CAPÍTULO

12 Utilidades

En este capítulo se discuten dos usos de las estructuras de datos en el marco de las denominadas *utilidades*: la compresión de datos y la generación de referencias cruzadas. La compresión de datos es una técnica muy importante en computación. Se emplea para reducir el tamaño de los ficheros guardados en disco (aumentando de esta forma su capacidad) y para aumentar la tasa de efectividad en la transmisión de datos mediante módems (al transmitirse menos datos repetidos). En la actualidad, prácticamente todos los módems realizan algún tipo de compresión de datos. Las tablas de referencias cruzadas constituyen una técnica de búsqueda y ordenación que se usa, por ejemplo, para hacer el índice de un libro.

En este capítulo veremos:

- La exposición de un algoritmo de compresión de datos denominado *algoritmo de Huffman*, y una descripción de cómo puede implementarse dicho algoritmo (su implementación completa queda fuera de los objetivos del texto).
- Una implementación de un programa de generación de referencias cruzadas que enumera, de forma ordenada, todos los identificadores que aparecen en un programa Java, indicando los números de línea donde aparece cada uno.

12.1 Compresión de ficheros

Una codificación estándar de C caracteres emplea $\lceil \log C \rceil$ bits.

El conjunto de caracteres ASCII está formado por unos 100 caracteres imprimibles. Para distinguir internamente estos caracteres se necesitan $\lceil \log 100 \rceil = 7$ bits. Siete bits permiten la representación de 128 caracteres, de modo que el conjunto de caracteres ASCII contiene además algunos caracteres no imprimibles. Se añade un octavo bit para poder realizar tests de paridad. Centrémonos, no obstante, en el hecho de que si el cardinal de un conjunto de caracteres es C, entonces se necesitan $\lceil \log C \rceil$ bits para una codificación de longitud fija estándar.

Supongamos que tenemos un fichero que contiene sólo caracteres *a*, *e*, *i*, *s*, *t*, además de espacios en blanco (*sp*) y caracteres de nueva línea (*nl*). Supongamos que la frecuencia de cada carácter es 10, 15, 12, 3, 4, 13 y 1, respectivamente. Como se muestra en la tabla de la Figura 12.1, este fichero ocuparía 174 bits, ya que contiene 58 caracteres y en este marco cada carácter necesitaría sólo 3 bits para ser representado.

Carácter	Código	Frecuencia	Bits totales
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
sp	101	13	39
nl	110	1	3
Total			**174**

Figura 12.1 Un esquema de codificación estándar.

En la vida real, los archivos pueden ser bastante grandes. Muchos ficheros de gran tamaño son salida de algunos programas, y normalmente existe una gran disparidad entre el número de apariciones de los caracteres más frecuentes y los menos frecuentes. Por ejemplo, muchos ficheros de datos de gran tamaño tienen una enorme cantidad de dígitos, blancos y caracteres de nueva línea, sin embargo tienen pocas q o x.

La reducción de la cantidad de bits necesarios para la representación de datos se llama *compresión*. La compresión tiene dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión).

Existen muchas situaciones en las que sería muy deseable reducir el tamaño de un fichero. Por ejemplo, como el espacio en disco es precioso en prácticamente cualquier máquina, disminuir el espacio necesario para almacenar los ficheros aumentaría la capacidad efectiva del disco. Cuando los datos se transmiten a través de las líneas telefónicas mediante un módem, la tasa de efectividad en la transmisión aumenta si puede reducirse la cantidad de datos transmitidos. Reducir la cantidad de bits necesarios para la representación de datos recibe el nombre de *compresión*. Realmente, la compresión se realiza en dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión). Una estrategia sencilla explicada en este capítulo permite ahorrar el 25 por ciento del espacio en ficheros típicos de gran tamaño, y hasta un 50 o 60 por ciento en muchos archivos de datos. Extensiones de esta técnica permiten obtener aún mejores resultados en la compresión.

En un sistema de codificación de longitud variable, los caracteres más frecuentes tienen la representación más corta.

La estrategia general es permitir que la longitud de la codificación cambie de carácter a carácter, haciendo que los caracteres más frecuentes tengan códigos más cortos. Nótese que cuando todos los caracteres aparecen con frecuencias similares, no podemos esperar un gran ahorro.

12.1.1 Códigos sin prefijos

En un *trie binario*, una rama izquierda representa un 0 y una rama derecha representa un 1. El camino a un nodo indica su representación.

El código binario de la Figura 12.1 se puede representar mediante el árbol binario de la Figura 12.2. En este árbol, los caracteres se almacenan sólo en las hojas. Los códigos de cada carácter se encuentran recorriendo el árbol desde la raíz hasta alcanzar dicho carácter, empleando un 0 para indicar la rama izquierda y un 1 para indicar la rama derecha. Por ejemplo, a *s* se llega yendo primero a la izquierda, y luego dos veces a la derecha. En consecuencia, su código es 011. Esta estructura recibe el nombre de *trie*[1] *binario* (pronunciado en inglés «trai»). Si cada carácter

[1] *N. del T.:* Al no existir una traducción habitual de este término hemos optado por conservar su nombre en inglés.

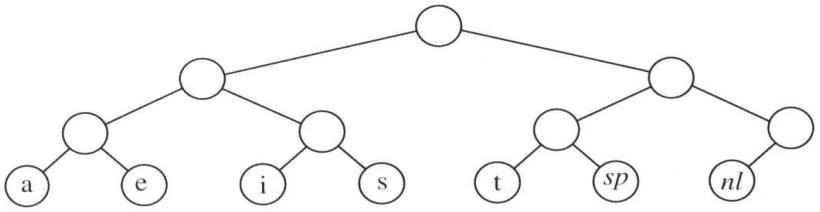


Figura 12.2 Representación mediante un árbol de la codificación original.

c_i se encuentra a profundidad d_i y aparece f_i veces en un fichero entonces el *coste* del código del mismo es igual a $\Sigma d_i f_i$.

Puede obtenerse una codificación mejor que la mostrada en la Figura 12.2 teniendo en cuenta que *nl* es hijo único. Colocándolo en un nivel superior (en el lugar de su padre) obtenemos un nuevo árbol mostrado en la Figura 12.3. Dicho árbol tiene un coste de 173 bits pero sigue estando lejos de ser óptimo.

Nótese que el árbol de la Figura 12.3 es un *árbol completo*, esto es, todos los nodos o son hojas o tienen dos hijos. Una codificación óptima cumple trivialmente esta propiedad; en caso contrario, como ya hemos visto, los nodos que tienen un único hijo pueden subirse un nivel.

En un *árbol completo* todos los nodos o son hojas o tienen dos hijos.

El que los caracteres se sitúen sólo en las hojas es importante, pues así cualquier secuencia de bits se decodifica sin ambigüedad posible. Consideremos por ejemplo la cadena codificada 010011110001011000100011 1. La Figura 12.3 muestra que 0 y 01 no son códigos de ningún carácter, pero 010 es el código de *i*. A continuación sigue 011, que representa una *s*. El código 11 siguiente es un carácter de nueva línea (*nl*). El resto de la cadena es *a*, *sp*, *t*, *i*, *e* y *nl*.

Los códigos de los caracteres pueden tener longitudes diferentes, y ningún código de carácter es un prefijo de cualquier otro código. Esta técnica de codificación se conoce como *código sin prefijos*. Recíprocamente, si permitimos que un carácter esté en un nodo que no sea una hoja, en general no será posible garantizar que la decodificación no sea ambigua.

En un *código sin prefijos* no existe ningún código de carácter que sea prefijo de otro código. Esto se garantiza si los caracteres están sólo en las hojas del árbol. Un código sin prefijos puede decodificarse siempre sin ambigüedad.

Combinando estos hechos, concluimos que nuestro problema básico se reduce a encontrar el árbol completo de coste mínimo (aplicando la definición dada de coste), en el que todos los caracteres se encuentren en las hojas. La Figura 12.4 muestra el árbol óptimo para nuestro alfabeto de muestra. Como puede verse en la Figura 12.5, esta codificación requiere sólo 146 bits. Nótese que siempre existen varias codificaciones óptimas. En concreto, a partir de cada una podemos encontrar otras distintas, intercambiando los hijos de algunos nodos (los que gustemos) del árbol.

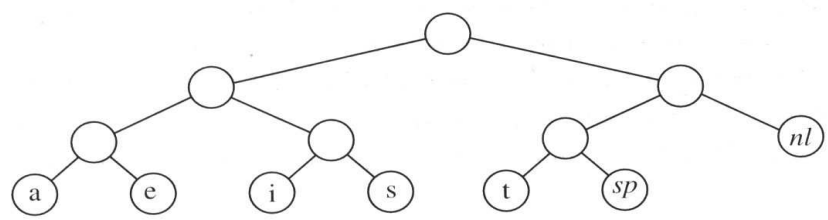


Figura 12.3 Un árbol de codificación un poco mejor.

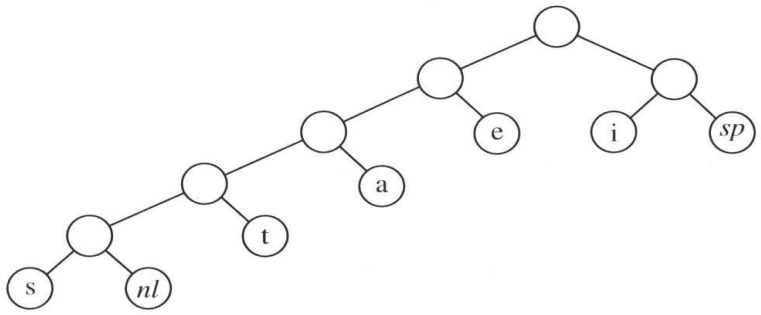


Figura 12.4 Árbol que genera el código óptimo sin prefijos.

12.1.2 Algoritmo de Huffman

El *algoritmo de Huffman* construye un código sin prefijos óptimo. Consiste en la mezcla repetida de dos árboles de peso mínimo.

¿Cómo se construye el árbol óptimo de codificación? El algoritmo que implementa el sistema de codificación se debe a Huffman, quien lo dio en 1952, y se conoce comúnmente como *algoritmo de Huffman*.

A lo largo de esta sección consideramos que el número de caracteres a codificar es C. El algoritmo de Huffman es el siguiente: se manejará un bosque de árboles. El *peso* de un árbol es igual a la suma de las frecuencias de sus hojas. Se seleccionan los dos árboles, T_1 y T_2, de menor peso y se forma un nuevo árbol cuyos subárboles son T_1 y T_2. Este proceso se repite $C - 1$ veces. Al empezar el algoritmo se tienen C árboles con un único nodo (un árbol por cada carácter). Al final del proceso sólo queda un árbol, que es justamente el árbol óptimo de Huffman. El Ejercicio 12.4 pide al lector que demuestre la corrección de este algoritmo.

Carácter	Código	Frecuencia	Bits totales
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
Total			**146**

Figura 12.5 Código sin prefijos óptimo.

Los árboles se mezclan colocándolos como hijo izquierdo y derecho de forma arbitraria.

Un ejemplo ilustrará el funcionamiento del algoritmo. La Figura 12.6 muestra el bosque de partida; el peso de cada árbol se indica con un número de tamaño pequeño al lado de la raíz. Se mezclan los dos árboles de menor peso, por medio de la nueva raíz $T1$, para así generar el bosque de la Figura 12.7. Hemos hecho de forma arbitraria que s sea el hijo izquierdo del nuevo árbol, pero la elección contraria sería perfectamente válida. El peso total del nuevo árbol es la suma de los pesos de los árboles que lo generan, esto es 4.

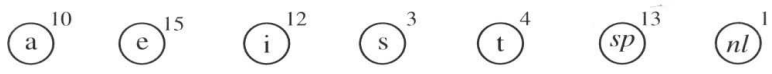


Figura 12.6 Configuración inicial del algoritmo de Huffman.

Figura 12.7 El algoritmo de Huffman tras la primera mezcla.

En este momento tenemos seis árboles, y de nuevo se seleccionan los dos de menor peso: $T1$ y t. Se mezclan en un nuevo árbol, con raíz $T2$ y peso 8. Esto se muestra en la Figura 12.8. El tercer paso mezcla $T2$ y a, creando $T3$, con peso $10+8=18$. La Figura 12.9 muestra el resultado de esta operación.

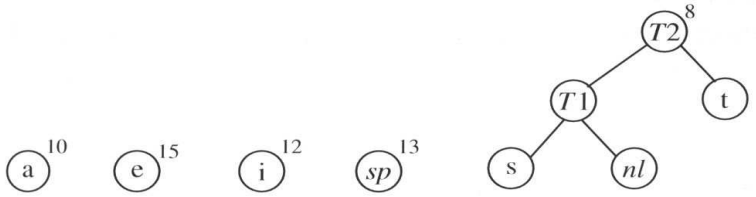


Figura 12.8 El algoritmo de Huffman tras la segunda mezcla.

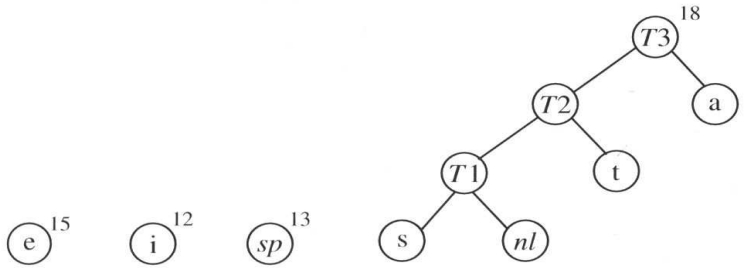


Figura 12.9 El algoritmo de Huffman tras la tercera mezcla.

Después de haber completado la tercera mezcla, los dos árboles de menor peso son los árboles con un único nodo correspondientes a los caracteres i y blanco. La Figura 12.10 muestra cómo dichos árboles se combinan en uno nuevo, con raíz $T4$. El quinto paso consiste en mezclar los árboles con raíz e y $T3$, ya que éstos son ahora los de menor peso. El resultado de este paso se muestra en la Figura 12.11.

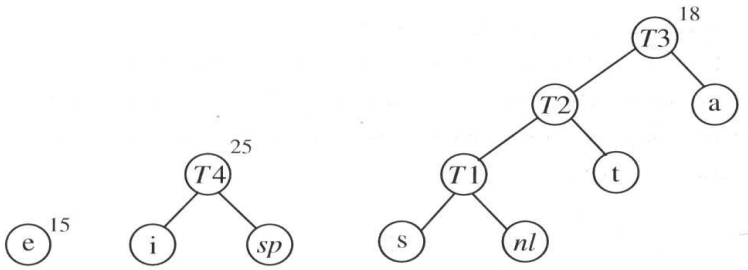


Figura 12.10 El algoritmo de Huffman tras la cuarta mezcla.

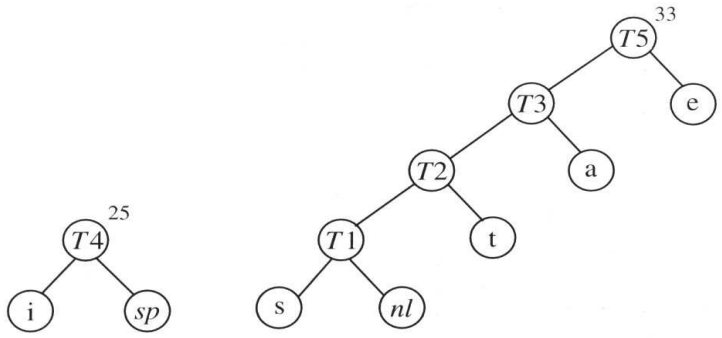


Figura 12.11 El algoritmo de Huffman tras la quinta mezcla.

Por último, se obtiene un árbol óptimo, el mostrado anteriormente en la Figura 12.4, combinando los dos árboles que nos quedan. La Figura 12.12 muestra el árbol óptimo, con raíz $T6$.

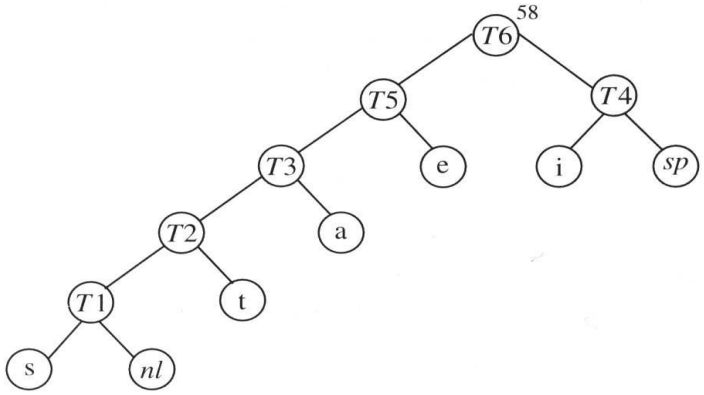


Figura 12.12 El algoritmo de Huffman tras la mezcla final.

12.1.3 La fase de codificación

El código puede representarse en un vector.

Una vez que el árbol ha sido construido, debemos decidir el código de cada carácter. Esto puede hacerse si cada nodo del árbol tiene información acerca de su padre, e indica si es su hijo derecho o izquierdo. Si tenemos C caracteres, entonces el árbol tiene $2C - 1$ nodos. En esta fase podemos emplear un vector con una entrada por cada nodo. Dichas entradas están compuestas por el peso, el padre y la información de si el nodo es hijo derecho o izquierdo.

La tabla de codificación para nuestro árbol de Huffman se muestra en la Figura 12.13. Cada carácter no está almacenado en el vector sino que se emplea como índice de la tabla (así `'a'` indexa la posición 97, que es su código ASCII, `'b'` indexa la posición 98, y así sucesivamente). Por ejemplo, el padre de *nl* es $T1$ y *nl* es su hijo derecho. Podemos escribir el código de *nl* generando recursivamente el código de $T1$ y luego el tipo de hijo que es. Nótese que $T6$ tiene a 0 como padre. Esto nos indica cuándo finalizar la recursión. Como el rango de los códigos ASCII se extiende desde 0 hasta 127, el vector tendrá desde la casilla 0 hasta la 254, para poder representar el hecho de que, además de las hojas, se necesitan, a lo sumo, $C - 1$ símbolos internos.

	Carácter	Peso	Padre	Tipo de hijo
0	a	10	9	1
1	e	15	11	1
2	i	12	10	0
3	s	3	7	0
4	t	4	8	1
5	*sp*	13	10	1
6	*nl*	1	7	1
7	T1	4	8	0
8	T2	8	9	0
9	T3	18	11	0
10	T4	25	12	1
11	T5	33	12	0
12	T6	58	0	

Figura 12.13 Tabla de codificación (los números de la izquierda son los índices del vector).

12.1.4 La fase de decodificación

Más tarde o más temprano, cuando queramos utilizar su información, necesitaremos descomprimir un fichero comprimido. Para hacerlo, debemos incluir cierta información extra en el fichero comprimido. Obviamente, deseamos incluir tan poca información como sea posible. Supongamos que estamos empleando el conjunto de caracteres ASCII.

Una posibilidad es almacenar la frecuencia de cada carácter. Esto requiere 128 enteros. Seguramente será suficiente hacerlo con enteros 24-bit, es decir, enteros 3-byte (probablemente, los enteros 16-bit no basten ya que es posible, si el fichero es grande, que algún carácter aparezca más de 65.535 veces). En total se necesitan 384 bytes para el almacenamiento del fichero.

Para el conjunto de caracteres ASCII podemos almacenar el árbol correspondiente con 255 bytes.

Una alternativa es almacenar la información del tipo de hijo y el padre de cada nodo. Como cada padre es un nodo completamente nuevo, su valor es un número entre 128 y 254. Como consecuencia, necesitamos 7 bits para almacenarlo. De hecho, podemos usar un carácter de 8 bits: sumando 128 al valor representado por los 7 bits menores obtendremos la posición del padre (la raíz puede usar la posición 255 como su padre). Esto deja libre el octavo bit (es decir, el más significativo) para representar el tipo de su hijo. Como consecuencia, se necesitan 255 bytes para almacenar la información auxiliar de la codificación.

12.1.5 Consideraciones prácticas

Un algoritmo eficiente no lee el fichero más de una vez. Existen muchas alternativas.

Antes de iniciar la compresión de datos, se necesita contabilizar la frecuencia de cada carácter. Si el fichero es lo suficientemente pequeño para poder ser almacenado en la memoria principal, podemos leer el fichero volcándolo en un vector, calcular las frecuencias de los caracteres y aplicar el algoritmo. Pero, ¿qué sucede si es demasiado grande?

Es obvio que no deseamos leer el fichero dos veces; la E/S a través de disco es extremadamente lenta. Una posibilidad es dividir el fichero en bloques que puedan ser almacenados en la memoria principal, y comprimir cada uno de ellos por separado. Muchos computadores modernas pueden almacenar muchos megabytes, así que no parece excesivo tener que almacenar repetidas tablas de códigos con 255 bytes.

Una alternativa es asumir que el fichero de entrada es «típico». Por ejemplo, es bien conocida la distribución de los caracteres en los programas implementados en Java. Una ventaja de este método es que no necesitamos almacenar en cada fichero la tabla de códigos. Esto es particularmente atractivo cuando los ficheros a almacenar no son muy grandes. Otras muchas posibilidades han sido exploradas, al igual que otros esquemas de compresión más complejos que funcionan mejor que la codificación de Huffman. *Véase* las referencias bibliográficas para una información más completa.

12.2 Generador de referencias cruzadas

Un *generador de referencias cruzadas* enumera todos los identificadores y las líneas en las que se encuentran. Es una aplicación habitual, similar a la creación del índice de un texto.

En esta sección diseñaremos un programa que examina un código Java y muestra todos los identificadores del mismo, junto a los números de línea en los que aparecen. Además, los identificadores se muestran ordenados alfabéticamente. Esto se conoce como *generación de referencias cruzadas*. Una aplicación en el terreno de la compilación es enumerar, para cada función, los nombres de las funciones que invoca directamente.

Sin embargo, éste es un problema general que aparece en muchos otros contextos. Por ejemplo, generaliza la creación del índice de un libro. Otro uso, la revisión ortográfica, se describe en el Ejercicio 12.16. Un revisor ortográfico detecta las palabras mal escritas en un documento, que se muestran junto a los números de líneas donde aparecen. Esto evita imprimir varias veces la misma palabra incorrecta y nos indica dónde se han producido los errores.

12.2.1 Ideas básicas

Cada identificador y el número de línea donde aparece se almacenarán en un objeto del tipo `IdNodo`. Mantendremos una colección ordenada de `IdNodos`. Cuando el fichero fuente se haya leído, nos desplazaremos a través de la colección, mostrando los identificadores y sus números de líneas correspondientes.

Una posibilidad es emplear una lista enlazada de objetos `IdNodo`. Cuando se lee un identificador, debemos comprobar si ya está en la lista. Si es así, añadimos la línea actual al objeto `IdNodo` referenciado mediante una llamada a `encontrar`. Si

no se encuentra en la lista, se crea un nuevo objeto `IdNodo` con el identificador y el número de línea, y se inserta en la lista enlazada. Después de que la entrada completa ha sido leída, podemos recorrer la lista enlazada y mostrar su contenido. Sin embargo, una lista enlazada ordenada tiene un comportamiento asintótico pobre, por lo que en su lugar emplearemos un árbol binario de búsqueda y haremos un recorrido en orden simétrico para obtener la salida final.

Almacenamos en una cola los números de línea en las que aparece un identificador. Los identificadores se almacenan en un árbol binario de búsqueda.

Cada objeto `IdNodo` almacena un identificador y los números de línea en las que aparece. El identificador se representa mediante un `String`. Los números de línea pueden representarse con una cola de enteros. Esto resulta adecuado ya que cuando mostramos los números de línea, aparecerán en el mismo orden en el que fueron recorridas, esto es, en orden creciente. Así cada `IdNodo` almacena una referencia a una cola a la que se le asigna memoria cuando se declara el `IdNodo`. La implementación se describe en la siguiente subsección.

12.2.2 Implementación en Java

La Figura 12.14 muestra la clase `IdNodo` (que sólo contiene componentes de acceso amistoso). Un `IdNodo` consiste en una cadena de caracteres y una referencia a una cola de enteros. El campo `String` llamado `palabra` se declara en la línea 9; `lineas`, que es la referencia a la cola de enteros, se declara en la lí-

```
1  /**
2   * Objeto básico, que se almacenará en un árbol de búsqueda,
3   * para el generador de referencias cruzadas.
4   * Implementa el interfaz Comparable, implementando
5   * compara y menorQue.
6   */
7  class IdNodo implements Comparable
8  {
9      String palabra; // Un identificador
10     Cola lineas;    // Líneas donde aparece
11
12         // Constructor
13     IdNodo( String laPalabra, int lineaActual )
14     {
15         palabra = new String( laPalabra );
16         lineas = new Cola( );
17         lineas.insertar( new Integer( lineaActual ) );
18     }
19
20         // Métodos de ordenación
21     public boolean menorQue( Comparable lder )
22     {
23         return palabra.compareTo( ( (IdNodo) lder ).palabra ) < 0;
24     }
25
26     public int compara( Comparable lder )
27     {
28         return palabra.compareTo( ( (IdNodo) lder ).palabra );
29     }
30 }
```

Figura 12.14 Clase `IdNodo`.

nea 10. El constructor implementado en las líneas 13 a 18, inicializa un `IdNodo` con un `String` y un número de línea. El componente `String` se inicializa en la forma habitual. `lineas` es inicializado creando una cola en la que se inserta `lineaActual`. El resto de la clase `IdNodo` implementa el interfaz `Comparable`.

```
1  // Interfaz AnalizadorJava: genera referencias cruzadas
2  //
3  // CONSTRUCCIÓN: con un objeto BufferedReader
4  // ********************OPERACIONES PÚBLICAS*********************
5  // void generaReferenciaCruzada( ) --> El nombre lo dice todo .
6  // ************************ERRORES*****************************
7  // Comprobación de errores sobre comentarios y comillas
8
9  import java.io.*;
10 import EstructurasDatos.*;
11 import Excepciones.*;
12 import Soporte.*;
13
14 /**
15  * Clase que genera referencias cruzadas para programas Java.
16  */
17 class AnalizadorJava
18 {
19     public AnalizadorJava( PushbackReader cadenaEntrada )
20     {
21         errores = 0;
22         car = '\0';
23         lineaActual = 1;
24         entrada = cadenaEntrada;
25         actualIdNodo = new IdNodo( "", 1 );
26     }
27
28     public void generaReferenciaCruzada( )
29       { /* Figura 12.19 */ }
30
31     private IdNodo actualIdNodo;
32
33     private static int esIdChar( ch )
34       { /* Figura 12.16 */ }
35     private String tomaString( )
36       { /* Figura 12.17 */ }
37     private int tomaSiguienteId( )
38       { /* Figura 12.18 */ }
39
40        // Los siguientes son los mismos que los de la Figura 11.2
41     private PushbackReader entrada; // La cadena entrada
42     private char car;               // Carácter actual
43     private int lineaActual;        // Línea actual
44     private int errores;            // Número de errores encontrados
45     private boolean siguienteCar( )
46     private void devolverCar( )
47     private void saltarComentario( int comienzo )
48     private void saltarCadena( char tipoCadena )
49     private void procesarBarra( )
50 }
```

Figura 12.15 Esquema de la clase para el generador de referencias cruzadas.

La clase de la Figura 12.15 es similar a la mostrada en la Figura 11.2, que era parte de un programa de equilibrado de símbolos (el código en Internet es una versión en la que se mezclan estas dos clases en una sola). Muchos de los métodos ya han sido explicados, por lo que no repetimos sus implementaciones. No se emplea herencia. La posibilidad de crear una clase abstracta y dos clases derivadas se deja como ejercicio al lector (Ejercicio 12.13).

Las rutinas de compilación son sencillas, aunque, como siempre, requieren bastante esfuerzo.

Las nuevas rutinas de compilación tratan el reconocimiento de identificadores. En la línea 31, se declara un objeto `IdNodo`. Éste almacenará el identificador que está siendo procesado. Esto es debido a que como manejamos un árbol de objetos `IdNodo`, necesitamos un `IdNodo` como parámetro de las rutinas `buscar` e `insertar`.

Un identificador de Java es una secuencia de caracteres alfanuméricos y guiones, con la restricción de que el primer carácter no puede ser un dígito. La rutina de la Figura 12.16 comprueba si el carácter dado puede formar parte de un identificador. El método `tomaString` de la Figura 12.17 asume que el primer carácter de un identificador ya ha sido leído, y almacenado en el atributo `car`. Esta rutina lee repetidamente caracteres hasta que se encuentra uno que no puede formar parte de los identificadores. En ese momento, volvemos al carácter anterior y devolvemos un `String`[2].

El método `tomaSiguienteId`, mostrado en la Figura 12.18, es similar a la rutina de la Figura 11.7. La diferencia es que aquí, en la línea 15, si se detecta el primer carácter de un identificador, rellenamos `actualIdNodo.palabra` con dicho identificador.

Si el elemento es encontrado, lo añadimos a la cola de referencias.

Una vez escritas todas las rutinas auxiliares, podemos implementar el método principal, `generaReferenciaCruzada`, que se muestra en la Figura 12.19. En las líneas 6 y 7 se define un árbol binario de búsqueda de objetos `IdNodo`. En las líneas desde la 10 hasta la 28 se procesa el fichero fuente. En las líneas 14 y 15, se ejecuta `buscar` para cada identificador, con el fin de saber si ya ha sido encontrado antes. Si `buscar` tiene éxito, en las líneas 16 y 17 insertamos la línea actual en la cola correspondiente. En caso contrario, se trata de un nuevo identificador, que añadimos en el árbol de búsqueda, en las líneas 21 a 26.

Si un elemento no es encontrado, creamos un nuevo `IdNodo`, inicializamos la cola con la línea actual, y lo insertamos en el árbol.

Recuérdese que el método `insertar` necesita un objeto `IdNodo`. Por tanto, llamando al constructor de la Figura 12.14 creamos un nuevo nodo, sin nombre. Colocaremos en el árbol una referencia a dicho `IdNodo`. Esto significa que en el `IdNodo` insertado en el árbol, su campo `lineas` referencia a la cola.

```
1  /**
2   * Devuelve true si car puede formar parte de un identificador Java
3   */
4  private boolean esIdCar( char car )
5  {
6      return car == '_' || Character.isUpperCase( car )
7                        || Character.isLowerCase( car )
8                        || Character.isDigit( car );
9  }
```

Figura 12.16 Rutina para comprobar si un carácter puede formar parte de un identificador.

[2] Java 1.1 tiene predefinidos métodos para comprobar si un carácter puede formar parte de un identificador. Para más detalles, *véase* el Apéndice C.1.1.

```
1  /**
2   * Devuelve un identificador de la entrada
3   * El primer carácter ya ha sido leído en car
4   */
5  private String tomaString( )
6  {
7      String tmpString = "";
8
9      for( tmpString += car; siguienteCar( ); tmpString += car )
10         if( !esIdCar( car ) )
11         {
12             devolverCar( );
13             break;
14         }
15
16     return tmpString;
17 }
```

Figura 12.17 Rutina que devuelve un `String` leído de la entrada.

```
1  /**
2   * Devuelve el siguiente identificador, saltándose los comentarios,
3   * las constantes String y las constantes de tipo carácter.
4   * Coloca el identificador en actualIdNodo.palabra y devuelve false
5   * cuando se alcanza el final de la entrada.
6   */
7  private boolean tomaSiguienteId( )
8  {
9      while( siguienteCar( ) )
10     {
11         if( car == '/' )
12             procesarBarra( );
13         else if( car == '\'' || car == '"' )
14             saltarCadena( car );
15         else if( !Character.isDigit( car ) &&
16                  esIdCar( car ) )
17         {
18             actualIdNodo.palabra = tomaString( );
19             return true;
20         }
21     }
22     return false; // Final de fichero
23 }
```

Figura 12.18 Rutina para asignar a `actualIdNodo.palabra` el siguiente identificador.

La salida se obtiene recorrriendo en orden simétrico el árbol, empleando una clase iterador.

Una vez que hemos construido el árbol de búsqueda, sólo tenemos que recorrerlo en orden simétrico. Para hacerlo, usamos la clase `OrdenSim` descrita en la Sección 17.4.2. El iterador visita los nodos del árbol binario en el orden adecuado. Se emplea de forma análoga a `ListaIter`. El iterador se declara en la línea 33, mientras que en la línea 34 se emplean los métodos estándar de iteración de la clase `OrdenSim`. En la línea 36, `esteNodo` es una referencia al objeto `IdNodo` almacenado en el nodo actual del árbol de búsqueda. En las líneas 39 y 40, se imprime la palabra y el primer número de línea, a la vez que éste último se saca de la cola (tenemos garantizado que la cola no está vacía). Mientras que la cola no esté vacía,

```
1  /**
2   * Muestra las referencias cruzadas.
3   */
4  public void generaReferenciaCruzada( )
5  {
6      ArbolBinarioBusqueda losIdentificadores =
7                                      new ArbolBinarioBusqueda( );
8
9          // Inserta los identificadores en el árbol de búsqueda
10     while( tomaSiguienteId( ) )
11     {
12         try
13         {
14             IdNodo esteNodo = (IdNodo)
15                     losIdentificadores.buscar( actualIdNodo );
16             esteNodo.lineas.insertar(
17                     new Integer( lineaActual ) );
18         }
19         catch( ElementoNoEncontrado e )
20         {
21             try
22             {
23                 losIdentificadores.insertar( new IdNodo
24                      ( actualIdNodo.palabra, lineaActual ) );
25             }
26             catch( ElementoDuplicado ex ) { } // No puede ocurrir
27         }
28     }
29         // Iteración a lo largo del árbol y salida
30         // de los identificadores y sus números de línea.
31     try
32     {
33         OrdenSimBus itr = new OrdenSimBus( losIdentificadores );
34         for( itr.primero( ); itr.esValido( ); itr.avanzar( ) )
35         {
36            IdNodo esteNodo = (IdNodo) itr.recuperar( );
37
38            // Muestra el identificador y la primera línea donde aparece
39            System.out.print( esteNodo.palabra + ": " +
40                    (Integer) esteNodo.lineas.quitarPrimero( ) );
41
42                // Imprime el resto de líneas donde aparece
43            while( !esteNodo.lineas.esVacia( ) )
44                System.out.print( ", " + (Integer)
45                                esteNodo.lineas.quitarPrimero( ) );
46            System.out.println( );
47        }
48    }
49    catch( ElementoNoEncontrado e ) { } // Árbol vacío
50    catch( Desbordamiento e ) { }       // No puede suceder
51 }
```

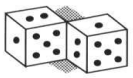
Figura 12.19 Algoritmo principal para generar referencias cruzadas[3].

imprimimos reiteradamente sus elementos gracias al bucle de las líneas 43 a 45. La instrucción de la línea 46 imprime una línea en blanco. No implementamos un programa `main`, ya que es prácticamente idéntico al de la Figura 11.10.

[3] *N. del T.:* La clase `OrdenSimBus` que aparece en la línea 33, es similar a la del Capítulo 17 `OrdenSim`. Sólo se diferencia de ella en que una árboles binarios de búsqueda, en lugar de árboles binarios no ordenados.

Resumen

En este capítulo hemos presentado las implementaciones de dos importantes aplicaciones. La compresión de ficheros es una destacada técnica que permite aumentar tanto la capacidad efectiva del disco como la velocidad efectiva del módem. Éste es un campo de investigación muy activo. La técnica sencilla presentada aquí, conocida como algoritmo de Huffman, consigue, habitualmente, una compresión del 25 por ciento de los ficheros de texto. Otros algoritmos y extensiones del algoritmo de Huffman obtienen todavía mejores resultados. El problema de las referencias cruzadas es un problema general, que tiene numerosas aplicaciones.



Elementos del juego

algoritmo de Huffman Algoritmo que construye un código sin prefijos óptimo. Se basa en la mezcla repetida de dos árboles de peso mínimo.

árbol completo Árbol cuyos nodos o bien son hojas o bien tienen dos hijos.

código sin prefijos Codificación en la que ningún código de carácter es un prefijo de otro código de carácter. En un trie, esto está garantizado cuando los caracteres se encuentran sólo en las hojas. Un código sin prefijos puede decodificarse sin ambigüedad ninguna.

compresión Es el hecho de reducir la cantidad de bits necesarios para la representación de datos. Se compone de dos fases: la fase de codificación (compresión) y la fase de decodificación (descompresión).

generador de referencias cruzadas Programa que enumera los identificadores de un programa y los números de línea en las que aparecen. Es una aplicación muy común, similar a la creación de un índice.

trie binario Estructura de datos en la que una rama izquierda representa un 0 y una rama derecha representa un 1. El camino a cada nodo indica su codificación.

Errores comunes

1. Cuando se implementa la compresión de datos no debe leerse el fichero de entrada más de una vez.
2. Es un error habitual emplear demasiada memoria para almacenar la tabla de la compresión. Esto limita el porcentaje de compresión que puede conseguirse en el proceso.

En Internet

El generador de referencias cruzadas está disponible en el directorio **Chapter12**. El nombre del fichero es:

JavaAnalyzer.java Es la versión inglesa de `AnalizadorJava`. El programa que incluye la rutina para el equilibrado de símbolos del Capítulo 11 se encuentra en el directorio **Part3**.

Ejercicios

Cuestiones breves

12.1. Muestre el árbol de Huffman que resulta de la siguiente distribución de símbolos de puntuación y dígitos: dos puntos (100), espacio (605), nueva línea (100), coma (705), 0 (431), 1 (242), 2 (176), 3 (59), 4 (185), 5 (250), 6 (174), 7 (199), 8 (205) y 9 (217).

12.2. Muchos sistemas tienen ya implementado un programa de compresión. Comprima distintos tipos de ficheros para determinar la tasa media de compresión de su sistema. ¿Cuál debe ser el tamaño de los ficheros para que la compresión resulte rentable?

12.3. ¿Qué sucede si se emplea un fichero comprimido con el algoritmo de Huffman para transmitir datos a través de la línea telefónica y se pierde un bit accidentalmente? ¿Qué se podría hacer al respecto?

Problemas teóricos

12.4. Pruebe la corrección del algoritmo de Huffman, siguiendo los siguientes pasos:

a) Muestre que no existe ningún nodo con un único hijo.
b) Muestre que los dos caracteres con menor frecuencia deben ser los dos nodos más profundos del árbol.
c) Muestre que los caracteres de dos nodos cualesquiera situados a la misma profundidad pueden intercambiarse sin afectar a la optimalidad.
d) Emplee inducción: según vamos mezclando árboles, considere como nuevo conjunto de caracteres uno abstracto cuyos elementos están asociados a las raíces de los árboles generados.

12.5. ¿Bajo qué condiciones el árbol de Huffman de caracteres ASCII genera un código de dos bits para un determinado carácter?, ¿y un código 20-bit?

12.6. Muestre que si los símbolos nos vienen dados ordenados por frecuencia, el algoritmo de Huffman puede implementarse en tiempo lineal.

Problemas prácticos

12.7. En el generador de referencias cruzadas use una lista ordenada en lugar de un árbol binario de búsqueda. ¿Cómo afecta este cambio a la eficiencia?

12.8. La desventaja de emplear una cola para almacenar los números de línea del generador de referencias cruzadas, es que la cola se vacía durante el proceso. Esto podría representar un problema en algunos casos. En lugar de guardar los números en una cola, use una lista enlazada. Implemente las siguientes estrategias y compare su eficiencia:

a) Reemplace la operación de `insertar` por una inserción al final de la lista. Para hacerlo debe recorrer dicha lista.
b) Guarde la lista enlazada y el objeto `ListaIter` que apunta al último elemento de la lista. La inserción al final de la lista enlazada debe ser más eficiente que en el apartado (*a*).

12.9. Combine los ejercicios 12.7 y 12.8, de modo que el generador de referencias cruzadas emplee una lista enlazada ordenada para almacenar la información de las palabras, la cual consiste en un `String`, una lista de números de línea y un `ListaIter` que apunta al final de la lista.

12.10. Modifique `IdNodo`, en el generador de referencias cruzadas, añadiendo un método `toString` que muestre el nombre y los números de línea de `IdNodo`. Simplifique el método `generaReferenciaCruzada` en la forma adecuada. Este empleo de `toString` es engañoso, ya que normalmente este método no altera el estado del objeto sobre el que actúa. Sin embargo, aquí vacía la cola de números de línea del objeto. ¿Qué puede hacerse para solucionar esta incongruencia?

12.11. Si una palabra aparece dos veces en la misma línea, el generador de referencias guardará dicho número de línea dos veces. Modifique el algoritmo, de modo que los duplicados sólo se enumeren una vez.

12.12. Modifique el generador de referencias cruzadas de modo que si una palabra aparece en líneas consecutivas, se especifique un rango. Por ejemplo,

```
if: 2, 4, 6-9, 11
```

Prácticas de programación

12.13. Descompoga el `AnalizadorJava` en tres clases: una clase base abstracta que recoja la funcionalidad común, y dos clases derivadas `TestBalance` y `GeneraReferenciaCruzada`.

12.14. Añada una componente GUI al generador de referencias cruzadas que incluya una opción para consultas avanzadas, como mostrar nombres específicos almacenados en el árbol binario de búsqueda (en lugar de mostrar todo su contenido).

12.15. Genere el índice de un libro. El fichero de entrada consiste en un conjunto de entradas del índice. Cada línea está formada por la cadena `IX:`, seguida de una entrada del índice escrita entre llaves y de un número de página también entre llaves. Cada símbolo `!` en una entrada de índice representa un nivel inferior. Los símbolos `|(` representan el inicio de un rango, y `|)` representa el final del rango. En algunas ocasiones, este rango puede ser la misma página. En ese caso, sólo debe indicar un único número de página. En otro caso, no debe colapsar o expandir los rangos indicados. Como ejemplo, la Figura 12.20 muestra un posible fichero de entrada y la Figura 12.21 muestra la salida correspondiente.

```
IX: {Series|(}                          {2}
IX: {Series!geométricas|(}              {4}
IX: {Constante de Euler}                {4}
IX: {Series!geométricas|)}              {4}
IX: {Series!aritméticas|(}              {4}
IX: {Series!aritméticas|)}              {5}
IX: {Series!armónicas|(}                {5}
IX: {Constante de Euler}                {5}
IX: {Series!armónicas|)}                {5}
IX: {Series|)}                          {5}
```

Figura 12.20 Entrada sencilla del Ejercicio 12.15.

```
Constante de Euler: 4, 5
Series: 2-5
  aritméticas: 4-5
  geométricas: 4
  armónicas: 5
```

Figura 12.21 Salida sencilla del Ejercicio 12.15.

12.16. Implemente un revisor ortográfico empleando una tabla hash. Asuma que el diccionario está divido en dos fuentes: un gran diccionario ya creado y un segundo fichero. Debe mostrar todas las palabras mal escritas y los números de línea en las que aparecen (observe que llevar la cuenta de estas palabras y los números de línea equivale a la generación de referencias cruzadas). Además, para cada palabra incorrecta debe enumerar todas las palabras en el diccionario que se obtienen mediante la aplicación de alguna de las siguientes reglas:

a) Añadir un carácter.
b) Eliminar un carácter.
c) Cambiar caracteres adyacentes.

12.17. Implemente un applet interactivo que muestre como se construye el árbol de Huffman.

12.18. Implemente un programa completo para la compresión de ficheros. No se olvide de incluir un algoritmo para la decompresión. Para ello, implemente una clase `BitStream`. Observe que si el número total de bits en el código de salida no es un múltiplo exacto del número de bits en un byte, pueden aparecer bits perdidos en el resultado comprimido. Debe definirse un mecanismo para ignorar dichos bits basura.

Bibliografía

El artículo original sobre el algoritmo de Huffman es [3]. Algunas variaciones se discuten en [2] y [4]. Otro esquema de compresión muy conocido es la *codificación Ziv-Lempel*, descrito en [7] y [6]. Trabaja generando series de códigos de longitud fija. Normalmente, deberíamos generar 4.096 códigos 12-bits para representar las subcadenas más frecuentes del fichero. [1] y [5] son buenos resúmenes de los esquemas de compresión más comunes.

1. T. Bell, I. H. Witten, y J. G. Cleary, «Modelling for Text Compression», *ACM Computing Surveys* **21** (1989) 557-591.
2. R. G. Gallager, «Variations on a Theme by Huffman», *IEEE Transactions on Information Theory* **IT-24** (1978), 668-674.
3. D. A. Huffman, «A Model for the Construction of Minimum Redundancy Codes», *Proceedings of the IRE* **40** (1952), 1098-1101.
4. D. E. Knuth, «Dynamic Huffman Coding», *Journal of Algorithms* **6** (1985), 163-180.
5. D. A. Lelewer y D. S. Hirschberg, «Data Compression», *ACM Computing Surveys* **19** (1987), 261-296.
6. T. A. Welch, «A Technique for High-Performance Data Compression», *Computer* **17** (1984), 8-19.
7. J. Ziv y A. Lempel, «Compression of Individual Sequences via Variable-Rate Coding», *IEEE Transactions on Information Theory* **IT-24** (1978), 530-536.