INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Estructuras de Datos



Practica 6 – Codificación de Huffman

Prof.: Franco Martínez Edgardo Adrián

Integrantes:

Mendoza Parra Sergio.

Salcedo Barrón Rubén Osmair.

Tejeda Martínez José Miguel.

MEXICO, D.F. a 12 de Enero del 2016

# INTRODUCCIÓN

El algoritmo de Huffman es un algoritmo para la construcción de códigos de Huffman, desarrollado por David A. Huffman en 1952 y descrito en A Method for the Construction of Minimum Redundancy Codes. [1]

Este algoritmo toma un alfabeto de n símbolos, junto con sus frecuencias de aparición asociadas, y produce un código de Huffman para ese alfabeto y esas frecuencias. Huffman propuso un algoritmo que obtiene una codificación prefija óptima, para ello construye un árbol binario de códigos de longitud variable de manera ascendente.

El algoritmo funciona de la siguiente manera:

• Parte de una secuencia inicial en la que los caracteres a codificar están colocados en orden creciente de frecuencia.

• Esta secuencia inicial se va transformando, a base de fusiones, hasta llegar a una secuencia con un único elemento que es el árbol de codificación óptimo.

CODIFICACIÓN

1. Se crean varios árboles, uno por cada uno de los símbolos del alfabeto, consistiendo cada uno de los árboles en un nodo sin hijos, y etiquetado cada uno con su símbolo asociado y su frecuencia de aparición.

2. Se toman los dos árboles de menor frecuencia, y se unen creando un nuevo árbol. La etiqueta de la raíz será la suma de las frecuencias de las raíces de los dos árboles que se unen, y cada uno de estos árboles será un hijo del nuevo árbol. También se etiquetan las dos ramas del nuevo árbol: con un 0 la de la izquierda, y con un 1 la de la derecha.

3. Se repite el paso 2 hasta que sólo quede un árbol.

DECODIFICACIÓN

1. A partir del árbol de codificación, comenzar a recorrer los caminos según los bits de la codificación. Al llegar a un nodo hoja se toma el valor de esta y coloca en el archivo original.

2. Se repite el paso 1 a partir del bit siguiente de la codificación comenzando un nuevo recorrido a partir de la raíz del árbol de la codificación.

3. La decodificación termina una vez se hallan recorrido todos los bits de la codificación.

# PLANTEAMIENTO DEL PROBLEMA

Las actividades a desarrollar se plantean en los siguientes puntos:

1. Entender el algoritmo de codificación voraz de Huffman e implementarlo en lenguaje C para codificar un archivo de texto.

2. Documentar y explicar el algoritmo y la implementación de este en C.

3. Construir el programa que sea capaz de reconstruir el archivo codificado a su codificación original.

# ALGORITMOS

Con base en el planteamiento del problema, se desarrolló la implementación del algoritmo de Huffman, basándonos en el pseudocódigo para la codificación y decodificación de la información:

PSEUDOCÓDIGO

ALGORITMO DE CODIFICACIÓN DE HUFFMAN

**función [código]= Huffman ( P )**

**P** = vector de probabilidades como entrada

**código** = tabla vacía inicialmente

Si (longitud de P <2) OR (Sum(P)!= 1)

error y sale

fin

**P\_ordenada** = ordena de mayor a menor(P)

Si la longitud es 2

asigna 0 y 1 según criterio

**código** = almacena probabilidad y código asignado en columnas adecuadas

en caso contrario

**P\_ordenada** = sustituye las dos 2 probabilidades menores por la suma de estas

**código** = almacena probabilidad y código asignado para cada probabilidad de las sumadas, en sus respectivas columnas

**Huffman (P\_ordenada)** comienza de nuevo con el nuevo vector ordenado

fin

CODIFICADOR HUFFMAN

**función [secuencia]= HuffmanCodificador ( informacion )**

**P** = calcula vector de probabilidades de **informacion**

**Sum** = suma de las probabilidades es cero inicialmente

para i=1 hasta fin del vector de **P**

**sum=sum+P(i)**

si **sum** > 1

error y sale

fin

fin

si **sum** es < 1

error y sale

fin

**código** dado por: huffman de P

**código (tabla)** = código + columna de símbolos asociados a cada probabilidad

**secuencia** = vacia

para i=1 hasta fin del vector de **informacion**

**simbolo** = busca en código el simbolo correspondiente a **información** (i)

**secuencia = [secuencia simbolo]**

fin

DECODIFICADOR HUFFMAN

**función [información]= HuffmanRecursivo ( secuencia, código )**

almacenado = vacio inicialmente, para almacenar bits recibidos

informacion = vacia inicialmente

para j=1 hasta fin de secuencia

almacenado = [almacenado secuencia(j)]

para i=1 hasta fin de la columna de códigos de código

si almacenado = codigo(columna de codigos,i)

informacion = [informacion codigo(columna de simbolos,i)]

fin

fin

fin

# IMPLEMENTACIÓN DE LOS ALGORITMOS

Diagrama de ejecución del programa:

Para codificar el texto fue necesario hacer uso de estructuras definidas por el usuario, con lo cual pudiésemos guardar la cantidad de apariciones de un carácter en el texto (frecuencias) para asi mismo definir los vectores de probabilidades, ordenarlos mezclar los árboles binarios.

typedef struct \_nodo

{

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

unsigned long int bits; // Valor de la codificación de la letra

char nbits; // Número de bits de la codificación

\_nodo \*sig; //Puntero a siguiente nodo de una lista enlazada

\_nodo \*cero; //Puntero a la rama cero de un árbol

\_nodo \*uno; //Puntero a la rama uno de un árbol

} tipoNodo;

Un árbol binario es un una serie de nodos que parten de una raíz, donde cada raíz puede albergar a lo mucho dos hijos que estos a su vez se comportan como raíces o semi-árboles de sus descendientes, en este caso cada uno se constituye por lo siguiente:

typedef struct \_nodo

{

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

unsigned long int bits; // Valor de la codificación de la letra

char nbits; // Número de bits de la codificación

\_nodo \*sig; //Puntero a siguiente nodo de una lista enlazada

\_nodo \*cero; //Puntero a la rama cero de un árbol

\_nodo \*uno; //Puntero a la rama uno de un árbol

} tipoNodo;

Como podemos observar, las partes esenciales en nuestro nodo son la frecuencia, nbits y la letra correspondiente a cada uno de ellos, de igual forma, para el almacenamiento de la codificación se usaron listas especiales, constituyéndose de la siguiente manera:

typedef struct \_tabla

{

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

\_tabla \*sig; //Siguiente elemento de la tabla

} tipoTabla;

De cierto modo, la lista de igual se conforma por nodos, los cuales guardan el número de bits de la codificación, lo cual servirá al momento de realizar el proceso de codificación y decodificación.

void Cuenta(tipoNodo\* &Lista, char c)

{

tipoNodo \*p, \*a, \*q;

if(!Lista) // Si la lista está vacía, el nuevo nodo será Lista

{

…

}

else

{

// Buscar el caracter en la lista (ordenada por letra)

…

// 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

… }

}

}

Con la funcion cuenta se almacenaron las frecuencias de aparición de cada carácter, usando la lista definida para determinar la longitud de la codificación. Las funciones ordenar e insertar orden en el programa sirvieron para ordenar las frecuencias de menor a mayor y asi iniciar con el proceso de codificación o decodificación, según sea el caso.

int Codificar(char \*in,char \*out)

{

tipoNodo \*Lista; // Lista de letras y frecuencias

tipoNodo \*Arbol; // Arbol de letras y frecuencias

…

// Fase 1: contar frecuencias

fe = fopen(in, "r");

while((c = fgetc(fe)) != EOF)

{

Longitud++; // Actualiza la cuenta de la longitud del fichero

Cuenta(Lista, c); // Actualiza la lista de frecuencias

}

fclose(fe);

// Ordenar la lista de menor a mayor

Ordenar(Lista);

// Crear el arbol

…

// Construir la tabla de códigos binarios

Tabla = NULL;

CrearTabla(Arbol, 0, 0);

// Crear fichero comprimido

fs = fopen(out, "wb");

// Escribir la longitud del fichero

fwrite(&Longitud, sizeof(long int), 1, fs);

…

// Escribir el número de elemenos de tabla

fwrite(&nElementos, sizeof(int), 1, fs);

// Escribir tabla en fichero

…

// Codificación del fichero de entrada

fe = fopen(in, "r");

dWORD = 0; // Valor inicial.

nBits = 0; // Ningún bit

while((c = fgetc(fe)) != EOF)

{

// Busca c en tabla:

t = BuscaCaracter(Tabla, c);

// Si nBits + t->nbits > 32, sacar un byte

while(nBits + t->nbits > 32)

{

…

}

// Extraer los cuatro bytes que quedan en dWORD

while(nBits>0)

{

…

}

...

}

La parte esencial en la codificación es el uso de la tabla para generar los bits a grabar en el archivo; nótese que para comenzar con la compresión de la información en el archivo siempre se guardará la longitud de los bits a codificar sizeof(long int), posteriormente se realiza a búsqueda en la tabla de frecuencias, de este modo si hay una aparición del carácter correspondiente (nBits + t->nbits > 32) se obtiene un byte guardándolo en el archivo. Como hacemos uso de octales en bits, es necesario obtener 4 bytes más para que de esa forma se complete la codificación de árbol ideal, lo cual es la clave para la función inversa.

int Decodificar(char \*in,char \*out)

{

...

// Crear un arbol con la información de la tabla

Arbol = (tipoNodo \*)malloc(sizeof(tipoNodo)); // un nodo nuevo

...

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

{

...

else // es un cero

if(q->cero) q = q->cero; // Si el nodo existe, nos movemos a él

else // Si no existe, lo creamos

{

...

}

j >>= 1; // Siguiente bit

}

// Ultimo Bit

if(p->bits & 1) // es un uno

q->uno = p;

else // es un cero

q->cero = p;

}

// Leer datos comprimidos y extraer al fichero de salida

// Bucle de lectura

do {

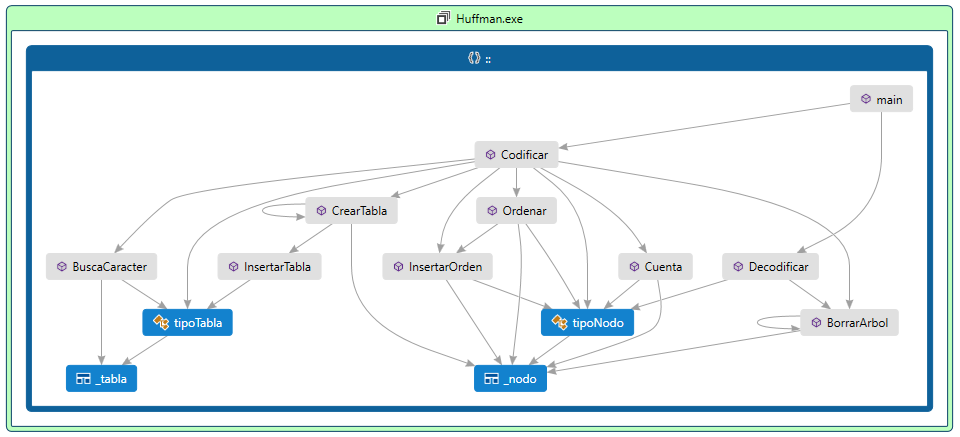
if(bits & 0x80000000) q = q->uno; else q = q->cero; // Rama adecuada

...

}

El proceso de decodificación hace uso de la longitud de la codificación y la frecuencia de apariciones, reconstruyendo el árbol binario. La lectura de la información codificada se realiza mediante octales, comprobando que nuestros auxiliares de cuenta coincidan con la codificación “uno” o “cero” que se almacenan en los nodos del árbol; de esta forma con el bucle de lectura podemos viajar hasta la rama adecuada y recuperar la letra o carácter correspondiente.

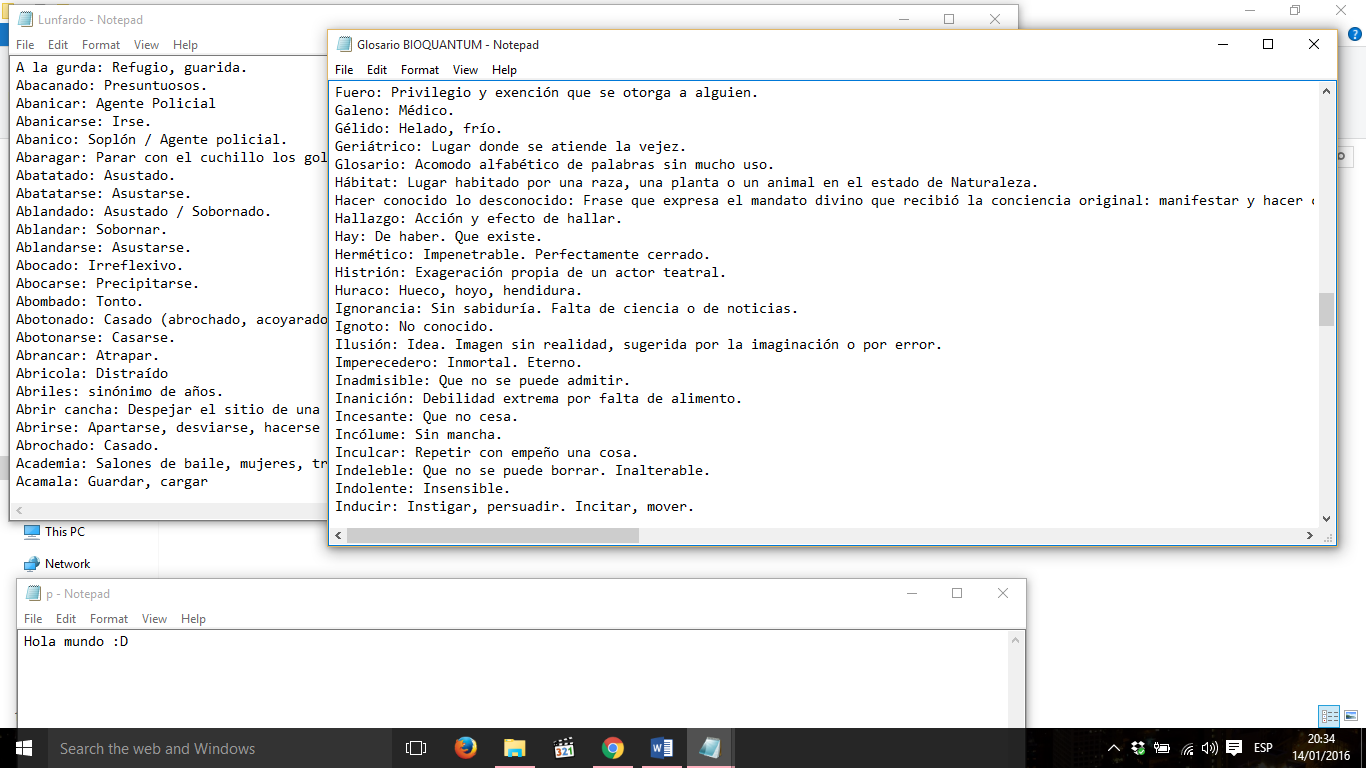
Con todo lo anterior, la implementación de la codificación y decodificación se ilustra en el siguiente diagrama. Las funciones codificar y decodificar son las que hacen el manejo de los archivos.



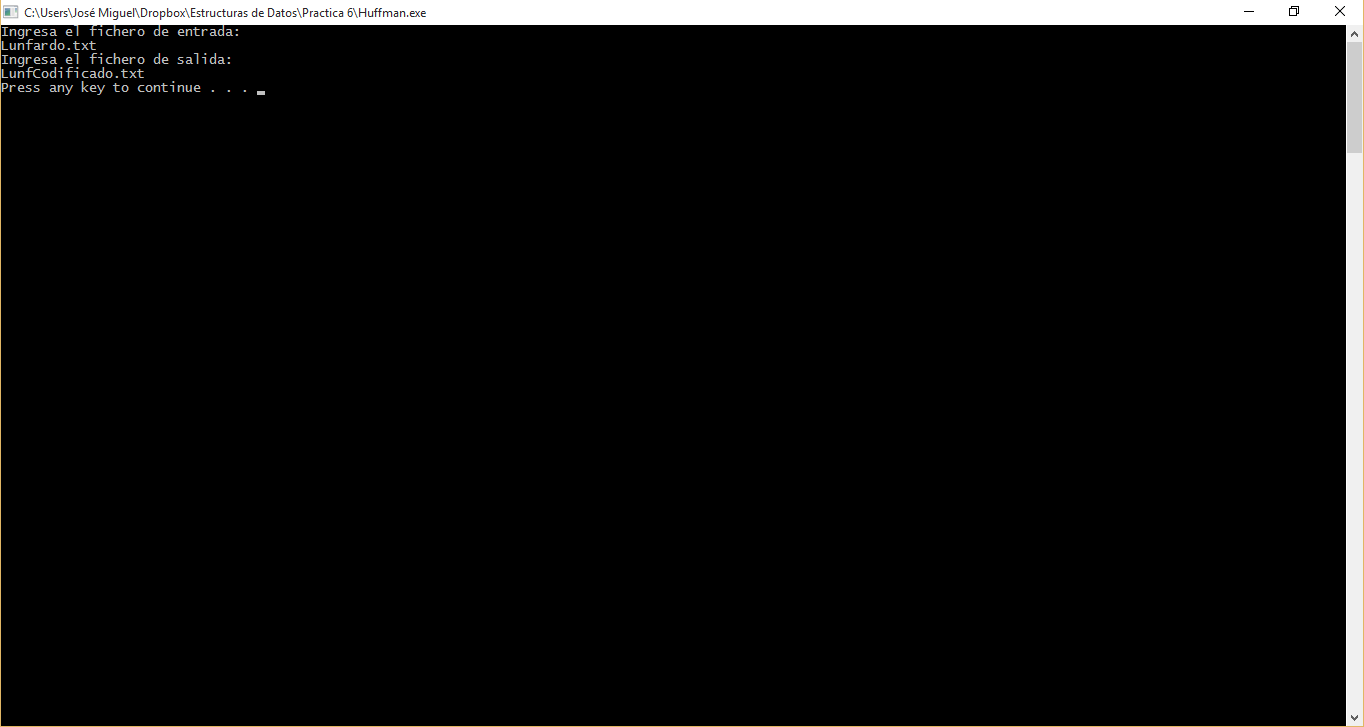
# ACTIVIDADES Y PRUEBAS

Para comprobar el funcionamiento del programa, fue necesario codificar archivos y decodificarlos.

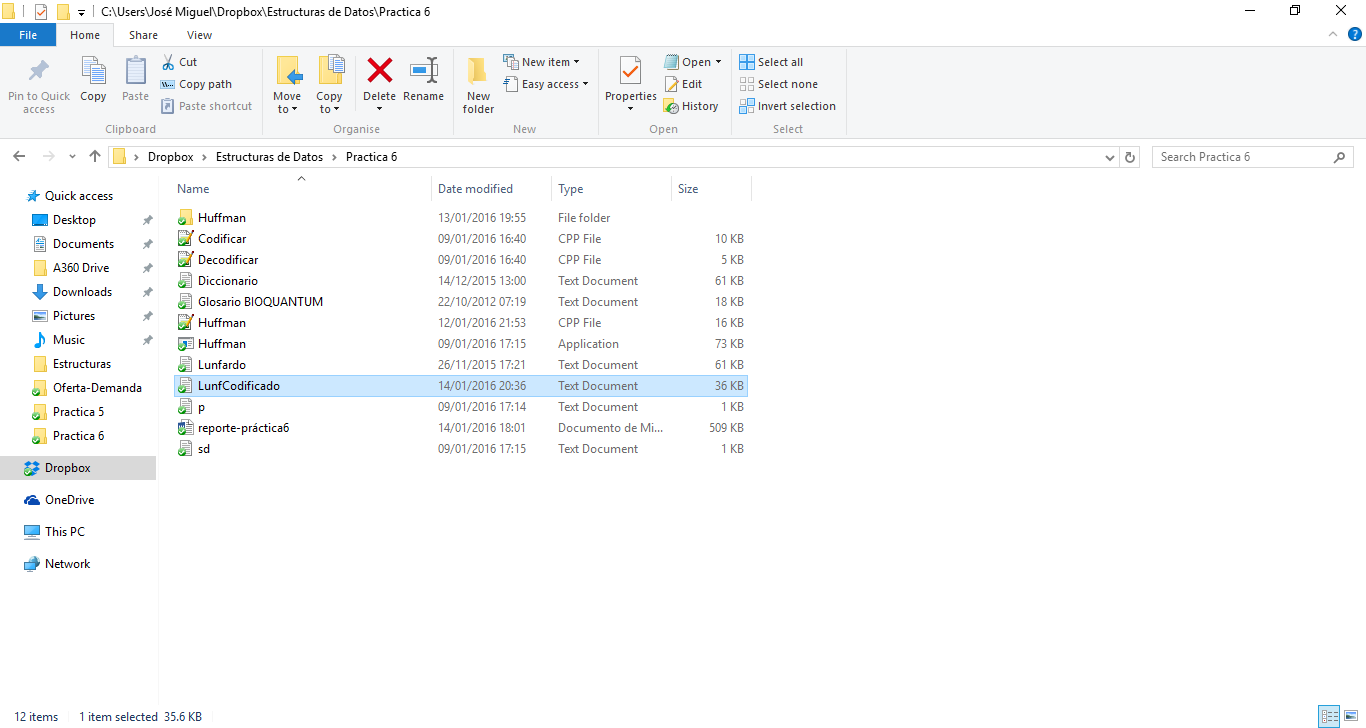
En la carpeta donde tenemos los ejecutables hemos almacenado archivos de texto de diferentes extensiones y con gran variedad de información:

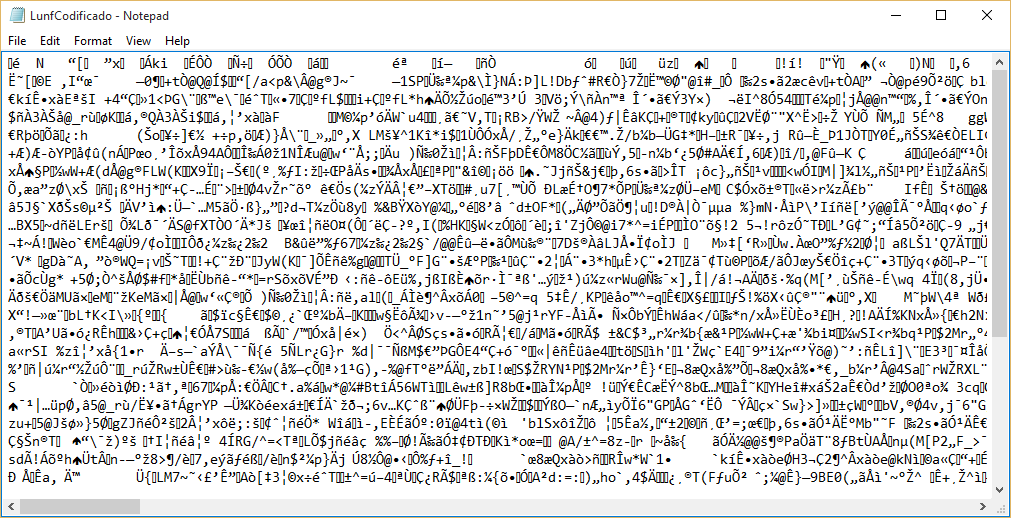


Como ejemplo ilustrativo mostraremos la codificación en el programa de alguno de estos archivos



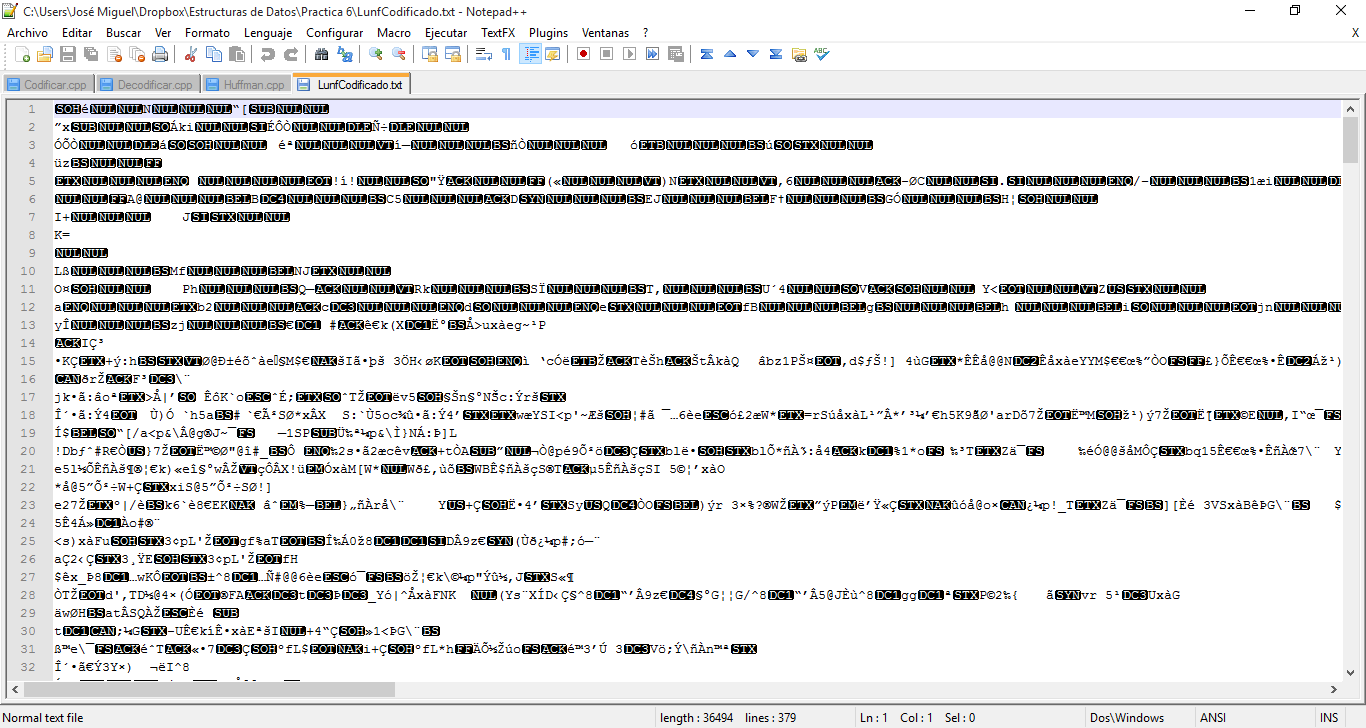
Como observamos, el fichero se guarda correctamente en la carpeta y si lo abrimos tenemos la información codificada:



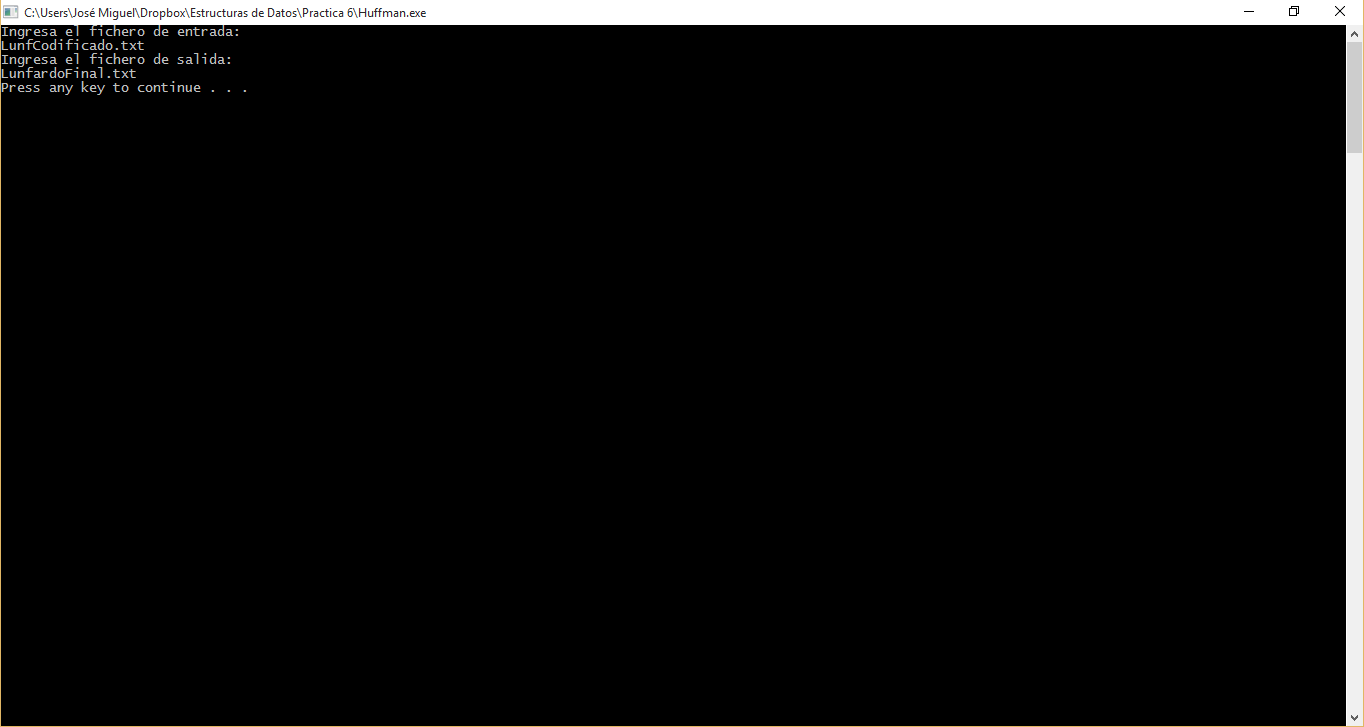


Nótese que el tamaño del archivo se redujo considerablemente, lo cual cumple el cometido de ahorrar espacio en el manejo de la información, además, la visualización de la información codificada depende del editor de texto en que se muestre.

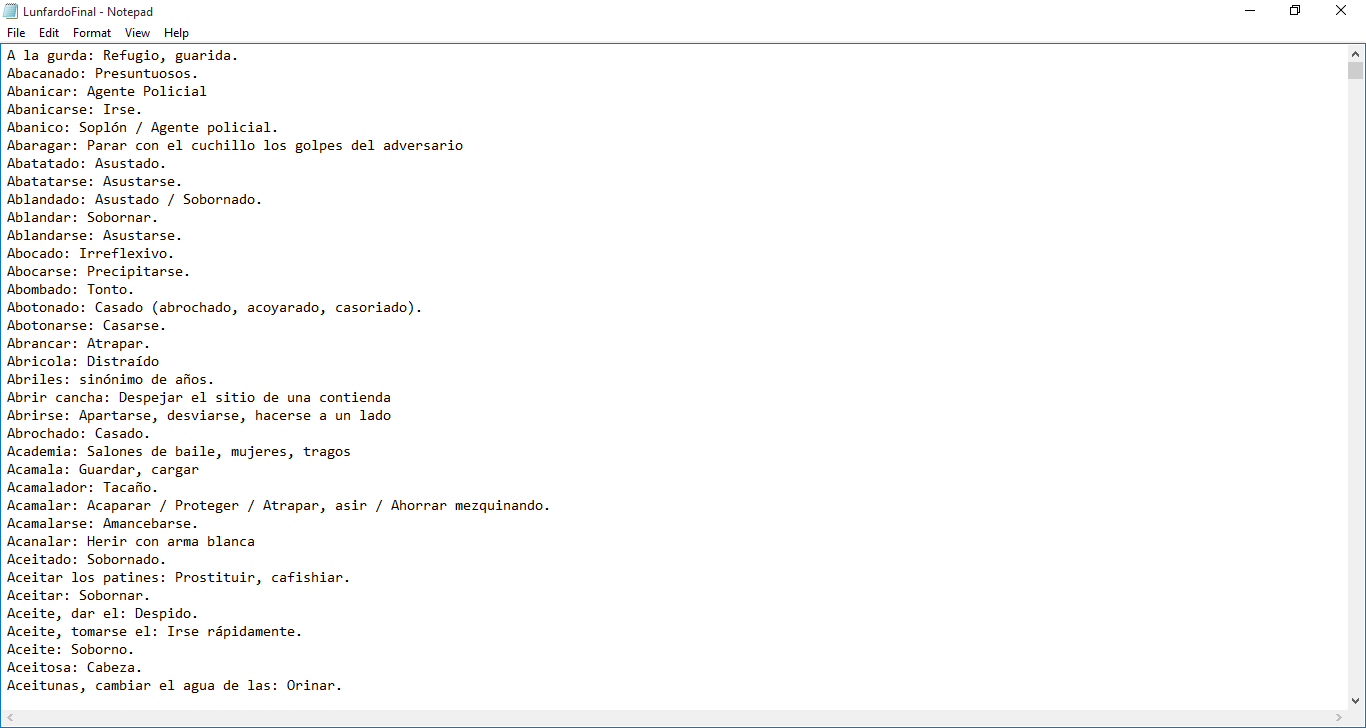
Notepad++



Finalmente en el proceso de decodificación insertamos como entrada nuestro archivo codificado.



Y comprobamos que la información se recupera correctamente:



GOTCHA!

# ERRORES DETECTADOS

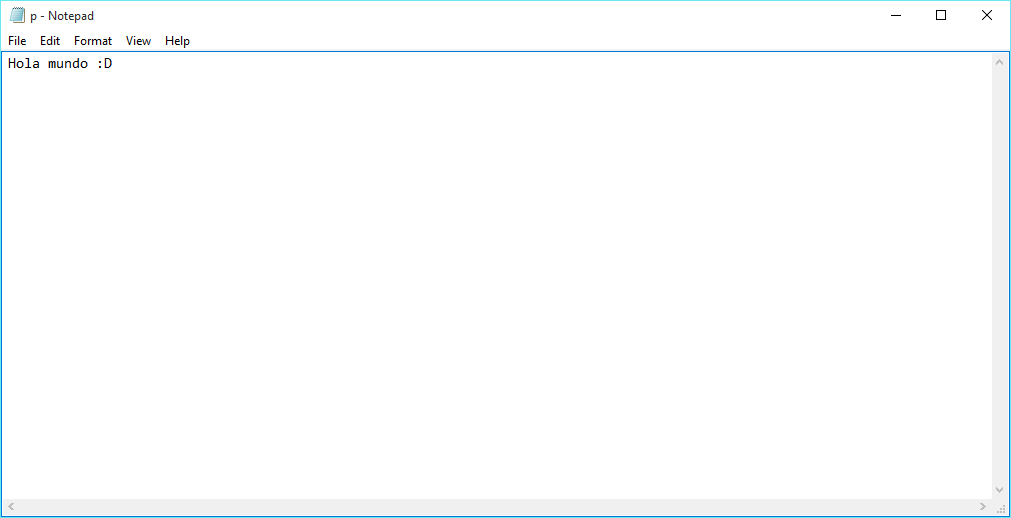
* El programa no codifica la información en modo carácter, lo realiza en modo binario; esto se debe a que en la implementación del código se realiza mediante codificación binaria conformando octales de bits (equivalentes a un byte), lo que al guardarse en un archivo de salida de texto se toman dichas secuencias en un equivalente de código ASCII

Si hacemos uso de alguna de herramienta, tal como “Ascii Text to Binary Converter” [2] podemos observar que si tomamos la información codificada de un archivo de texto tiene su equivalente en código binario. En las pruebas del programa se hizo la conversión de un mensaje que tenía un archivo de texto diciendo “Hola mundo :D”, el cual una vez codificado mostraba “ : D H a d l m n o u ”‹±®À”; si usamos un convertidor de este equivalente ASCII que muestra un editor de texto a un visualizador de binario obtendremos la codificación en 0 y 1 que pedía como requisito la práctica:

00001010 00100000 00100000 00100000 00001011 00100000 00100000 00100000 00100000 00000011 00100000 00100000 00100000 00000010 00111010 00000101 00100000 00100000 00100000 00000011 01000100 00001000 00100000 00100000 00100000 00000100 01001000 00001001 00100000 00100000 00100000 00000100 01100001 00000010 00100000 00100000 00100000 00000100 01100100 00000011 00100000 00100000 00100000 00000100 01101100 00100000 00100000 00100000 00100000 00000100 01101101 00000001 00100000 00100000 00100000 00000100 01101110 00000110 00100000 00100000 00100000 00000100 01101111 00000010 00100000 00100000 00100000 00000011 01110101 00000111 00100000 00100000 00100000 00000100 10000000011101 00000101 10000000111001 10110001 10101110 11000000

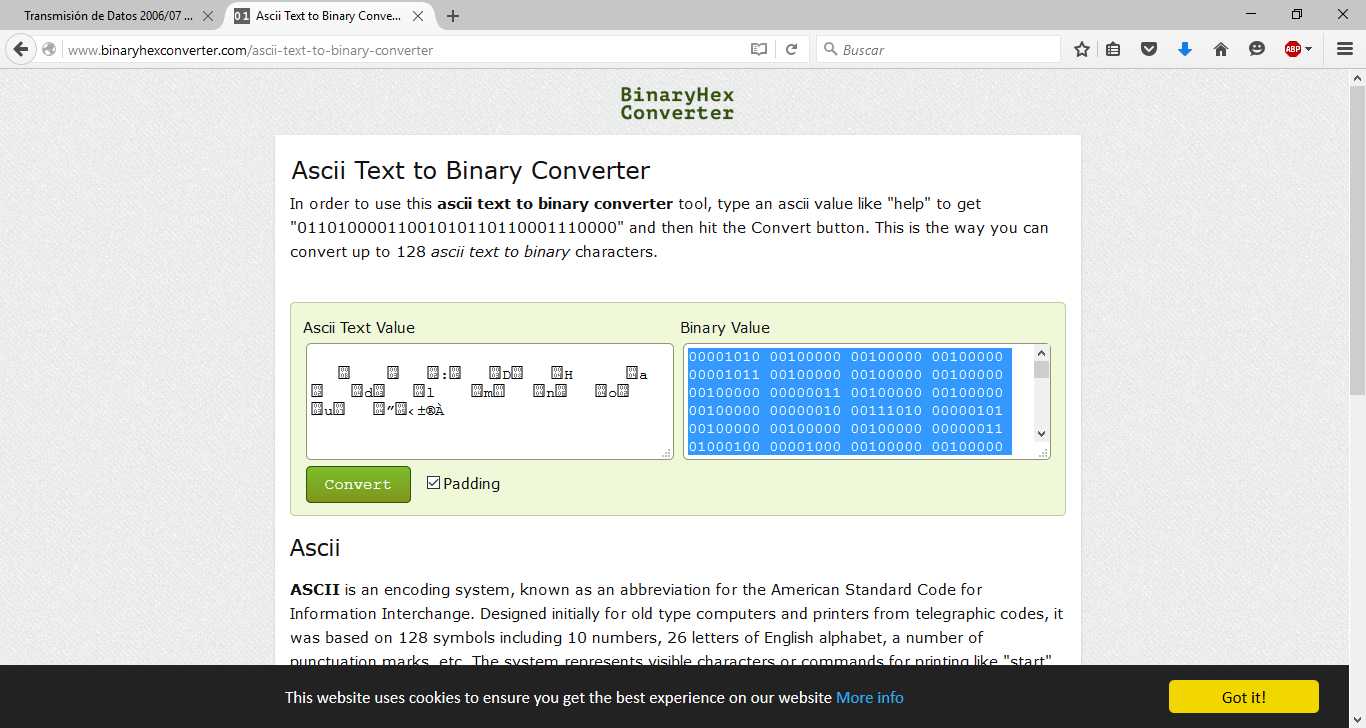
Esta secuencia binaria no es directamente una traducción del texto plano a codificar, es el equivalente de la concatenación de todos los árboles que se crearon en la ejecución del programa y que equivalen a las hojas del árbol donde se almacena cada uno de los caracteres y su frecuencia de aparición.

Texto a Codificar:



Texto codificado:

Conversión a caracteres binarios por la representación del editor de texto:



# POSIBLES MEJORAS

* Los valores que actualmente se usan en las estructuras definidas (\_nodo, \_tabla) pueden trasladarse a las implementaciones de los TAD’s propuestos en clases, de modo que en una implementación equivalente se utilizara el TAD Árbol Binario y el TAD Lista.
* En la codificación del código se podría sustituir la parametrización de codificar cada octal (8 apariciones) o las restricciones de decodificación (hexadecimal que solicita la rama o árbol óptimo <0x8000000000>) para que almacene el carácter 0 o 1 dependiendo de que rama le corresponde en el árbol binario.

# CONCLUSIONES

**Mendoza Parra Sergio:** En está practica se vio lo que es el algoritmo de Huffman en el cual dependiendo de un archivo o cualquier cosa se hace un el TAD árbol el código binario de lo que se quiere hacer, nosotros implementamos un algoritmo de Huffman que dependiendo de un archivo de tipo texto se viera de tipo binario desafortunadamente no pudimos implementarlo al 100% lo que es la parte de la encriptación o des encriptación por lo que lo hicimos con listas pero de igual manera es muy eficiente al igual que el del árbol binario de búsqueda y por ultimo ésta práctica estuvo un poco difícil entender o implementar más que nada lo que era el algoritmo de Huffman en la parte de árbol binario.

**Salcedo Barrón Rubén Osmair**: En esta práctica vimos el poder que tiene la codificación de Huffman para poder reducir en espacio y efectividad los archivos, además que también se puede aplicar en diferentes situaciones así sean muchos caracteres o demás aun así seguirá siendo efectivo por bien que está hecha esa codificación. Esta práctica en lo particular me gusto ya que tienes que razonar bien el funcionamiento y recorrer un esquema para ver cómo implementar bien esta codificación a pesar de que estuvo complicado al principio pudimos entender cómo funcionaba el programa para así poder explicarlo.

**Tejeda Martínez José Miguel**: El Algoritmo de Huffman es una poderosa herramienta usada en la actualidad para la compresión de la información. Basándose en la frecuencia de apariciones de caracteres, es posible el método de codificar o en su caso encriptar la información que tenemos protegiendo el contenido o resguardando algún mensaje en un espacio más pequeño.

En el desarrollo de los sistemas actuales, lograr algoritmos eficaces que ahorren espacio en el manejo de datos es esencial para mejorar la eficacia ahorrando recursos; en este caso pudimos ver eso, ya que teniendo una entrada de información pudimos obtener datos codificados que ahorraran considerablemente el espacio que necesitamos para almacenar.

# ANEXOS

CÓDIGO DEL PROGRAMA

/\*

Algoritmo de Huffman

AUTOR: Mendoza Parra Sergio, Salcedo Barron Ruben Osmair, Tejeda Martinez Jose Miguel (C) Diciembre 2015

VERSIÓN: 1.0

DESCRIPCIÓN: Implementar el algoritmo de codificación de Huffman para codificar archivos de texto bajo lenguaje C.

• Implementar codificación voraz de Huffman

• Implementar el algoritmo de decodificación

OBSERVACIONES:

COMPILACIÓN: g++ Huffman.cpp -o "Nombre del ejecutable"

EJECUCION: ./"Nombre del ejecutable" (Linux)

"Nombre del ejecutable".exe (Windows)

\*/

//Librerias

#include<stdio.h>

#include<stdlib.h>

//Definiciiones del usuario

#define PAUSE system("pause") //Pausar el sistema (Windows)-> Sustituir por getchar() en Linux

#define CLEAR system("cls") //Borrar pantalla (Windows)-> Sustituir por clear en Linux

/\*Tipo nodo para árbol o Lista de árboles

El propósito es dual, sirve como elemento de una lista enlazada

Cuando se usa el puntero sig, y como elemento de un árbol cuando

se usan los punteros cero y uno \*/

typedef struct \_nodo

{

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

unsigned long int bits; // Valor de la codificación de la letra

char nbits; // Número de bits de la codificación

\_nodo \*sig; //Puntero a siguiente nodo de una lista enlazada

\_nodo \*cero; //Puntero a la rama cero de un árbol

\_nodo \*uno; //Puntero a la rama uno de un árbol

} tipoNodo; //Nombre de tipo

//Nodo para construir una lista para la tabla de codigos

typedef struct \_tabla

{

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

\_tabla \*sig; //Siguiente elemento de la tabla

} tipoTabla; //Nombre del tipo

//Variables globales

tipoTabla \*Tabla;

//Declaración de funciones

void Cuenta(tipoNodo\* &Lista, char c);

void Ordenar(tipoNodo\* &Lista);

void InsertarOrden(tipoNodo\* &Cabeza, tipoNodo \*e);

void BorrarArbol(tipoNodo \*n);

void CrearTabla(tipoNodo \*n, int l, int v);

void InsertarTabla(char c, int l, int v);

tipoTabla \*BuscaCaracter(tipoTabla \*Tabla, char c);

int Codificar(char \*in,char \*out);

int Decodificar(char \*in,char \*out);

int main(void)

{

unsigned int o; //Opciones de menú

char filein[30],fileout[30]; //archivo de entrada - archivo de salida

do

{

puts("\t\tAlgoritmo de Huffman\n\nQue deseas realizar?");

puts("1. Codificacion de texto\n2. Decodificacion de texto\n3. Salir");

scanf("%d",&o);

CLEAR;

switch (o)

{

case 1:

puts("Ingresa el fichero de entrada (recuerda agregar la extension de archivo):");

scanf("%s",filein); //archivo con texto plano

puts("Ingresa el fichero de salida (recuerda agregar la extension de archivo):");

scanf("%s",fileout); //archivo de salida con codificacion

Codificar(filein,fileout);

PAUSE;

CLEAR;

break;

case 2:

puts("Ingresa el fichero de entrada:");

scanf("%s",filein); //archivo codificado

puts("Ingresa el fichero de salida:");

scanf("%s",fileout); //archivo decodificado a texto plano

Decodificar(filein,fileout);

PAUSE;

CLEAR;

break;

default:

break;

};

}while(o!=3);

return 0;

}

/\*int Codificar(char \*in,char \*out);

Recibe: ruta de archivo de texto plano, ruta de salida del archivo codificado

Obervaciones: el nombre de archivo debe acompañarse de su extensión. Ej: "archivo1.txt"

\*/

int Codificar(char \*in,char \*out)

{

tipoNodo \*Lista; // Lista de letras y frecuencias

tipoNodo \*Arbol; // Arbol de letras y frecuencias

FILE \*fe, \*fs; // Ficheros de entrada y salida

char c; // variables auxiliares

tipoNodo \*p;

tipoTabla \*t;

int nElementos; // Número de elementos en tabla

long int Longitud = 0; // Longitud del fichero original

unsigned long int dWORD; // Doble palabra usada durante la codificación

int nBits; // Número de bits usados de dWORD

Lista = NULL;

// Fase 1: contar frecuencias

fe = fopen(in, "r");

while((c = fgetc(fe)) != EOF)

{

Longitud++; // Actualiza la cuenta de la longitud del fichero

Cuenta(Lista, c); // Actualiza la lista de frecuencias

}

fclose(fe);

// Ordenar la lista de menor a mayor

Ordenar(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

// Construir la tabla de códigos binarios

Tabla = NULL;

CrearTabla(Arbol, 0, 0);

// Crear fichero comprimido

fs = fopen(out, "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);

// 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;

}

// Codificación del fichero de entrada

fe = fopen(in, "r");

dWORD = 0; // Valor inicial.

nBits = 0; // Ningún bit

while((c = fgetc(fe)) != EOF)

{

// 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

}

// 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;

}

fclose(fe); // Cierra los ficheros

fclose(fs);

// Borrar Arbol

BorrarArbol(Arbol);

// Borrar Tabla

while(Tabla)

{

t = Tabla;

Tabla = t->sig;

free(t);

}

return 0;

}

/\* void Cuenta(tipoNodo\* &Lista, char c);

Descripción: Actualiza la cuenta de frecuencia del carácter c.

El puntero a Lista se pasa por referencia, ya que debe poder cambiar

ya sea por que la lista esté vacía, o porque el nuevo elemento sea el primero \*/

void Cuenta(tipoNodo\* &Lista, 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

{

// 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

}

}

}

/\*void Ordenar(tipoNodo\* &Lista);

Descripción: 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

}

}

/\*void InsertarOrden(tipoNodo\* &Cabeza, tipoNodo \*e);

Descripcion: 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 en NULL, e 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

}

}

/\*void CrearTabla(tipoNodo \*n, int l, int v);

Descripción; 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);

}

/\*void InsertarTabla(char c, int l, int v);

Descripción; Insertar un elemento en la tabla \*/

void InsertarTabla(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 1º

{

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

}

}

/\*tipoTabla \*BuscaCaracter(tipoTabla \*Tabla, char c);

Descripción: Buscar un caracter en la tabla, devielve un puntero al elemento de la tabla \*/

tipoTabla \*BuscaCaracter(tipoTabla \*Tabla, char c)

{

tipoTabla \*t;

t = Tabla;

while(t && t->letra != c) t = t->sig;

return t;

}

/\*void BorrarArbol(tipoNodo \*n);

Descripción: Función recursiva para borrar un arbol \*/

void BorrarArbol(tipoNodo \*n)

{

if(n->cero) BorrarArbol(n->cero);

if(n->uno) BorrarArbol(n->uno);

free(n);

}

int Decodificar(char \*in,char \*out)

{

tipoNodo \*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 \*p, \*q; // Auxiliares

unsigned char a;

int i, j;

// Crear un arbol con la información de la tabla

Arbol = (tipoNodo \*)malloc(sizeof(tipoNodo)); // un nodo nuevo

Arbol->letra = 0;

Arbol->uno = Arbol->cero = NULL;

fe = fopen(in, "rb");

fread(&Longitud, sizeof(long int), 1, fe); // Lee el número de caracteres

fread(&nElementos, sizeof(int), 1, fe); // Lee el número de elementos

for(i = 0; i < nElementos; i++) // Leer todos los elementos

{

p = (tipoNodo \*)malloc(sizeof(tipoNodo)); // 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;

// 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 \*)malloc(sizeof(tipoNodo)); // 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 \*)malloc(sizeof(tipoNodo)); // 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;

}

// Leer datos comprimidos y extraer al fichero de salida

bits = 0;

fs = fopen(out, "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;

// 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

fclose(fs); // Cerramos ficheros

fclose(fe);

BorrarArbol(Arbol); // Borramos el árbol

return 0;

}

# BIBLIOGRAFÍA

#### [1] Edgardo Adrián Franco Martínez, Practica 0**6**: ["**Codificación de Huffman**"](http://eafranco.com/docencia/estructurasdedatos/files/practicas/Practica05.pdf)  **Diciembre** de 2015. [En línea]. Disponible en: <http://eafranco.com/docencia/estructurasdedatos/files/practicas/Practica06.pdf>

#### [2] BinaryHexConverter.com, Ascii Text to Binary Converter. Enero de 2016. [En línea]. Disponible en:

<http://www.binaryhexconverter.com/ascii-text-to-binary-converter>