante un array dichas operaciones tienen un coste constante e independiente del tamaño de la lista. Sin embargo, esta clase no es la más adecuada si el tamaño de la lista va a estar creciendo y decreciendo continuamente, ya que implica reservar una memoria para el array interno; y las operaciones de borrado o inserción en el medio de la lista tienen un coste lineal con el tamaño de ésta, ya que potencialmente implican desplazar hacia adelante (en la inserción) o hacia atrás (en el borrado) todos los elementos de la lista. LinkedList por la contra tiene un rendimiento malo (lineal en el número de elementos) para las operaciones de acceso aleatorio, ya que pueden implicar recorrer todos los elementos de la lista. Sin embargo, las operaciones de borrado o inserción en el medio, o el crecer o decrecer dinámicamente su tamaño tienen un coste constante e independiente del número de elementos que contenga la lista (si excluimos el tiempo necesario para localizar el elemento que queremos borrar o la posición donde queremos realizar la inserción). Dependiendo de la aplicación concreta para la cual necesitemos la lista será mejor optar por una u otra implementación.

En el listado 2 podemos ver un código en el cual se emplea una LinkedList para almacenar un conjunto de **Strings** (nombres de personas). **Empleando el método que devuelve el iterador definido en la interfaz Collection recorremos la lista hacia adelante mostrando su contenido por consola**. Después eliminamos un elemento y la recorremos hacia atrás empleando el iterador definido en la interfaz List.

```
package estructurasdedatos;

import java.util.*;

public class Ejemplo2 {

    public static void main(String[] args) {
        List<String> lista = new LinkedList<String>();
        lista.add("María");
        lista.add("Pedro");
        lista.add("Jose");
        lista.add("Marcos");

        Iterator<String> it = lista.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }

        lista.remove("Pedro");// it debe descartarse
        System.out.println("Pedro borrado");
```

```
        ListIterator<String> it2 =
lista.listIterator(lista.size());
        while (it2.hasPrevious()) {//esta vez vamos hacia
atrás
            System.out.println(it2.previous());
        }
    }
}
```

Observa como a pesar de que se ha empleado un LinkedList, la variable en la cual almacenamos la lista es de tipo List; es decir, nuestro código no tiene ninguna dependencia con la clase concreta que está implementando la lista más allá de la línea donde ésta fue creada. Por tanto, con sólo cambiar la línea de código:

```
List<String> lista = new LinkedList<String>();
```

por la línea:

```
List<String> lista = new ArrayList<String>();
```

habremos cambiado de una a otra implementación, sin necesidad de tener que modificar nada más en el código (puedes ver esto en el código Ejemplo3, que omitimos aquí ya que el único cambio que tiene es el mostrado en las líneas anteriores).

Nuevamente, haré énfasis en que usar una u otra implementación tiene más impacto en el rendimiento del programa que en la forma de programar. Demostrar esto es el objeto de Ejemplo4; este código crea un ArrayList y LinkedList; a continuación les añade a cada uno de ellos 1 millón de números enteros, y realiza sobre ambos varias operaciones de insertado y borrado de elementos al principio, al final y en medio de la estructura de datos. Mostramos aquí un fragmento del código:

```
        List<Integer> listaEnlazada = new
LinkedList<Integer>();
        List<Integer> listaArray = new ArrayList<Integer>();

        for (int i = 0; i < UN_MILLON; i++) {
            listaEnlazada.add(i);
            listaArray.add(i);
        }

        //insertar al principio
        long t = System.nanoTime();
        for (int i = 0; i < REPETICIONES; i++) {
```

```
            listaEnlazada.add(0, Integer.SIZE);
        }
        System.out.println("LinkedList insertando al
principio: " + (System.nanoTime() - t) / REPETICIONES);

        t = System.nanoTime();
        for (int i = 0; i < REPETICIONES; i++) {
            listaArray.add(0, Integer.SIZE);
        }
        System.out.println("ArrayList insertando al
principio: " + (System.nanoTime() - t) / REPETICIONES);

        //insertar al final
        t = System.nanoTime();
        for (int i = 0; i < REPETICIONES; i++) {
            listaEnlazada.add(listaEnlazada.size(),
Integer.SIZE);
        }
        System.out.println("LinkedList insertando al final: "
+ (System.nanoTime() - t) / REPETICIONES);

        t = System.nanoTime();
        for (int i = 0; i < REPETICIONES; i++) {
            listaArray.add(listaArray.size(), Integer.SIZE);
        }
        System.out.println("ArrayList insertando al final: "
+ (System.nanoTime() - t) / REPETICIONES);
```

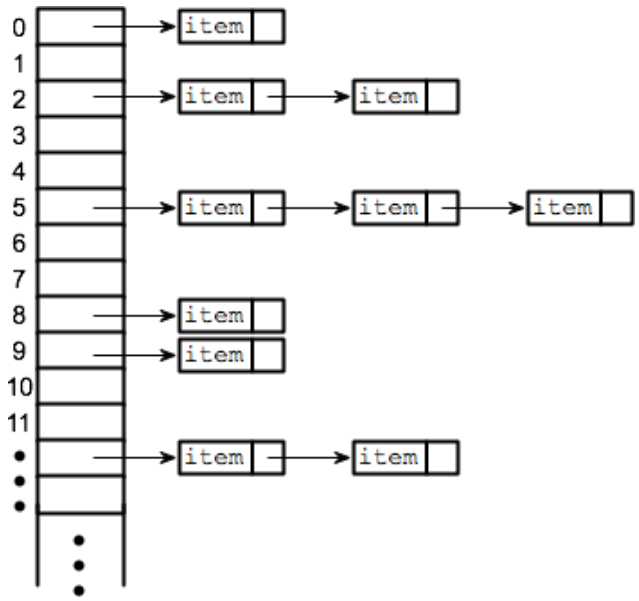
En la salida de consola que produce este ejemplo puede observarse como efectivamente hay considerables diferencias de rendimiento realizando exactamente la misma operación sobre cada una de las dos implementaciones diferentes de la lista:

```
LinkedList insertando al principio: 140
ArrayList insertando al principio: 309862
LinkedList insertando al final: 28
ArrayList insertando al final: 22
LinkedList insertar en medio: 3428675
ArrayList insertando en medio: 149078
LinkedList eliminando al principio: 111
ArrayList eliminando al principio: 235453
LinkedList eliminando al final: 18
ArrayList eliminando al final: 696
LinkedList eliminando en medio: 3399718
ArrayList eliminando en medio: 94752
```

Finalmente, decir que LinkedList implementa también la interfaz Java java.util.Queue<E>; esto es, puede comportarse como una cola. En Java también tenemos una clase java.util.Stack <E> que representa una pila. Puedes consultar más información sobre ellas en sus javadoc.

4.4 Tablas hash

Los arrays nos proporcionan un acceso aleatorio, si están ordenados, podemos buscar en ellos elementos de un modo muy eficiente. Pero tienen un pobre rendimiento cuando queremos insertar nuevos datos en ellos. Las listas tienen un buen rendimiento para insertar nuevos datos, pero no tienen acceso aleatorio ni puede buscarse elementos en ellas de modo eficiente. Un compromiso entre las características de ambos es el de las tablas hash. Estas estructuras permiten localizar dentro de ellas elementos de un modo muy eficiente, y permiten además insertar elementos de modo eficiente.



Una forma de visualizar una tabla hash es como un array estático de listas. Dado un elemento determinado que queremos buscar o insertar, tenemos una función h(elemento) que nos da directamente la posición dónde está ese elemento dentro de la tabla, o donde debe insertarse. A esta función se la denomina **función hash**. Una forma sencilla de implementar esta función es la siguiente: supongamos que tenemos un procedimiento para asignar unívocamente un número entero a cada uno de nuestros elementos. Supongamos que nuestra tabla tiene un tamaño de, digamos, 512 posiciones. La posición en la que tendría que ir cada elemento sería (número entero del elemento)%512.

Es posible que varios elementos tengan el mismo hash. Por ello en cada posición de la tabla se almacena una lista con todos los elementos que tienen el mismo hash. Cuando varios elementos diferentes producen un mismo hash se dice que hay **colisiones** en la tabla. Es interesante diseñar una función hash que minimicen las colisiones.

Si, por ejemplo, tuviésemos una función hash que asigna primero el número entero "0" a todos los elementos, y después calcula 0%512=0, introduciría todos los elementos en la primera posición de la tabla. Por ello, todos los elementos se almacenarían en una única lista que se encuentra en esa posición. Éste sería el peor de los casos posibles. No estamos obteniendo ninguna ventaja respecto a usar una lista. Si estuviésemos en una situación ideal en la que nunca se produce una colisión, tenemos una estructura de datos que nos permite recuperar cualquier elemento, insertarlo o borrarlo en un tiempo O(1), ya que dado el elemento podemos determinar inmediatamente en qué posición de la tabla está, y ese será el único elemento que esté en esa posición.

Cuando hay colisiones, el rendimiento de la tabla hash se degradará. Si suponemos que el tamaño de la lista más grande que se encuentra dentro de la tabla es k, en el peor caso nos llevará O(k) localizar un elemento dentro de la tabla. Suponiendo que la función hash se comporta de modo ideal y distribuye equitativamente todos los elementos entre cada una de las posiciones de la tabla, si tenemos n elementos y la tabla tiene m posiciones, el tamaño de las listas en cada una de las posiciones en el caso en el que n > m será n%m, y el rendimiento de una operación de búsqueda de un dato en la tabla será O(n%m).

Resumiendo, para las tablas hash:

- **Tiempo de acceso a los datos**: en una tabla hash bien diseñada debería haber pocas o ninguna colisiones, por lo que el tiempo de acceso a los datos es O(1). En el caso de haber colisiones, el tiempo de acceso será proporcional al tamaño de la lista más larga.
- **Son estructuras dinámicas**: aunque el tamaño del array es constante, en cada una de sus posiciones hay una lista, por lo que esta estructura puede crecer indefinidamente.
- **Operaciones de borrado e inserción**: si no hay colisiones, o hay muy pocas, el tiempo de estas operaciones es O(1). En el caso de haber colisiones, el tiempo de acceso será proporcional al tamaño de la lista más larga.
- **Operaciones de búsqueda**: la tabla hash se encuentra ordenada por los valores hash, que para obtener un rendimiento adecuado deben ser dispersos y, por tanto, no se podrán ordenar en base a un criterio predefinido. Sin embargo, **calculando el hash de un elemento, es muy eficiente encontrar la posición correspondiente con un determinado elemento, ya que obtendremos la posición en la que se encuentra almacenado en caso de estar presente en la colección**. Si no hay colisiones el tiempo de acceso a los datos es O(1). En el caso de haber colisiones, el tiempo de acceso será proporcional al tamaño de la lista más larga.

4.5 La clase HashSet de Java

La interfaz Set no añade ningún método respecto a la interfaz Collection, pero se usa para representar conjuntos de elementos que no admiten duplicados. Por tanto, un Set necesita conocer si los objetos que contiene son o no iguales entre sí y esto, como veremos, no es completamente trivial.

Existen dos implementaciones principales de esta interfaz: HashSet y TreeSet. La primera almacena los datos utilizando una tabla hash. A partir de un dato, es posible de un modo simple y rápido calcular en cuál de las listas que contiene el array debe encontrarse dicho dato; esto es, a partir de un dato podemos calcular su hash, que de un modo directo nos indicará la posición del array. Mientras el tamaño de todas las listas sea reducido y no haya colisiones, el acceso a los datos es altamente eficiente.

HashSet tiene dos parámetros que afectan a su rendimiento: el tamaño del array (que se conoce como capacidad) y su factor de carga. Este segundo parámetro es una medida de cuanto se permite que el array inicial se llene antes de incrementarlo de tamaño. Cuando el número de datos contenidos en la estructura excede el producto del factor de carga y de la capacidad, se duplica la capacidad de la estructura. Es posible especificar en el constructor de la clase HashSet ambos parámetros; habitualmente, un factor de carga de 0.75 (el que emplea el constructor por defecto de la clase) suele proporcionar un rendimiento bastante adecuado, aunque es posible jugar con estos parámetros para mejorar el rendimiento de una estructura tipo hash para una aplicación concreta.

Para que un HashSet funcione correctamente sobre instancias de clases que nosotros hemos creado (en el caso de clases de la librería estándar habitualmente quien las creó ya se ha encargado de resolver el problema que mencionaremos a continuación) debemos asegurarnos que los objetos instancias de dichas clases tienen bien definida la relación de igualdad. Esto es, que la clase define perfectamente cuando dos objetos son iguales. Para comparar objetos el framework de colecciones con frecuencia recurrirá al método equals de la clase Object:

```
boolean equals(Object obj)
```

Este método debe devolver true cuando el objeto que se le pasa como argumento es igual al objeto sobre el cual se invocó el método. En la implementación de la clase Object, este método sólo devuelve true cuando ambos objetos son idénticos, es decir, cuando son el mismo objeto. A menudo éste no es el comportamiento que queremos: no nos importa si son exactamente o no el mismo objeto, sino si su contenido es el mismo. Así por ejemplo los objetos f1 y f2 obtenidos de la forma siguiente:

```
Float f1 = new Float (4.345F);
Float f2 = new Float (4.345F);
```

no son idénticos, ya que f1 y f2 son dos objetos diferentes que están en posiciones diferentes de la memoria RAM. Por tanto, f1 == f2 es false. Sin embargo, son iguales ya que tienen el mismo contenido.

Para emplear cualquier clase que implemente la interfaz Set una de las cosas que debemos hacer es definir de un modo adecuado la relación de igualdad de las clases que queramos añadir al conjunto. Para ello debemos de sobrescribir el método equals de la clase Object de un modo adecuado. Como podemos ver en el Ejemplo5, lo que este método debe hacer es comprobar en primer lugar si el objeto que se le ha pasado como argumento es null; en caso afirmativo debemos devolver false. A continuación, debemos comprobar si el objeto que se le ha pasado como argumento pertenece a la misma clase que el objeto sobre el cual se ha invocado el método, y a continuación realizar las comprobaciones pertinentes sobre los atributos del objeto. Estas comprobaciones dependen

completamente de la naturaleza del objeto que estemos representando mediante la clase, y de cómo se defina la igualdad para dichos objetos.

Sin embargo, esto no va a ser suficiente para emplear la clase HashSet. Además de definir la relación de igualdad entre objetos, para poder emplear esta clase **debemos asegurarnos que si dos objetos son iguales ambos tienen asociados un mismo hashcode**. El hashcode de un objeto se obtiene invocando el método int hashCode(), método que nuevamente está definido en la clase Object. Este es el método en el cual se va a basar la tabla hash para determinar la posición del array en la cual se debería encontrar la lista que contenga al objeto, en el caso de que el objeto esté contenido en la tabla. En la implementación por defecto de Object el hashcode se basa en la identidad del objeto (habitualmente, en la posición de la memoria RAM en la cual se encuentra almacenado) y no en su igualdad; por tanto, **también debemos sobrescribir este método de tal modo que cuando dos objetos sean iguales según el método equals ambos tengan también el mismo hashcode.**

Lo dicho en el párrafo anterior es imprescindible para que las cosas funcionen: si dos objetos iguales no devuelven el mismo hashcode esos objetos no serán manipulados de modo adecuado por el framework de colecciones. Sin embargo, idealmente, para que estos objetos sean manipulados además de correctamente de modo eficiente, los hashcode de las distintas instancias de la clase deben estar distribuidos uniformemente sobre todo el rango posible de valores de hashcode, y objetos muy diferentes deberían presentar hashcodes muy diferentes.

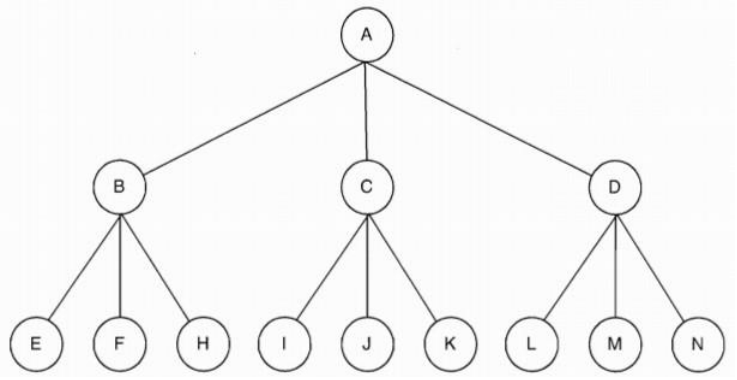
En el Ejemplo5 podríamos haber cumplido el contrato del método hashCode(), simplemente devolviendo el atributo valor de la clase MiEntero. Sin embargo, **en este caso dos objetos diferentes que contengan números consecutivos tendrían hashcodes muy similares, lo cual no es bueno para el rendimiento de una tabla hash.** La implementación del método hashCode() que se muestra en el Ejemplo5 es la propuesta por Joshua Bloch en su libro Effective Java y se basa en la realización de varias operaciones de bits sobre un valor entero, y en añadir un valor de entropía aleatoria inicial. Poniéndolo en cristiano, y teniendo en cuenta que Bloch es quien ha diseñado el framework de Collections, es la implementación óptima para nuestro caso. Si nuestra clase tuviese más atributos sería recomendable tener en cuenta el valor de todos los atributos al generar el hashcode. Bloch en su libro proporciona un complejo y efectivo algoritmo para la generación de códigos hash de calidad para casos más complejos que el nuestro.

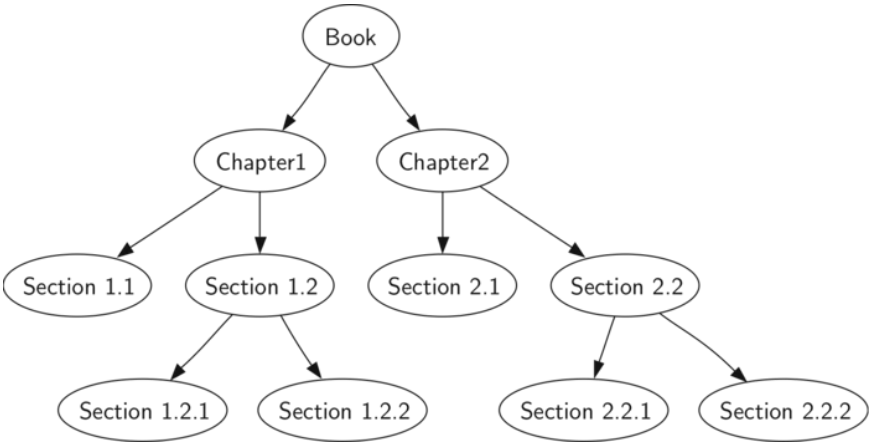
```
package estructurasdedatos;

import java.util.*;

class MiEntero {

    private int valor;

    public MiEntero(int valor) {
        this.valor = valor;
    }
```

```
    public boolean equals(Object objeto) {
        if (objeto == null) {
            return false;
        }
        if (!(objeto instanceof MiEntero)) {
            return false;
        }
        if (((MiEntero) objeto).valor == this.valor) {
            return true;
        } else {
            return false;
        }
    }

    public int hashCode() {
        return 37 * 17 + (int) (valor ^ (valor >>> 32));
    }
}

public class Ejemplo5 {

    public static void main(String[] args) {
        Set<MiEntero> set = new HashSet<MiEntero>();
        set.add(new MiEntero(25));
        set.add(new MiEntero(7));
        set.add(new MiEntero(72));
        System.out.println(set.contains(new MiEntero(5)));
        System.out.println(set.contains(new MiEntero(25)));
    }
}
```

4.6 Árboles

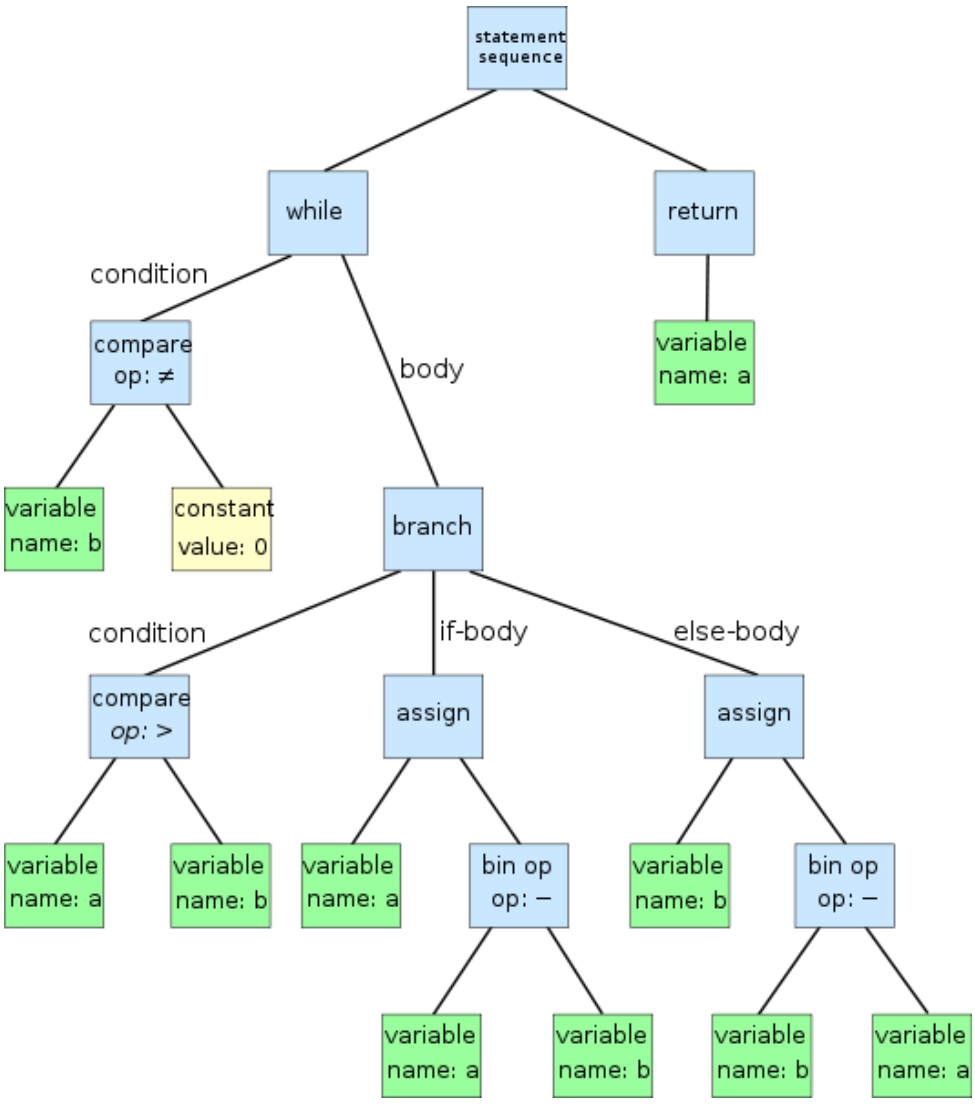
Las tablas hash son muy eficientes cuando no hay colisiones o cuando el número de éstas es bajo. Pero cuando el número de elementos es elevado comparado con el tamaño del array base de la tabla, inevitablemente habrá muchas colisiones. Los árboles nos proporcionan una alternativa a esta estructura. Los árboles pueden permitir encontrar (y borrar o insertar) elementos de un modo eficiente sin imponer restricciones en el número de elementos con el que trabajamos, más allá de las capacidades de almacenamiento del equipo que usemos.



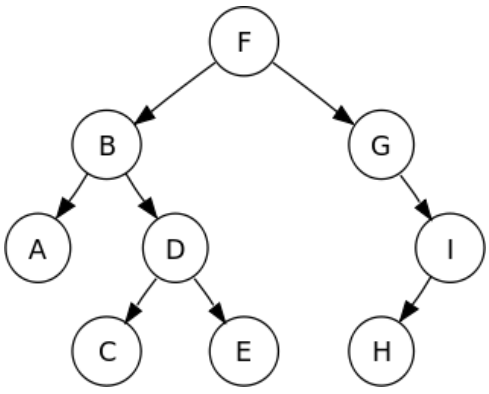


Los árboles representan estructuras jerárquicas en las que cada nodo tiene un único antecesor, pero puede tener varios sucesores. Existe un elemento denominado **raíz** sin antecesor del cual derivan todos los demás. A los elementos que no tienen hijos, esto es, a los que se encuentran en el extremo "inferior" del árbol se le suele denominar **nodos hoja**. Los árboles son la primera estructura de datos no lineal que vemos en este tema; es decir, un elemento de un árbol puede estar conectado a más de un par de elementos.

Hay algunos datos que se representan de un modo natural a través de árboles. Por ejemplo, el árbol genealógico de una persona, o la organización jerárquica de un libro o unos apuntes de clase (como se muestra en la imagen) son datos que se representan de un modo natural de un modo jerárquico. Incluso las sentencias de lenguaje natural, y los lenguajes de programación, tienen su representación más sencilla en modo de árbol jerárquico (aunque sobre este punto no vamos a profundizar ya que se encuentra fuera de los objetivos de esta asignatura; no obstante, el siguiente ejemplo te permitirá hacerte una idea de cómo funciona esto).

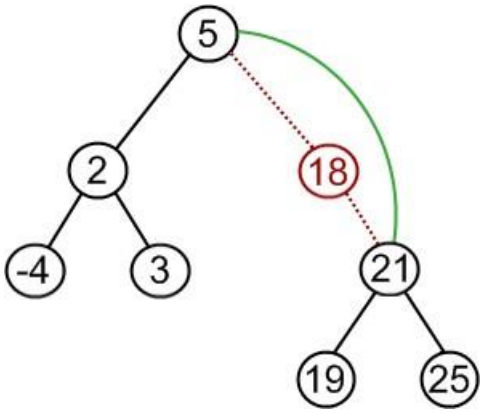


Un tipo muy común de árboles son los **árboles binarios**, donde cada nodo tiene como máximo dos descendientes. La principal utilidad en computación de los árboles binarios es almacenar elementos que tienen definida una relación de orden de un modo altamente eficiente. Podemos construir un árbol binario ordenado partiendo de la raíz, y colocando un elemento menor que la raíz a su izquierda, y un elemento mayor que la raíz a su derecha. A continuación repetimos una y otra vez esta misma operación para cada uno de los nodos. El resultado final es una estructura que nos va a permitir localizar de un modo muy eficiente cualquier elemento. Para ello, partimos de la raíz del árbol binario. Si el elemento que buscamos es mayor que la raíz, nos vamos por la rama de la derecha. Si es menor, nos vamos por la rama de la izquierda. Repetimos esta operación una y otra vez para cada nodo, hasta que alcancemos el nodo deseado.



Una vez hemos comparado nuestro elemento con el elemento raíz del árbol, y decidimos ir por una de las dos ramas, podemos olvidarnos de la otra rama, y de todos los elementos que cuelgan de ella. Esto quiere decir que, si inicialmente teníamos n nodos en el árbol, suponiendo que el árbol estuviese bien balanceado, después de comparar nuestro elemento con el nodo raíz podemos olvidarnos de la mitad de los elementos; nos centramos ahora sólo en una de las ramas que contiene n/2 elementos. Volvemos a comparar nuestro elemento con el nodo de la rama en cuestión, y nuevamente decidimos irnos por una de sus dos ramas, ignorando la mitad de los elementos restantes, quedando ahora sólo n/4 elementos. Después de haber descendido k niveles en el árbol, sólo nos quedan $n/(2^k)$ elementos entre los cuales podría estar el que buscamos. $n/(2^k)$ debe ser siempre mayor que 1, sino ya no quedarían elementos para buscar. Por tanto, debe de cumplirse siempre la relación $n/(2^k) > 1$. Si despejamos el valor de k, que es el número de veces que tendremos que comparar nuestro dato con un nodo del árbol, obtenemos $k \leq \log_2(n)$. Es decir, en un tiempo O(log(n)) habremos localizado nuestro elemento si estaba en el árbol.

Este proceso no es tan eficiente como localizar un elemento dentro de una tabla hash, que tiene una complejidad O(1), suponiendo que no ha habido colisiones. Pero es muy eficiente, y tiene la ventaja de que el árbol puede ser todo lo grande que quiera, y puede crecer dinámicamente, y permiten elegir un criterio de ordenación. Las tablas, aunque pueden crecer dinámicamente, tienen un proceso de crecimiento altamente no trivial, y requiere una reorganización mayor del array base. En el caso de los árboles binarios, basta con insertar/borrar nuevos nodos (cosa que no siempre tiene que suceder en las hojas del árbol; también puede suceder en medio; en la figura se muestra cómo sería el borrado de un nodo en medio de un árbol).



Resumiendo, las propiedades de un árbol binario ordenado serían las siguientes:

- **Tiempo de acceso a los datos**: necesitaremos como mucho $O(\log_2(n))$ operaciones para localizar cualquier dato en el árbol.
- **Son estructuras dinámicas**: podemos insertar borrar nuevos nodos en cualquier parte del árbol con sólo reorganizar las conexiones entre los nodos.
- **Operaciones de borrado e inserción**: la operación de inserción en si sólo requiere modificar un número de conexiones bien definido por lo que tiene complejidad O(1). Localizar el nodo a borrar o insertar tiene una complejidad $O(\log_2(n))$.

- **Operaciones de búsqueda**: las búsquedas son muy eficientes; la estructura sido diseñada precisamente para buscar cosas de modo eficiente. Cualquier búsqueda sucede como mucho en $O(\log_2(n))$.

4.7 La clase Java TreeSet

La clase TreeSet es una implementación de la interfaz Set que representa un conjunto ordenado de objetos que emplea internamente un árbol binario. El orden en el que se basa esta clase para ordenar los elementos es lo que en Java se llama "el orden natural" de los objetos, que viene definido por el método **int compareTo(T o) de la interfaz Comparable<T>.** Otra forma alternativa de indicar cuál es el orden, si no podemos hacer que nuestros objetos implementen dicha interfaz, es proporcionarle al TreeSet en su constructor un objeto **Comparator<T>,** objeto que define un método **int compare(T o1,T o2)** que permite comparar dos instancias de la clase T. El entero que devuelve el método compareTo debe ser negativo si el objeto que se le pasó como argumento va después que el objeto sobre el cual se invocó el método; 0 si ambos objetos son iguales y negativo si el objeto que se le pasó como argumento va después que el objeto sobre cuál se invocó el método. El método compare debe comportarse de un modo análogo, indicando si el primero de esos argumentos va antes (devuelve un valor negativo), después (devuelve un valor positivo) o es igual (devuelve 0) que el segundo parámetro.

Tanto si implementamos la interfaz Comparable como si usamos un Comparator **los resultados de ambos deben de ser compatibles con el método equals**, ya que dicho método también será empleado internamente por la estructura. Esto quiere decir que, si según el método equals dos objetos son iguales, al compararlos el entero que devuelva el método de la interfaz o del comparador debe ser "0", o en caso contrario la estructura permitirá elementos repetidos, algo que no debería ocurrir de acuerdo a la definición de árbol.

El hecho de que un TreeSet sea un conjunto ordenado permite localizar los elementos dentro del conjunto de un modo altamente eficiente: la operación de localizar un elemento, añadirlo o eliminarlo tiene un coste log (n) respecto al tamaño del conjunto.

En el Ejemplo6 podemos ver cómo hemos creado una clase Nombre que representa el nombre de una persona y su apellido. El orden natural (**definido a través del método compareTo**) de la clase indica que las instancias de la clase se ordenarán primero por nombre, y si el nombre es igual se ordenarán por apellido. **Observa como para construir esta clase hemos tenido que sobrescribir también los métodos equals y hashCode**. También hemos creado una clase Comparador que implementa un orden diferente del orden natural: nos permitirá ordenar un conjunto de Nombres por apellido y, si el apellido es igual, se recurrirá al nombre. Esta es una de las utilidades de las clases Comparator: empleando distintos Comparators puedo ordenar una misma colección de objetos en base a diferentes criterios.

En el main del Ejemplo6 creamos un primer TreeSet para el cual no indicamos ningún Comparator, por lo que empleará el orden natural. A continuación, empleando un iterador mostramos sus elementos.

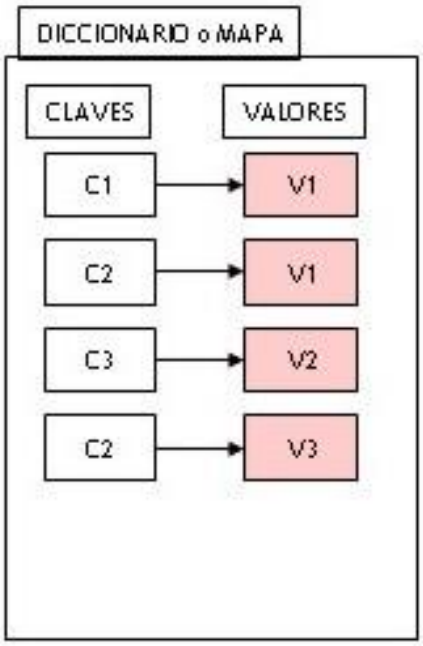
Después creamos otro TreeSet y esta vez especificamos que se emplee el Comparator que ordenará los objetos por apellido. En esta ocasión no empleamos un iterador para recorrer la colección, sino que empleamos el bucle "foreach" de Java. Este bucle permite recorrer cualquier colección de elementos; hasta ahora sólo lo habíamos empleado para recorrer los elementos de un array, pero podemos emplearlo sobre cualquier contenedor del framework de colecciones. A menudo esto resulta mucho más práctico para procesar una colección de elementos que emplear iteradores.

```
package estructurasdedatos;

import java.util.*;

class Nombre implements Comparable<Nombre> {

    String nombre, apellido;

    public Nombre(String nombre, String apellido) {
        this.nombre = nombre;
        this.apellido = apellido;
    }

    public int compareTo(Nombre objeto) {
        if (nombre.equals(objeto.nombre)) {
            return apellido.compareTo(objeto.apellido);
        } else {
            return nombre.compareTo(objeto.nombre);
        }
    }

    @Override
    public boolean equals(Object objeto) {
        if (objeto == null) {
            return false;
        }
        if (!(objeto instanceof Nombre)) {
            return false;
        }
        Nombre objetoNombre = (Nombre) objeto;
        if (objetoNombre.nombre == this.nombre &&
objetoNombre.apellido == this.apellido) {
            return true;
        } else {
            return false;
        }
    }

    @Override
    public int hashCode() {
        return 37 * 17 + (int) (nombre.hashCode() ^
(nombre.hashCode() >>> 16))
                + (int) (apellido.hashCode() ^ (apellido.hashCode()
>>> 16));
```

```
    }

    @Override
    public String toString() {
        return nombre + " " + apellido;
    }
}

class ComparadorOrdenPorApellido implements Comparator<Nombre> {

    public int compare(Nombre objeto1, Nombre objeto2) {
        if (objeto1.apellido.equals(objeto2.apellido)) {
            return objeto1.nombre.compareTo(objeto2.nombre);
        } else {
            return objeto1.apellido.compareTo(objeto2.apellido);
        }
    }
}

public class Ejemplo6 {

    public static void main(String[] args) {

        Set<Nombre> setOrdenNatural = new TreeSet<Nombre>();
        setOrdenNatural.add(new Nombre("Juan", "Pérez"));
        setOrdenNatural.add(new Nombre("Antonio", "Rodríguez"));
        setOrdenNatural.add(new Nombre("Francisco", "Casado"));
        Iterator<Nombre> it = setOrdenNatural.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }

        Set<Nombre> setOrdenMedianteComparador = new
TreeSet<Nombre>(new ComparadorOrdenPorApellido());
        setOrdenMedianteComparador.addAll(setOrdenNatural);
        System.out.println("\nOrdenados  por  apellido:");
        for (Nombre elemento : setOrdenMedianteComparador) {
            System.out.println(elemento);
        }
    }
}
```

4.8 Diccionarios o mapas

Hasta ahora todas las estructuras de datos que hemos visto almacenan directamente el dato en cuestión, y para recuperarlo necesitamos tener una copia de ese dato, o ese mismo dato. En ocasiones en el mundo real nos encontramos con situaciones en las cuales no se emplea el propio dato para recuperarse a sí mismo, sino que empleamos algún tipo de "clave" para recuperar los datos.



Un diccionario o mapa es una estructura dinámica de datos formada por una serie de parejas (clave, valor). Conociendo la clave es posible recuperar el valor correspondiente. Una guía telefónica o un diccionario son ejemplos del mundo real que funcionan de un modo similar a esta estructura. En ellos, la clave sería el nombre de la persona cuyo teléfono queremos averiguar o la palabra cuya definición estamos buscando, y el valor sería el número de teléfono y la definición, respectivamente.

En un mapa, los distintos elementos pueden encontrarse ordenados o no según su clave, y dependiendo del tipo de mapa es posible que se permitan claves repetidas o que no se permita la repetición claves.

Las operaciones típicas que se realizan sobre un diccionario o mapa son:

- Insertar una pareja (clave, valor).
- Eliminar una pareja (clave, valor).
- Recuperar el valor asociado a una clave.
- Comprobar si una clave pertenece o no a la estructura.
- Listar todas las claves.
- Listar todos los valores.
- Borrar todos los elementos.

Internamente los mapas suelen emplear o bien una estructura tipo tabla hash o bien una estructura tipo árbol binario ordenado para ordenar las claves y poder recuperarlas de modo eficiente. Cada clave tiene un enlace a su valor correspondiente y, en consecuencia, una vez se ha recuperado la clave es sencillo recuperar el valor. La eficiencia de un mapa es la misma que la de la estructura interna que está

empleando para mantener ordenadas las claves; es decir, es la misma que la de una tabla hash o un árbol binario, según corresponda.

4.9 La interfaz Map de java y sus implementaciones más comunes

Se trata de la otra interfaz raíz de la jerarquía de colecciones de Java. Es una estructura dinámica de datos formada por una serie de parejas (clave, valor); es decir, esta interfaz representa a una estructura tipo mapa. Conociendo la clave es posible recuperar el valor correspondiente de un modo altamente eficiente.

Los principales métodos de este interfaz son:

- V put(K clave, V valor): permite añadir un par (clave, valor) al mapa. Devuelve el valor añadido si la operación se realizó correctamente o null en caso contrario.
- void putAll(Map<K,V> mapa): añade el contenido del mapa argumento al mapa sobre cual se invoca el método.
- void clear(): elimina todo el contenido del mapa.
- V remove(K clave): elimina del mapa la clave que se le pasa como argumento. Devuelve el valor eliminado del mapa, o null sino se pudo realizar la operación.
- V get(K clave): recibe como argumento una clave, y devuelve el valor correspondiente o null si dicha clave no está en el mapa.
- boolean containsKey(K clave): devuelve true si el mapa contiene la clave.
- boolean containsValue(K value): devuelve true si el mapa contiene el valor.
- int size(): devuelve el tamaño del mapa.
- boolean isEmpty(): devuelve true si el mapa está vacío, false en caso contrario.
- Set keySet(): devuelve un Set con todas las claves del mapa.
- Collection values(): devuelve una Collection con todos los valores del mapa.

Existen dos implementaciones principales de este interfaz: **HashMap y TreeMap**. La primera implementa la interfaz basándose en una tabla hash, por lo que su contenido no estará ordenado en base a un criterio predefinido. La segunda implementa la interfaz basándose en un árbol en el cual los datos están ordenados por valor ascendente de clave, empleando para este orden o bien el orden natural de las claves o bien un comparador que se le pasa en el constructor.

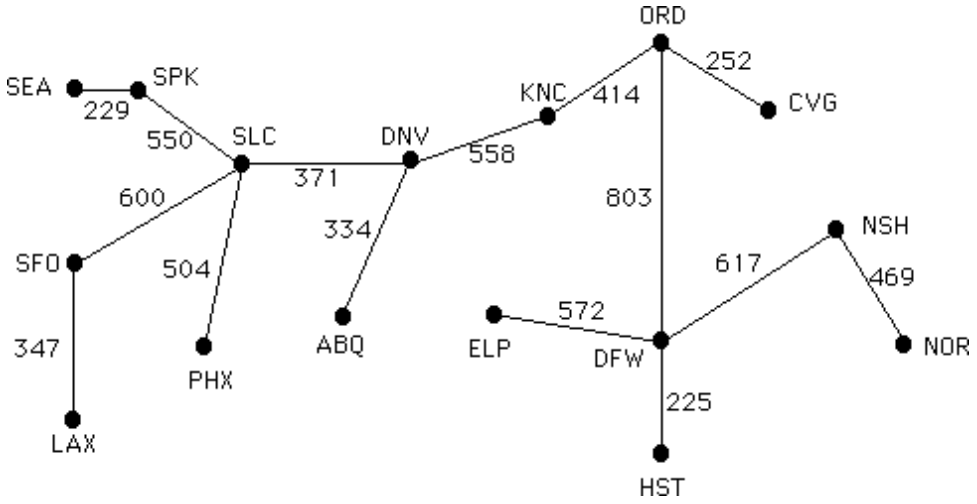
En el Ejemplo7 podemos ver un ejemplo de uso de las clases HashMap y TreeMap. Primero creamos un HashMap empleando de clave un entero y de valor un String. Introducimos unas cuantas entradas, y las

mostramos por pantalla empleando un iterador que itera sobre la colección de claves. Podemos ver cómo empleando esta implementación no tenemos ninguna garantía acerca del orden en el cual se van a almacenar los elementos. Después construimos un TreeMap con las mismas entradas. Como podemos ver en el código, en este caso las entradas estarán ordenadas por valor ascendente de la clave. Nuevamente, para que sirva de ejemplo, hemos empleado dos formas diferentes de recorrer cada una de las dos colecciones: en el primer caso hemos usado un iterador, en el segundo un bucle tipo "foreach".

```
package estructurasdedatos;

import java.util.*;

public class Ejemplo7 {

    public static void main(String[] args) {
        Map<Integer, String> mapaEnteroString = new
HashMap<Integer, String>();
        mapaEnteroString.put(11, "Once");
        mapaEnteroString.put(2, "Dos");
        mapaEnteroString.put(16, "Dieciséis");
        Iterator<Integer> it =
mapaEnteroString.keySet().iterator();
        while (it.hasNext()) {
            System.out.println(mapaEnteroString.get(it.next()));
        }

        Map<Integer, String> mapaEnteroString2 = new
TreeMap<Integer, String> ();
        mapaEnteroString2.putAll(mapaEnteroString);
        System.out.println( "\nOrdenados:");
        for (String elemento : mapaEnteroString2.values()) {
           System.out.println(elemento);
        }
    }
}
```

4.10 Grafos

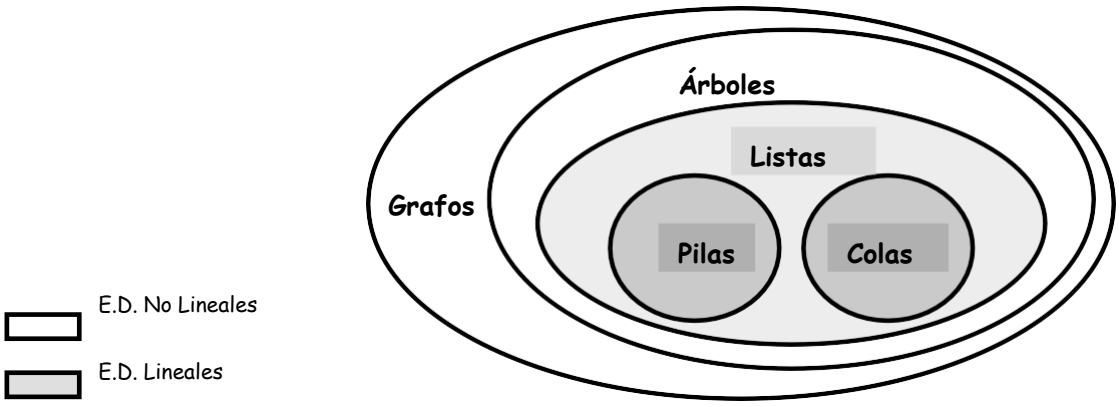
Un grafo es un conjunto de nodos conectados mediante arcos sin ningún tipo de restricción entre las conexiones que se pueden establecer entre un nodo y los demás. Si, por ejemplo, quisiésemos representar gráficamente las distancias que hay entre distintas ciudades de un país que están conectadas de modo directo entre sí esta sería la estructura más adecuada para representar esta información.



.

Los grafos también se emplean, por ejemplo, para representar rutas metabólicas dentro de una célula. Son la estructura de datos más potente y versátil, aunque también la más compleja. La librería estándar de Java no tiene soporte para trabajar con esta estructura de datos. Aunque hay algunas librerías no estándar que permiten trabajar con este tipo de datos, a menudo es necesario escribir código a medida para trabajar con esta representación. En el próximo tema veremos algoritmos para trabajar con grafos.

Finalmente, terminar con este estudio de estructuras de datos diciendo que mediante un grafo puede representarse cualquier cosa que se pueda representar con un árbol, con una lista, con una pila o con una cola. Del mismo modo, con un árbol se puede representar cualquier tipo de lista. Las listas son el caso más particular de estructura, y los árboles y grafos son estructuras más generales.



5 Algoritmos de la librería estándar de Java: la clase Collections

En el tema pasado hemos visto varios algoritmos para buscar datos de modo eficiente y para ordenarlos. En ocasiones, uno tiene que recurrir a escribir su propio algoritmo de búsqueda o ordenación de datos por motivos de eficiencia o porque el problema con el que está trabajando es muy particular. Pero a menudo podemos recurrir a clases de la librería estándar de Java para estas tareas. Más concretamente, a la clase Collections. En esta clase hay una multitud de métodos estáticos que permiten realizar todo tipo de operaciones sobre colecciones de datos. Algunos de sus métodos más relevantes son:

- static int binarySearch(List<T> list, T key): busca en la lista que se le pasa como argumento el objeto que se le pasa como segundo argumento empleando el método de la búsqueda binaria. Devuelve el índice de la lista donde está el objeto, o un valor negativo si no lo ha encontrado. Antes de emplear este método la lista de datos debe haber sido ordenada según su orden natural; en caso contrario el resultado de la llamada a este método no está definido.
- static <T> void copy(List<T> dest, List<T> src): copia la segunda lista en la primera.
- static <T> void fill(List<T> list, T obj): llena la lista que se le pasa como argumento con copias del segundo argumento.
- static T max(Collection<T> coll): devuelve el máximo de todos los elementos de la colección, según el orden natural. También es posible especificar un comparador.
- static T min(Collection<T> coll): devuelve el mínimo de todos los elementos de la colección, según el orden natural. También es posible especificar un comparador.
- static <T> boolean replaceAll(List<T> list, T oldVal, T newVal): reemplaza todas las ocurrencias del segundo argumento en la lista que se le pasa como primer argumento por el tercer argumento.
- void sort(List<T> list): ordena la lista que se le pasa como argumento empleando el orden natural. También es posible especificar un comparador.

En esta clase también existen otros métodos que proporcionan funcionalidad más avanzada, como, por ejemplo, crear copias no modificables (de sólo lectura) de una colección de elementos. **Te recomiendo que consultes el javadoc de esta clase para familiarizarte con la funcionalidad que proporciona.**

En el Ejemplo8 podemos ver cómo funcionan los métodos que nos permiten realizar búsquedas binarias y ordenaciones.

```
package estructurasdedatos;

import java.util.*;

public class Ejemplo8 {

    public static void main(String[] args) {
        List<String> lista = new LinkedList<String>();
        lista.add("María");
        lista.add("Pedro");
        lista.add("Jose");
        lista.add("Marcos");

        System.out.println("La posición de Pedro es "
                + Collections.binarySearch(lista, "Pedro"));

        Collections.sort(lista);
        System.out.println("Lista en orden alfabético:");
        for (String elemento : lista) {
            System.out.println(elemento);
        }
    }
}
```

Dentro del paquete java.util también podemos encontrar una clase con una funcionalidad muy parecida a la de la clase de Collections, pero orientada a realizar operaciones sobre arrays: la clase Arrays. Esta clase nos permite ordenar arrays, buscar elementos sobre ellos, copiarlos, etcétera. Si nuestros datos no están almacenados en una estructura del framework de Colections, sino en un array, esta será la clase que debamos emplear para este tipo de operaciones.

6 Algunos comentarios más sobre estructuras de datos en Java

El framework de Collections es una extensa y potente librería que recoge múltiples estructuras de datos en las cuales podemos apoyarnos para construir nuestros programas. Las estructuras de datos que hemos presentado en este artículo, aunque son las más empleadas, son sólo una pequeña fracción. Dentro de esta librería tenemos versiones de sólo lectura de todas las estructuras que hemos presentado en este documento, así como versiones sincronizadas pensadas para trabajar en entornos multithread (algunas de ellas, optimizadas para realizar múltiples lecturas y pocas escrituras; otras optimizadas para realizar múltiples escrituras). La librería también cuenta con colas con prioridad, mapas de listas, listas secuenciales, colecciones cuyos elementos son referencias débiles que pueden ser

reclamados por el Garbage Collector si la máquina virtual se está quedando sin memoria, y un largo etcétera.

Dentro de este framework también contamos con múltiples colecciones de algoritmos, contenidos en las clases Collections y Arrays, para realizar muchas de las operaciones más frecuentes que podamos querer realizar tanto sobre estas estructuras de datos, como sobre arrays. Según mi experiencia, dentro de este framework es posible encontrar casi cualquier estructura de datos que uno pueda necesitar, salvo un grafo. Para cualquier otra cosa, es improbable que no encuentres una colección de datos adecuada dentro del framework.

Por último, decirte que nunca deberías emplear en un programa las clases Vector o Hastable (sí, la "t" va con minúscula; esta clase es tan vieja que ni siquiera sigue los convenios de nomenclatura Java), clases que no pertenecen al framework de collections. Estas son clases legacy que se incorporaron a Java antes de que existiese el framework y que, por tanto, antes de Java 1.2 eran las únicas estructuras de datos que el lenguaje aportaba. Estas clases no estaban integradas dentro del framework de Collections, aunque en Java 1.2 se modificaron para que implementasen las interfaces List y Map, respectivamente. Sin embargo, estas clases están sincronizadas, esto quiere decir que siempre que se trabaja con ellas se obtiene una penalización en rendimiento en cualquier operación que se realiza sobre ellas, aunque nuestro código esté empleando un único thread. En la actualidad, no existe ningún motivo para emplearlas, a pesar de que por motivos históricos se usan mucho más de lo que se debiese: muchos programadores que aprendieron Java cuando estas clases eran las únicas alternativas nunca se han molestado en aprender nada nuevo.