# Ficha 13

# Compresión de Datos

**1.) Consideraciones iniciales.**

Luego de haber visto las ideas detrás del *Algoritmo de Huffman* para sustituir un sistema de codificación binaria de longitud fija por otro de longitud variable y luego de haber visto detalles de gestión de bits, podemos intentar el desarrollo de un prototipo de programa que permita la compresión efectiva de un archivo. Ese prototipo se brinda en el modelo *Compresion* que acompaña a estas notas.

El modelo brinda una clase *Compressor*, la cual provee los métodos *compress()* y *decompress()* que llevan a cabo las tareas indicadas sobre un archivo cuyo nombre toman como parámetro. El resto de las clases del modelo se presentaron ya, en ocasión del análisis del *Algoritmo de Huffman*.

La primera (y quizás única) gran cuestión detrás del desarrollo del compresor, es que el programa debería ser capaz de procesar archivos de cualquier contenido: no debería comprimir sólo archivos de texto, sino archivos de cualquier origen: el archivo podría contener registros de datos (o sea, una mezcla de datos numéricos, de texto y boolean) o podría ser un archivo que contenga una imagen digitalizada en algún formato (*bmp*, *jpg*, etc.) o podría ser una planilla de cálculo o un archivo *exe*. Y con mayor o menor eficiencia el programa debería poder efectuar sobre él el proceso de compresión.

Se podría pensar que esto realmente es un gran problema: al fin y al cabo, el *Algoritmo de Huffman* está pensado para proveer codificación binaria de longitud variable en reemplazo de *ASCII* u otra técnica de longitud fija, pero partiendo de la base de saber la frecuencia de aparición de cada *caracter* en el mensaje o archivo a comprimir. Y si se habla de caracteres, se está presuponiendo un archivo de texto...

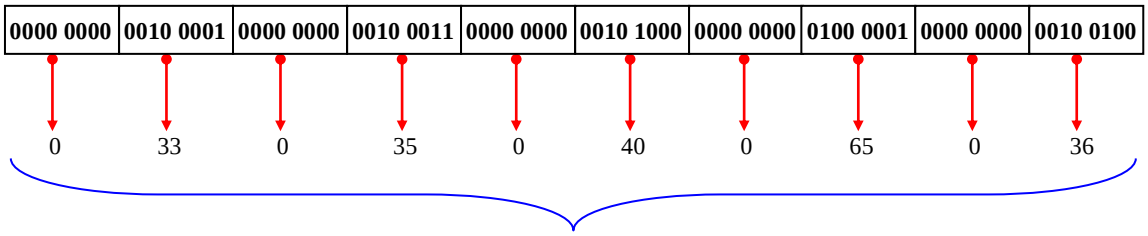
Sin embargo, el proceso puede generalizarse tranquila y transparentemente, simplemente pensando en *bytes* en lugar de pensar en *caracteres*... Todo archivo finalmente es una secuencia de bytes almacenada en disco, sin importar el sentido original que el creador del archivo dió a esos bytes. Si pensamos que en última instancia cada caracter se representa con un byte, y que un caracter no es otra cosa que un valor numérico entre 0 y 255, entonces no hay ningún problema en abrir un archivo *cualquiera*, leerlo byte por byte, y contar cuantas veces aparece ese *byte* en el archivo (tomando a cada byte que venga como si originalmente se hubiera grabado un caracter).

Así, supongamos que alguien ha creado un archivo de datos y ha grabado en él las edades (supongamos valores *short*) de los alumnos de un curso. Por lo tanto, lo que se grabó fue una secuencia de pares de bytes (formato *short*). Si en el archivo se grabaron las siguientes edades {33, 35, 40, 65, 36} entonces quedó así:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **0000 0000** | **0010 0001** | **0000 0000** | **0010 0011** | **0000 0000** | **0010 1000** | **0000 0000** | **0100 0001** | **0000 0000** | **0010 0100** |

Si ahora se quiere aplicar sobre ese archivo un proceso de compresión basado en Huffman, no hay problema: se debe contar cuántas veces aparece cada *byte*. Podemos ver que si miramos los bytes (y tomamos sus valores en base 10) tenemos que el valor 0 aparece cinco veces, el 33 aparece una vez, el 35 aparece dos veces, el 40 aparece una vez, el 65 aparece una vez y el 36 una vez. Podemos (si

queremos) considerar que el 0, el 33, el 35, el 40, el 65 y el 36 son los caracteres 0, 33, 35, 40, 65 y 36 respectivamente de la tabla ASCII (o la que sea que se esté usando) pero *no es estrictamente necesario hacerlo*. Sólo necesitamos la distribución de frecuencias de cada byte tomado por su *valor numérico*:




| | | |
|---|---|---|
| byte: 0 | ASCII: NUL | frecuencia: 5 |
| byte: 33 | ASCII: ! | frecuencia: 1 |
| byte: 35 | ASCII: # | frecuencia: 1 |
| byte: 40 | ASCII: ( | frecuencia: 1 |
| byte: 65 | ASCII: A | frecuencia: 1 |
| byte: 36 | ASCII: $ | frecuencia: 1 |

## 2.) Estructura básica de un compresor basado en el Algoritmo de Huffman.

La clase *Compressor* presentada en el modelo que acompaña a esta ficha, solo contiene un atributo *n* que almacena la cantidad de símbolos o bytes de la tabla de entrada. En definitiva, *n* es la cantidad de símbolos o bytes diferentes que había en el archivo, y que serán las hojas del Árbol de Huffman):

```
public class Compressor
{
   // cantidad de simbolos / bytes diferentes del archivo de entrada.
   private int n;

   /**
    *  Crea un compresor.
    */
   public Compressor()
   {
     n = 0;
   }

    // resto de la clase aquí…

}
```

El único constructor de la clase simplemente asigna el valor 0 en el atributo n, y el resto del trabajo se especifica en los métodos *compress()* y *decompress()*. La creación del árbol de Huffman se hace en particular cuando se comienza con el proceso de compresión de un archivo, y por eso no es el constructor quien hace la tarea: suponemos que el mismo compresor puede comprimir distintos archivos (con distintos árboles de Huffman) en distintos momentos.

La clase provee entonces dos métodos esenciales: *compress()* y *decompress()*. El primero de ellos realiza los siguentes pasos:

**a.)** Toma el nombre de un archivo como parámetro.
**b.)** Abre el archivo y determina la frecuencia de aparición de cada byte en ese archivo.
**c.)** Crea un árbol de Huffman en base a esa tabla de frecuencias obtenida.
**d.)** Produce otro archivo, comprimido en base a la codificación Huffman derivada del árbol creado.

El proceso de conteo de frecuencias de cada byte, se hace con el ya conocido algoritmo basado en un arreglo de contadores y acceso directo. Al final se recorre el arreglo para saber cuántos bytes realmente aparecieron en el archivo, y saber así el valor final del atributo *cantSignos*:

```
 // determinar la frecuencia de aparición por byte en el archivo fuente...
int c[] = new int[256];
while(fuente.getFilePointer() < fuente.length())
{
   int b = fuente.readUnsignedByte();
   c[b]++;
}

// determinar cuantos simbolos / bytes diferentes habia...
n = 0;
for(int i = 0; i < 256; i++)
{
   if( c[i] != 0 ) { n++; }
}
```

Una vez creada la tabla de distribución de frecuencias, la creación del árbol se lleva adelante de la siguiente forma:

```
// inicializar un Arbol de Huffman con espacio para n simbolos...
HuffmanTree ht = new HuffmanTree(n);
int ind = 0;
for(int i = 0;  i < 256;  i++)
{
   if(c[i] != 0)
   {
      ht.setNode((byte)i, c[i], ind);
      ind++;
   }
}

// crear el arbol y obtener el codigo Huffman de cada simbolo / byte...
ht.encode();
```

El método *setNode()* de la clase *HuffmanTree* asigna los caracteres (bytes) en el arreglo que representa al árbol, y junto con esos caracteres almacena la frecuencia de cada uno. Con la frecuencia y el índice que cada caracter tiene en el arreglo que representa al árbol, va creando también la cola de prioridades para poder recuperar rápidamente los dos caracteres de menos frecuencia. El método *encode()* de la clase *HuffmanTree* es el que finalmente crea el árbol completo, tomando los dos elementos de menor frecuencia, asignando a ellos un nodo unión, y repitiendo el proceso hasta llegar a la raiz. Al terminar de

armarlo, obtiene la codificación Huffman de cada byte recorriendo hacia arriba el árbol y almacenando esos códigos en un *ArrayList* (contenido en la clase *HuffmanTree*).

Un problema no menor que debe estudiarse es el siguiente: una vez comprimido el archivo, en algún momento se querrá lanzar el proceso de descompresión. Pero el descompresor deberá conocer el árbol de Huffman *exacto* que se usó para comprimir: no bastaría con que el descompresor tenga la tabla de frecuencias, pues sabemos que el árbol de Huffman no es necesariamente único para una tabla de frecuencias dada. Por lo tanto, el árbol usado en el proceso de compresión *deberá almacenarse en el archivo comprimido*, al inicio del mismo a modo de información de cabecera o metadatos. Esto implica que además de los propios datos comprimidos, el archivo tendrá información adicional sin la cual el descompresor no podría cumplir su trabajo. Pero esa información adicional agrega bytes al archivo, y podría darse entonces que al finalizar el proceso no se gane mucho en la compresión, o *incluso que se pierda*: si el archivo original es pequeño podría ocurrir que el archivo comprimido sea más grande que el original... Obviamente, en casos así la compresión no tiene sentido y debe usarse simplemente el archivo original.

El método *compress()* almacena de la siguiente manera el árbol (y otra información valiosa, como el tamaño y el nombre del archivo original) en el archivo comprimido:

```
// ...guardar el nombre y la extension del archivo original...
comprimido.writeUTF(fileName);

// ...guardar la longitud en bytes del archivo original...
long ta = fuente.length();
comprimido.writeLong(ta);

// ...guardar la cantidad de simbolos (hojas) del arbol...
comprimido.writeInt(n);

// ...guardar la tabla de bytes / simbolos tal como esta en el arbol...
for(int i = 0; i < n; i++)
{
  byte simbolo = ht.getSymbol(i);
  comprimido.writeByte(simbolo);
}

// ...guardar ahora el arreglo que representa al arbol...
HuffmanTreeNode a[] = ht.toArray();
for(int i = 0; i < a.length; i++)
{
  // ... y por cada nodo, guardar todos sus datos...
  comprimido.writeInt(a[i].getFrequency());
  comprimido.writeInt(a[i].getFather());
  comprimido.writeBoolean(a[i].isLeft());
  comprimido.writeInt(a[i].getLeft());
  comprimido.writeInt(a[i].getRight());
}
```

Note que posiblemente no sea necesario almacenar *todos* estos valores en el archivo: por ejemplo, al guardar el arreglo que representa al árbol no seria necesario guardar el valor del índice de los hijos

izquierdo y derecho de cada hoja, pues sabemos que esos índices son –1. Simplemente, para las primeras *n* vueltas del ciclo no harían falta los dos últimos *writeInt()* que se ven en el ciclo... Este y otros detalles que permiten afinar la implementación, quedarán como tarea del alumno.

Finalmente, luego de todo lo anterior, el método *compress()* lanza el proceso de compresión propiamente dicho. Para eso, se posiciona el file pointer al inicio del archivo original (había sido abierto y leido hasta el final para contar la frecuencia de cada byte), y se leen uno por uno, nuevamente, todos los bytes. Por cada byte, se recupera del árbol de Huffman su respectivo código binario Huffman. Al mismo tiempo, se cuenta con una variable *salida* con valor inicial 0x0000 ( o sea, 0000 0000 0000 0000 ). Esa variable se declara de tipo *int*, aunque en realidad se usará sólo su último byte (el menos significativo): esto es así para evitar problemas de desborde, como se vio en la Ficha de gestión de bits. Usando operadores a nivel de bits de Java, cambiamos los bits que correspondan en la variable *salida* para que esta almacene la secuencia de Huffman del byte leido desde el archivo, llevando la cuenta de cuántos bits se cambiaron ya. Cuando esa cuenta llega a 8, el byte de salida se graba en el archivo comprimido y se vuelve a cero el valor de la variable *salida* para recomenzar el proceso. La secuencia de operaciones es la siguiente:

```
// comenzar el proceso de compresion (por fin...)
int mascara = 0x00000080;  // el valor 0000 0000 1000 0000
int salida  = 0x00000000;  // el valor 0000 0000 0000 0000
int bit = 0;           // en que bit vamos?

// procesar el archivo de entrada byte a byte, desde el inicio...
fuente.seek(0);
while(fuente.getFilePointer() < fuente.length())
{
   // ...leer un byte desde el archivo fuente...
   int b = fuente.readUnsignedByte();

   // ... pedir el codigo Huffman del byte / simbolo leido...
   ArrayList<Byte> hc = ht.getCodeBySymbol((byte)b);

   // ...analizar bit a bit el codigo de Huffman del simbolo...
   for(int i = 0; i < hc.size(); i++)
   {
      if(hc.get(i) == 1)
      {
         // ...si era 1, bajarlo al byte de salida
         salida = salida | mascara;
      }

      // ...correr el uno a la derecha en la mascara...
      mascara = mascara >>> 1;

      // ...contar el bit procesado...
      bit++;

      // ...analizar si es hora de grabar el byte de salida...
      if(bit == 8)
      {
```

```
            // ...se llenó... grabar el byte menos significativo..
            comprimido.writeByte((byte)salida);

            // ... resetear variables de trabajo...
            bit = 0;
            mascara = 0x00000080;
            salida  = 0x00000000;
        }
    }
}

// ...controlar si el último byte de salida quedo incompleto...
if (bit != 0)
{
    // ...grabar el byte menos significativo de la variable de salida...
    comprimido.writeByte((byte)salida);
}

// cerrar archivos y terminar el proceso
comprimido.close();
fuente.close();
```

### 3.) El proceso de descompresión.

El método *decompress()* realiza el proceso inverso:

- **a.)** Toma el nombre de un archivo compirmido como parámetro.
- **b.)** Abre el archivo y recupera el nombre y el tamaño del archivo original.
- **c.)** También recupera los datos necesarios para rearmar el árbol, y arma el árbol nuevamente.
- **d.)** Produce otro archivo, descomprimiendo el que tomó en base a la codificación Huffman derivada del árbol recuperado.

La recuperación de los datos del archivo original más la creación del árbol a partir de los datos recuperados desde el archivo comprimido es directa:

```
// ...empezar por la longitud del archivo original...
long ta = comprimido.readLong();

// ...cantidad de simbolos de la tabla (o sea, la cantidad de hojas)...
n = comprimido.readInt();

// ...crear de nuevo el Arbol de Huffman en memoria...
HuffmanTree ht = new HuffmanTree(n);

// ...recuperar los simbolos originales...
for(int i = 0; i < n; i++)
{
    byte simbolo = comprimido.readByte();
    ht.setSymbol(simbolo, i);
}
```

```
// ...leer ahora los datos del arbol y reasignarlos...
int t = n * 2 - 1;  // ...cantidad total de nodos del arbol...
for(int i = 0; i < t; i++)
{
  // ...por cada nodo, recuperar sus datos y volver a armar el arbol...
  int f  = comprimido.readInt();          // frecuencia
  int padre = comprimido.readInt();       // padre
  boolean left = comprimido.readBoolean(); // es izquierdo?
  int hi = comprimido.readInt();          // hijo izquierdo
  int hd = comprimido.readInt();          // hijo derecho
  HuffmanTreeNode nh = new HuffmanTreeNode(f, padre, left, hi, hd);
  ht.setNode(nh, i);
}

// ...volver a crear los codigos de Huffman de cada simbolo / byte...
ht.makeCodes();
```

Y luego se lanza el proceso de descompresión propiamente dicho. Se lee cada byte del archivo comprimido, y se analizan uno por unos sus bits (usando máscaras y operadores a nivel de bits). Cada vez que se detecta un 1 en el byte leido, se baja en el árbol por una rama derecha, o se baja por una izquierda si el bit analizado era un cero. Si se llega a una hoja del árbol al descender, se recupera el caracter (byte) que corresponde a esa hoja y se graba en el archivo de salida (que es el que se quiere regenerar...) El proceso se detiene cuando el número de bytes grabados coincide con la cantidad de bytes que tenía el archivo original:

```
// ...obtener el arreglo que representa al arbol y el indice de la raiz...
HuffmanTreeNode a[] = ht.toArray();
int raiz =  a.length - 1;

// ...comenzar la fase de descompresion (por fin...)

// ...ubicar un indice auxiliar en la raiz del arbol...
int nodo = raiz;

// ...contador de bytes que se llevan descomprimidos...
long cb = 0;

// ...leer y analizar byte por byte el archivo comprimido...
while(comprimido.getFilePointer() < comprimido.length())
{
   // ...leer un byte sin signo desde el archivo...
   int b = comprimido.readUnsignedByte();

   // ... iniciar la mascara para el analisis bit a bit...
   int mascara = 0x00000080;

   // ...y comenzar el analisis bit a bit...
   for(int bit = 0; bit < 8 && cb != ta; bit++)
   {
```

```
      // ...consultar el valor del bit actual...
      int aux = b & mascara;

      // ... y bajar en el arbol por la rama que corresponda...
      if(aux == mascara)
      {
         // ...si el bit era un uno, bajar por la derecha...
         nodo = a[nodo].getRight();
      }
      else
      {
         // ...si el bit era un cero, bajar por la izquierda...
         nodo = a[nodo].getLeft();
      }

      // ...correr el 1 a la derecha en la mascara...
      mascara = mascara >>> 1;

      // ...controlar si se llego a una hoja (un simbolo original)...
      if (a[nodo].getLeft() == -1)
      {
         // ...es una hoja... grabar el byte / simbolo que hay en ella...
         byte salida = ht.getSymbol(nodo);
         nuevo.writeByte(salida);

         // ...contar el byte que se acaba de recuperar...
         cb++;

         // ...y volver a la raiz del arbol...
         nodo = raiz;
      }
   }
}

// ...cerrar todo y terminar...
nuevo.close();
comprimido.close();
```