16

Compresión de archivos mediante el algoritmo de Huffman

Contenido

Objetivos del capítulo

- Estudiar y analizar el algoritmo de Huffman.
- Desarrollar una aplicación que, basándose en este algoritmo, permita comprimir y descomprimir archivos.
- Proveer implementaciones concretas para las *interfaces* de los objetos que se utilizan en el programa. Estas implementaciones estarán a cargo del lector y constituyen el ejercicio en sí mismo.
- Aprender sobre como manejar archivos en Java.

Competencias específicas

- Implementar aplicaciones orientadas a objetos que creen y manipulen archivos para guardar y recuperar información.

David A. Huffman (1925 - 1999). Hizo importantes aportes al estudio de aparatos finitos, circuitos aleatorios, síntesis de procedimientos, y diseño de señales. Es más conocido por su "código de Huffman", un sistema de compresión y codificación de longitud variable (fue el resultado de su proyecto de fin de carrera en el MIT).
Es usado, por ejemplo, en redes de computadoras y televisión.

16.1 Introducción

Generalmente utilizamos múltiplos del *byte* para dimensionar diferentes cantidades de información. Por ejemplo: “tengo un disco rígido con más de 100 *gigabytes* de música” o “la base de datos de la compañia supera el *terabyte* de información” o “no puedo enviarte la foto por mail porque pesa 300 *megabytes*”.

En cambio, cuando queremos dimensionar el caudal de información que se transmite por un canal durante una determinada cantidad de tiempo hablamos de “bits por unidad de tiempo”. Por ejemplo: “el servicio de video *on demand* requiere un ancho de banda no menor a 3 *megabits* por segundo” o “conseguí la canción que buscaba en mp3 *rippeada* a 320 kbps (kilobits por segundo)”.

Los programas de compresión de información como el WinRAR o el WinZIP utilizan algoritmos que permiten reducir esas cantidades de *bytes*. Cuando “zippeamos” un archivo obtenemos un nuevo archivo que contiene la misma información que el anterior pero ocupa menos espacio en disco.

En este capítulo estudiaremos el “algoritmo de Huffman”, desarrollado por David A. Huffman y publicado en “*A method for the construction of minimun redundancy codes”*, en 1952. El algoritmo permite comprimir información basándose en una idea simple: utilizar menos de 8 bits para representar a cada uno de los *bytes* de la fuente de información original.

Por ejemplo, si utilizamos los bits 01 (dos bits) para representar el carácter ‘A’ (o el *byte* 01000001), cada vez que aparezca una ‘A’ estaremos ahorrando 6 bits. Entonces cada 4 caracteres ‘A’ que aparezcan en el archivo ahorraremos 24 bits, es decir, 3 *bytes*. Luego, cuanto mayor sea la cantidad de caracteres ‘A’ que contenga el archivo o la fuente de información original, mayor será el grado de compresión que podremos lograr. Así, a la secuencia de bits que utilizamos para recodificar cada carácter la llamamos “código Huffman”.

El algoritmo consiste en una serie de pasos a través de los cuales podremos construir los códigos Huffman para reemplazar los *bytes* de la fuente de información que queremos comprimir, generando códigos más cortos para los caracteres más frecuentes y códigos más largos para los *bytes* que menos veces aparecen.

Algoritmo de Huffman

Alcance del ejercicio

Implementar un compresor de archivos basado en el algoritmo de Huffman es un excelente ejercicio que nos inducirá a aplicar los principales conceptos que estudiamos desde el comienzo del libro: *arrays*, archivos y listas.

El algoritmo utiliza un árbol binario para generar los códigos Huffman que reemplazarán a los *bytes* de la fuente de información original. El hecho de que aún no hayamos estudiado el tema del “árbol binario” representa una excelente oportunidad para aplicar los conceptos de abstracción y encapsulamiento que vimos en los capítulos de programación orientada a objetos.

Es decir, este capítulo persigue tres objetivos:

- * Estudiar y analizar el algoritmo de Huffman.
- * Desarrollar una aplicación que, basándose en este algoritmo, permita comprimir y descomprimir archivos.
- * Proveer implementaciones concretas para las *interfaces* de los objetos que utilicemos en el programa. Estas implementaciones estarán a cargo del lector y constituyen el ejercicio en sí mismo.

16.2 El algoritmo de Huffman

El algoritmo de Huffman provee un método que permite comprimir información mediante la recodificación de los *bytes* que la componen. En particular, si los *bytes* que se van a comprimir están almacenados en un archivo, al recodificarlos con secuencias de bits más cortas diremos que lo comprimimos.

La técnica consiste en asignar a cada *byte* del archivo que vamos a comprimir un código binario compuesto por una cantidad de bits tan corta como sea posible. Esta cantidad será variable y dependerá de la probabilidad de ocurrencia del *byte*. Es decir: aquellos *bytes* que más veces aparecen serán recodificados con combinaciones de bits más cortas, de menos de 8 bits. En cambio, se utilizarán combinaciones de bits más extensas para recodificar los *bytes* que menos veces se repiten dentro del archivo. Estas combinaciones podrían, incluso, tener más de 8 bits.

Los códigos binarios que utilizaremos para reemplazar a cada *byte* del archivo original se llaman “códigos Huffman”.

Veamos un ejemplo. En el siguiente texto:

COMO COME COCORITO COME COMO COSMONAUTA

el carácter ‘O’ aparece 11 veces y el carácter ‘C’ aparece 7 veces. Estos son los caracteres que más veces aparecen y por lo tanto tienen la mayor probabilidad de ocurrencia. En cambio, los caracteres ‘I’, ‘N’, ‘R’, ‘S’ y ‘U’ aparecen una única vez; esto significa que la probabilidad de hallar en el archivo alguno de estos caracteres es muy baja.

Como ya sabemos, para codificar cualquier carácter se necesitan 8 bits (1 *byte*). Sin embargo, supongamos que logramos encontrar una combinación única de 2 bits con la cual codificar al carácter ‘O’, una combinación única de 3 bits con la cual codificar al carácter ‘M’ y otra combinación única de 3 bits con la cual codificar al carácter ‘C’.

Byte o Carácter	Codificación
O	01
M	001
C	000

Si esto fuera así, entonces para codificar los primeros 3 caracteres del texto anterior solo necesitaríamos 1 *byte*, lo que nos daría una tasa de compresión del 66.6%.

Carácter	C	O	M	…
Byte	000	01	001	…

Ahora, el *byte* 00001001 representa la secuencia de caracteres ‘C’, ‘O’, ‘M’, pero esta información solo podrá ser interpretada si conocemos los códigos binarios que utilizamos para recodificar los *bytes* originales. De lo contrario, la información no se podrá recuperar.

Para obtener estas combinaciones de bits únicas, el algoritmo de Huffman propone seguir una serie de pasos a través de los cuales obtendremos un árbol binario llamado “arbol Huffman”. Luego, las hojas del árbol representarán a los diferentes caracteres que aparecen en el archivo y los caminos que se deben recorrer para llegar a esas hojas representarán la nueva codificación del carácter.

A continuación, analizaremos los pasos necesarios para obtener el árbol y los códigos Huffman que corresponden a cada uno de los caracteres del texto expresado más arriba.

16.2.1 Paso 1 - Contar la cantidad de ocurrencias de cada carácter

El primer paso consiste en contar cuántas veces aparece en el archivo cada carácter o *byte*. Como un *byte* es un conjunto de 8 bits, resulta que solo existen $2^8 = 256$ *bytes* diferentes. Entonces utilizaremos una tabla con 256 registros para representar a cada uno de los 256 *bytes* y sus correspondientes contadores de ocurrencias.

	Carácter	*n*
0		
:		
32	*ESP*	5
:		
65	A	2
:		
67	C	7
:		
69	E	2
:		
73	I	1

	Carácter	*n*
:		
:		
77	M	5
78	N	1
79	O	11
:		
82	R	1
83	S	1
84	T	2
85	U	1
:		
255		

Fig. 16.1 Tabla de ocurrencias. Indica la cantidad de veces que aparece cada carácter.

16.2.2 Paso 2 - Crear una lista enlazada

Conociendo la cantidad de ocurrencias de cada carácter, tenemos que crear una lista enlazada y ordenada ascendentemente por dicha cantidad. Primero los caracteres menos frecuentes y luego los que tienen mayor probabilidad de aparecer y, si dos caracteres ocurren igual cantidad de veces, entonces colocaremos primero al que tenga menor valor numérico. Por ejemplo: los caracteres 'I', 'N', 'R', 'S' y 'U' aparecen una sola vez y tienen la misma probabilidad de ocurrencia entre sí; por lo tanto, en la lista ordenada que veremos a continuación los colocaremos ascendentemente según su valor numérico o código ASCII.

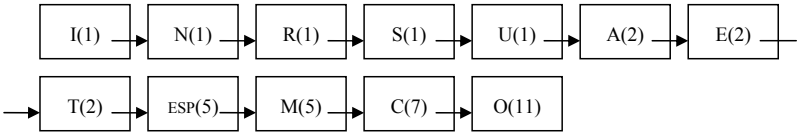

Fig. 16.2 Lista enlazada de caracteres ordenada por probabilidad de ocurrencia.

Los nodos de la lista tendrán la siguiente estructura:

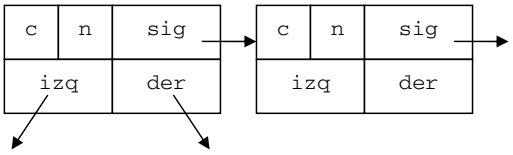

Fig. 16.3 Estructura del nodo de la lista enlazada.

El campo `c` representa el carácter (o *byte*) y el campo `n` la cantidad de veces que aparece dentro del archivo. El campo `sig` es la referencia al siguiente elemento de la lista enlazada. Los campos `izq` y `der`, más adelante, nos permitirán implementar el árbol binario. Por el momento no les daremos importancia; simplemente pensemos que son punteros a `null`.

16.2.3 Paso 3 - Convertir la lista enlazada en el árbol Huffman

Vamos a generar el árbol Huffman tomando "de a pares" los nodos de la lista. Esto lo haremos de la siguiente manera: sacamos los dos primeros nodos y los utilizamos para crear un pequeño árbol binario cuya raíz será un nuevo nodo que identificaremos con un carácter ficticio **1* (léase "asterisco uno") y una cantidad de ocurrencias igual a la suma de las cantidades de los dos nodos que estamos procesando. En la rama derecha colocamos al nodo menos ocurrente (el primero); el otro nodo lo colocaremos en la rama izquierda.

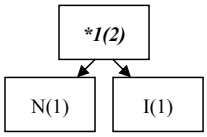

Fig. 16.4 Proceso de los dos primeros nodos de la lista.

Luego insertamos en la lista al nuevo nodo *(raíz)* respetando el criterio de ordenamiento que mencionamos más arriba. Si en la lista existe un nodo con la misma cantidad de ocurrencias (que en este caso es 2), la inserción la haremos a continuación de este.

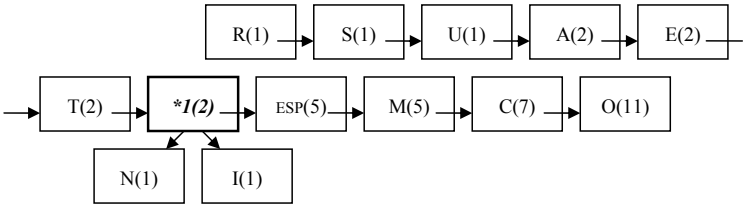

Fig. 16.5 Estado de la lista una vez finalizado el proceso de los dos primeros nodos.

Ahora repetimos la operación procesando nuevamente los dos primeros nodos de la lista: R(1) y S(1):

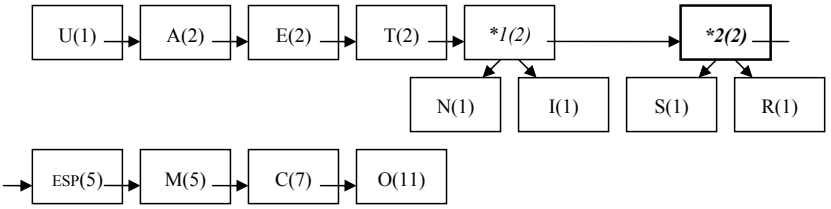

Fig. 16.6 Estado de la lista luego de procesar el siguiente par de nodos.

Luego continuamos con este proceso hasta que la lista se haya convertido en un árbol binario cuyo nodo raíz tenga una cantidad de ocurrencias igual al tamaño del archivo que queremos comprimir.

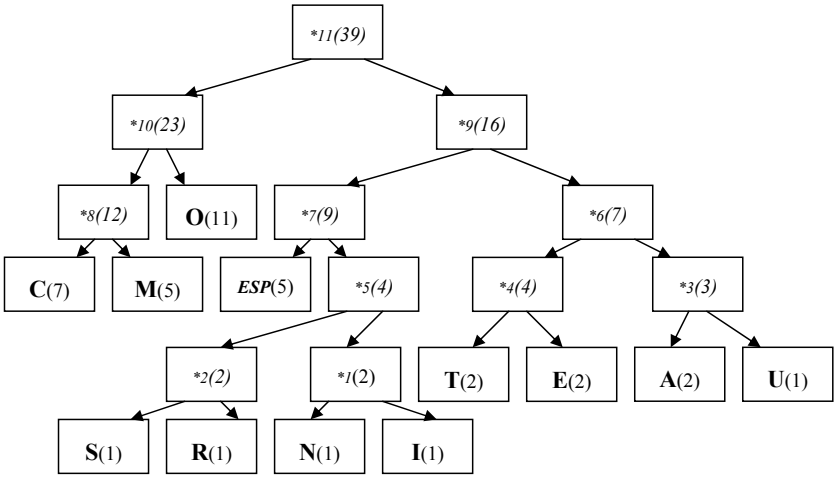

Fig. 16.7 Árbol Huffman que obtenemos luego de procesar todos los nodos de la lista.

Observemos que el árbol resultante tiene caracteres ficticios en los vértices y caracteres reales en las hojas.

16.2.4 Paso 4 - Asignación de códigos Huffman

El siguiente paso será asignar un código Huffman a cada uno de los caracteres reales que se encuentran ubicados en las hojas del árbol. Para esto, consideraremos el camino que se debe recorrer para llegar a cada hoja. El código se forma concatenando un 0 (cero) por cada tramo que avanzamos hacia la izquierda y un 1 (uno) cada vez que avanzamos hacia la derecha. Por lo tanto, el código Huffman que le corresponde al carácter 'O' es 01, el código que le corresponde al carácter 'M' es 001 y el código que le corresponde al carácter 'S' es 10100.

Como podemos ver, la longitud del código que el árbol le asigna a cada carácter es inversamente proporcional a su cantidad de ocurrencias. Los caracteres más frecuentes reciben códigos más cortos mientras que los menos frecuentes reciben códigos más largos.

Agreguemos los códigos Huffman (*cod*) y sus longitudes (*nCod*) en la tabla de ocurrencias.

	Carácter	*n*	*cod*	*nCod*
0				
:				
32	ESP	5	100	3
:				
65	A	2	1110	4
:				
67	C	7	000	3
:				
69	E	2	1101	4
:				
73	I	1	10111	5

:	Carácter	*n*	*cod*	*nCod*
:				
77	M	5	001	3
78	N	1	10110	5
79	O	11	01	2
:				
82	R	1	10101	5
83	S	1	10100	5
84	T	2	1100	4
85	U	1	11111	5
:				
255				

Fig. 16.8 Tabla de ocurrencias completa.

16.2.4.1 Longitud máxima de un código Huffman

El lector podrá pensar que, en el peor de los casos, el código Huffman más largo que se asignará a un carácter será de 8 bits porque esta es la cantidad original con la que el carácter está codificado, pero esto no es así.

La longitud máxima que puede alcanzar un código Huffman dependerá de la cantidad de símbolos que componen el alfabeto con el que esté codificada la información. De modo que, sea *n* la cantidad de símbolos que componen un alfabeto, entonces, en el peor de los casos, el código Huffman que se asignará a un carácter podrá tener hasta *n*/2 bits.

En nuestro caso el alfabeto está compuesto por cada una de las 256 combinaciones que admite un *byte*; por lo tanto, tenemos que estar preparados para trabajar códigos de, a lo sumo, 256/2 = 128 bits.

Obviamente, el hecho de asignar códigos de menos de 8 bits a los caracteres más frecuentes compensa la posibilidad de que, llegado el caso, se utilicen más de 8 bits para codificar los caracteres que menos veces aparecen.

16.2.5 Paso 5 - Codificación del contenido

Para finalizar, generamos el archivo comprimido reemplazando cada carácter del archivo original por su correspondiente código Huffman, agrupando los diferentes códigos en paquetes de 8 bits ya que esta es la menor cantidad de información que podemos manipular.

C	O	M	O	*[esp]*	C	...
000	01	001	01	100	000	...
00001001			01100000			...

Fig. 16.9 Compresión de los primeros *bytes* del archivo original.

16.2.6 Proceso de decodificación y descompresión

Para descomprimir un archivo necesitamos disponer del árbol Huffman utilizado para su codificación. Sin el árbol la información no se podrá recuperar. Por este motivo el algoritmo de Huffman también puede utilizarse como algoritmo de encriptación.

Supongamos que, de alguna manera, podemos rearmar el árbol Huffman, entonces el algoritmo para descomprimir y restaurar el archivo original es el siguiente:

1. Rearmar el árbol Huffman y posicionarnos en la raíz.
2. Recorrer "bit por bit" el archivo comprimido. Si leemos un 0 descendemos un nivel del árbol posicionándonos en su hijo izquierdo. En cambio, si leemos un 1 descendemos un nivel para posicionarnos en su hijo derecho.
3. Repetimos el paso 2 hasta llegar a una hoja. Esto nos dará la pauta de que hemos decodificado un carácter y lo podremos grabar en el archivo que estamos restaurando.

16.3 Aplicación práctica

Analizaremos una implementación del algoritmo de Huffman para desarrollar dos programas que nos permitirán comprimir y descomprimir archivos.

16.3.1 Compresor de archivos (`szzip.java`)

Este programa recibirá como parámetro en línea de comandos el nombre del archivo que debe comprimir. Luego procesará el archivo indicado y como resultado del proceso

generará dos nuevos archivos que contengan la información comprimida (`.szdat`) y el árbol Huffman que se utilizó durante la codificación (`.szcod`). El árbol Huffman será necesario para poder, luego, decodificar y descomprimir la información.

Por ejemplo, si en línea de comandos invocamos al compresor de la siguiente manera:

```
C:\>java szzip dibujo.bmp
```

el programa generará los archivos `dibujo.bmp.szdat` y `dibujo.bmp.szcod`, cuyos contenidos serán la información codificada y el árbol Huffman respectivamente.

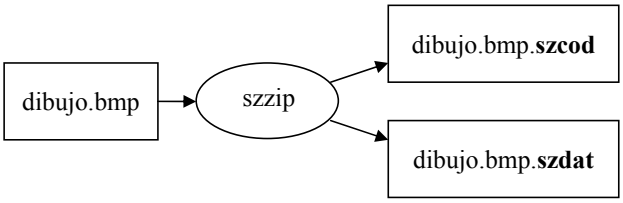

Fig. 16.10 szzip.java –Comprime dibujo.bmp y genera los archivos .szcod y .szdat.

16.3.2 Descompresor de archivos (`szunzip.java`)

El programa descompresor debe realizar la tarea inversa al programa compresor. Para esto, recibirá como parámetro en línea de comandos el nombre del archivo que se quiere restaurar y, como resultado del proceso, generará el archivo original.

Por ejemplo, si en línea de comandos invocamos al programa descompresor de la siguiente manera:

```
C:\>java szunzip dibujo.bmp
```

el resultado será el siguiente:

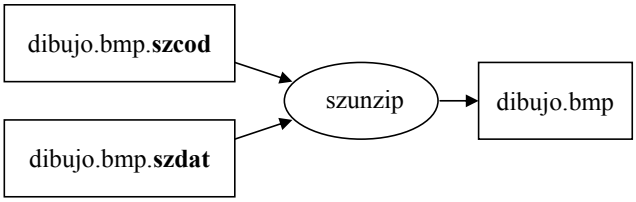

Fig. 16.11 szunzip.java – Dados los archivos `.szcod` y `.szdat`, restaura el archivo original.

16.3.3 Estructura del archivo `.szcod` (árbol Huffman)

Como ya vimos, la única forma de descomprimir y recuperar la información que ha sido codificada en un archivo `.szdat` (comprimido) es recomponiendo el árbol Huffman que se utilizó para codificarla. Por este motivo el compresor, además de generar el archivo comprimido, debe generar un archivo que contenga la información que nos permita recomponer el árbol Huffman. Este archivo tendrá la extensión `.szcod` y su estructura será la siguiente:

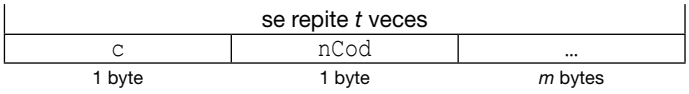

Fig. 16.12 Estructura del archivo `.szcod`.

Si consideramos que el árbol tiene exactamente *t* hojas entonces los *bytes* del archivo `.szcod` repetirán *t* veces el siguiente patrón:

- 1 *byte* que contiene el carácter (o valor numérico) representado por la hoja del árbol.
- 1 *byte* que indica la longitud del código Huffman con el que dicho carácter fue codificado.
- *m bytes* que contienen los bits que componen el código Huffman asignado al carácter, donde *m* = *nCod*/8 + (*nCod*%8 !=0 ? 1:0).

16.3.4 Estructura del archivo `.szdat` (archivo comprimido)

Este archivo contiene la información comprimida reemplazando cada *byte* del archivo original por su correspondiente código Huffman. Para generarlo tenemos que recorrer el archivo original y, por cada *byte* leído, debemos escribir la secuencia de bits que compone su código.

La tabla que veremos a continuación tiene tres filas: en la primera se muestran los caracteres del archivo original, en la segunda se muestra el código Huffman asociado al carácter y en la tercera se agrupan los códigos Huffman en paquetes de 8 bits para obtener así los *bytes* que forman el archivo comprimido.

C	O	M	O		C	O	M	E		C	O	C	O	R	I
000	*01*	*001*	*01*	*100*	*000*	*01*	*001*	*1101*	*100*	*000*	*01*	*000*	*01*	*10101*	*10111*
00001001			01100000			01001110			**1**1000000			**1**0000110			**101**10111

T	O		C	O	M	E		C	O	M
1100	*01*	*100*	*000*	*01*	*001*	*1101*	*100*	*000*	*01*	*001*
11000110			**0**0000100			**1**1101100		00001001		

O		C	O	S	M	O	N	A	U	T	A	
01	*100*	*000*	*01*	*10100*	*001*	*01*	*10110*	*1110*	*11111*	*1100*	*1110*	
01100000			011010000			**01**011011		**0**11101111		**1 1**110011		**10**000000

Fig. 16.13 Codificación completa del texto analizado.

Como vemos, solo necesitamos 16 *bytes* para representar los 39 *bytes* del texto original. Además, grabaremos en este archivo, en los primeros 8 *bytes*, un valor numérico entero de tipo `long` que indique la cantidad total de *bytes* codificados o, en otras palabras, la longitud que tenía el archivo original.

El problema que se presentará a la hora de generar el archivo `.szdat` es el siguiente: los códigos Huffman se forman con sucesiones de bits de longitud variable. Habrá códigos de menos de 8 bits y códigos más extensos. Como en un archivo solo podemos escribir cantidades de *z* bits, siendo *z* un múltiplo de 8, tendremos que "concatenar" los códigos hasta formar un grupo de 8 bits. Recién allí los podremos escribir. Muy probablemente un mismo *byte* contenga más de un código Huffman.

Resolver este punto implica cierto grado de complejidad que excede el objetivo de este trabajo. Por esto, como parte del *setup* del ejercicio, se provee la clase `UFile`, cuyos métodos `writeBit` y `readBit` encapsulan la lógica necesaria para resolverlo, permitiéndole al programador abstraerse del problema. Además, los métodos `writeLength` y `readLength` le permitirán al programador grabar en los primeros *bytes* la longitud del archivo original y luego recuperarla.

Análogamente, para "navegar" a través de las hojas del árbol binario, el *setup* del ejercicio incluye la clase `UTree`, cuyo método `next` retornará la siguiente hoja del árbol cada vez que se lo invoque, comenzando por la hoja ubicada "más a la izquierda" hasta llegar a la hoja ubicada "más a la derecha". El método, además, retornará el código Huffman con el que se debe codificar al carácter representado por la hoja que retornó.

16.3.5 El setup del ejercicio

Llamaremos *setup* al conjunto de recursos (clases e *interfaces*) que se proveen para que el lector pueda desarrollar el ejercicio.

El *setup* incluye las siguientes clases utilitarias:

- `UFactory` – Factoría de objetos.
- `UFile` – Permite leer y escribir bits en un archivo.
- `UTree` – Permite navegar a través de las hojas de un árbol binario.

Más adelante, veremos ejemplos de cómo usar cada una de estas clases.

También se incluye la clase `Node`, con la que implementaremos la lista enlazada y el árbol binario. Su código fuente es el siguiente:

```
package jhuffman.def;

public class Node
{
   private int c;
   private long n;

   private Node sig = null;
   private Node izq = null;
   private Node der = null;

   // :
   // setters y getters...
   // :
}
```

El *setup* incluye también el código fuente de los programas `szzip` y `szunzip` y de todas las *interfaces* en las que se basa su codificación. Todo esto lo analizaremos más adelante.

16.3.5.1 Self testing (autotest)

Por último, se incluye en el *setup* un conjunto de programas de "autotest" para que el lector (programador) pueda probar las implementaciones a medida que las va desarrollando.

Estos programas invocan y prueban el funcionamiento de cada uno de los métodos programados en las diferentes implementaciones, indicándole al programador si el resultado de la prueba fue exitoso o no.

Un resultado no exitoso significa que la implementación tiene un error y, mientras este no sea corregido, no tiene sentido seguir avanzando con la implementación de otra *interface*.

En cambio, un resultado exitoso solo indica que la prueba fue satisfactoria para ese caso en particular pero no garantiza que la implementación esté libre de errores.

16.4 Análisis de clases y objetos

Hagamos un breve repaso de la estrategia que utilizaremos para implementar el compresor/descompresor de archivos destacando en negrita los objetos que intervienen en el proceso. Veamos:

Recorreremos el **archivo que se va a comprimir** para crear una **tabla de ocurrencias** en la que vamos a contar cuántas veces aparece cada carácter. Luego crearemos una **lista enlazada** donde cada nodo representará a cada uno los diferentes caracteres, ordenada de menor a mayor según su probabilidad de aparición. La lista la convertiremos en un **árbol binario**. Los diferentes caminos hacia las hojas del árbol nos darán los **códigos Huffman** que nos permitirán recodificar cada carácter. Luego asignaremos en cada registro de la tabla su código Huffman correspondiente y, a partir de esto, generaremos el **archivo de códigos**. Finalmente, con los códigos Huffman que tenemos en la tabla recorreremos el archivo que se va a comprimir para sustituir cada *byte* por su nuevo código, generando así el **archivo comprimido**.

En el proceso intervienen los siguientes objetos:

- Archivo que se va a comprimir
- Tabla de ocurrencias
- Lista enlazada
- Árbol binario
- Código Huffman
- Archivo de códigos
- Archivo comprimido

El análisis de las relaciones entre estos objetos nos ayudará a definir sus *interfaces*.

Veamos:

Recorreremos el **archivo que se va a comprimir** para crear una **tabla de ocurrencias** en la que vamos a contar cuántas veces aparece cada carácter.

```
TablaOcurrencias tabla  = archivoAComprimir.crearTablaOcurrencias();
```

Luego crearemos una **lista enlazada** con los diferentes caracteres, ordenada de menor a mayor según su probabilidad de aparición.

```
ListaOrdenada lista = tabla.crearLista();
```

La lista la convertiremos en un **árbol binario**.

```
ArbolHuffman arbol = lista.convertirEnArbol();
```

Los diferentes caminos hacia las hojas del árbol nos darán los **códigos Huffman** que nos permitirán recodificar cada carácter. Luego asignaremos en cada registro de la tabla su código Huffman correspondiente.

```
tabla.cargarCodigosHuffman(arbol);
```

A partir de esto generaremos el **archivo de códigos**.

```
archivoDeCodigos.grabar(tabla);
```

Finalmente, con los códigos Huffman que tenemos en la tabla recorreremos el archivo que se va a comprimir para sustituir cada *byte* por su nuevo código, generando así el **archivo comprimido**.

```
archivoComprimido.grabar(archivoAComprimir,tabla);
```

Los tipos de datos de estos objetos serán *interfaces*. Esto nos permitirá avanzar en el análisis del ejercicio y abstraernos de la implementación que, como ya dijimos, quedará a cargo del lector.

Objeto	*Interface*
archivo que se va a comprimir	IFileInput
tabla de ocurrencias	ITable
lista enlazada	IList
árbol binario	ITree
código Huffman	ICode
archivo de códigos	IFileCode
archivo comprimido	IFileCompressed

Fig. 16.14 *Interfaces* de los objetos.

16.4.1 szzip.java

Veamos el código fuente del compresor, donde podremos apreciar la gran importancia de separar las *interfaces* de las implementaciones.

Respecto del programa en sí, el lector verá que es lineal. Comenzamos instanciando la clase que representa al archivo de entrada. Luego le pedimos que cree la tabla de ocurrencias. A la tabla le pedimos que cree la lista enlazada y a esta le pedimos el árbol Huffman. Luego utilizamos el árbol para cargar los códigos en la tabla. Por último, instanciamos la clase que representa el archivo de códigos y lo generamos utilizando la información contenida en la tabla. Más adelante instanciamos el archivo comprimido y lo generamos en función del archivo de entrada y de la tabla que contiene los códigos Huffman asociados a cada *byte*.

```
package jhuffman;

import jhuffman.def.*;
import jhuffman.util.UFactory;

public class szzip
{
   public static void main(String[] args)
   {
      // abrimos el archivo original (archivo que se va a comprimir)
      IFileInput inputFile = (IFileInput) UFactory.getObject("inputFile");
      fi.setFilename(args[0]);

      // obtenemos la tabla de ocurrencias
      ITable table = inputFile.createTable();

      // obtenemos la lista enlazada
      IList list = table.createSortedList();

      // convertimos la lista en arbol
      ITree tree = list.toTree();

      // asignamos los codigos en la tabla
      table.loadHuffmanCodes(tree);

      // abrimos el archivo de codigo
      IFileCode codeFile = (IFileCode) UFactory.getObject("codeFile");
      codeFile.setFilename(args[0] + ".szcod");
```

```
      // grabamos el archivo tomando los codigos del arbol
      codeFile.save(table);

      // abrimos el archivo comprimido
      IFileCompressed compressFile = (IFileCompressed) UFactory.getObject("compressFile");
      compressFile.setFilename(args[0] + ".szdat");

      // grabamos el archivo comprimido
      compressFile.save(inputFile,table);
   }
}
```

Los únicos objetos que instanciamos en `szzip` son `inputFile` (archivo de entrada o archivo original y que queremos comprimir), `codeFile` (archivo que grabaremos con los códigos Huffman) y `compressedFile` (archivo comprimido).

Como no conocemos sus implementaciones porque de hecho no existen aún, utilizamos la clase `UFactory` que se provee con el *setup* del ejercicio para crear instancias desentendiéndonos de sus implementaciones.

16.4.2 `szunzip.java`

El descompresor es más simple. Instanciamos el archivo de códigos Huffman y le pedimos que restaure el árbol. Luego utilizamos el árbol y el archivo comprimido para restaurar el archivo original.

```
package jhuffman;

import jhuffman.def.*;
import jhuffman.util.UFactory;

public class szunzip
{
   public static void main(String[] args)
   {
      // abrimos el archivo de codigos
      IFileCode codeFile = (IFileCode) UFactory.getObject("codeFile");
      codeFile.setFilename(args[0] + ".szcod");

      // leemos el archivo y generamos el arbol
      ITree tree = codeFile.load();

      // abrimos el archivo comprimido
      IFileCompressed compressFile = (IFileCompressed) UFactory.getObject("compressFile");
      compressFile.setFilename(args[0] + ".szdat");

      // abrimos el archivo original (archivo que se va a restaurar)
      IFileInput inputFile = (IFileInput) UFactory.getObject("inputFile");
      inputFile.setFilename(args[0]);

      // recuperamos el archivo original
      compressFile.restore(inputFile,tree);
   }
}
```

16.5 Interfaces e implementaciones

Llegó el momento de programar. Ahora definiremos los métodos de cada una de las *interfaces* para que el lector, como programador, desarrolle las implementaciones correspondientes y haga que los programas `szzip` y `szunzip` funcionen.
La especificación de la tarea que cada método debe realizar está documentada en el mismo código fuente de la *interface* . Por esto, sugiero prestar especial atención a esta información.

16.5.1 `ICode.java` - Interface de los códigos Huffman

Esta *interface* representa a un código Huffman, es decir, una secuencia de, a lo sumo, 128 bits.

```
package jhuffman.def;

public interface ICode
{
   // retorna el i-esimo bit (contando desde cero, de izquierda a derecha)
   public int getBitAt(int i);

   // retorna la longitud de este codigo Huffman (valor entre 1 y 128)
   public int getLength();

   // inicializa el codigo Huffman con los caracteres de sCod
   // que seran "unos" o "ceros"
   public void fromString(String sCod);
}
```

16.5.2 `ITree.java` - Interface del árbol binario o árbol Huffman

`ITree` representa al árbol Huffman.

```
package jhuffman.def;

public interface ITree
{
   // asigna la raiz del arbol (es decir: primer y unico nodo de la lista)
   public void setRoot(Node root);

   // retorna la raiz del arbol
   public Node getRoot();

   // en cada invocacion retorna la siguiente hoja del arbol
   // comenzando desde la que esta "mas a la izquierda"
   // NOTA: utilizar el metodo next de la clase UTree
   public Node next(ICode cod);
}
```

16.5.3 `IList.java` - Interface de la lista enlazada

La *interface* `IList` representa una lista enlazada, ordenada de menor a mayor según la cantidad de ocurrencias de cada carácter. Si dos caracteres aparecen la misma cantidad de veces entonces se considera menor al que tenga menor código ASCII o valor numérico. Los caracteres ficticios se consideran alfabéticamente mayores que los demás.

```
package jhuffman.def;

import java.util.Comparator;

public interface IList
{
   // desenlaza y retorna el primer nodo de la lista
   public Node removeFirstNode();

   // agrega el nodo n segun el critero de ordenamiento explicado mas arriba
   public void addNode(Node n);

   // convierte la lista en un arbol y lo retorna
   // NOTA: usar el metodo removeFirstNode
   public ITree toTree();

   // retorna una instancia (implementacion) de Comparator<Node> tal
   // que permita determinar si un nodo precede o no a otro segun
   // el criterio de ordenamiento antes mencionado
   public Comparator<Node> getComparator();
}
```

16.5.4 `ITable.java` - Interface de la tabla de ocurrencias

La *interface* `ITable` representa la tabla de ocurrencias donde llevaremos la cuenta de cuántas veces aparece cada carácter dentro del archivo que vamos a comprimir.

```
package jhuffman.def;

public interface ITable
{
   // incrementa el contador relacionado al caracter (o byte) c
   public void addCount(int c);

   // retorna el valor del contador asociado al caracter (o byte) c
   public long getCount(int c);

   // crea la lista enlazada
   public IList createSortedList();

   // almacena en la tabla el codigo Huffman asignado a cada caracter
   // NOTA: usar el metodo next del parametro tree
   public void loadHuffmanCodes(ITree tree);

   // retorna el codigo Huffman asignado al caracter c
   public ICode getCode(int c);
}
```

16.5.5 `IFileInput.java` - Interface del archivo que vamos a comprimir o a restaurar

```
package jhuffman.def;

public interface IFileInput
{
   // asigna el nombre del archivo
   public void setFilename(String filename);

   // retorna el nombre del archivo
   public String getFilename();

   // crea y retorna la tabla de ocurrencias con los contadores de los
   // diferentes bytes que aparecen en el archivo
   public ITable createTable();

   // retorna la longitud del archivo
   public long getLength();
}
```

16.5.6 `IFileCode.java` - Interface del archivo de códigos Huffman

Representa al archivo `.szcod` que se genera durante la compresión y que contiene los códigos Huffman asociados a cada carácter del archivo original.

```
package jhuffman.def;

public interface IFileCode
{
   // asigna el nombre del archivo
   public void setFilename(String f);

   // graba el archivo tomando los codigos Huffman de la tabla
   // NOTA: usar el metodo writeBit de la clase UFile
   public void save(ITable table);

   // lee el archivo (ya generado) y construye el arbol Huffman
   // NOTA: usar el metodo readBit de la clase UFile.
   // Tambien es probable que se necesite utilizar el metodo parseInt
   // de la clase Integer para convertir un numero binario en un
   // valor entero, asi:
   // int v = Integer.parseInt("10101",2); // retorna el valor 21
   public ITree load();
}
```

16.5.7 `IFileCompressed.java` - Interface del archivo comprimido

Representa al archivo `.szdat` que contiene la información codificada y comprimida.

```
package jhuffman.def;

public interface IFileCompressed
{
   // asigna el nombre del archivo
   public void setFilename(String filename);

   // retorna el nombre del archivo
   public String getFilename();

   // graba el archivo comprimido recorriendo el archivo de entrada
   // y reemplazando cada caracter por su correspondiente codigo Huffman.
   // Al principio del archivo debe grabar un long con la longitud en bytes
   // que tenia el archivo original
   // NOTA: usar los metodos writeLength y writeBit de la clase UFile.
   public void save(IFileInput inputFile, ITable table);

   // restaura el archivo original recorriendo el archivo comprimido
   // y, por cada bit leido, se desplaza por las ramas del arbol
   // hasta llegar a la hoja que contiene el caracter por escribir.
   // Recordar que los primeros bytes del archivo .szdat indican la longitud
   // en bytes del archivo original
   // NOTA: usar los metodos readLength y readBit de la clase UFile
   public void restore(IFileInput inputFile, ITree tree);
}
```

16.6 Manejo de archivos en Java

Para implementar los métodos `save` y `restore` de la *interface* `IFileCompressed`, el método `save` de la *interface* `IFileCode` y el método `createTable` de la *interface* `IFileInput` será imprescindible conocer parte de la API a través de la cual los programas Java pueden leer y escribir archivos.

En esta sección veremos algunos ejemplos simples que nos permitirán comprender cómo funciona la administración de archivos en Java.

16.6.1 Leer un archivo (clase `FileInputStream`)

La clase `FileInputStream` provee el método `read` a través del cual podemos leer "*byte* por *byte*" el contenido de un archivo que abriremos "para lectura".

En el siguiente ejemplo abrimos un archivo y, por cada *byte*, incrementamos un contador. Al finalizar mostramos el valor del contador que coincidirá con el tamaño (en *bytes*) del archivo que leímos.

```
package jhuffman.util.demos;

import java.io.FileInputStream;

public class TestFileInputStream
{
```

```
   public static void main(String[] args)
   {
      FileInputStream fis = null;

      try
      {
         // el nombre del archivo se debe pasar en linea de comandos
         String nombreArchivo = args[0];

         // abrimos el archivo
         fis = new FileInputStream(nombreArchivo);

         // contador para contar cuantos bytes tiene el archivo
         int cont=0;

         // leemos el primer byte
         int c = fis.read();

         // iteramos mientras no llegue el EOF, representado por -1
         while( c!=-1 )
         {
            cont++;

            // leemos el siguiente byte
            c = fis.read();
         }

         System.out.println(nombreArchivo+" tiene "+cont+" bytes");
      }

      catch(Exception ex)
      {
         ex.printStackTrace();
         throw new RuntimeException(ex);
      }
      finally
      {
         try
         {
            // cerramos el archivo
            if( fis!=null ) fis.close();
         }
         catch(Exception ex)
         {
            ex.printStackTrace();
            throw new RuntimeException(ex);
         }
      }
   }
}
```

El método `read` lee un *byte* y retorna su valor, que será mayor o igual que cero y menor o igual que 255. Al llegar el *end of file*, para indicar que no hay más *bytes* para leer, el método retornará -1.

Para asegurarnos de que el archivo quede correctamente cerrado, invocamos al método `close` dentro de la sección `finally` de un gran bloque *try-catch-finally*.

16.6.2 Bytes sin bit de signo

Java provee el tipo de datos `byte` que permite representar un valor numérico entero en 1 *byte* de memoria. El problema es que este tipo de datos es signado y, al no existir el modificador *unsigned*, una variable de tipo `byte` solo puede contener valores comprendidos entre -128 y 127.

Para trabajar con archivos binarios necesitaremos leer *bytes* sin bit de signo, es decir, valores numéricos enteros comprendidos entre 0 y 255. Dado que el tipo `byte` no soporta este rango de valores, en Java se utiliza el tipo `int` para representar *unsigned bytes*.

16.6.3 Escribir un archivo (clase `FileOutputStream`)

La clase `FileOutputStream` representa un archivo que se abrirá para escritura. El método `write` permite escribir "*byte* por *byte*" en el archivo.

En el siguiente programa escribiremos una cadena de caracteres (una sucesión de *bytes*) en un archivo. Tanto el nombre del archivo como el texto de la cadena que vamos a escribir deben especificarse como argumentos en línea de comandos.

```
package jhuffman.util.demos;

import java.io.FileOutputStream;

public class TestOutputStream
{
   public static void main(String[] args)
   {
      FileOutputStream fos = null;

      try
      {
         String nombreArchivo = args[0];
         String textAGrabar = args[1];

         // abrimos el archivo para escritura
         fos = new FileOutputStream(nombreArchivo);

         // recorremos la cadena
         for( int i=0; i<textAGrabar.length(); i++ )
         {
            int c = textAGrabar.charAt(i);

            // grabamos el siguiente byte
            fos.write(c);
         }
      }
      catch(Exception ex)
      {
         ex.printStackTrace();
         throw new RuntimeException(ex);
      }
      finally
      {
```

```
            try
            {
               if( fos!=null ) fos.close();
            }
            catch(Exception ex)
            {
               ex.printStackTrace();
               throw new RuntimeException(ex);
            }
         }
      }
}
```

16.6.4 Buffers de entrada y salida (clases `BufferedInputStream` y `BufferredOutputStream`)

Como estudiamos en su momento, el uso de *buffers* incrementa notablemente el rendimiento de las operaciones de entrada y salida.

En Java, las clases `BufferredInputStream` y `BufferredOutputStream` implementan los mecanismos de *buffers* de lectura y escritura respectivamente.

El siguiente programa graba en un archivo de salida el contenido de un archivo de entrada, es decir, copia el archivo.

```
package jhuffman.util.demos;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;

public class TestBuffersIO
{
   public static void main(String[] args)
   {
      BufferedInputStream bis = null;
      BufferedOutputStream bos = null;

      try
      {
         // nombre de archivo origen
         String desde = args[0];

         // nombre de archivo destino
         String hasta = args[1];

         bis = new BufferedInputStream(new FileInputStream(desde));
         bos = new BufferedOutputStream(new FileOutputStream(hasta));

         // leemos el primer byte desde el buffer de entrada
         int c = bis.read();

         while( c!=-1 )
         {
```

```
            // escribimos el byte en el buffer de salida
            bos.write(c);

            // leemos el siguiente byte
            c = bis.read();
         }

          // vaciamos el contenido del buffer
         bos.flush();
      }
      catch(Exception ex)
      {
         ex.printStackTrace();
         throw new RuntimeException(ex);
      }
      finally
      {
         try
         {
            if( bos!=null ) bos.close();
            if( bis!=null ) bis.close();
         }
         catch(Exception ex)
         {
            ex.printStackTrace();
            throw new RuntimeException(ex);
         }
      }
   }
}
```

Por defecto, el tamaño de *buffer* tanto de lectura como de escritura es de 8192 *bytes* (8 KB). Podemos especificar el tamaño de *buffer* con el que queremos trabajar pasándolo como segundo argumento en el constructor.

Por ejemplo, en las siguientes líneas de código utilizamos como tamaño de *buffer* el valor que el usuario indique como tercer argumento en línea de comandos.

```
int bufferSize = Integer.parseInt(args[2]);
bis = new BufferedInputStream(new FileInputStream(args[0]),bufferSize);
bos = new BufferedOutputStream(new FileOutputStream(args[1]),bufferSize);
```

16.7 Clases utilitarias

Las clases utilitarias son recursos que se incluyen en el *setup* del ejercicio con el objetivo de abstraer de determinadas cuestiones al programador. Dichas cuestiones están relacionadas con los siguientes temas:

- Recorrer el árbol binario, ya que este tema lo estudiaremos más adelante.
- Leer y escribir bits en archivos, para evitar distraer la atención del programador.
- Factoría de objetos.

A continuación, veremos cómo usar cada una de estas clases.

16.7.1 La clase `UTree` - Recorrer el árbol binario

La clase `UTree` permite recorrer el árbol Huffman a través de sus hojas. Para esto, provee el método `next` que, en cada invocación, retornará la siguiente hoja o `null` cuando no tenga más hojas para retornar. Veamos un ejemplo que ilustra su uso:

```
package jhuffman.util.demos;

import jhuffman.def.Node;
import jhuffman.util.UTree;

public class UTreeDemo
{
   public static void main(String[] args)
   {
      // obtenemos el arbol Huffman
      Node root = crearArbolHuffman();

      // instanciamos un objeto de la clase UTree a partir de la raiz del arbol
      UTree uTree = new UTree(root);

      // buffer donde el metodo next asignara los codigos Huffman
      StringBuffer codHuffman = new StringBuffer();

      // obtenemos la primera hoja
      Node hoja = uTree.next(codHuffman);

      // iteramos mientras el metodo no retorne null
      while( hoja!=null )
      {
         // mostramos la hoja leida
         String s = ((char)hoja.getC())+"("+hoja.getN()+") ";
         System.out.println(s + "codigo = {"+codHuffman+"}");

         // obtenemos la siguiente hoja
         hoja = uTree.next(codHuffman);
      }
   }

   // sigue mas abajo
   // :
```

La primera vez que invocamos al método `next` retornará la hoja del árbol ubicada "más a la izquierda". Cada invocación subsiguiente avanzará hacia la derecha hasta llegar a la última hoja. Luego retornará `null`. El método, además, asigna en el *stringBuffer* que recibe como parámetro el código Huffman correspondiente a la hoja que retornó.

La salida de este programa será:

```
C(7) CODIGO = {000}
M(5) CODIGO = {001}
A(2) CODIGO = {010}
U(1) CODIGO = {011}
 (5) CODIGO = {100}
S(1) CODIGO = {10100}
R(1) CODIGO = {10101}
N(1) CODIGO = {10110}
I(1) CODIGO = {10111}
T(2) CODIGO = {1100}
E(2) CODIGO = {1101}
A(2) CODIGO = {1110}
U(1) CODIGO = {1111}
```

Veamos el método `crearArbolHuffman` donde se “hardcodea” el árbol correspondiente al texto “COMO COME COCORITO COME COMO COSMONAUTA”.

```
   // :
   // viene de mas arriba

   private static Node crearArbolHuffman()
   {
      // nivel 5 (ultimo nivel)
      Node nS = node('S', 1, null, null);
      Node nR = node('R', 1, null, null);
      Node nN = node('N', 1, null, null);
      Node nI = node('I', 1, null, null);

      // nivel 4
      Node a2 = node(256+2, 2, nS, nR);
      Node a1 = node(256+1, 2, nN, nI);
      Node nT = node('T', 2, null, null);
      Node nE = node('E', 2, null, null);
      Node nA = node('A', 2, null, null);
      Node nU = node('U', 1, null, null);

      // nivel 3
      Node nC = node('C', 7, null, null);
      Node nM = node('M', 5, null, null);
      Node nESP = node(' ', 5, null, null);
      Node a5 = node(256+5, 4, a2, a1);
      Node a4 = node(256+4, 4, nT, nE);
      Node a3 = node(256+3, 3, nA, nU);

      // nivel 2
      Node a8 = node(256+8, 12, nC, nM);
      Node nO = node('O', 3, nA, nU);
      Node a7 = node(256+7, 9, nESP, a5);
      Node a6 = node(256+6, 7, a4, a3);

      // nivel 1
      Node a10 = node(256+10, 23, a8, nO);
      Node a9  = node(256+9, 16, a7, a6);

      // nivel 0 (raiz)
      Node a11 = node(256+11, 39, a10, a9);

      return a11;
   }

   private static Node node(int c, long n, Node izq, Node der)
   {
      Node node = new Node();
      node.setC(c);
      node.setN(n);
      node.setIzq(izq);
      node.setDer(der);
      node.setSig(null);
      return node;
   }
}
```

16.7.2 La clase `UFile` - Leer y escribir bits en un archivo

La lógica del compresor de archivos basado en el algoritmo de Huffman consiste en reemplazar cada *byte* del archivo original por su correspondiente código Huffman que, generalmente, será una secuencia de menos de 8 bits.

El problema que surge es el siguiente: la mínima unidad de información que podemos escribir (o leer) en un archivo es 1 *byte* (8 bits). No podemos escribir o leer "bit por bit" a menos que lo simulemos.

Por ejemplo: podemos desarrollar un método `escribirBit` que reciba el bit (1 o 0) que queremos escribir y lo guarde en una variable de instancia hasta juntar un paquete de 8 bits. En cada invocación juntará un bit más. Así, cuando haya reunido los 8 bits podrá grabar un *byte* completo en el archivo.

Para leer, el proceso es inverso: deberíamos desarrollar el método `leerBit` que en la primera invocación lea un *byte* desde el archivo y retorne el primero de sus bits. Durante las próximas 7 invocaciones deberá retornar cada uno de los siguientes bits del *byte* que ya tiene leído. En la invocación número 9 no tendrá más bits para entregar, por lo que deberá volver al archivo para leer un nuevo *byte*.

La clase `UFile` hace exactamente esto. Veamos un ejemplo:

```
package jhuffman.util.demos;

import java.io.FileInputStream;
import java.io.FileOutputStream;

import jhuffman.util.UFile;

public class UFileDemo
{
   public static void main(String[] args)
   {
      // grabamos en DEMO.dat los bits: 1011101101110
      grabarBits("DEMO.dat","1011101101110");

      // recorremos DEMO.dat "bit x bit" e imprimimos cada bit leido
      leerBits("DEMO.dat");
   }

   private static void grabarBits(String nomArch, String bits)
   {
      FileOutputStream fos = null;

      try
      {
         // abrimos el archivo para grabar los bits
         fos = new FileOutputStream(nomArch);

         // instanciamos UFile
         UFile uFile = new UFile(fos);

         // recorremos la cadena de bits que queremos escribir
         for(int i=0; i<bits.length(); i++)
```

```
        {
            // obtenemos el i-esimo bit (1 o 0)
            int bit = bits.charAt(i)-'0';

            // lo grabamos en el archivo
            uFile.writeBit(bit);
        }

        // si quedo un paquete a medio formar lo completamos con
        // ceros a la derecha y lo grabamos
        uFile.flush();
    }
    catch(Exception ex)
    {
        ex.printStackTrace();
        throw new RuntimeException(ex);
    }
    finally
    {
        try
        {
            if(fos!=null) fos.close();
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }
}

private static void leerBits(String nomArch)
{
    FileInputStream fis = null;

    try
    {
        // abrimos el archivo para grabar los bits
        fis = new FileInputStream(nomArch);

        // instanciamos UFile
        UFile uFile = new UFile(fis);

        // recorremos la cadena de bits
        int bit = uFile.readBit();

        // cuando no haya mas bits retornara un valor negativo
        while( bit>=0 )
        {
            // mostramos el bit
            System.out.println(bit);

            // leemos el proximo bit
            bit = uFile.readBit();
        }
```

```
      }
      catch(Exception ex)
      {
         ex.printStackTrace();
         throw new RuntimeException(ex);
      }
      finally
      {
         try
         {
            if(fis!=null) fis.close();
         }
         catch(Exception ex)
         {
            ex.printStackTrace();
            throw new RuntimeException(ex);
         }
      }
   }
}
```

16.7.3 La clase `UFactory` - Factoría de objetos

Esta clase implementa una factoría de objetos similar a las que estudiamos en el capítulo de programación orientada a objetos. Las implementaciones de las diferentes *interfaces* se configuran en un archivo llamado `factory.properties` que debe estar ubicado en el directorio de corrida del programa.

Por ejemplo, si el archivo `factory.properties` tiene las siguientes líneas:

```
table=ejercicio.imple.ITableImple
code=ejercicio.imple.ICodeImple
```

entonces, si las clases `ejercicio.imple.ITableImple` y `ejercicio.imple.ICodeImple` son implementaciones de las *interfaces* `ITable` e `ICode` respectivamente, en nuestro programa podremos instanciar "objetos table" y "objetos code" sin preocuparnos por conocer su implementación; simplemente nos manejamos al nivel de sus *interfaces*.

```
package jhuffman.util.demos;

import jhuffman.def.ICode;
import jhuffman.def.ITable;
import jhuffman.util.UFactory;

public class UFactoryDemo
{
   public static void main(String[] args)
   {
      // instanciamos la tabla
      ITable table = (ITable) UFactory.getObject("table");

      // instanciamos un codigo Huffman
      ICode code = (ICode) UFactory.getObject("code");
```

```
        //
        // ...
        //
    }
}
```

16.8 Resumen

En este capítulo estudiamos el algoritmo de Huffman y lo aplicamos al desarrollo de un compresor/descompresor de archivos.

Este desarrollo nos permitió poner a prueba muchos de los conceptos que estudiamos en los capítulos anteriores y aplicar también otros conceptos que, aun sin haberlos estudiado, pudimos utilizar gracias al poder de encapsulamiento que proveen las clases.

El hecho de usar *interfaces* me permitió a mí, como programador más experimentado, ocupar el rol de diseñador de la aplicación desentendiéndome de su implementación ya que esta tarea quedó a cargo del lector.

La clase `UTree`, por ejemplo, nos permitió recorrer las hojas del árbol binario abstrayéndonos de la naturaleza no lineal y recursiva de este tipo de estructura de datos.

En los próximos capítulos, estudiaremos algoritmos y estructuras de datos con mayor nivel de complejidad, estructuras no lineales como árboles y grafos, algoritmos recursivos, métodos de ordenamiento altamente eficientes, etcétera.

16.9 Contenido de la página Web de apoyo

El material marcado con asterisco (*) solo está disponible para docentes.

16.9.1 Mapa conceptual

16.9.2 Autoevaluaciones

16.9.3 Videotutorial

16.9.3.1 Algoritmo de Huffman

16.9.4 Presentaciones*