IC2001 Estructuras de Datos

Grupo 2

Jose Pablo Aguero Mora 2021126372

Katerine Guzmán Flores 2019390523

Manual técnico – Proyecto 3A

I Semestre 2022

Índice:

Manual técnico

Descripción:

Este proyecto constituye un programa que funciona como compresor y descompresor de archivos por un método que utiliza árboles de Huffman para agrupar conjuntos repetidos de información, generando de esta forma una composición por probabilidad de una nueva estructura de la información que establece un peso en el archivo menor al original.

Este programa está implementado de tal forma que puede ser utilizado desde la terminal del sistema operativo pasándole los parámetros que deseamos para la compresión o descompresión de los mismos.

De igual forma, se puede comprobar este proceso de minimización de peso si se verifica internamente las propiedades de los archivos originales y procesados, para confirmar de esta manera el correcto procedimiento.

A la hora de descomprimir un archivo se puede comprobar su estado en comparación con el original, así se establece una verificación base sobre el contenido que debería tener el nuevo archivo recuperado.

Structs utilizados:

Cada uno de los procesos tiene structs diferentes, en el caso de la compresión cuenta con dos de estas estructuras, una de ellas se utiliza en la creación del árbol (que originalmente funciona como una lista enlazada) y la otra en la base de la lista que genera la tabla de códigos binarios (codificación según carácter).

- Struct para árbol binario de compresión y lista enlazada inicial:

```
//////// Structs correspondientes del compresor ////////
typedef struct _nodo
{
    unsigned char letra;    /* Letra a la que hace referencia el nodo */
    int frecuencia;         /* veces que aparece la letra en el texto o las letras */
                            /* de los nodos de las ramas cero y uno */
    struct _nodo* sig;      /* Puntero a siguiente nodo de una lista enlazada */
    struct _nodo* cero;     /* Puntero a la rama cero de un árbol */
    struct _nodo* uno;      /* Puntero a la rama uno de un árbol */
} tipoNodo;                /* Nombre de tipo */
```

- Struct para lista usada en la tabla de códigos binarios (compresión):

```
/* Nodo para construir una lista para la tabla de codigos */
typedef struct _tabla
{
    unsigned char letra;    /* Letra a la que hace referencia el nodo */
    unsigned long int bits; /* Valor de la codificación de la letra */
    unsigned char nbits;    /* Número de bits de la codificación */
    struct _tabla* sig;     /* Siguiente elemento de la tabla */
} tipoTabla;               /* Nombre del tipo */
//////// Fin structs correspondientes del compresor ////////
```

- Struct para el árbol del descompresor:

```
//////// Structs correspondientes del descompresor ////////
typedef struct _nodo_D
{
    char letra;             /* Letra a la que hace referencia el nodo */
    unsigned long int bits; /* Valor de la codificación de la letra */
    char nbits;             /* Número de bits de la codificación */
    _nodo_D* cero;            /* Puntero a la rama cero de un árbol */
    _nodo_D* uno;             /* Puntero a la rama uno de un árbol */
} tipoNodo_D;               /* Nombre del tipo */
//////// Fin structs correspondientes del descompresor ////////
```

Implementación del main:

Para poder comprender los procesos que se llevan a cabo en el programa es necesario visualizar la forma en como se ejecuta el mismo, ya que está diseñado para ser una aplicación de consola que se utiliza por medio de comandos. Al ser de esta forma se debe establecer un uso correcto de los argumentos que entran al main.

Originalmente el main contiene un contador de argumentos y un arreglo de punteros que llevan a arreglos de char, esto es necesario para poder manejar cadenas de texto las cuales son los comandos recibidos al ejecutar el programa.

El programa se configura para que tenga una estructura definida mediante la cual entra información, esta es: nombre del programa + modo de procesamiento (compresión / descompresión) + nombre del archivo a procesar (con extensión) + nombre del nuevo archivo a crear (con extensión).

Por lo tanto, el main se encarga inicialmente de comprobar el segundo argumento, referente al tipo de procesamiento que se le va a dar al archivo de origen. De esta forma decide si debe ejecutar una compresión o descompresión (dependiendo si el parámetro es “e” o “d”). Una vez que toma la decisión ejecuta la función correspondiente.

```
////////// Función main que decide si se comprime o descomprime el archivo seleccionado ////////////
void main(int argc, char* argv[]) {
    char opcion = argv[1][0];

    if (opcion == 'e') {
        Comprimir(argc, argv);
        printf("Proceso de compresion finalizado");
    }
    else {
        Descomprimir(argc, argv);
        printf("Proceso de descompresion finalizado");
    }

}
```

```
               3 archivos      2 216 079 bytes
               2 dirs  660 247 293 952 bytes libres

D:\Proyectos C++\Proyecto 3A y 3B\Para documentar\3A\x64\Debug>3A.exe e texto.txt procesado.comp
Proceso de compresion finalizado
D:\Proyectos C++\Proyecto 3A y 3B\Para documentar\3A\x64\Debug>
```

Funcionalidad Comprimir:

Una de las primeras tareas por realizar es generar una lista de frecuencias o sistema de apariciones según el carácter para poder calcular qué tipo de caracteres tienen mayor presencia en el archivo.

Pero para empezar se deben inicializar las variables, la lista, el árbol y los punteros a utilizar para crear el árbol de Huffman y todas sus dependencias.

```
int Comprimir(int argc, char* argv[]) // Compresion
{
    tipoNodo* Lista;       /* Lista de letras y frecuencias */
    tipoNodo* Arbol;       /* Arbol de letras y frecuencias */

    FILE* fe, * fs;         /* Ficheros de entrada y salida */
    unsigned char c;       /* variables auxiliares */
    tipoNodo* p;
    tipoTabla* t;
    int nElementos;        /* Número de elementos en tabla */
    long int Longitud = 0; /* Longitud del fichero original */
```

Así como se puede observar, se genera una lista para guardar la frecuencia de apariciones por carácter y el árbol que va a ir actualizando cada simplificación.

Además, las variables que determinan el uso del archivo de origen y el de destino.

Variables auxiliares para guardar datos momentáneos y la longitud del primer archivo.

El primer paso importante es calcular las frecuencias según carácter, por lo que se guardan estos contadores de apariciones en “Lista” para posteriormente ser ordenados.

```
Lista = NULL;
/* Fase 1: contar frecuencias */
fopen_s(&fe, argv[2], "r");
do {
    c = fgetc(fe);
    if (feof(fe)) {
        break;
    }
    Longitud++;        /* Actualiza la cuenta de la longitud del fichero */
    Cuenta(&Lista, c); /* Actualiza la lista de frecuencias */
} while (1);
fclose(fe);
```

Así como se observa en la imagen se abre el archivo de origen (el cual representa el tercer argumento) y se toma un carácter leído, con esto se va aumentando el contador de longitud además de que se llama a la subrutina Cuenta, la cual se encarga de generar la frecuencia.

Dentro de cuenta se realiza un proceso inicial para determinar cómo actuar si la lista de frecuencias está vacía (es decir, cuando registra elemento en lista por primera vez).

```
void Cuenta(tipoNodo** Lista, unsigned char c)
{
    tipoNodo* p, * a, * q;

    if (!*Lista)  /* Si la lista está vacía, el nuevo nodo será Lista */
    {
        *Lista = (tipoNodo*)malloc(sizeof(tipoNodo));   /* Un nodo nuevo */
        (*Lista)->letra = c;                             /* Para c */
        (*Lista)->frecuencia = 1;                        /* en su 1ª aparición */
        (*Lista)->sig = (*Lista)->cero = (*Lista)->uno = NULL;
    }
    else
    {
```

De esta forma se crea un nuevo nodo como se puede ver, usando la letra encontrada y dejando su frecuencia en 1.

Ahora el proceso consiste en aumentar el contador de frecuencia según la cantidad de apariciones que el programa ha ido registrando.

```
    else
    {
        /* Buscar el caracter en la lista (ordenada por letra) */
        p = *Lista;
        a = NULL;
        while (p && p->letra < c)
        {
            a = p;      /* Guardamos el elemento actual para insertar */
            p = p->sig; /* Avanzamos al siguiente */
        }
        /* Dos casos: */
        /* 1) La letra es c se encontró */
        if (p && p->letra == c) p->frecuencia++; /* Actualizar frecuencia */
        else
            /* 2) La letra c no se encontró*/
        {
            /* Insertar un elemento nuevo */
            q = (tipoNodo*)malloc(sizeof(tipoNodo));
            q->letra = c;
            q->frecuencia = 1;
            q->cero = q->uno = NULL;
            q->sig = p;        /* Insertar entre los nodos p */
            if (a) a->sig = q;  /* y a */
            else *Lista = q;    /* Si a es NULL el nuevo es el primero */
        }
    }
}
```

Se va recorriendo la lista y si el elemento actual es igual al parámetro c (que es la unidad que la función está buscando) entonces procede a aumentar el contador en 1 (representando la frecuencia de aparición en el archivo).

Y en caso de que no lo encuentre es porque representa un carácter que no ha visto antes, por lo que es necesario crear un nuevo nodo que quede listo para las siguientes iteraciones donde podrían encontrar elementos iguales a él para seguir aumentando las frecuencias.

Una vez que finaliza este proceso obtenemos una lista enlazada simple con las frecuencias listas (este proceso de “Cuenta”, termina cuando ya se han evaluado todos los elementos del archivo de origen).

Pero antes de generar el árbol con cada uno de estos nodos de frecuencia es necesario ordenar la lista de menor a mayor según ese mismo número de apariciones.

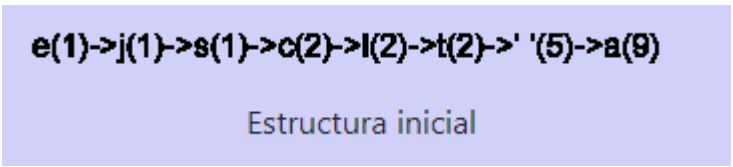
```
/* Ordenar la lista de menor a mayor */
Ordenar(&Lista);
```

```
/* Ordena Lista de menor a mayor por frecuencias */
/* De nuevo pasamos el puntero a lista por referencia */
void Ordenar(tipoNodo** Lista)
{
    tipoNodo* Lista2, * a;

    if (!*Lista) return; /* Lista vacia */
    Lista2 = *Lista;
    *Lista = NULL;
    while (Lista2)
    {
        a = Lista2;             /* Toma los elementos de Lista2 */
        Lista2 = a->sig;
        InsertarOrden(Lista, a); /* Y los inserta por orden en Lista */
    }
}
```

Se realiza una verificación de que la lista no esté vacía, a partir de aquí se hace una copia de la lista en una auxiliar, ya que la lista original se limpia (se declara en NULL) y va tomando elemento por elemento para ingresarlo en la función “InsertarOrden” que se encarga de verificar los tamaños de frecuencia y reordenar la lista según eso.

```c
/* Inserta el elemento e en la Lista ordenado por frecuencia de menor a mayor */
/* El puntero a Cabeza se pasa por referencia */
void InsertarOrden(tipoNodo** Cabeza, tipoNodo* e)
{
    tipoNodo* p, * a;

    if (!*Cabeza) /* Si Cabeza es NULL, es el primer elemento */
    {
        *Cabeza = e;
        (*Cabeza)->sig = NULL;
    }
    else
    {
        /* Buscar el caracter en la lista (ordenada por frecuencia) */
        p = *Cabeza;
        a = NULL;
        while (p && p->frecuencia < e->frecuencia)
        {
            a = p;      /* Guardamos el elemento actual para insertar */
            p = p->sig; /* Avanzamos al siguiente */
        }
        /* Insertar el elemento */
        e->sig = p;
        if (a) a->sig = e;   /* Insertar entre a y p */
        else *Cabeza = e;    /* el nuevo es el primero */
    }
}
```

Se van tomando pares de nodos (“e” y “p” en donde “e” es el actual) y se verifica cuál es mayor y cuál es menor (según la frecuencia) para colocarlo justo en medio de ambos. Este proceso se repite con todos los elementos de la lista auxiliar (que estaba desordenada) hasta que se acaben los nodos y quede finalmente la nueva lista ordenada.

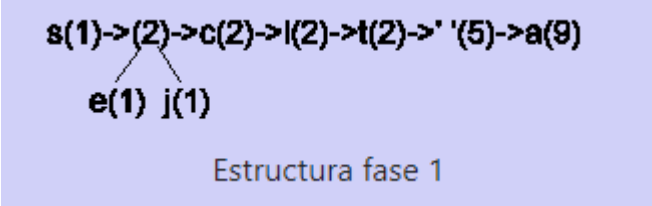


Estructura inicial

Con esto se crea el árbol inicial a partir de la nueva lista.

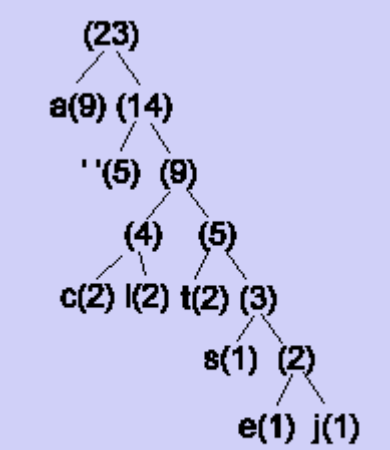
```
/* Crear el arbol */
Arbol = Lista;
while (Arbol && Arbol->sig) /* Mientras existan al menos dos elementos en la lista */
{
    p = (tipoNodo*)malloc(sizeof(tipoNodo)); /* Un nuevo árbol */
    p->letra = 0;                            /* No corresponde a ninguna letra */
    p->uno = Arbol;                          /* Rama uno */
    Arbol = Arbol->sig;                      /* Siguiente nodo en */
    p->cero = Arbol;                         /* Rama cero */
    Arbol = Arbol->sig;                      /* Siguiente nodo */
    p->frecuencia = p->uno->frecuencia +
        p->cero->frecuencia;      /* Suma de frecuencias */
    InsertarOrden(&Arbol, p);                /* Inserta en nuevo nodo */
}                                           /* orden de frecuencia */
```

Se acomodan los dos primeros nodos en diferentes ramas del árbol y al final se vuelve a llamar a InsertarOrden para reordenar los elementos restantes. De esta forma se obtiene la siguiente estructura.



Estructura fase 1

Recordemos que este proceso se encuentra en un ciclo que se repite hasta que se reordene toda la antigua lista en el nuevo árbol de la misma forma, es decir tomando pares de nodos y comparando sus frecuencias hasta que todos estén en una rama distinta.



Estructura final

Cuando esto se cumple es necesario asignarle un bit 0 a las ramas de la izquierda y un bit 1 a los de la derecha, para esto se requiere una tabla de códigos binarios que se va a asignar con el nuevo árbol ordenado.

```
/* Construir la tabla de códigos binarios */
Tabla = NULL;
CrearTabla(Arbol, 0, 0);
```

```
/* Función recursiva para crear Tabla */
/* Recorre el árbol cuya raiz es n y le asigna el código v de l bits */
void CrearTabla(tipoNodo* n, int l, int v)
{
    if (n->uno)  CrearTabla(n->uno, l + 1, (v << 1) | 1);
    if (n->cero) CrearTabla(n->cero, l + 1, v << 1);
    if (!n->uno && !n->cero) InsertarTabla(n->letra, l, v);
}
```

Se inicializa la tabla en NULL y entra a su función recursiva que decide nodo por nodo del árbol la selección de bits. Si tiene un elemento en la rama del 1 entonces sigue recorriendo el árbol, pero aumenta el contador de bits y en este caso se utiliza un OR con un corrimiento de bits a la izquierda en donde siempre va a dar 1.

En el caso de que tenga un nodo en la referencia de la rama del cero realiza lo mismo, pero avanza un nodo por esa rama, igualmente aumenta el contador de bits en 1 y hace un corrimiento de bits a la izquierda.

Finalmente, la tercera condición es para los nodos hoja (que no tienen más ramas), cuando se llega a estos se insertan en la tabla con los índices calculados, el contador de bits, la letra que representa esos datos y la combinación de bits.

InsertarTabla funciona muy similar a InsertarOrden solo que esta vez realiza una copia de los datos del nodo y verifica si ya existe un nodo así en la tabla, recordemos que esta tabla inicialmente solo estaba inicializada en NULL.

```c
/* Insertar un elemento en la tabla */
void InsertarTabla(unsigned char c, int l, int v)
{
    tipoTabla* t, * p, * a;

    t = (tipoTabla*)malloc(sizeof(tipoTabla)); /* Crea un elemento de tabla */
    t->letra = c;                              /* Y lo inicializa */
    t->bits = v;
    t->nbits = l;

    if (!Tabla)         /* Si tabla es NULL, entonces el elemento t es el 1° */
    {
        Tabla = t;
        Tabla->sig = NULL;
    }
    else
    {
        /* Buscar el caracter en la lista (ordenada por frecuencia) */
        p = Tabla;
        a = NULL;
        while (p && p->letra < t->letra)
        {
            a = p;      /* Guardamos el elemento actual para insertar */
            p = p->sig; /* Avanzamos al siguiente */
        }
        /* Insertar el elemento */
        t->sig = p;
        if (a) a->sig = t;  /* Insertar entre a y p */
        else Tabla = t;    /* el nuevo es el primero */
    }
}
```

Si no está registrada esa letra entonces se inserta como primera, de no ser así (en la sección del else) busca el nodo directamente menor (estaba ordenado por frecuencia o apariciones). Al inicio de la función se editan los valores de t (un nodo auxiliar que será ingresado a la tabla) con los valores de bits que se calcularon en CrearTabla, se hace una copia de estos y se prepara a la estructura para el ingreso de un nuevo elemento.

```c
/* Crear fichero comprimido */
fopen_s(&fs, argv[3], "wb");
/* Escribir la longitud del fichero */
fwrite(&Longitud, sizeof(long int), 1, fs);
/* Cuenta el número de elementos de tabla */
nElementos = 0;
t = Tabla;
while (t)
{
    nElementos++;
    t = t->sig;
}
/* Escribir el número de elemenos de tabla */
fwrite(&nElementos, sizeof(int), 1, fs);
```

A partir de aquí ya la información está lista, solo queda escribir toda esta estructura en el archivo de destino. Por lo tanto, se crea un nuevo archivo con el nombre del cuarto parámetro ingresado y se prepara con la cantidad de elementos de la tabla, para esto se recorre la misma con un contador hasta que llegue al final de la misma.

```
/* Escribir tabla en fichero */
t = Tabla;
while (t)
{
    fwrite(&t->letra, sizeof(char), 1, fs);
    fwrite(&t->bits, sizeof(unsigned long int), 1, fs);
    fwrite(&t->nbits, sizeof(char), 1, fs);
    t = t->sig;
}
```

De esta forma se hace una copia de la tabla y se va recorriendo nodo por nodo, se escribe la letra, el código binario y la cantidad de bits, este representa el encabezado necesario para luego poder descomprimir el archivo. Esto solo representa parte de la información para poder recomponerlo es por eso que el peso del archivo comprimido es menor. Parte de esta información necesaria para descomprimir es la codificación.

```
/* Codificación del fichero de entrada */
fopen_s(&fe, argv[2], "r");
dWORD = 0; /* Valor inicial. */
nBits = 0; /* Ningún bit */
do {
    c = fgetc(fe);
    if (feof(fe)) {
        break;
    }
    /* Busca c en tabla: */
    t = BuscaCaracter(Tabla, c);
    /* Si nBits + t->nbits > 32, sacar un byte */
    while (nBits + t->nbits > 32)
    {
        c = dWORD >> (nBits - 8);          /* Extrae los 8 bits de mayor peso */
        fwrite(&c, sizeof(char), 1, fs);  /* Y lo escribe en el fichero */
        nBits -= 8;                       /* Ya tenemos hueco para 8 bits más */
    }
    dWORD <<= t->nbits; /* Hacemos sitio para el nuevo caracter */
    dWORD |= t->bits;   /* Insertamos el nuevo caracter */
    nBits += t->nbits;  /* Actualizamos la cuenta de bits */
} while (1);
/* Extraer los cuatro bytes que quedan en dWORD*/
while (nBits > 0)
{
    if (nBits >= 8) c = dWORD >> (nBits - 8);
    else c = dWORD << (8 - nBits);
    fwrite(&c, sizeof(char), 1, fs);
    nBits -= 8;
}
```

Como se puede ver va recorriendo el archivo original y cada carácter que encuentra lo procesa mediante la función BuscaCaracter.

```
/* Buscar un caracter en la tabla, devielve un puntero al elemento de la tabla */
tipoTabla* BuscaCaracter(tipoTabla* Tabla, unsigned char c)
{
    tipoTabla* t;

    t = Tabla;
    while (t && t->letra != c) t = t->sig;
    return t;
}
```

Esta recibe la tabla de códigos binarios y el carácter del archivo original, de esta forma busca el nodo de este carácter en la tabla y devuelve una referencia del mismo.

Y mediante un ciclo while se extraen los 8 bits más pesados de ese nodo. En las últimas líneas del ciclo se puede ver cómo se utiliza el corrimiento de bits para preparar a las variables auxiliares para el siguiente nodo.

En la variable dWORD se corren los bits hacia la izquierda la cantidad que esté registrada en el nodo calculado de la tabla (para guardar el espacio necesario). Luego con el operador OR agrega ese carácter y por último a la cantidad de bits se le suma lo que valga el carácter ingresado.

Usando este método, al final quedan 4 bytes en la variable auxiliar por lo que se va pasando uno por uno al archivo de destino y se van limpiando los originales.

```
fclose(fe);  /* Cierra los ficheros */
fclose(fs);

/* Borrar Arbol */
BorrarArbol(Arbol);

/* Borrar Tabla */
while (Tabla)
{
    t = Tabla;
    Tabla = t->sig;
    free(t);
}

return 0;
```

Finalmente, se cierra el proceso de los dos archivos, se borra el contenido del árbol y se borran todos los elementos de la tabla de códigos binarios, esto para liberar el uso en memoria dinámica. Con esto finaliza el algoritmo de compresión, lo cual nos deja con un archivo de destino con el peso reducido.

Funcionalidad descomprimir:

Para descomprimir el fichero original es necesario conocer el código asignado a cada carácter, así como su longitud en bits. Si esta información sobre la codificación no se incluye en el archivo de destino el programa puede servir como un sistema de encriptación, ya que solo podrían visualizar los datos los usuarios que tengan acceso a la manera como se descompuso el archivo original.

En este caso como se está trabajando con un compresor/descompresor es vital incluir la codificación. Este proceso se detalla al final de la sección “Funcionalidad comprimir”.

```
/////// Función principal para descomprimir archivos ///////
int Descomprimir(int argc, char* argv[]) // Descomprimir
{
    tipoNodo_D* Arbol;        /* Arbol de codificación */
    long int Longitud;       /* Longitud de fichero */
    int nElementos;          /* Elementos de árbol */
    unsigned long int bits; /* Almacen de bits para decodificación */
    FILE* fe, * fs;          /* Ficheros de entrada y salida */

    tipoNodo_D* p, * q;       /* Auxiliares */
    unsigned char a;
    int i, j;
```

Al igual que en el caso de compresión, inicialmente se declaran los punteros y las variables que dan paso a cada una de las estructuras usadas durante la ejecución.

```
/* Crear un arbol con la información de la tabla */
Arbol = (tipoNodo_D*)malloc(sizeof(tipoNodo_D)); /* un nodo nuevo */
Arbol->letra = 0;
Arbol->uno = Arbol->cero = NULL;
fopen_s(&fe, argv[2], "rb");
fread(&Longitud, sizeof(long int), 1, fe); /* Lee el número de caracteres */
fread(&nElementos, sizeof(int), 1, fe); /* Lee el número de elementos */
```

El árbol se inicializa con valores por defecto, como la letra en 0 y las ramas en NULL, además de eso se abre el archivo de origen (que en este caso está comprimido) y se obtiene la cantidad de caracteres y elementos.

```
for (i = 0; i < nElementos; i++) /* Leer todos los elementos */
{
    p = (tipoNodo_D*)malloc(sizeof(tipoNodo_D)); /* un nodo nuevo */
    fread(&p->letra, sizeof(char), 1, fe); /* Lee el carácter */
    fread(&p->bits, sizeof(unsigned long int), 1, fe); /* Lee el código */
    fread(&p->nbits, sizeof(char), 1, fe); /* Lee la longitud */
    p->cero = p->uno = NULL;
```

Se crea un ciclo para ir leyendo cada elemento del archivo y en cada elemento se saca la letra asociada al nodo, la cantidad de bits y el código binario que lo representa, esto se ingresa en un nuevo nodo interno del ciclo for.

```
    /* Insertar el nodo en su lugar */
    j = 1 << (p->nbits - 1);
    q = Arbol;
    while (j > 1)
    {
        if (p->bits & j) /* es un uno*/
            if (q->uno) q = q->uno;   /* Si el nodo existe, nos movemos a él */
            else                      /* Si no existe, lo creamos */
            {
                q->uno = (tipoNodo_D*)malloc(sizeof(tipoNodo_D)); /* un nodo nuevo */
                q = q->uno;
                q->letra = 0;
                q->uno = q->cero = NULL;
            }
        else /* es un cero */
            if (q->cero) q = q->cero; /* Si el nodo existe, nos movemos a él */
            else                      /* Si no existe, lo creamos */
            {
                q->cero = (tipoNodo_D*)malloc(sizeof(tipoNodo_D)); /* un nodo nuevo */
                q = q->cero;
                q->letra = 0;
                q->uno = q->cero = NULL;
            }
        j >>= 1;  /* Siguiente bit */
    }
    /* Ultimo Bit */
    if (p->bits & 1) /* es un uno*/
        q->uno = p;
    else           /* es un cero */
        q->cero = p;
}
```

En el valor de j se hace un corrimiento de bits según el código binario del nodo actual. Se verifica si el bit actual tiene asociado un “1” o un “0” y si ya existe un nodo de este tipo se omite su creación y nada más lo localiza. Tal proceso se repite con todos, incluyendo el último bit y a su vez se repite con todos los elementos de los nodos.

```
/* Leer datos comprimidos y extraer al fichero de salida */
bits = 0;
fopen_s(&fs, argv[3], "w");
/* Lee los primeros cuatro bytes en la dobel palabra bits */
fread(&a, sizeof(char), 1, fe);
bits |= a;
bits <<= 8;
fread(&a, sizeof(char), 1, fe);
bits |= a;
bits <<= 8;
fread(&a, sizeof(char), 1, fe);
bits |= a;
bits <<= 8;
fread(&a, sizeof(char), 1, fe);
bits |= a;
j = 0; /* Cada 8 bits leemos otro byte */
q = Arbol;
```

Ahora se procede a abrir el nuevo archivo con el nombre incluido en el cuarto parámetro a la hora de ejecutar el programa. Se leen los 4 primeros bytes del archivo comprimido (que incluyen información importante de la codificación) mediante una variable auxiliar “a” que ayuda a insertar a la cadena de bits real mediante el operador “|=” y al finalizar con un grupo de 8 bits (un byte), pasa al siguiente (esto con los primeros 4).

```
/* Bucle */
do {
    if (bits & 0x80000000) q = q->uno; else q = q->cero; /* Rama adecuada */
    bits <<= 1;           /* Siguiente bit */
    j++;
    if (8 == j)           /* Cada 8 bits */
    {
        i = fread(&a, sizeof(char), 1, fe); /* Leemos un byte desde el fichero */
        bits |= a;                  /* Y lo insertamos en bits */
        j = 0;                      /* No quedan huecos */
    }
    if (!q->uno && !q->cero)        /* Si el nodo es una letra */
    {
        putc(q->letra, fs);         /* La escribimos en el fich de salida */
        Longitud--;                 /* Actualizamos longitud que queda */
        q = Arbol;                    /* Volvemos a la raiz del árbol */
    }
} while (Longitud);                /* Hasta que acabe el fichero */
/* Procesar la cola */
```

Finalmente se realiza un ciclo según la longitud del archivo comprimido en donde primero se verifica la rama adecuada en la cual posicionarse según las cadenas de bits registradas previamente. A partir de aquí, los procesos internos se realizan en bloques de bytes, por lo tanto, se usa un contador base llamado “j” para verificar cuando ya se ha leído un byte completo y si es así entonces lee uno nuevo desde el fichero, agrega el código binario a la cadena y reinicia el contador de bits de nuevo.

Por otro lado, si el nodo actual se reconoce como una letra entonces se escribe directamente en el archivo de destino (que va a ser el archivo recompuesto), además resta en uno la longitud (para que el ciclo tenga un punto de parada cuando se acaben los elementos del fichero comprimido) y se devuelve el puntero q a la raíz del árbol para repetir el mismo proceso. De esta forma cada letra se va agregando de forma ordenada al archivo de destino.

```
    fclose(fs);                         /* Cerramos ficheros */
    fclose(fe);

    BorrarArbol_D(Arbol);                /* Borramos el árbol */
    return 0;
}
```

Al igual que en el caso de la compresión se cierran los dos archivos utilizados y se borra el árbol de reconstrucción para liberar la memoria utilizada.

Base de los algoritmos:

La base original de los algoritmos fue tomada del trabajo de Salvador Pozo Coronado para su debida adaptación:

Pozo, S. (2021). Compresión de Huffman. Tomado de: https://conclase.net/blog/item/huffman