Obligatorio 2 - Compresión de datos: Parte 1

22 de abril de 2025

1. Generalidades

La aprobación de este curso se consigue mediante la correcta implementación de tres pequeños proyectos de programación (que llamaremos obligatorios). Éstos son propuestos en tres momentos sucesivos del curso, aumentando en complejidad, y forman parte de un mismo paquete, alimentándose mutuamente. Los programas desarrollados en la primer entrega son utilizados en la segunda y en la tercera. Algo similar pasa entre lo que se desarrolle en la segunda y tercera entrega. Cada obligatorio será entregado a través de una página web habilitada para tales fines, con fecha límite de entrega señalada en la misma página. Estas entregas se complementan con pruebas parciales escritas cuyo objetivo es evaluar aspectos más teóricos relacionados con el propio obligatorio.

Es importante recalcar que **tanto la prueba escrita como el proyecto entregado son individuales.** El sistema de recepción de entregas, además de almacenar los archivos enviados por los estudiantes, realiza un control de copias contra las entregas de otros estudiantes así como de programas similares que se encuentran en la web. Ese programa es capaz de detectar copias "maquilladas", es decir donde se cambiaron nombres de variables u otras formas de ocultar una copia. Este asunto debe ser bien entendido. No tenemos ningún inconveniente en que discutan soluciones, miren en la web, etc., pero lo entregado debe ser realmente el producto de vuestro trabajo y si se detecta que hubo copia ello implica una sanción que puede implicar la pérdida del curso e incluso sanciones mayores, tal como está especificado en el reglamento de la Facultad.

En caso de ser posible, el sistema intentará además compilar y ejecutar la entrega de cada estudiante, a fin de dar un mínimo de información respecto de qué tan bien funciona la entrega. Dependiendo del caso, esta evaluación preliminar estará o no disponible.

La evaluación preliminar mencionada anteriormente **no** determina la nota obtenida, siendo ésta definida por una evaluación global por parte de los docentes que incluye los obligatorios, los parciales y la participación en clase.

1.1. Formato del archivo a entregar

El archivo entregado debe ser un archivo comprimido en formato **zip** (NO se aceptan archivos en formato rar), de nombre **nombre1_nombre2_apellido1_appelido2.zip** en el que los fuentes están en la raíz del zip. El contenido del archivo debe incluir los siguientes elementos (que deben estar en la raíz del mismo y no en un directorio interno):

- Todos los archivos fuente creados por el estudiante (**.h** y **.c**)
- Un archivo **Makefile** para compilar el o los programas requeridos en el trabajo.

La resolución de este obligatorio consiste en la generación de un ejecutable de nombre `obligatorio2`. Supongamos que su nombre es Juan Pablo Perez Fernandez, y usted generó dichos ejecutables a partir de los archivos **obligatorio2.c**, la biblioteca con las funciones para codificar y decodificar según Huffman **libHuf.a**, que se genera desde el archivo **libHuf.c**, además de la biblioteca **libbits.a** que se genera desde **bits.c** y sus correspondientes archivos de encabezado **bits.h** y **libHuf.h**. Entonces debe subir un archivo de nombre **Juan_Pablo_Perez_Fernandez.zip** con el siguiente contenido:

```
obligatorio2.c
bits.c
bits.h
libHuf.c
libHuf.h
Makefile
```

El **Makefile** en este caso podría ser así:

```
all: libbits.a libHuf.a obligatorio2
COPT=-Wall -ansi -ggdb
LDFLAGS=-L./
LDLIBS= -lHuf -lbits -lm

obligatorio2: obligatorio2.o
  cc $(COPT) -o $@ obligatorio2.o $(LDFLAGS) $(LDLIBS)

.c.o:
  cc $(COPT) -c $<

libbits.a: bits.o
  ar rcs $@ $<

libHuf.a: libHuf.o
  ar rcs $@ $<

clean:
  rm -rf *.o obligatorio2 libbits.a libHuf.a
```

En el archivo **Makefile** anterior se puede generar la biblioteca **libbits.a**, también compilar los programas para el obligatorio2 (obligatorio2.c, libHuf.c), el primero llama a la biblioteca libHuf.a que contiene las funciones utilizadas. Para ello se debe invocar el **Makefile** de la siguiente manera:

```
make libHuf.a
make obligatorio2
```

La primera línea genera la biblioteca **libHuf.a** que se utiliza en el obligatorio2, O simplemente se puede utilizar el comando `make` para generar la biblioteca y el ejecutable del obligatorio2 (usando la biblioteca **libHuf.a** que fue generada en el obligatorio 1):

```
make
```

Por último con el fin de limpiar el directorio de trabajo y realizar una nueva compilación se utilizan sendos comandos:

```
make clean
make
```

Nota 1: Pueden crear un archivo **.zip** desde la máquina virtual con el comando `zip`; la sintaxis es, desde la carpeta de trabajo:

```
$zip -r nombre_archivo.zip .
```

En el ejemplo anterior, sería `zip -r Juan_Pablo_Perez_Fernandez.zip`

Nota 2: El **Makefile** presentado anteriormente sirve para generar la biblioteca y compilar los programas, pero el que se utilizará para evaluar el trabajo de los estudiantes será el que aparezca junto con el programa de prueba, que es algo distinto.

Nota 3: Se dispone, junto con esta letra, el archivo **TablaDeCodificacion.txt**, que contiene los códigos necesarios que deben utilizar para comprimir y descomprimir los mensajes.

Nota 4: Se sugiere inicialmente probar sus programas en archivos de texto más cortos creados por ustedes mismos, a fin de verificar la codificación visualmente con un editor binario como el bless.

Nota 5: Se dispone, junto con esta letra, el archivo **discurso.txt**, que sirve para hacer pruebas. El mismo está formado por un discurso. Sugerimos que lo codifiquen y luego lo comparen con el archivo decodificado usando la función `diff` de Linux.

1.2. Metodología de trabajo

Algunas recomendaciones generales sobre cómo trabajar con proyectos como los que se proponen aquí:

- Simplicidad (KISS - Keep It Simple, Stupid). No complicar el código más allá de lo requerido.
- Prolijidad. No importa cuánto aburra, documentar bien lo que se hace es fundamental; es muy fácil olvidarse lo que uno mismo hizo. Esto incluye la inclusión de comentarios y el uso de variables con nombres autoexplicativos, si es posible.
- Incrementalidad. Implementar y probar de a pequeños pasos. "No construir un castillo de entrada". Es muy difícl encontrar las causas de un problema si se prueba todo simultáneamente.

2. Introducción al problema

El problema que se plantea en este obligatorio, la compresión de señales digitales, depende directamente de la codificación de símbolos, que es de gran importancia en muy diversas áreas de la ingeniería eléctrica, en particular en telecomunicaciones y en electrónica. Si bien desde el punto de vista teórico y formal las herramientas para trabajar con este tipo de problemas se ven más adelante en la carrera, es posible trabajar con, y comprender informalmente, algunos algoritmos importantes y algunos conceptos asociados.

La compresión de señales consiste en codificar una secuencia de símbolos con un fin específico, por ejemplo para enviar un mensaje a través de un *canal de comunicación* o para almacenarlo en memoria. En esos casos es necesario asociar a cada símbolo un código binario único, de tal manera que sea posible decodificar el mensaje (en el receptor si estamos hablando de un canal de comunicación o al leer la memoria en el caso de un mensaje almacenado). Qué código asociar a cada símbolo, y cómo asociarlos para poder comprimir los datos y posterioriormente descomprimirlos, es todo un mundo. Buscar códigos que compriman el mensaje de manera tal que aprovechemos el canal de comunicación de manera óptima ha sido un desafío durante muchos años. Por ejemplo, dado que ese canal tiene una capacidad limitada (un cierto *ancho de banda*) podemos buscar la forma de trasmitir la información contenida en nuestro mensaje con la menor cantidad de bits posible. Hablamos entonces de *codificación para comprimir*.

Es interesante destacar que la codificación también es utilizada con otros propósitos, por ejemplo puede ser interesante asociar códigos que tengan la propiedad de identificar si hubo algún error, y eventualmente qué error fue y corregirlo si ocurrió. Esos códigos se llaman *códigos correctores de errores* y son muy utilizados en electrónica (para defender la integridad de los datos ante los efectos del ruido en un sistema electrónico) y en telecomunicaciones (para mejorar la integridad de los datos ante el ruido en un canal de comunicación).

En otras circunstancias podemos estar interesados en ocultar ante terceros el contenido de la información trasmitida, y para ello definir una codificación que asocie códigos a símbolos de una manera que sólo nosotros y el destinatario del mensaje conozcamos y que sea muy difícil de descubrir

por terceras personas. Esta tercera forma de codificar da lugar a toda un área llamada *criptografía*, que se ocupa de desarrollar métodos específicos de codificación y decodificación con ese fin.

La figura 1 ilustra el esquema general de trabajo. A los efectos de este obligatorio usaremos los conceptos con poca rigurosidad, de manera más bien intuitiva. Para una definición precisa deberán esperar a los cursos específicos. De todas maneras, a fin de trabajar en este obligatorio, definiremos de manera muy general algunos conceptos: Un *símbolo* es un elemento comprensible por un ser humano, por ejemplo una letra en el alfabeto o un número. Un *mensaje* es una serie de símbolos. Llamaremos *código* a la representación de un símbolo en forma binaria, es decir como una secuencia de unos y ceros. En este contexto el *medio* es el lugar en que se almacenan o por el que se trasmiten los datos en forma de códigos, es decir una zona de memoria o un canal de comunicación respectivamente. La codificación puede hacerse de tal manera que los *códigos* sean difícilmente descifrables (en el caso de la criptografía), que tengan redundancia para poder darnos cuenta de si hubo errores (en el caso de los códigos correctores de errores) o que en media usen un número pequeño de bits (en el caso que se busque comprimir).

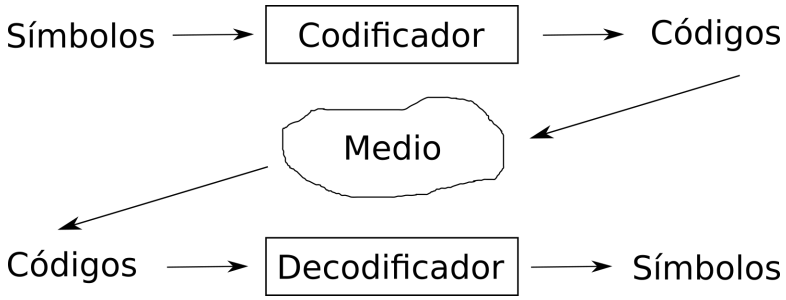

Figura 1: Esquema general de trabajo, un mensaje compuesto por N símbolos es codificado por el *Codificador*, generando N códigos que son trasmitidos a través de un canal o almacenados en memoria. En el esquema estos dos casos están representados por la nube llamada *Medio*. El *Decodificador* convierte los códigos recibidos a través del canal de comunicación o leídos de memoria y los convierte nuevamente en símbolos como los que conforman el mensaje original.

En este obligatorio vamos a trabajar con el conjunto de símbolos definidos en la codificación ASCII (American Standard Code for Information Interchange). Este estandar incluye las letras y cifras, así como una serie de caracteres de control que permiten escribir un texto. La codificación ASCII establece ciertos códigos para esos símbolos[1], con la propiedad de que todos esos códigos tienen un tamaño fijo, 8 bits, y es ampliamente utilizada en las computadoras para identificar los símbolos asociados a las teclas o para imprimir en pantalla.

En términos generales es posible leer un archivo de texto, byte por byte, y conociendo la codificación ASCII, sabemos qué símbolo es representado por ese conjunto de 8 bits. Del mismo modo, si tenemos una serie de símbolos alfanuméricos podemos escribir una sucesión de bytes correspondientes a sus respectivos códigos ASCII.

A lo largo del curso realizaremos tres obligatorios a partir de los cuales generaremos una biblioteca de manejo de bits y funciones que nos permitan manejar la codificación y decodificación de datos conociendo una tabla de codificación. Al final del curso podremos comprimir y descomprimir datos utilizando un método que calcula la tabla de codificación y podremos implementar una cadena como la que se representa en la figura 1.

Los tres obligatorios que realizaremos son los siguientes:

- Obligatorio 1: Construcción de una biblioteca para manejo de bits.
- Obligatorio 2: Compresión y descompresión de datos con codificación dada.
- Obligatorio 3: Generación de la tabla de codificación usando Huffman..

[1] http://www.asciitable.com/

En este obligatorio se debe generar un ejecutable, que se usará para comprimir un archivo de texto o descomprimir un archivo comprimido, según los argumentos de la línea de comando. La compresión implica leer un mensaje sin comprimir (o mensaje original) desde un archivo, codificarlo utilizando la tabla de codificación proporcionada y escribir dicha codificación en otro archivo. La descompresión implica leer un mensaje codificado desde un archivo, decodificarlo y escribir la decodificación en otro archivo. Noten que al descomprimir un mensaje se deberá obtener el mensaje original.

3. Punto de partida

Para desarrollar este obligatorio se debe utilizar la biblioteca de bits desarrollada en el obligatorio 1.

Nota: Tengan en cuenta que dicha biblioteca puede ser modificada según sea conveniente.

4. Compresión de datos: tabla de codificación dada

Como se mencionó anteriormente, para comprimir un mensaje es necesario codificarlo mediante la reducción del número de bits utilizados. Si un *mensaje* es codificado con *códigos* de largo fijo, la longitud del *mensaje* codificado lm será de $lm = nbS * lc$, donde lm es el largo del mensaje en bits, nbS es el número de *símbolos* y lc es el largo del *código* en bits. Por ejemplo si utilizamos la codificación ASCII extendida (que utiliza 8 bits para cada *símbolo* y contiene 256 *símbolos* diferentes, entonces un archivo de 200 caracteres pesará $lm = 200 * 8$, es decir 200 bytes (recordar que cada byte está formado por 8 bits). La técnica de compresión de datos que utilizaremos busca realizar una codificación de los símbolos mediante la generación de *códigos* óptimos, en el sentido de generar un *mensaje* lo más corto posible conteniendo la misma información, lo que permite ocupar menor espacio en memoria o consumir menos ancho de banda en una trasmisión.

La técnica de compresión de datos en cuestión consiste en hacer una estadística de la aparición de los *símbolos* en el *mensaje* y codificar con menos bits los *símbolos* más frecuentes y con más bits los *símbolos* menos frecuentes. En cursos más avanzados de la carrera podrán estudiar muchas cosas sobre codificación: cómo medir la cantidad de información que trasmite un mensaje, por ejemplo. Acá nos concentraremos en los aspectos prácticos de este tipo de codificación a fin de implementarla y de esa manera aprender algo de programación en *C*.

Para tener una idea del procedimiento, los invitamos a mirar el video desarrollado por nuestro compañero Gastón Notte, docente del Centro Universitario Regional del Litoral Norte, en Paysandú, que se encuentra en: //Obligatorio 2 - Ejemplo de codificación y decodificación

Estamos utilizando la codificación de Huffman, que garantiza que ningún código está incluido en otro.

En este obligatorio supondremos que alguien ha realizado la estadística de aparición de los símbolos, ha definido la codificación de cada símbolo y nos ha proporcionado la tabla de codificación resultante (es decir la que establece la correspondencia entre *símbolos* y *códigos*). Dado que contamos con la tabla de codificación, nos concentraremos en generar la codificación y decodificación a partir de dicha tabla para llevar a cabo la compresión y descompresión de datos.

En términos generales la codificación asocia un *código* de largo variable a un *símbolo*. Ello significa que es necesario ir leyendo cada *símbolo* y asociarle el *código* correspondiente. La codificación debe almacenarse en una estructura que tenga los siguientes campos:

- unsigned char simbolo; /* el *símbolo* */
- int nbits; /* el número de bits del *código* correspondiente */
- unsigned int codigo; /* el *código* correspondiente. En este entero, se ocupan los *nbits* menos significativos y se llena el resto con ceros. */

Dicha estructura debe definirse de la siguiente manera en **libHuf.h**:

```
typedef struct Simbolo{
    unsigned char valor;
    int nbits;
    unsigned int codigo;
}simbolo;
```

Noten que en este caso nunca utilizaremos códigos de más de 32 bits de largo. A los efectos prácticos usaremos la tabla de codificación proporcionada por el equipo docente. Dicha tabla deberá leerse y cargarse en memoria desde un archivo de texto llamado *TablaDeCodificacion.txt*. Dicho archivo tiene la siguiente estructura interna:

- Una fila que contiene el valor nbS (entero que corresponde al número de símbolos de la tabla).
- nbS filas, donde cada fila contiene los valores de la estructura *codificacion* (previamente definida). Dichos valores están separados por un espacio.
- Cada fila del archivo *TablaDeCodificacion.txt* finaliza con un salto de línea.

Noten que en la tabla aparecen todos los símbolos codificados en ASCII, por tanto deben leerse e interpetar qué representan y armar el valor correspondiente. Una función deberá leer el archivo *TablaDeCodificacion.txt* que está codificada en texto plano, convertir los valores leídos y llenar la tabla de codificación con la estructura señalada para cada *símbolo*. En el archivo *TablaDeCodificacion.txt* los datos se han guardado de la siguiente manera: el símbolo en hexadecimal, el número de bits en entero y el código en hexadecimal.

Noten que en este caso la tabla de codificación es innecesariamente larga pues cada código está almacenado en un entero (32 bits) y muchos de sus bits son inútiles.

A fin de manejar los posibles errores que pueden aparecer al ejecutar una función, se deben definir codigos de error que serán devueltos por las funciones. Para ello utlizaremos una enumeración de la manera siguiente en **libHuf.h**:

```
typedef enum codigo_error{
    TODO_OK = 0,
    ERROR_LECTURA = 1,
    ARCHIVO_INEXISTENTE = 2,
    ERROR_ESCRITURA = 3,
    CODIGO_MUY_LARGO = 4,
    ERROR_MEMORIA = 5,
}CodigoError;
```

4.1. Codificación

Al codificar un mensaje formado por *nbS* símbolos, deberemos concatenar los *códigos* correspondientes a esos *símbolos*. Como los *códigos* tienen un largo variable vamos a ir concatenándolos en un entero de 32 bits, ésto es dado que nunca utilizaremos códigos de más de 32 bits de largo. Así por ejemplo, si tuviéramos un mensaje formado por 3 *símbolos* de largo 13, 12 y 7 bits respectivamente, formaríamos un entero de 32 bits con el primer *código* ocupando los 13 bits más significativos, luego 12 bits correspondientes al segundo *símbolo* y finalmente los 7 bits menos significativos para el tercer *símbolo*. Está claro que en términos generales tendremos mensajes formados por muchos *símbolos* y por tanto se formarán cadenas de bits muy largas (aunque es claro que para un mensaje corto la concatenación de bits pueden ser de un largo menor a 1 entero). En cualquier caso, esas cadenas de bits las podemos segmentar en bytes de 8 bits a efectos por ejemplo de guardarlos en un archivo. Cada

vez que tengamos 8 bits los escribiremos en el mensaje de salida. Siguiendo esta línea es de notar que con alta probabilidad el último byte tendrá bits sobrantes. Es decir que la combinación de *códigos* no dará un número de bits que sea múltiplo de 8. Para solucionar llenaremos de ceros los bits menos significativos sobrantes del último entero que se desee escribir en el archivo de salida. Tendremos que saber cuantos bits llenamos de ceros de este modo, cosa que se guardará en la variable *NbStuff* que formará parte de la información trasmitida al principio del mensaje.

El proceso de cómo concatenar varios símbolos en un entero se ilustra en la figura 2.

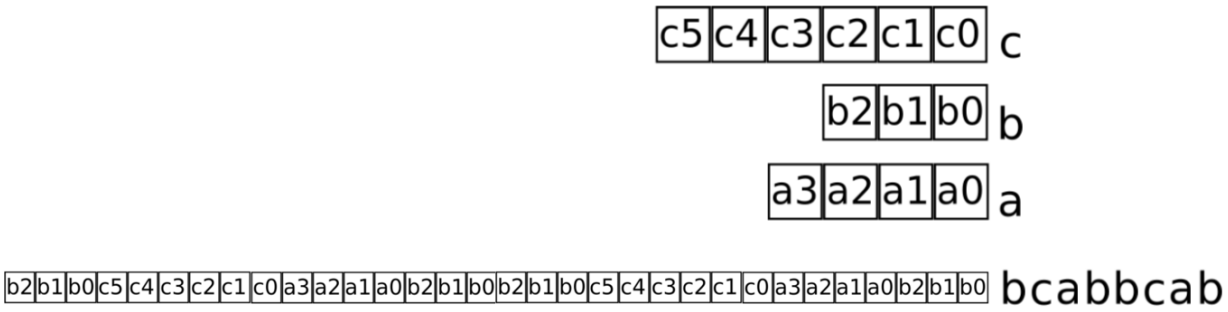

Figura 2: Ejemplo de la concatenación de tres *códigos*, correspondientes a los *símbolos* que se muestran en las primeras tres filas de esta figura. El *símbolo* `a` es codificado con 4 bits, el *símbolo* `b` con 3 bits y el *símbolo* `c` con 6 bits. Si la secuencia codificada es `bcabbcab` (que casualmente configura un paquete de 32 bits, es decir 4 bytes), su concatenación es como aparece en la cuarta fila de la figura. Este proceso implica que al leer cada carácter del archivo codificado se comience a escribir en el archivo de salida el código correspondiente al primer símbolo leido y así sucesivamente. Recordar que si quedan bits libres, estos se deben llenar con ceros. El número de bits de relleno para completar un byte completo lo llamaremos **NbStuff**.

El mensaje codificado estará formado por el valor de **NbStuff** seguido de la sucesión de códigos correspondientes a los símbolos del mensaje. Más precisamente, el archivo codificado estará formado por:

- Un carácter indicando el valor de **NbStuff**, es decir el número de bits de relleno al final del mensaje.
- Una sucesión de bytes conteniendo el mensaje codificado.

Para ejecutar la codificación se utilizará el siguiente procedimiento:

- El programa principal (que contiene el *main*), llamado *obligatorio2.c*, lee los parámetros de entrada (en la línea de comandos) que serán los siguientes:
  1. Una letra que puede ser C (en caso de que se esté codificando), D (en caso de que se esté decodificando) o T (para imprimir la tabla de cdificación leída),
  2. el nombre del archivo de entrada, a codificar si la opción en el primer parámetro fue C, o a decodificar si la opción en el primer parámetro fue D.
  3. el nombre del archivo de salida, codificado si la opción en el primer parámetro fue C, o decodificado si la opción en el primer parámetro fue D.
  4. El nombre del archivo donde está la tabla de codificación.

Por ejemplo, una llamada podría ser:

```
./obligatorio2 C NombreArchivo.txt ArchivoCodificado TablaDeCodificacion.txt
```

En ese caso el programa *obligatorio2* es llamado para codificar el archivo llamado *NombreArchivo.txt* y generar el archivo codificado que lleva por nombre *ArchivoCodificado*. Noten que la tabla de codificación y el archivo a codificar tienen una terminación *.txt*, indicando que están en texto plano y pueden ser leídos por un editor de texto, mientras que el archivo codificado no tiene terminación, indicando que se trata de un archivo binario. Se trata de convenciones, pero es importante saber que los archivos codificados no utilizan una codificación comprensible por un editor de textos.

Otra llamada al programa podría ser:

```
./obligatorio2 D ArchivoCodificado ArchivoDecodificado.txt TablaDeCodificacion.txt
```

En ese caso el programa *obligatorio2* es llamado para decodificar el archivo llamado *ArchivoCodificado* y generar el archivo decodificado que lleva por nombre *ArchivoDecodificado.txt*.

Noten que en este caso *NombreArchivo.txt* y *ArchivoDecodificado.txt* deben ser idéntcos.

El archivo codificado debe escribirse como una secuencia binaria. Para ello deben utilizar las funciones `fwrite` (y no `fprintf`) para la tabla de codificación y `fputc` o `fputchar` para el mensaje. Naturalmente deberá untilizar las funciones complementarias `fread` (y no `fscanf`) y `fgetc` o `fgetchar` respectivamente.

Nota 1: Para facilitar el test de las funciones es importante cumplir el siguiente criterio: el cierre de archivos (**fclose**) se deben hacer después de llamar a las funciones de la biblioteca *libHuf.a* y no dentro de ellas.

Nota 2: Las liberaciones de memoria (**free**) se deben hacer dentro de las funciones de la biblioteca *libHuf.a*, salvo que el alcance de esa memoria reservada salga de dichas funciones. Por ejemplo en la función *leer_tabla_codificacion* de la biblioteca *libHuf.a*, se reserva la memoria para almacenar la tabla de símbolos *tablaCod*. Luego se invocan las funciones *codificarConTabla* y *decodificarConTabla* que usan la tabla de símbolos *tablaCod* y recién después de volver de dicha función (cuando no necesitamos más ese arreglo *tablaCod*) lo liberamos usando el **free**.

Las funciones deben implementarse en **libHuf.c** y sus respectivas definiciones deben ir en **libHuf.h**.

A continuación se detalla una lista de funciones necesarias para implementar la codificación.

- Una función que deberá leer la tabla de codificación que se encuentra en el archivo *TablaDeCodificacion.txt* y cargarla en un arreglo de estructuras de tipo *simbolo*, que llamaremos *tablaCod*. Esta función devolverá el *CodigoError* correspondiente, por ejemplo *TODO_OK* si no tuvo problema o el código correspondiente si hubo algún error, por ejemplo si el puntero al archivo TablaDeCodificacion es *NULL*. El primer argumento es un puntero al archivo *TablaDeCodificacion.txt*, el segundo argumento es un puntero doble al arreglo *tablaCodificacion* y el tercer argumento es un puntero a un entero con el número de símbolos en dicha tabla.

  ```
  CodigoError leer_tabla_codificacion_txt(FILE *fpTdC, simbolo** tablaCod, int* nbS)
  ```

  Dicha función se llamará desde el programa principal para leer la tabla de codificación. Nótese que la función crea con memoria dinámica la tabla y devuelve tanto la tabla como el valor *nbS*.

- Deberá escribirse una función *codificarConTabla* a ser llamada desde el *main* (luego de haberse llamado a la función *leer_tabla_codificacion_txt*), a la que se le pasarán los punteros al archivo

conteniendo el mensaje a codificar y al archivo que contendrá el mensaje codificado, la tabla de codificación y el número de símbolos *nbS*. Dicha función debe definirse de la siguiente manera:

```
CodigoError codificarConTabla(FILE *fpIn, FILE *fpOut, simbolo* tablaCod, int nbS)
```

En dicha función se pide que lea el mensaje a codificar y se codifique, para esto es necesario implementar la siguiente función:

1. Una función que deberá leer el archivo que contiene el mensaje a codificar y devolver los *símbolos* leídos en un arreglo de caracteres sin signo. Recuerden que estos símbolos están en ASCII y por tanto cada uno cabe en un caracter. Dicho arreglo se llamará *Msj* y será un vector de tamaño *nbM caracteres*, cada uno contendrá uno de los símbolos leídos. Se sugiere crear el arreglo *Msj* dentro de la presente función, que pasará por parámetro tanto el puntero al arreglo *Msj* como el puntero a *nbM* (que corresponde al número de caracteres del mensaje). Dentro de la función se llena dicho arreglo y posteriormente se podrá acceder a su contenido en otras funciones. Para ello se debe utilizar memoria dinámica. La función debe devolver el código de error correspondiente.

   ```
   CodigoError leer_archivo_txt(FILE* fpIn, unsigned char **Msj, int* nbM)
   ```

   Para saber el tamaño de *Msj* debemos conocer *nbM* y eso se puede saber a partir del tamaño del archivo utilizando las funciones `fseek()`, `ftell()` y `rewind()`.

La función *codificarConTabla* deberá tomar el arreglo *Msj*, leer cada *símbolo* contenido en él, buscar el *código* correspondiente en la tabla de codificación e ir armando el mensaje codificado mediante la concatenación de dichos *códigos*. La función debe devolver el código de error correspondiente. Esta función debe tomar el archivo *fpOut* e ir escribiendo en el mismo el mensaje codificado.

En el archivo codificado debe aparecer primero el valor $NbStuff$ como carácter y a continuación la sucesión de *códigos* correspondientes a los *símbolos* del *mensaje*, como una sucesión de bytes (utilizar `fputc`). El último carácter debe contener en sus bits menos significativos *NbStuff* bits a cero.

5. Decodificación

Para decodificar un mensaje es necesario utilizar la tabla de codificación.

Para decodificar un archivo codificado se debe ejecutar el programa *obligatorio2* con la opción D, que leerá un archivo codificado y la tabla de codificación y utilizándola vaya leyendo los *códigos* y asociando a cada uno de ellos el *símbolo* correspondiente y a partir de ello escriba en un archivo de salida el mensaje reconstruido en código **ASCII**.

El programa principal (*ogbligatorio2*) lee los parámetros de entrada en la línea de comando que serán:

1. Una opción que será una de las siguientes letras:
   - *C*: esta opción quiere decir que se va a codificar.
   - *D*: esta opción quiere decir que se decodifica el archivo de entrada y se genera el archivo decodificado.
   - *T*: esta opción quiere decir que se lee el arhivo con la tabla de codificación y se escribe dicha tabla de codificación en el archivo de salida. Sirve para verificar que leímos correctamente la tabla.
2. el nombre del archivo de entrada, que es un archivo binario si la opción en el primer parámetro es D o un archivo de texto si la opción en el primer parámetro fue C o T.

3. el nombre del archivo de salida, que es un archivo de texto si la opción en el primer parámetro es D o T o un archivo binario si la opción en el primer parámetro fue C.

4. un nombre de archivo en que se almacenará la tabla de decodificación.

Por ejemplo, una llamada podría ser:
`./obligatorio2 D ArchivoCodificado ArchivoDecodificado.txt Tabla.txt`

En ese caso se lee el archivo *ArchivoCodificado*, se decodifica, se crea el archivo *ArchivoDecodificado.txt* con el mensaje decodificado, utilizando el archivo *Tabla.txt* con la tabla de codificación.

A continuación se detalla una lista de funciones necesarias para implementar la decodificación *libHuf.c*.

- Deben crear una función *decodificarConTabla* que se invoca desde el main una vez analizados los parámetros:

  ```
  CodigoError decodificarConTabla(FILE* fpIn, FILE* fpOut, simbolo* Tabla, int NbS)
  ```

  El primer argumento apuntará al archivo codificado, el segundo apuntará al archivo donde se guardará el mensaje decodificado, el tercer argumento es la tabla de codificación y el cuarto argumento un entero con el número de símbolos en dicha tabla. La función devuelve *CodigoError*.

  Recuerden que el archivo donde está el mensaje codificado tiene la siguiente estructura:

  - Un carácter indicando el valor de **NbStuff**, es decir el número de bits de relleno al final del mensaje.
  - Una sucesión de bytes conteniendo el mensaje codificado.

  Dentro de la función se lee el archivo codificado apuntado por *fpIn*. Se estima el número de caracteres del mensaje y en función de ello se reserva memoria para una tabla de caracteres *Msj*, donde estarán el conjunto de bytes que configuran el mensaje mismo. Noten que estos bytes están codificados y por tanto cada uno de ellos no corresponde necesariamente a un símbolo. Luego se toman los caracteres que están en *Msj* y los interpretan de acuerdo a la *Tabla de Codificación*, que debe ser leída previamente y pasada como argumento. El resultado decodificado se escribe en un archivo de salida apuntado por *fpOut*.

  Recuerden que debemos reservar al interior de esta función la memoria para el mensaje a leer. Para ello debemos estimar el número de caracteres que lo compone. Podemos calcular su tamaño en bits utilizando `ftell` y `fseek`. Recuerden que al principio del archivo codificado hay un carácter correspondiente a *NbStuff* (este carácter no forma parte del mensaje codificado).

  Las siguientes funciones deben ser creadas y utilizadas:

- Una función *indiceEnTabla* que busque el *código* correspondiente en la tabla de codificación y devuelva el índice, como un entero, indicando la posición de dicho elemento en la tabla de codificación. En caso de no encontrar el *símbolo* debe devolver -1. Noten que el *código* tiene un largo variable de *nbits* y que debemos comparar esa cantidad de bits al buscar en la tabla de *códigos*.

  ```
  int indiceEnTabla(unsigned int codigo, int nbits, simbolo* tablaCod, int NbS)
  ```

  La función tendrá los siguientes parámetros:

  - Un entero sin signo con el código buscado.
  - Un entero con el número de bits del código buscado.
  - Un puntero a la tabla de codificación *tablaCod*. Previamente cargada
  - Un entero indicando el número de símbolos *NbS*.

- Deben crear también una función para imprimir la tabla de codificación leída, en el puntero a archivo *out*. El formato debe ser el mismo que en la tabla leída en codificar y el primer elemento debe ser el valor de NbS. Noten que si quieren imprimir en pantalla pueden pasar como parámetro *stdout*:

  ```
  CodigoError salvar_codigos(simbolo* tablaCod, int NbS, FILE* out)
  ```

  La función tendrá los siguientes parámetros:

  - Un puntero a la tabla de codificación *tablaCod*. Previamente cargada
  - Un entero indicando el número de símbolos *NbS*.
  - Un puntero al archivo donde se imprimirán los códigos, si se pone *stdout* se imprimen en pantalla.

  La función devuelve *CodigoError*.

Noten que se debe tener cuidado con el órden en que se procesan los *símbolos* y se crean los *códigos* en relación a como quedan en el archivo codificado. En la codifiación los códigos se van escribiendo uno tras otro en el archivo de salida. Al decodificar se debe tener cuidado de proceder en el orden apropiado para reconstruir la secuencia de entrada. Vease la figura 2.

Una vez completado el obligatorio 2 se debe implementar el sistema completo tal como se muestra en la figura 1. Para ello se debe utilizar un documento de texto como por ejemplo el que llamamos *discurso.txt* y se puede proceder de la manera siguiente:

- *./obligatorio2 C discurso.txt discursoCodificado TablaDeCodificacion.txt*
- *./obligatorio2 D discursoCodificado DiscursoRecuperado.txt*
- diff *discurso.txt discursoRecuperado.txt*

Noten que en este caso utilizamos el archivo de entrada *discurso.txt* que está en ASCII y podemos leerlo con un editor de texto (salvo los caracteres invisibles del ASCII) mientras que el archivo codificado *discursoCodificado* no está en ASCII. La función *diff* indica si hay diferencias entre el archivo original y el recuperado.

También es interesante que comparen el tamaño de *discurso.txt* con el tamaño de *discursoCodificado*, y comprobar que el archivo codificado pesa menos que el archivo original, y por lo tanto que la compresión se llevó a cabo correctamente. Para conocer el tamaño de los archivos pueden utilizar el comando "ls -lh".

Sugerimos que se creen pequeños archivos de texto de test para ir avanzando de manera incremental. Por ejemplo proporcionamos 4 archivos de prueba *Prueba1.txt*, *Prueba2.txt*, *Prueba3.txt* y *Prueba4.txt* que son pequeños y permiten ir viendo cómo funciona el sistema y dónde están los eventuales errores. Sugerimos que prueben que el sistema funciona con todos los archivos, pues puede ser que uno de ellos no falle por alguna configuración particular pero otro archivo si genere un error.

El programa de prueba va a verificar todas las funciones indicadas en la letra de este obligatorio. La carpeta con el programa de prueba tiene un archivo binario que se llama *discursoCodificadoTest*. Este archivo se obtiene a partir de *discurso.txt* con nuestra implementación.

De esta manera, el programa de prueba ejecutará *./obligatorio2 C discurso.txt discursoCodificado TablaDeCodificacion.txt* y va a comparar los archivos para mostrar las diferencias entre el archivo *discursoCodificado* que va a generar la aplicación construida por ustedes y nuestro *discursoCodificadoTest*, que está precargado.

Las diferencias se van a mostrar en la consola, de manera que ayude a despulgar vuestra implementación.