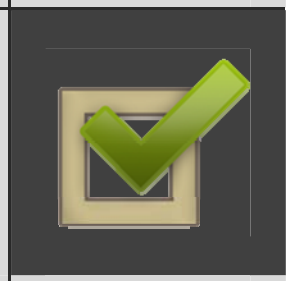
ALGORITMOS Y ESTRUCTURA DE DATOS



Compresor Huffman

Compresión de archivos mediante el algoritmo de Huffman

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 el video 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 WinRAR o 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.

Aquí 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 su byte equivalente: 01000001), cada vez que aparezca una 'A' estaremos economizando 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. A la secuencia de bits que utilizamos para recodificar a cada carácter la llamamos: código Huffman.

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

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: *arrays*, archivos y listas enlazadas.

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.

Es decir, este ejercicio 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 vayamos a utilizar en el programa. Estas implementaciones estarán a cargo del alumno y constituyen el ejercicio en sí mismo.

El algoritmo de Huffman

Introducción

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. Por 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.

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 para codificar al carácter 'C'.

Byte o Carácter	Codificación
O	01
M	001
C	000

Si esto fuese así, entonces para codificar los primeros 3 caracteres del texto anterior alcanzará 1 byte, lo que nos dará 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 propuesto más arriba.

El algoritmo paso a paso

Paso 1 - Contar la cantidad de ocurrencias de cada carácter

El primer paso será contar cuántas veces aparece cada carácter (o byte) dentro del archivo; y para esto debemos tener en cuenta las siguientes consideraciones.

Un byte consiste en un conjunto de 8 bits que pueden combinarse de $2^8 = 256$ maneras diferentes. Esto significa que sea cual fuere la cantidad de bytes que contenga un archivo, éstos se distribuirán entre no más de 256 variantes. Por supuesto, algunas más frecuentes que otras.

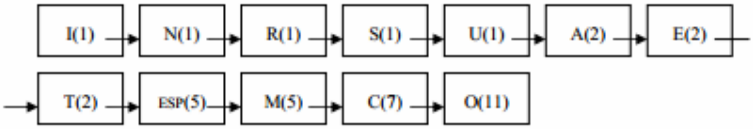
Entonces, para contar cuantas veces aparece cada byte dentro del archivo utilizaremos una tabla (*array*) de 256 filas. Cada fila con su correspondiente contador de ocurrencias.

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

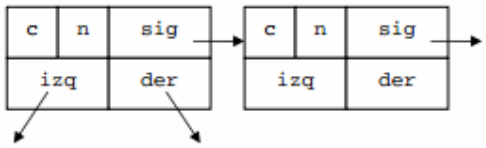
	Carácter	*n*
:		
:		
77	M	5
78	N	1
79	O	11
:		
82	R	1
83	S	1
84	T	2
85	U	1
:		
255		

Paso 2 - Crear una lista enlazada

Conociendo cuantas veces aparece 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.



Los nodos de la lista tendrán la siguiente estructura:

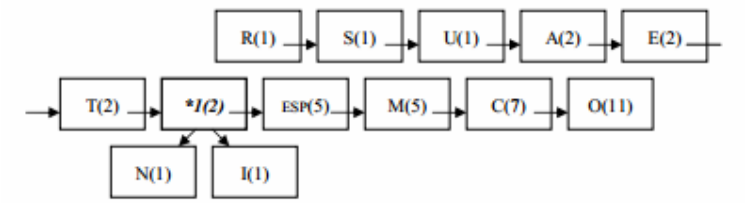


El campo `c` representa al carácter (o byte) propiamente dicho; y el campo `n` indica la cantidad de veces que dicho carácter aparece dentro del archivo. El campo `sig` es la referencia al siguiente nodo de la lista enlazada. Los campos `izq` y `der`, más adelante, nos permitirán implementar el árbol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

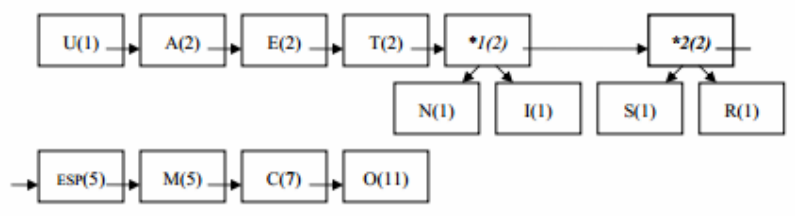
Paso 3 - Convertir la lista enlazada en un árbol Huffman

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 *n* 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.

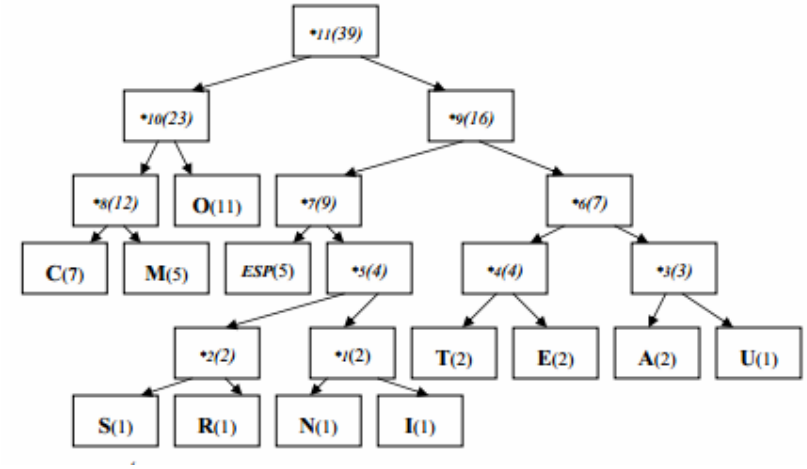
Luego insertamos en la lista al nuevo nodo (raíz) respetando el criterio de ordenamiento que mencionamos más arriba. Si en la lista existe un nodo con la misma cantidad de ocurrencias (que en este caso es 2), la inserción la haremos a continuación de éste.



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



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



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

Paso 4 - Asignación de los códigos Huffman

El siguiente paso será asignar un código Huffman a cada uno de los caracteres reales que, como vimos, se encuentran ubicados en las hojas del árbol.

Para esto debemos considerar el camino que une a la raíz del árbol con cada una de las hojas. 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.

Podemos verificar que la longitud del código que el árbol Huffman le asigna a cada carácter es inversamente proporcional a su cantidad de ocurrencias. Los caracteres más frecuentes reciben códigos más cortos y los que son menos frecuentes reciben códigos más largos.

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

	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				

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 de bits con que se representa a cada carácter; sin embargo 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, si *n* es 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 con códigos de, a lo sumo, $256/2 = 128$ bits.

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

Paso 5 - Codificación y decodificación del contenido

Codificación (compresión)

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			...

Decodificación (descompresión)

Para descomprimir un archivo necesitamos disponer del árbol Huffman que ha sido utilizado para su codificación; sin el árbol la información no se podrá recuperar. Por este motivo el algoritmo de Huffman es también un método de encriptación.

Supongamos que, de alguna manera, podemos reconstituir 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 nos posicionamos 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; entonces lo podremos grabar en el archivo que estamos restaurando.

Implementación del algoritmo

Analizaremos una implementación del algoritmo de Huffman con la cual podremos desarrollar dos programas para comprimir y descomprimir archivos.

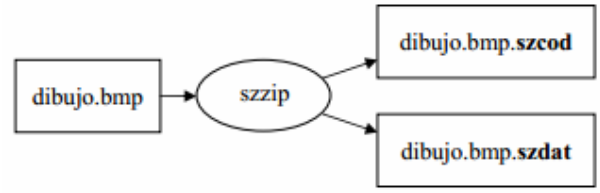
Compresor de archivos

Este programa recibirá 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 conteniendo 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 invocamos al compresor de la siguiente manera:

Implementación Java	Implemencación C++
`C:\>java szzip dibujo.bmp`	`C:\>szzip zip dibujo.bmp`

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



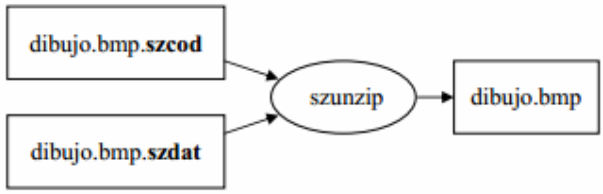
Descompresor de archivos

El programa descompresor debe realizar la tarea inversa al programa compresor. Para esto, recibirá 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:

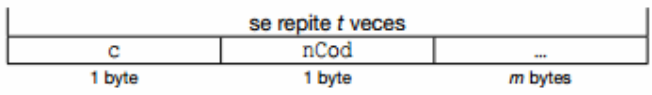
Implementación Java	Implemencación C++
`C:\>java szunzip dibujo.bmp`	`C:\>szzip unzip dibujo.bmp`

el resultado será el siguiente:



Estructura del archivo `.szcod` (árbol Huffman)

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



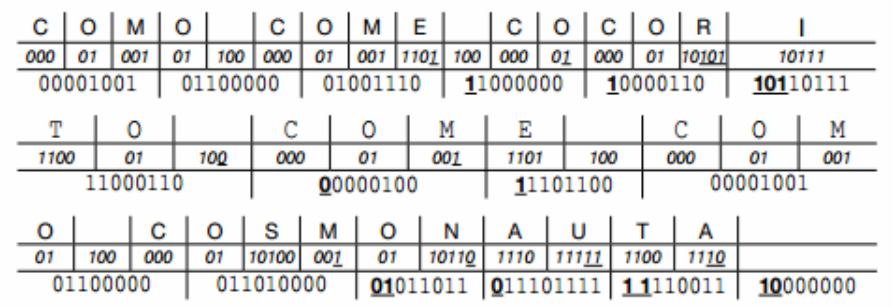
Si consideramos que el árbol tiene exactamente *t* hojas entonces los bytes del archivo `.szcod` repetirán *t* veces el siguiente patrón:

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

Estructura del archivo `.szdat` (archivo comprimido)

Este archivo contiene la información comprimida, reemplazando a 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 los bytes que forman el archivo comprimido.



Solo necesitamos 16 bytes para representar a los 39 bytes del texto original.

Además, grabaremos en este archivo, en los primeros 8 bytes, un valor numérico entero de tipo `long` indicando 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 proveen recursos que incluyen funciones tales como: `writeBit` y `readBit` que encapsulan la lógica necesaria para resolverlo, permitiéndole al programador abstraerse del problema. Además, las funciones `writeLength` y `readLength` le permitirán al programador grabar en los primeros bytes del archivo 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 (o **TAD**) `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 debe codificarse al carácter representado por la hoja que retornó.

Setup del ejercicio

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

El *setup* incluye las siguientes clases utilitarias:

- `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 que ilustran cómo debemos usar cada una de estas clases.

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

```
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 programa `sszzip` y `szunzip` y de todas las clases en las que se basa su codificación. Todo esto lo analizaremos más adelante.

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 interfases. Veamos:

Recorreremos el archivo que queremos 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 estos objetos se presentan en la siguiente tabla; la implementación de sus métodos quedará a cargo del alumno.

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`

Programa compresor (`szzip.java`)

Veamos el código fuente del compresor. El lector verá que es lineal. Comenzamos instanciando la clase que representa al archivo de entrada. Luego le pedimos crear la tabla de ocurrencias. A la tabla le pedimos que 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.

```
public class szzip
{
   public static void main(String[] args)
   {
      // abrimos el archivo original que vamos a comprimir
      IFileInput inputFile = new IFileInput();
      inputFile.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 = new IFileCode();
      codeFile.setFilename(args[0] + ".szcod");
```

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

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

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

Programa descompresor (`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.

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

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

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

      // abrimos el archivo original (el que vamos a restaurar)
      IFileInput inputFile = new IFileInput();
      inputFile.setFilename(args[0]);

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

Clases e implementaciones

Llegó el momento de programar. Ahora definiremos los métodos de cada una de las clases para que el alumno, como programador, complete sus implementaciones y haga que los programas `szzip` y `szunzip` funcionen correctamente.

La especificación de la tarea que cada método debe realizar está documentada en el mismo código fuente. Por esto, sugiero prestar especial atención a dicha información.

`ICode.java` – Clase de los códigos Huffman

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

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

```
      return 0;
   }

   // retorna la longitud del codigo Huffman (valor entre 1 y 128)
   public int getLength()
   {
      return 0;
   }

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

`ITree.java` – **Clase del árbol binario o árbol Huffman**

La clase `ITree` representa al árbol Huffman.

```
public class 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()
   {
      return null;
   }

   // 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)
   {
      return null;
   }
}
```

`IList.java` - **Clase de la lista enlazada**

La clase `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.

```
public class IList
{
   // desenlaza y retorna el primer nodo de la lista
   public Node removeFirstNode()
   {
      return null;
   }
   // 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()
   {
      return null;
   }

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

`ITable.java` - **Clase de la tabla de ocurrencias**

La clase `ITable` **representa la tabla de ocurrencias que indicará cuantas veces aparece cada carácter dentro del archivo que vamos a comprimir.**

```
public class 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)
   {
      return 0;
   }

   // crea la lista enlazada
   public IList createSortedList()
   {
      return null;
   }

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

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

`IFileInput.java` – **Clase del archivo que vamos a comprimir o a restaurar**

```
public class IFileInput
{
   // asigna el nombre del archivo
   public void setFilename(String filename)
   {
   }
```

```
   // retorna el nombre del archivo
   public String getFilename()
   {
      return null;
   }

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

   // retorna la longitud del archivo
   public long getLength()
   {
      return 0;
   }
}
```

`IFileCode.java` – **Clase 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.

```
public class 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 writeBitde la clase UFile
   public void save(ITable table)
   {
   }

   // lee el archivo (ya generado) y construye el arbol Huffman
   // NOTA:usar el metodo readBitde 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()
   {
      return null;
   }
}
```

`IFileCompressed.java` – **Clase del archivo comprimido**

Representa el archivo `.szdat` que contiene la información comprimida.

```
public class IFileCompressed
{
   // asigna el nombre del archivo
   public void setFilename(String filename)
   {
   }
```

```
   // retorna el nombre del archivo
   public String getFilename()
   {
      return null;
   }

   // 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)
   {
      return null;
   }

   // 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)
   {
   }
}
```

Manejo de archivos en Java

Para implementar los métodos `save` y `restore` de la clase `IFileCompressed`, el método `save` de la clase `IFileCode` y el método `createTable` de la clase `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.

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.

```
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*.

Bytes sin bit de signo

Java provee el tipo de datos `byte` que permite representar un valor numérico entero en 1 byte de longitud. El problema es que este tipo de dato 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 repre-sentar *unsigned bytes*.

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 en línea de comandos.

```
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);
         }
      }
   }
}
```

Buffers (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.

```
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
         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);
```

Clases utilitarias

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

- Recorrer el árbol binario, tema que corresponde a otras materias.
- Leer y escribir bits en archivos.

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

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:

```
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
         char c = hoja.getC();
         int n = hoja.getN();
         String cod = codHuffman.toString();
         System.out.println(c+","+n+","+cod);

         // 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}
O(11) codigo = {01}
 (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', 11, null, null);
      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;
   }
}
```

Clase `UFile` - Leer y escribir bits

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 reu-nido 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:

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

      // recorremos DEMO.dat 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 los 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 no se completo un paquete lo completo con ceros y lo grabo
         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 fi s = null;
      try
      {
         // abrimos el archivo para grabar los bits
         fis = new FileInputStream(nomArch);
         // instanciamos UFile
         UFile uFile = new UFile(fi s);
         // recorremos la cadena de bits
         int bit = uFile.readBit();

         // cuando no haya mas bits retornara 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) fi s.close();
         }
         catch(Exception ex)
         {
            ex.printStackTrace();
            throw new RuntimeException(ex);
         }
      }
   }
}
```