



# Practica 3

## *Codificación y Decodificación mediante algoritmo de Huffman*

Algoritmo de Huffman

Jeong Jeong Paola

Lemus Ruiz Mariana Elizabeth

López López Oscar Manuel

Mora Ayala Jose Antonio



## CONTENIDO

Practica 3 .....	1
Introducción.....	4
Definición del Problema.....	7
Análisis Teórico .....	8
Número de códigos Huffman .....	8
Altura del Árbol de Huffman.....	9
Codificación Aritmética.....	9
Complejidad Algorítmica .....	10
Pasos del algoritmo de huffman.....	11
descripción del algoritmo (manual y teórico).....	15
Codificación .....	15
Estructuras necesarias: .....	15
Función fillList (Listaenlazada *lista, char c) .....	17
Analisis de Complejidad.....	17
Función CopyList(Listaenlazada lista).....	20
Descripción: .....	20
Análisis.....	20
Función PrintList(Listaenlazada lista).....	23
Descripción .....	23
Analisis.....	23
Función OrderList(Listaenlazada *Lista).....	24
Descripción .....	24
Análisis:.....	24
Función HuffmanTree () .....	27
Descripción: .....	27
Análisis:.....	27
Función binaryRoute (Listaenlazada *arbollImpresión, int pos, int bit, int izq, int der, int *auxBin).....	30
Descripción .....	30
Análisis.....	31



Función createBinaryList.....	33
Descripción .....	33
Análisis.....	34
Función crearFrecuencias.....	37
Descripción .....	37
Análisis.....	37
Función crearCodificacionDAT.....	39
Descripción: .....	40
Análisis:.....	42
Análisis.....	45
Función imprimirTiempos .....	47
Decodificación .....	48
Descripción .....	48
Análisis.....	49
Función Craerdecodificación .....	50
Descripción .....	52
Análisis.....	52
Pruebas tablas comparativas y gráficas .....	55
Cuestionario.....	66
Conclusiones.....	68
Código.....	70
Funciones.c.....	75
Main.c .....	84
Archivo Decodificación .....	86
FuncionesD.h.....	86
FuncionesD.c .....	90
Decodificador.c .....	99
Bibliografía .....	100
Bibliografía.....	100



## INTRODUCCIÓN

### Árbol

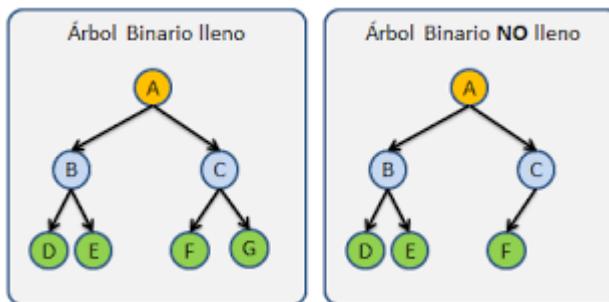
Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas. En un árbol existe un nodo especial denominado raíz. Así mismo, un nodo del que sale alguna rama recibe el nombre de nodo de bifurcación o nodo rama y un nodo que no tiene ramas recibe el nombre de nodo hoja.

Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas. En un árbol existe un nodo especial denominado raíz. Así mismo, un nodo del que sale alguna rama, recibe el nombre de nodo de bifurcación o nodo rama y un nodo que no tiene ramas recibe el nombre de nodo hoja.

Es útil limitar los árboles en el sentido de que cada nodo sea a lo sumo de grado 2. De esta forma cabe distinguir entre subárbol izquierdo y subárbol derecho de un nodo. Los árboles así formados, se llaman árboles binarios.

### Árbol Binario

A los árboles ordenados de grado dos se les conoce como árboles binarios ya que cada nodo del árbol no tendrá más de dos descendientes directos. Las aplicaciones de los árboles binarios son muy variadas ya que se les puede utilizar para representar una estructura en la cual es posible tomar decisiones con dos opciones en distintos puntos. La representación gráfica de un árbol binario es la siguiente:



Hay tres maneras de recorrer un árbol: en inorden, preorden y postorden. Cada una de ellas tiene una secuencia distinta para analizar el árbol como se puede ver a continuación:

Inorden:

- Recorrer el subárbol izquierdo en inorden
- Examinar la raíz.
- Recorrer el subárbol derecho en inorden

Preorden:

- Examinar la raíz.
- Recorrer el subárbol izquierdo en preorden.
- Recorrer el subárbol derecho en preorden.

Postorden:

- Recorrer el subárbol izquierdo en postorden.
- Recorrer el subárbol derecho en postorden.
- Examinar la raíz.

### Algoritmo de Huffman

La compresión de datos digitales, imágenes digitales, es el proceso de reducción del volumen de datos para representar una determinada cantidad de información. Es decir, un conjunto de datos puede contener datos redundantes que son de poca relevancia o son datos que se repiten en el conjunto, los cuales si se identifican pueden ser eliminados.

Con su estudio, Huffman superó a su profesor, quien había trabajado con el inventor de la teoría de la información Claude Shannon con el fin de desarrollar un código similar. Huffman solucionó la mayor parte de los errores en el algoritmo de codificación Shannon-Fano. La solución se basaba en el proceso de construir el árbol de abajo hacia arriba en vez de arriba hacia abajo. La idea de Huffman fue identificar los píxeles que aparecen con mayor frecuencia en una imagen y asignarles representaciones cortas. Los píxeles que ocurren con menor frecuencia en una imagen se les asigna representaciones largas.

Para obtener el código de Huffman de cada carácter hay que construir un árbol binario de nodos, a partir de una lista de nodos, cuyo tamaño depende del número de caracteres,  $n$ . Los nodos contienen dos campos, el símbolo y el peso (frecuencia de aparición).

La técnica utilizada por el algoritmo de Huffman, consiste en la creación de un árbol binario en el que se etiquetan los nodos hoja con los caracteres, junto a sus frecuencias, y de forma consecutiva se van uniendo cada pareja de nodos que menos frecuencia sumen, pasando a crear un nuevo nodo intermedio etiquetado con dicha suma. Se procede a realizar esta acción hasta que no quedan nodos hoja por unir a ningún nodo superior, y se ha formado el árbol binario.

Cada nodo del árbol puede ser o bien un nodo hoja o bien un nodo interno. Inicialmente se considera que todos los nodos de la lista inicial son nodos hoja del árbol. Al ir construyendo el árbol, los nodos internos tendrán un peso y dos nodos hijos, y opcionalmente un enlace al nodo padre que puede servir para recorrer el árbol en ambas direcciones. Por convención el bit 0 se asocia a la rama izquierda y el bit 1 a la derecha. Una vez finalizado el árbol contendrá  $n$  nodos hijos y  $n - 1$  nodos internos.

Posteriormente de que se etiquetan las aristas que unen cada uno de los nodos con ceros y unos (hijo derecho e izquierdo, respectivamente, por ejemplo), el código resultante para cada carácter es la lectura, siguiendo la rama, desde la raíz hacia cada carácter (o viceversa) de cada una de las etiquetas de las aristas.

## DEFINICIÓN DEL PROBLEMA

- Implementar el algoritmo de codificación de Huffman para codificar archivos de cualquier tipo.
  - Implementar la codificación voraz de Huffman
  - Implementar el algoritmo de decodificación
- Medir y comprobar las ventajas de tamaño de los archivos una vez realizadas distintas codificaciones de los archivos.
- Medir los tiempos de ejecución de las implementaciones, ya sea codificación y decodificación y realizar estadísticas de comprensión.
- Detallar el procedimiento de análisis que determina una cota  $O$  de complejidad para el código implementado (asignar su correspondiente  $O$  por cada módulo). [1]



## ANÁLISIS TEÓRICO

Para el análisis teórico de este complejo algoritmo uno necesita basarse en los diferentes apartados que tuvieron que ser implementados para poder llevar a cabo la realización de esta práctica, pues los temas que abarca competen a aquellos que tienen que ver con Listas enlazadas (en este caso decidió realizar la implementación de una lista enlazada simple y la implementación de un árbol).

El algoritmo comienza construyendo una lista de todos los símbolos del alfabeto correspondiente a un fragmento de la información que se va a codificar, el listado se organiza en forma descendente según el orden de las probabilidades de ocurrencia de cada uno de los símbolos dentro de la cadena. Luego se construye un árbol, con un símbolo en cada hoja. Esto se hace en iteraciones, donde cada uno de los dos símbolos con los valores más pequeños de probabilidad se seleccionan y se agregan a la parte superior del árbol parcial, se eliminan de la lista y se reemplazan por un símbolo auxiliar que representa los dos símbolos originales. Cuando la lista se reduce a un solo símbolo auxiliar (que representa el alfabeto completo), el árbol es completo, siendo este último símbolo auxiliar a raíz del árbol. Luego se recorre el árbol para determinar los códigos de los símbolos.

A continuación, se muestra la estructura del algoritmo en el método de Huffman de codificación mediante su representación de una estructura de datos de tipo árbol binario.

Como primer paso se debe identificar y describir la información y su representación de la siguiente forma. Sea una cadena de símbolos, que representa en su forma natural cierta cantidad de información que se quiere codificar y comprimir mediante el método de Huffman, entonces se define el conjunto como:

*S: { $s_i$  es el símbolo que representa un fragmento de la información}*

*S: { $s_0, s_1, s_2, s_3 \dots \dots \dots s_{n-1}, s_n$ }*

Donde cada elemento del conjunto *S* es un símbolo que representa un fragmento o unidad mínima de la información que se quiere procesar; por lo tanto, el conjunto tiene un número de elementos ( $n - 1$ ), que depende directamente del tipo, la cantidad y la naturaleza de la información [1]

### NÚMERO DE CÓDIGOS HUFFMAN

Dado que el código de Huffman no es único, se puede preguntar cuántos códigos diferentes se pueden obtener a partir de un conjunto dado de símbolos. Si este conjunto tiene símbolos, el



árbol de Huffman contiene  $- 1$  nodos interiores o nodos distintos de la raíz, esto quiere decir que, si por cada nivel del árbol se intercambian las etiquetas de sus ramas, al asignar el bit correspondiente, este produce un código Huffman diferente; de esta manera en cada rama de todos los niveles del árbol al intercambiar las etiquetas, genera un nuevo código, por lo tanto, el número total de árboles de código de Huffman diferentes es igual a

$$2n - 1$$

## ALTURA DEL ÁRBOL DE HUFFMAN

La altura del árbol Huffman a veces puede ser importante porque el valor de la altura también es la longitud del código más largo que genero el árbol, por esta razón una forma de optimización del código Huffman, es buscar el árbol con la menor altura dentro de los posibles generados; para encontrarlo se debe hallar la varianza que más cercana a cero se encuentre, ya que el árbol Huffman más corto se crea cuando los símbolos tienen probabilidades iguales o muy cercanas, ósea cuando la desviación estándar del conjunto de probabilidades tiende a cero. [2]

## CODIFICACIÓN ARITMÉTICA

Las implementaciones reales de la codificación aritmética son muy similares a las de codificación Huffman, aunque superan a estas últimas en la realidad, el método Huffman asigna un número entero de bits a cada símbolo, mientras que la codificación aritmética asigna un único código extenso a la cadena de entrada completa. Por ejemplo, idealmente a un símbolo con probabilidad 0,4 se le debería asignar un código de 1.32 bits, pero será codificado con 2 bits usando el método Huffman. Es por esta razón que la codificación aritmética tiene el potencial de comprimir datos en el límite teórico. La codificación aritmética combina un modelo estadístico con un paso de codificación que consiste en algunas operaciones aritméticas. El modelo más sencillo tendría una complejidad temporal lineal de  $N(\log(n) + a) + S n$ , donde  $N$  es el número total de símbolos de entrada,  $n$  es el número real de símbolos distintos,  $a$  es la aritmética a ser realizada y  $S$  es el tiempo requerido, si se necesita, para mantener las estructuras de datos internas [3].

## COMPLEJIDAD ALGORÍTMICA

Un algoritmo es un conjunto finito de acciones que dan solución a un determinado problema; en particular en las ciencias de la computación, un algoritmo es una serie de instrucciones ordenadas que logran dar solución a determinado problema de procesamiento de información; los algoritmos se pueden clasificar no solo porque lleven a cabo la tarea para la que fueron diseñados, también pueden clasificarse en cuanto a los costos de su ejecución; entendiéndose por costo, la cantidad de recursos físicos de almacenamiento y unidades de tiempo que consume para su ejecución, desde que inicia hasta que termina la tarea, a este indicador se le conoce como medida de la complejidad del algoritmo o también como medida de la eficiencia.

La complejidad de tiempo para codificar cada carácter único en función de su frecuencia es  $O(n \log n)$ . La extracción de la frecuencia mínima de la cola de prioridad se realiza  $2 * (n-1)$  veces y su complejidad es  $O(n \log n)$ . Por tanto, la complejidad general es  $O(n \log n)$ .

## PASOS DEL ALGORITMO DE HUFFMAN

Para explicar este algoritmo vamos a utilizar un ejemplo. Supongamos que tenemos la siguiente cadena:

**BCAADDCCACACAC**

Cada carácter ocupa un total de 8 bits de memoria. En total hay 15 caracteres, entonces tenemos que  $8 \times 15 = 120$  bits que se necesitan para usar esta cadena. Una vez teniendo esta cadena utilizamos el algoritmo de Huffman:

1. Contamos la frecuencia que tiene cada carácter dentro de la cadena, es decir,

1	3	5	6
B	D	A	C

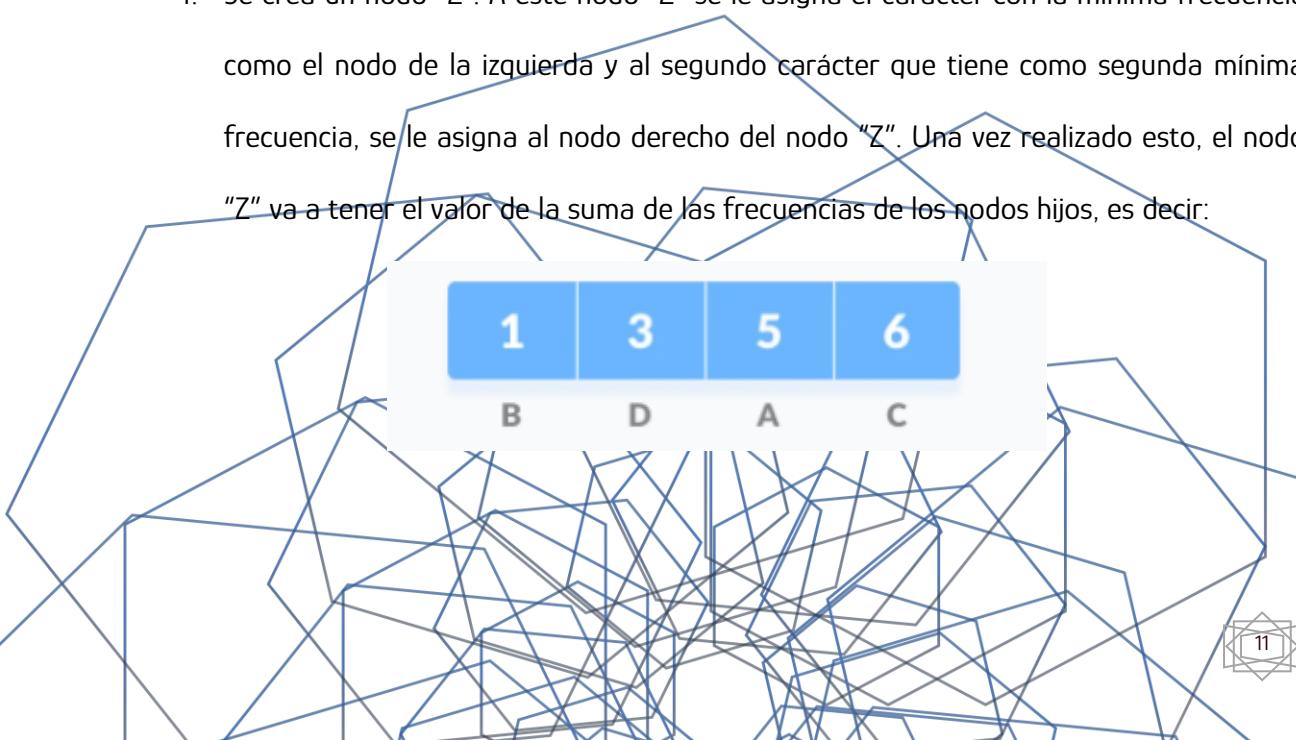
2. Ordenamos la frecuencia de menor a mayor, teniendo como resultado esta cadena

"Q":

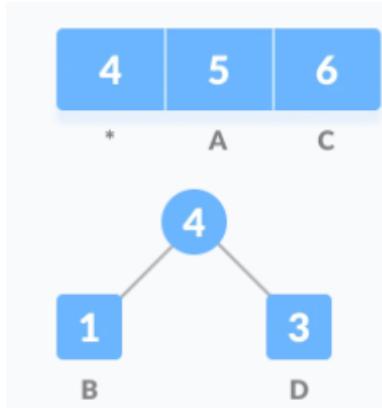
1	3	5	6
B	D	A	C

3. Vamos a tomar cada carácter diferente como un nodo de un árbol binario A.

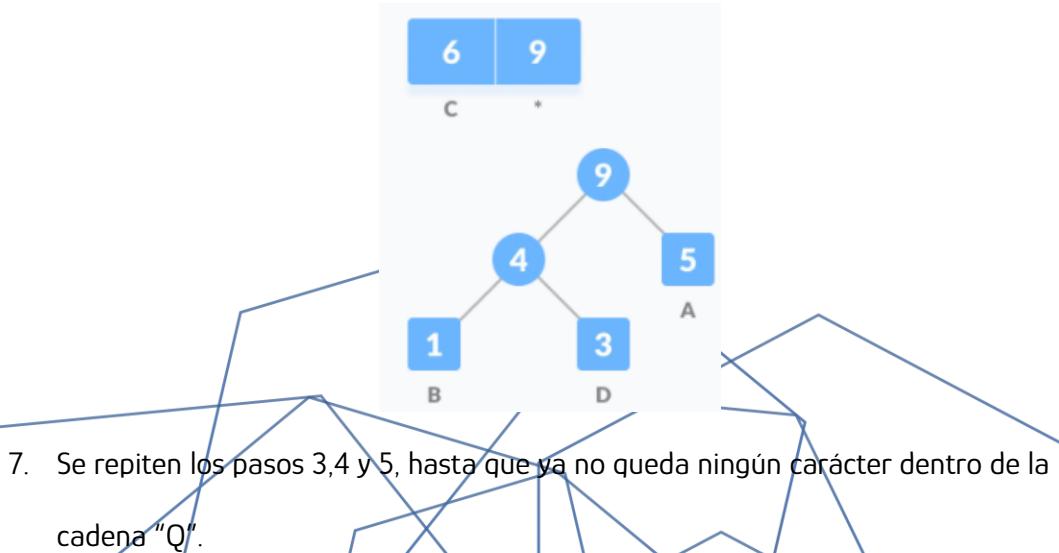
4. Se crea un nodo "Z". A este nodo "Z" se le asigna el carácter con la mínima frecuencia como el nodo de la izquierda y al segundo carácter que tiene como segunda mínima frecuencia, se le asigna al nodo derecho del nodo "Z". Una vez realizado esto, el nodo "Z" va a tener el valor de la suma de las frecuencias de los nodos hijos, es decir:

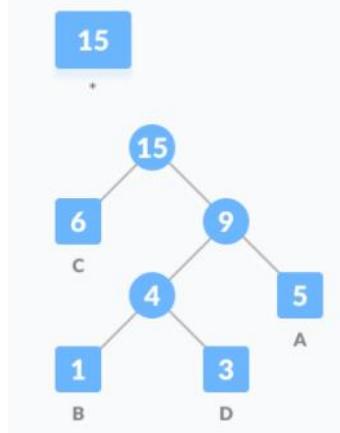


"B" se encuentra en el primer lugar con la mínima frecuencia de 1, y en segundo lugar está "D" con la frecuencia de 3, entonces el valor del árbol A1, sería 4:

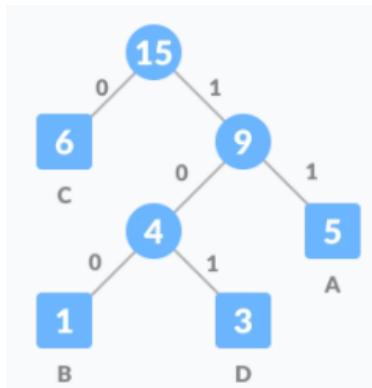


5. Quitamos a "B" y "D" de la cadena "Q" y el resultado del nodo raíz del árbol A1, se inserta a la cadena "Q".
6. Luego se vuelve a insertar un nodo "Z" arriba del nodo del árbol A1, esto se realiza, poniendo el tercer carácter con la mínima frecuencia como el nodo hijo a la derecha del nodo del árbol A1, en este caso el carácter "A" con una frecuencia de 5, será el nodo derecho del nuevo nodo raíz "Z".

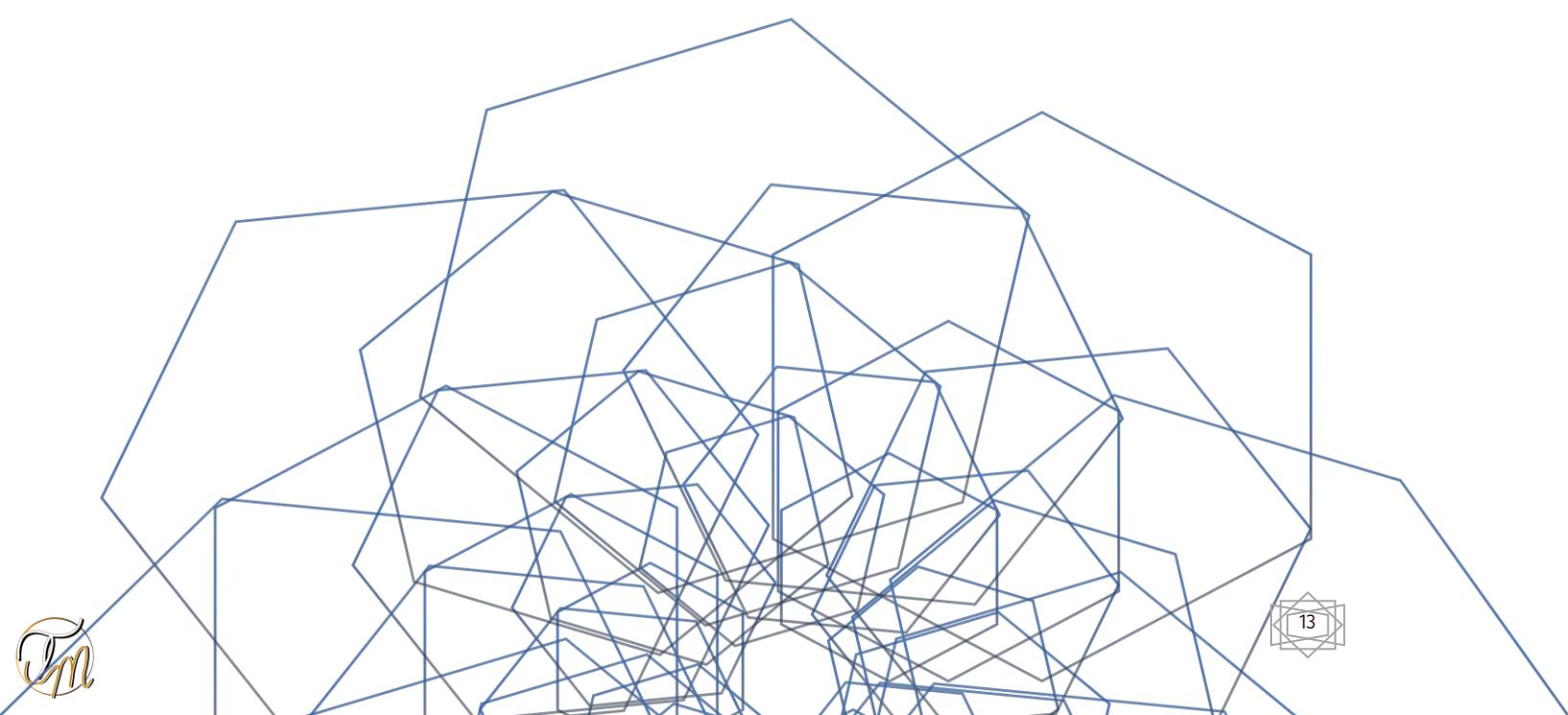




8. Ahora, para todas las ramas que se encuentran en la izquierda, se les asigna un 0, y para los de la derecha, se les asigna un 1:



Una vez realizado este procedimiento podemos observar que el número de bits a utilizar se reduce después de usar el algoritmo de Huffman:



Character	Frequency	Code	Size
A	5	11	$5*2 = 10$
B	1	100	$1*3 = 3$
C	6	0	$6*1 = 6$
D	3	101	$3*3 = 9$
$4 * 8 = 32$ bits		15 bits	28 bits

Antes de usar el algoritmo Huffman, se necesitaban 120 bits para poder procesar la cadena donde se repetían varios caracteres, pero después, solo se necesitaron 28 bits. [4] [5]

## **DESCRIPCIÓN DEL ALGORITMO (MANUAL Y TEÓRICO)**

# CODIFICACIÓN

El algoritmo de Huffman es una técnica la cual nos ayuda a comprimir datos para poder reducir el tamaño de este sin perder ninguna característica importante. Esta técnica nos ayuda específicamente cuando se quieren procesar datos en donde vienen repetidos varios caracteres.

Supongamos que tenemos los datos de "hello world", normalmente, para cada carácter, se le asigna una serie de bits diferentes, pero gracias al algoritmo de Huffman esto se puede evitar

A continuación se realiza la descripción de cada una de las funciones involucradas para el desarrollo del programa de codificación de Huffman

## **ESTRUCTURAS NECESARIAS:**

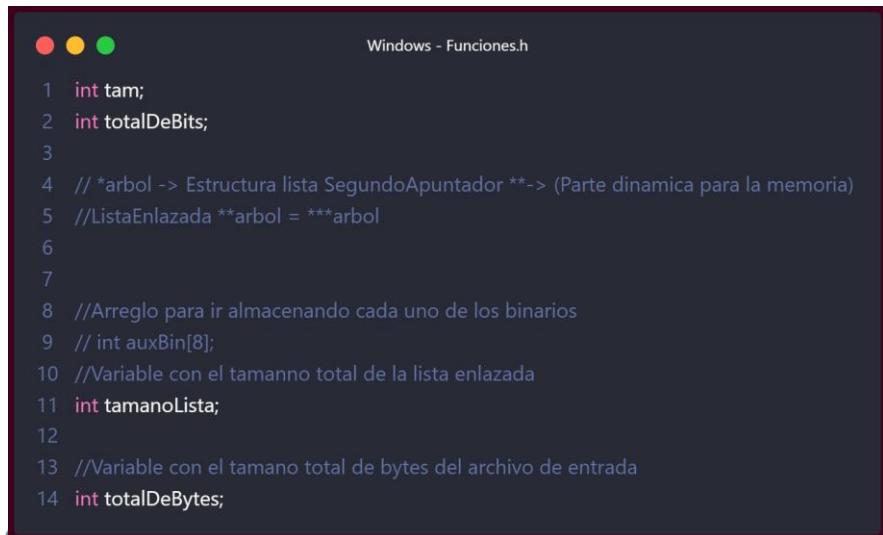


Para la realización de la práctica comenzaremos declarando las estructuras correspondientes que serán necesarias para la creación de la lista enlazada (mejor visto como una cola LIFO) ya que nuestro último dato ingresado será el primer en ser extraído en caso de que fuera necesario, dato a tener en consideración ya que por eso será necesario crear una función que nos devuelva la lista en el orden inverso.



```
Windows -  
1 typedef struct Lista  
2 {  
3     NodoInfo inf;  
4     struct Lista *sig, *izq, *der;  
5 } Nodolista;  
6  
7 typedef Nodolista *Listaenlazada;  
8 //Sin el typedef  
9 // Nodolista *lista  
10  
11 Nodolista **arbol;
```

De igual forma nos auxiliamos de la forma en que podemos, por decirlo de alguna forma resumir la declaración de un apuntador de tipo nodolista, para poder ir iterando y creando los nodos auxiliares de una forma mas sencilla conforme necesitemos recurrir a la creación de nodos para la lista o en su momento el árbol, como podemos observar cada nodo también contiene en su estructura un lado derecho y uno izquierdo lo cual ya proporciona la estructura de un árbol



```
Windows - Funciones.h  
1 int tam;  
2 int totalDeBits;  
3  
4 // *arbol -> Estructura lista SegundoApuntador **-> (Parte dinamica para la memoria)  
5 //ListaEnlazada **arbol = ***arbol  
6  
7  
8 //Arreglo para ir almacenando cada uno de los binarios  
9 // int auxBin[8];  
10 //Variable con el tamanno total de la lista enlazada  
11 int tamanoLista;  
12  
13 //Variable con el tamanno total de bytes del archivo de entrada  
14 int totalDeBytes;
```

De igual manera se hace la declaración de unas variables globales que son necesarias para el desarrollo del algoritmo así como para el control y acceso a ellas, para de esta forma no estar pasando en todo momento a las distintas funciones

```
Windows - Funciones.c

1 void fillList(Listaenlazada *lista, char c)
2 {
3     Listaenlazada laux;
4     laux = *lista;
5     if (*lista == NULL)
6     {
7         *lista = (Nodolista *)malloc(sizeof(Nodolista));
8         (*lista)->inf.frec = 1;
9         (*lista)->inf.c = c;
10        (*lista)->sig = NULL;
11        (*lista)->izq = NULL;
12        (*lista)->der = NULL;
13    }
14    else
15    {
16        do
17        {
18            if ((laux->inf.c) == c)
19            {
20                laux->inf.frec = laux->inf.frec + 1;
21                return;
22            }
23            laux = laux->sig;
24        } while (laux != NULL);
25        Listaenlazada nuevo;
26        nuevo = (Nodolista *)malloc(sizeof(Nodolista));
27        nuevo->inf.frec = 1;
28        nuevo->inf.c = c;
29        nuevo->sig = *lista;
30        nuevo->izq = NULL;
31        nuevo->der = NULL;
32        *lista = nuevo;
33    }
34 }
```

## FUNCIÓN FILLLIST (LISTAENLAZADA \*LISTA, CHAR C)

Como podemos observar a nuestro lado izquierdo tenemos la declaración y el algoritmo que sigue esta función de realizar la inserción de los distintos elementos dentro de una lista, como sabemos cuando la lista está vacía insertaremos el primer elemento, en caso contrario debemos realizar un recorrido de la cabeza en una posición y convertir a nuestro nuevo nodo en la cabeza actual de la cabeza, de igual manera debemos realizar la comprobación de que el carácter en cuestión no existe, si existe no se inserta como tal, simplemente se le aumenta la frecuencia

### ANALISIS DE COMPLEJIDAD

Empecemos analizando el ciclo más interno, en este caso el do-while:



```

    laux->inf.frec = laux->inf.frec += 1;

    return;
}

laux = laux->sig;
}while(laux != NULL)

```

El **bloque interno** solo realiza una operación aritmética y de asignación, siendo de  $O(1)$ .

Tanto la **comparación del if** como la **siguiente instrucción secuencial**, son operaciones constantes, por lo tanto, todo el bloque dentro del do es de  $O(1)$ .

Finalmente, podemos observar que se va recorriendo la lista enlazada un elemento a la vez, y el **ciclo** se detiene cuando el siguiente elemento es nulo (es decir, el final de la lista), por lo tanto, el peor caso es que se vaya a insertar el último elemento de una lista con  $n$  elementos, por lo tanto, el bloque entero es de complejidad:  $O(n)$ .

Las siguientes instrucciones secuenciales que suceden al ciclo do-while son:

```

Listaenlazada nuevo;
nuevo = (Nodolista *) malloc(sizeof(Nodolista));
nuevo -> inf.frec = 1;
nuevo->inf.c = c;
nuevo->sig = *lista;
nuevo->izq = NULL;
nuevo->der = NULL;
*lista = nuevo;

```

Vemos que únicamente se trata de un bloque donde se crea un nuevo nodo con el byte y su frecuencia, sin embargo, todas son operaciones constantes, este bloque tiene una complejidad  $O(1)$ .

Por regla de la suma, el bloque que posee en secuencia al ciclo do-while y a estas instrucciones de asignación es de complejidad  $O(n)$ .

Ahora analicemos el bloque exterior compuesto principalmente por una condicional:

```

Listaenlazada laux;

laux = *lista;

if( *lista == NULL)

{

    *lista = (Nodolista *) malloc(sizeof(Nodolista));

    (*lista)->inf.frec = 1;

    (*lista)->inf.c = c;

    (*lista)->sig = NULL;

    (*lista)->izq = NULL;

    (*lista)->der = NULL;

}

else

{

    //bloque de código con complejidad O(n)

}

```

Apliquemos la regla de las decisiones para determinar la complejidad de la función completa, en la primera parte tenemos una serie de asignaciones, es decir operaciones constantes y, por ende, el primer bloque del condicional es de complejidad  $O(1)$ . Por otra parte, el siguiente bloque fue el que analizamos previamente, dando una complejidad de  $O(n)$ , así, aplicamos la regla de decisiones y seleccionamos la complejidad mayor de ambos casos, dando una complejidad total de  $O(n)$ .

## FUNCIÓN COPYLIST(LISTAENLAZADA LISTA)

### DESCRIPCIÓN:

```
Windows - Funciones.c

1 void copyList(Listaenlazada lista)
2 {
3     int pos;
4
5     if (lista == NULL)
6     {
7         printf("La lista se encuentra vacía\n");
8         exit;
9     }
10    arbol = (Nodolista **)malloc(tam * sizeof(Nodolista *));
11    for (pos = 0; pos < tam; pos++)
12    {
13        Nodolista *hoja;
14        hoja = (Nodolista *)malloc(sizeof(Nodolista));
15        hoja->inf.frec = 0;
16        hoja->inf.c = '\n';
17        hoja->izq = NULL;
18        hoja->der = NULL;
19        hoja->sig = NULL;
20        arbol[pos] = hoja;
21    }
22
23    for (lista, pos = 0; lista != NULL; pos++, lista = lista->sig)
24    {
25        arbol[pos]->inf.frec = lista->inf.frec;
26        arbol[pos]->inf.c = lista->inf.c;
27    }
28 }
```

El funcionamiento que realiza la función de copiar lista es bastante sencillo, simplemente recibe una lista que no se encuentra vacía, en caso de que se encuentre vacía esta devolverá un mensaje dandonos dicho aviso, en el caso contrario comenzaremos a emplear uso de nuestro árbol declarado de forma global el cual consiste en un doble apuntador.

Comenzamos reservando la memoria necesaria para el árbol de Huffman y comenzamos con la copia de la lista, la cual se realiza mediante un bucle ue se iterara de acuerdo al tamaño que esta tenga (mismo que se encuentra almacenado en una variable global), esta incremente posterior a que se realice una impresión de la lista, pues la variable ira incrementando conforme haya elementos que imprimir, es así como sabemos cuantos tenemos en nuestra lista ya creada.

Así mismo vamos realizando la inserción

de nuestros nodos en el arreglo del árbol y vamos inicializando los lados de cada una de las hojas del árbol en NULL posteriormente mediante otro ciclo for iremos evaluando e iterando sobre nuestro árbol y mientras este no se encuentre vacío iremos asignando a cada nodo la información pertinente que debería tener, es la creación de diversos arboles que después serán sometidos a una unión.

### ANÁLISIS

Comencemos por el primer ciclo for:

```
for (pos = 0; pos < tam; pos++)
```

```
{  
    Nodolista *hoja;
```

```

    hoja = (Nodolista *)malloc(sizeof(Nodolista));
    hoja->inf.frec = 0;
    hoja->inf.c = '\n';
    hoja->izq = NULL;
    hoja->der = NULL;
    hoja->sig = NULL;
    arbol[pos] = hoja;
}

```

El bloque de código dentro del ciclo simplemente se trata de asignaciones por lo que tiene una complejidad de orden  $O(1)$ .

Posteriormente pasamos al número de veces que este ciclo se repite, donde es fácil ver que la variable pos recorrerá todo el arreglo árbol por lo que su complejidad es  $O(n)$ .

Por regla del producto, este bloque posee complejidad  $O(n)$ .

Pasemos al segundo ciclo for:

```

for (lista, pos = 0; lista != NULL; pos++, lista = lista->sig)
{
    arbol[pos]->inf.frec = lista->inf.frec;
    arbol[pos]->inf.c = lista->inf.c;
}

```

El bloque interno tiene complejidad constante pues se trata de puras asignaciones.  $O(1)$ .

La condición para que el ciclo cese es que el elemento actual sea nulo, y se irá recorriendo de uno en uno. Por regla del peor caso, la complejidad es lineal.  $O(n)$ .

Por regla del producto, este bloque posee complejidad  $O(n)$ .

Finalmente, analicemos el primer bloque compuesto por una condicional:

```

int pos;
if (lista == NULL)
{

```

```
printf("La lista se encuentra vacia\n");

exit;

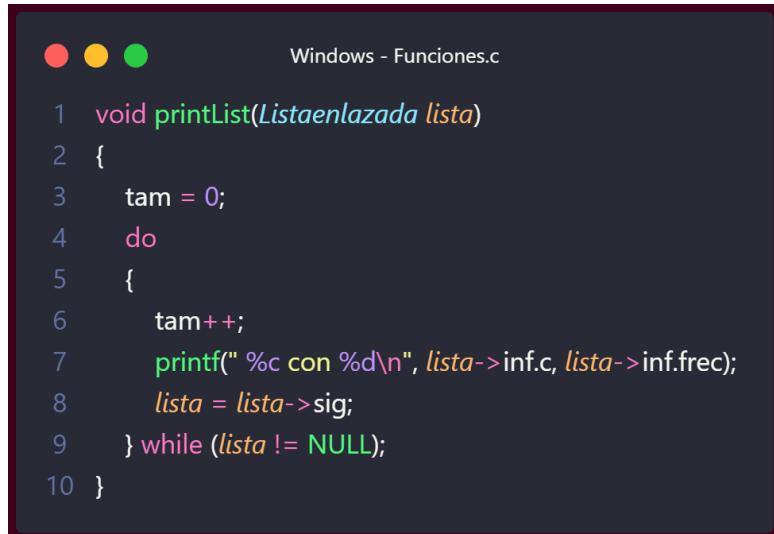
}

arbol = (Nodolista **)malloc(tam * sizeof(Nodolista *));

//bloque de complejidad O(n)
```

En la condicional establecemos que si la lista es nula, se salga del programa, por lo que no supondría un coste alguno, sin embargo, hay que aplicar la regla de las condicionales y por lo tanto, esta función posee una complejidad:  $O(n)$ .

## FUNCIÓN PRINTLIST(LISTAENLAZADA LISTA)



```
Windows - Funciones.c

1 void printList(Listaenlazada lista)
2 {
3     tam = 0;
4     do
5     {
6         tam++;
7         printf(" %c con %d\n", lista->inf.c, lista->inf.frec);
8         lista = lista->sig;
9     } while (lista != NULL);
10 }
```

### DESCRIPCIÓN

El funcionamiento de este es muy sencillo, simplemente vamos iterando sobre la lista que recibimos y junto a un ciclo do while que tiene como condición principal que nuestra lista no se encuentre vacía realizará el aumento de la variable tamaño, así como la impresión de información que existe en el nodo actual,

posteriormente pasamos al siguiente elemento de la lista y así iremos iterando, cuando nos encontramos situados en un elemento nulo o bien sin información alguna terminará nuestro ciclo.

### ANÁLISIS

Esta función consta de un solo bloque de código:

```
tam = 0;

do
{
    tam++;

    printf(" %c con %d\n", lista->inf.c, lista->inf.frec);

    lista = lista->sig;

} while (lista != NULL);
```

En el interior del ciclo do-while nos encontramos con dos instrucciones constantes, sin embargo, vemos que llamamos a la función "printf", así que aplicando regla de la suma la complejidad de este bloque será  $O(\max(O(1), O(\text{printf})))$ , por conveniencia, supongamos que la complejidad de imprimir una cadena es constante, así el bloque interno es de  $O(1)$ .

El número de veces que se iterará serán las necesarias hasta que el elemento actual de la lista sea nulo, aplicamos regla del peor caso y tenemos una complejidad  $O(n)$ .

Por regla del producto, esta función posee complejidad  $O(n)$ .



## FUNCIÓN ORDERLIST(Listaenlazada \*LISTA)

```
Windows - Funciones.c
1 void orderList(Listaenlazada *lista)
2 {
3     Listaenlazada actual, siguiente;
4     int t;
5     char c;
6     actual = *lista;
7     while (actual->sig != NULL)
8     {
9         siguiente = actual->sig;
10
11        while (siguiente != NULL)
12        {
13            if (actual->inf.frec >= siguiente->inf.frec)
14            {
15                t = siguiente->inf.frec;
16                c = siguiente->inf.c;
17                siguiente->inf.frec = actual->inf.frec;
18                siguiente->inf.c = actual->inf.c;
19                actual->inf.frec = t;
20                actual->inf.c = c;
21            }
22            siguiente = siguiente->sig;
23        }
24        actual = actual->sig;
25        siguiente = actual->sig;
26    }
27 }
```

### DESCRIPCIÓN

El funcionamiento de `orderList` esta determinado mediante dos listas auxiliares de las cuales necesitamos disponer para poder proceder a realizar el ordenamiento de los elementos que se encuentran dentro de la lista que recibimos, esta es necesario que sea enviada por referencia dado que estaremos modificando dicha lista para poder realizar uso de esta posteriormente, para realizar este ordenamiento nos estaremos basando principalmente en las frecuencias, por lo cual necesitaremos dos variables mas una que almacenara el carácter de forma temporal y otra que almacenara la frecuencia, al momento de realizar la comparación, en dado caso de que la frecuencia en efecto sea mayor o igual (de suma importancia que se considera la igualdad pues es una de las condiciones que pone el algoritmo de Huffman), dicho elemento se situara en la parte delantera de la lista, almacenamos tanto el valor de la frecuencia como el carácter y procedemos a realizar el cambio, este es una implementación del método de la burbuja prácticamente, solo que siendo operado con estructuras y con listas

### ANÁLISIS:

Observamos que el grueso de esta función radica en los ciclos while anidados, empecemos por el más interno:



```

siguiente->inf.frec = actual->inf.frec;
siguiente->inf.c = actual->inf.c;
actual->inf.frec = t;
actual->inf.c = c;
}

siguiente = siguiente->sig;
}

```

Se realiza una **condicional**, si se cumple, entra a un boque de complejidad  $O(1)$  pues se trata únicamente de asignaciones, por otro lado, si no se cumple la condición se ejecuta una sola instrucción de asignación, es decir,  $O(1)$ . Por regla de las condicionales, el bloque tiene complejidad  $O(1)$ .

Este ciclo while **recorrerá todo el arreglo a partir del nodo "siguiente"**, pues con cada iteración el elemento actual se desplaza una posición y se rompe el ciclo hasta que dicho elemento sea nulo (el final de la lista), por lo que el número de veces que se repetirá variará con las iteraciones del ciclo más externo, por lo que hay que aplicar la regla del peor caso (cuando recorre todo el arreglo), así el ciclo es de complejidad  $O(n)$ .

Por regla del producto, el bloque más interno de esta función es de  $O(n)$ .

Pasemos al ciclo exterior:

```

while (actual->sig != NULL)
{
    siguiente = actual->sig;
    //ciclo interno de complejidad O(n)
    actual = actual->sig;
    siguiente = actual->sig;
}

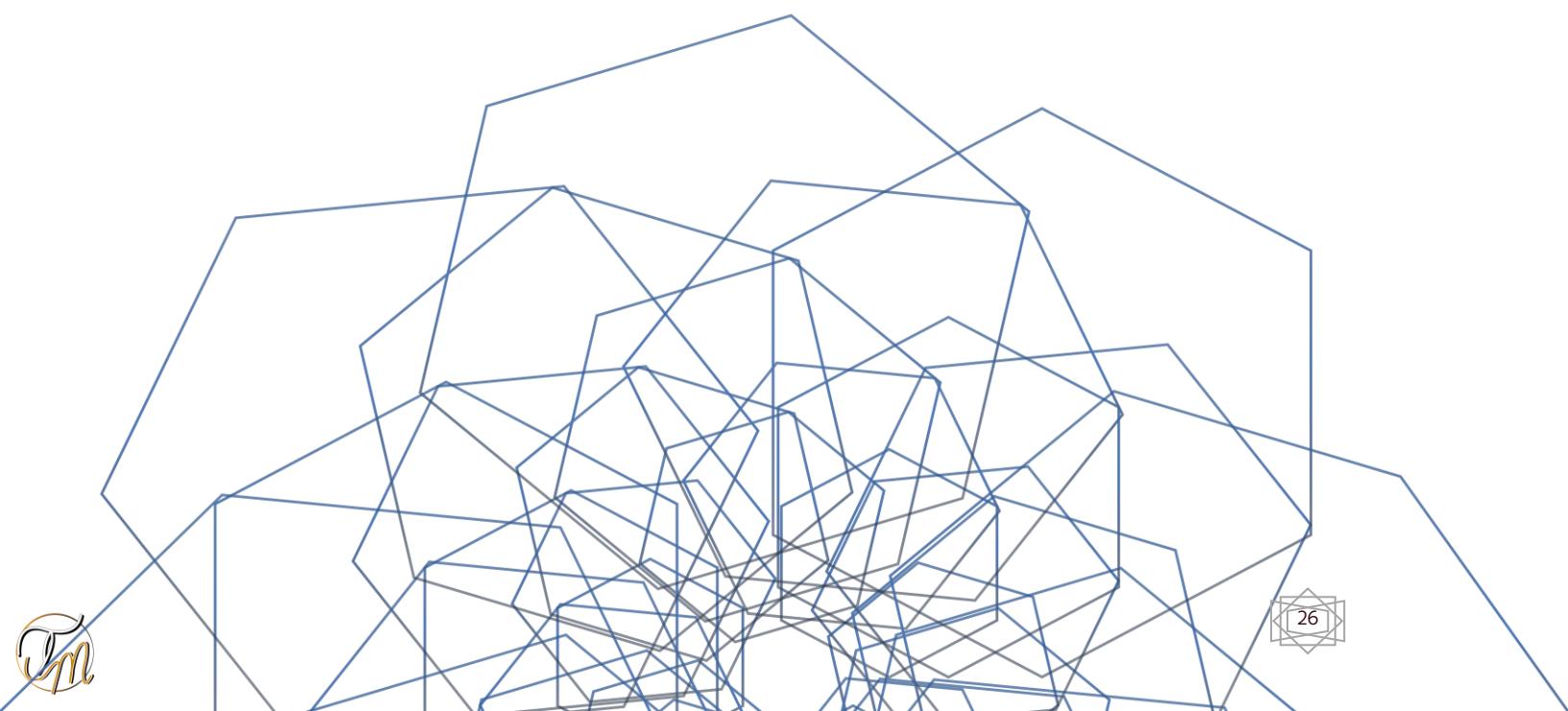
```

En el bloque interno ejecutamos la regla de la suma, pues tenemos principalmente dos bloques secuenciales: el de las **asignaciones** y el que **analizamos previamente**, dando como resultado una complejidad de  $O(n)$ .

El ciclo externo recorrerá todo el arreglo, pues previamente se inicializa en la primera posición y se desplazara al siguiente elemento con cada iteración, así hasta que el elemento actual sea nulo (el final de la lista), por lo que concluimos que este bloque tiene una complejidad  $O(n)$ .



Por regla del producto, multiplicamos las complejidades anidadas, dando como resultado una función de complejidad  $O(n^2)$ .



## FUNCIÓN HUFFMANTREE()

```
Windows - Funciones.c

1 void huffmanTree()
2 {
3     Nodolista *temp;
4     while (arbol[1] != NULL)
5     {
6         Nodolista *aux;
7         aux = (Nodolista *)malloc(sizeof(Nodolista));
8         aux->inf.frec = arbol[0]->inf.frec + arbol[1]->inf.frec;
9         aux->inf.c = '\0';
10        aux->izq = arbol[0];
11        aux->der = arbol[1];
12        for (int i = 0; i < tam - 1; i++)
13        {
14            arbol[i] = arbol[i + 1];
15        }
16        arbol[tam - 1] = NULL;
17        arbol[0] = aux;
18        for (int i = 0; i < tam - 2; i++)
19        {
20            if (arbol[i]->inf.frec >= arbol[i + 1]->inf.frec)
21            {
22                temp = arbol[i + 1];
23                arbol[i + 1] = arbol[i];
24                arbol[i] = temp;
25            }
26        }
27        tam--;
28    }
29 }
```

### DESCRIPCIÓN:

El algoritmo que realiza el árbol de huffman necesita inicialmente de un nodo temporal, y mientras que nuestro arreglo de árboles en la posición 1 sea distando del vacío comenzaremos a realizar el ciclo, con otro nodo auxiliar comenzamos la reserva de memoria correspondiente y comenzaremos con el proceso pertinente accediendo a cada uno de los elementos de la estructura que compete a los nodos involucrados en los árboles, sumando dichas frecuencias y asignando el árbol pertinente en el lado derecho o lao izquierdo, en el nodo donde estamos colocando la suma de las frecuencias de los toros dos árboles colocaremos como carácter representativo el terminador de línea o bien EOF y con un ciclo nos iremos recorriendo sobre los árboles dando al último árbol creado la igualdad con el vacío, nuevamente realizamos la comprobación con respecto a las

frecuencias y terminamos decrementando el tamaño en cuestión.

### ANÁLISIS:

Nuevamente encontramos anidaciones de ciclos, así que empezemos por el bloque más interno que posee dos ciclos:

```
Nodolista *aux;
aux = (Nodolista *)malloc(sizeof(Nodolista));
aux->inf.frec = arbol[0]->inf.frec + arbol[1]->inf.frec;
aux->inf.c = '\0';
```



```

aux->izq = arbol[0];

aux->der = arbol[1];

for (int i = 0; i < tam - 1; i++)

{

    arbol[i] = arbol[i + 1];

}

arbol[tam - 1] = NULL;

arbol[0] = aux;

for (int i = 0; i < tam - 2; i++)

{

    if (arbol[i]->inf.frec >= arbol[i + 1]->inf.frec)

    {

        temp = arbol[i + 1];

        arbol[i + 1] = arbol[i];

        arbol[i] = temp;

    }

}

tam--;

```

El código dentro del primer ciclo for es constante, pues se trata de una asignación, por otra parte el ciclo se iterará prácticamente todo el arreglo ( $n - 1$  veces), por lo que en cota O queda como  $O(n)$ .

El segundo ciclo for posee una condicional dentro: si se cumple ejecuta un bloque de complejidad  $O(1)$ , pues se trata de puras asignaciones; por otra parte si no se cumple la condición, no se ejecuta nada, por lo que podríamos decir que no tiene coste, sin embargo, hay que aplicar regla de la condicional y nos que una complejidad interna de  $O(1)$ . El ciclo por sé se ejecutará  $n - 2$  veces, por lo que se redondea en complejidad  $O(n)$ . Así todo este bloque tiene complejidad  $O(n)$ .

Finalmente aplicamos regla de la suma para el código analizado junto con las **instrucciones constantes** que acompañan a éste, así tenemos que la complejidad del bloque más interno de la función es:  $O(\text{mayor}(O(n), O(1), O(n))) = O(n)$ .

```

Nodolista *temp;
while (arbol[1] != NULL)
{
    //Bloque interno de complejidad O(n)
}

```

El ciclo while se ejecutará hasta que  $\text{árbol}[1]$  sea nulo, también si vemos las siguientes líneas en el código interno:

```

arbol[tam - 1] = NULL;
arbol[0] = aux;
//Segundo for
tam--;

```

Nos percatamos que con cada iteración el tamaño del arreglo disminuirá en uno y que el último elemento se le asigna el valor nulo, por lo que eventualmente, la segunda posición ( $\text{árbol}[1]$ ) será nula. Sin embargo, en su peor caso se ejecutara todo el arreglo, así, el ciclo externo tiene una complejidad  $O(n)$ .

Por regla del producto, la complejidad total es de:  $O(n^2)$ .

## FUNCIÓN BINARYROUTE (LISTAENLAZADA \*ARBOLIMPRESION, INT POS, INT BIT, INT IZQ, INT DER, INT \*AUXBIN)

```

Windows - Funciones.c

1 void binaryRoute(Listaenlazada *arbollImpresion, int pos, int bit, int izq, int der, int *auxBin)
2 {
3
4     //Obtendremos el binario usando como base el recorrido en preorder de un arbol
5     if (*arbollImpresion != NULL)
6     {
7
8         //Iremos insertando el valor del bit dependiendo si nos vamos por izquierda con valor 0
9         //o si vamos por la derecha con 1, en la primera llamada en el main como no hemos
10        //recorrido ninguna pos dentro del arbol por eso tendremos el valor de -1 donde no
11        //guardaremos ningun valor dentro del arreglo
12        if (pos != -1)
13        {
14            auxBin[pos] = bit;
15        }
16
17        //Si nos encontramos en un nodo hoja donde tenemos la frecuencia y la letra haremos
18        //las siguientes operaciones, en caso de que no seguiremos recorriendo el arbol
19        if ((*arbollImpresion)->izq == NULL && (*arbollImpresion)->der == NULL)
20        {
21
22            //Haremos un ciclo que dependiendo de las veces que nos hayamos desplazado en la
23            //izquierda o la derecha insertaremos los valores guardados en nuestro arreglo
24            //auxiliar en nuestra variable dentro de la estructura
25            for (int i = 0; i < (izq + der); i++)
26            {
27
28                //Igualamos los datos dentro de nuestro auxiliar con nuestro valor dentro
29                //de la estructura
30                (*arbollImpresion)->inf.binario[i] = auxBin[i];
31            }
32
33            //Como en este punto la frecuencia de las letras ya no nos es util lo remplazaremos
34            //con el numero de bites que se registraron, aun que el arreglo tiene los 8 bits,
35            //cuando escribamos en el archivo .dat esto nos sera de ayuda para
36            //insertar los valores dentro del arreglo
37            (*arbollImpresion)->inf.frec = izq + der;
38        }
39        else
40        {
41            //Dependiendo hacia donde nos desplazemos sumaremos en 1 la pos para insertar en
42            //el arreglo, ademas de que llevaremos el conteno de los desplazamiento hacia la
43            //izquierda o hacia la derecha
44            binaryRoute(&(*arbollImpresion)->izq, pos + 1, 0, izq + 1, der, auxBin);
45            binaryRoute(&(*arbollImpresion)->der, pos + 1, 1, izq, der + 1, auxBin);
46        }
47    }
48 }
49 }
```

### DESCRIPCIÓN

La ruta binaria simplemente se basa en el algoritmo de recorrer un árbol en forma postorden el cual ya fue descrito en la introducción a este documento, simplemente comenzamos de la raíz vamos a izquierda y luego por los elementos de la derecha todo esto se va realizando de una forma recursiva para que cuando el árbol se encuentre en uno de los nodos hojas (aquellos que a su lado izquierdo y derecho ya no tienen más elementos) comenzaremos con la inserción e bits dentro del arreglo binaria que se estableció en un inicio dentro de una de las estructuras. Este bit se va insertando conforme haya sido recibido, pues uno de los parámetros que estamos mandando es si se trata de un 0 o de un 1 si se mueve a la izquierda o derecha respectivamente, así como la cantidad de veces que se va moviendo de izquierda y derecha para poder realizar la inserción bits con un ciclo for que tendrá como límite la cantidad de veces

que se movió de derecha a izquierda hasta haber llegado a un nodo hoja



## ANÁLISIS

Esta función es recursiva, donde la condición inicial la encontramos hasta el final del recorrido (las hojas), pues es donde los nodos izquierdos y derechos son nulos; analicemos el coste de esta condición inicial, ubicada en el if que posee un ciclo en su interior:

```

if ((*arbolImpresion)->izq == NULL && (*arbolImpresion)->der == NULL)

{
    for (int i = 0; i < (izq + der); i++)

    {
        (*arbolImpresion)->inf.binario[i] = auxBin[i];

    }

    (*arbolImpresion)->inf.frec = izq + der;

}

else

{
    binaryRoute(&(*arbolImpresion)->izq, pos + 1, 0, izq + 1, der,auxBin);

    binaryRoute(&(*arbolImpresion)->der, pos + 1, 1, izq, der + 1,auxBin);

}

```

Como el ciclo comienza cuando se encuentra una hoja, el número de veces variará dependiendo de la altura a la que se encuentre dicha hoja en el árbol, por lo que debemos aplicar la regla del peor caso, y como sabemos, la altura máxima de un árbol es  $n$ , por lo tanto, la condición inicial es de complejidad  $O(n)$ .

Si no se cumple la condición, se hacen dos llamadas recursivas a la siguiente posición tanto de la derecha como de la izquierda, es decir,  $n + 1$ , hasta llegar a una hoja, así, nuestro modelo recursivo queda como:

$$T(n) = T(n + 1) + T(n + 1) + n = 2T(n + 1) + n$$

$$T(n) - 2T(n + 1) = 1^n n$$

Hacemos el cambio  $T(n) = x, b = 1, d = 1$ :

$$(x - 2)(x - 1)^2 = 0$$

$$r_1 = 2, r_2 = 1, r_3 = 1$$

$$T(n) = c_1 2^n + c_2 1^n + c_3 n 1^n$$

Aplicamos la condición inicial, si  $T(n) = n$ :

$$n = c_1 2^n + c_2 1^n + c_3 n 1^n$$

Vemos que  $c_1 = 0, c_2 = 0, c_3 = 1$

$$T(n) = 1n1^n = n \in O(n)$$



Windows - Funciones.c

```

1 void createBinaryList(Listaenlazada arbolImpresion, Listaenlazada *binaryList)
2 {
3
4     //Haremos este proceso mientras el valor no sea NULL, apoyandonos en el
5     //recorrido en preorder de la funcion
6     if (arbolImpresion != NULL)
7     {
8
9         //Si estamos en un nodo hoja haremos lo siguiente, caso de que no sea un
10        //nodo hoja nos desplazaremos a la izquierda o derecha
11        if (arbolImpresion->izq == NULL && arbolImpresion->der == NULL)
12        {
13
14             //en caso de que la lista binaria sea NULL, crearemos un nuevo nodo con el
15             //primer valor encontrado en el nodo hoja del arbol a este primer nodo
16             //de la lista que tendra nuestras frecuencias, letra y binario
17             if (*binaryList == NULL)
18             {
19
20                 *binaryList = (Nodolista *)malloc(sizeof(Nodolista));
21                 (*binaryList)->inf.c = arbolImpresion->inf.c;
22                 (*binaryList)->inf.frec = arbolImpresion->inf.frec;
23
24                 //Insertamos el numero binario de la letra segun el arbol de huffman
25                 for (int i = 0; i < arbolImpresion->inf.frec; i++)
26                     (*binaryList)->inf.binario[i] = arbolImpresion->inf.binario[i];
27
28                 (*binaryList)->sig = NULL;
29             }
30
31
32         else
33         {
34
35             //En caso de que no sea el primer nodo lo que haremos es de igual forma
36             //crear un nuevo nodo que insertaremos dentro de la lista con los datos
37             //del siguiente nodo hoja encontrado en el arbol
38             Listaenlazada nuevoNodo;
39             nuevoNodo = (Nodolista *)malloc(sizeof(Nodolista));
40             nuevoNodo->inf.c = arbolImpresion->inf.c;
41             nuevoNodo->inf.frec = arbolImpresion->inf.frec;
42
43             //Insertamos el valor binario del arbol en la lista para esta letra
44             for (int i = 0; i < arbolImpresion->inf.frec; i++)
45                 nuevoNodo->inf.binario[i] = arbolImpresion->inf.binario[i];
46
47             //actualizamos la cabeza de la lista enlazada
48             nuevoNodo->sig = *binaryList;
49             *binaryList = nuevoNodo;
50         }
51     }
52 }
```



Windows - Funciones.c

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

## FUNCIÓN CREATEBINARYLIST

### DESCRIPCIÓN

La función que se dedica a crear la lista binaria recibe como parámetros principales el árbol y la lista binaria, el ciclo se realizará este proceso mientras el valor no sea NULL, apoyándose en el recorrido en preorder de la función, en caso de que la lista binaria sea NULL, crearemos un nuevo nodo con el primer valor encontrado en el nodo hoja del árbol a este primer nodo de la lista que tendrá nuestras frecuencias, letra y binario.

Insertamos el numero binario de la letra según el árbol de huffman

En caso de que no sea el primer nodo lo que haremos es de igual forma crear un nuevo nodo que insertaremos dentro de la lista con los datos del siguiente nodo hoja encontrado en el árbol. Insertamos el valor binario del árbol en la lista para esta letra. actualizamos la cabeza de la lista enlazada.

~~En caso de que no estemos en un nodo hoja nos seguiremos desplazando dentro del árbol~~

```

Windows - Funciones.c

1     else
2     {
3
4         //En caso de que no estemos en un nodo hoja nos seguiremos
5         //desplazando dentro del arbol
6         createBinaryList(arbolImpresion->izq, &(*binaryList));
7         createBinaryList(arbolImpresion->der, &(*binaryList));
8     }
9 }
10 }
```

## ANÁLISIS

Esta función posee una recursividad, así que analicemos el bloque que posee a la condición inicial

```

if (*binaryList == NULL)
{
    *binaryList = (Nodolista *)malloc(sizeof(Nodolista));
    (*binaryList)->inf.c = arbolImpresion->inf.c;
    (*binaryList)->inf.frec = arbolImpresion->inf.frec;
    for (int i = 0; i < arbolImpresion->inf.frec; i++)
        (*binaryList)->inf.binario[i] = arbolImpresion->inf.binario[i];

    (*binaryList)->sig = NULL;
}

else
{
    Listaenlazada nuevoNodo;
    nuevoNodo = (Nodolista *)malloc(sizeof(Nodolista));
    nuevoNodo->inf.c = arbolImpresion->inf.c;
    nuevoNodo->inf.frec = arbolImpresion->inf.frec;
    for (int i = 0; i < arbolImpresion->inf.frec; i++)
        nuevoNodo->inf.binario[i] = arbolImpresion->inf.binario[i];
}
```

```

nuevoNodo->inf.binario[i] = arbolImpresion->inf.binario[i];

nuevoNodo->sig = *binaryList;

*binaryList = nuevoNodo;

}

```

Vemos que se trata de una condicional, donde los dos caminos son muy similares, pues ambos poseen un ciclo for en su interior y unas instrucciones secuenciales junto a éste (dichas instrucciones son de complejidad  $O(1)$ ). El ciclo for en ambos caso itera desde 0 hasta  $inf.freq$ , es decir, la frecuencia del nodo actual, por lo que en este caso, aunque sabemos que variará la frecuencia en cada nodo, es muy difícil suponer cuál es el peor caso (pues la frecuencia de un nodo es la suma de las frecuencias de sus nodos hijos o en caso de una hoja, la frecuencia del carácter en el archivo), lo que si sabemos, es que dicha frecuencia, de manera implícita está en función del tamaño del archivo (es decir,  $n$ ), pues las hojas contienen las frecuencias de los elementos de éste, así podemos suponer que la condición inicial es de cota  $O(n)$ .

Ahora veamos la llamada recursiva:

```

if (arbolImpresion != NULL)
{
    if (arbolImpresion->izq == NULL && arbolImpresion->der == NULL)
    {
        //Condición inicial de cota O(n)
    }
    else
    {
        createBinaryList(arbolImpresion->izq, &(*binaryList));
        createBinaryList(arbolImpresion->der, &(*binaryList));
    }
}

```

El bloque completo posee una condicional al inicio, sin embargo, si esta no se cumple, no se ejecuta nada (coste cero), por lo que usamos regla de la condicional y la cota de complejidad corresponderá a la de su interior, en el cual, nuevamente nos encontramos una condición, la que dicta si se cumple la condición inicial o ejecuta las llamadas recursivas. Dichas llamadas

recursivas llaman a la siguiente posición del árbol tanto por la izquierda como por la derecha, hasta encontrar una hoja ( $T(n)$ ), por ende, se cada llamada corresponde a  $T(n + 1)$ , y nuestro modelo recursivo queda como:

$$T(n) = T(n + 1) + T(n + 1) + n = 2T(n + 1) + n$$

$$T(n) - 2T(n + 1) = 1^n n$$

Hacemos el cambio  $T(n) = x, b = 1, d = 1$ :

$$(x - 2)(x - 1)^2 = 0$$

$$r_1 = 2, r_2 = 1, r_3 = 1$$

$$T(n) = c_1 2^n + c_2 1^n + c_3 n 1^n$$

Aplicamos la condición inicial, si  $T(n) = n$ :

$$n = c_1 2^n + c_2 1^n + c_3 n 1^n$$

Vemos que  $c_1 = 0, c_2 = 0, c_3 = 1$

$$T(n) = 1n 1^n = n \in O(n)$$

## FUNCIÓN CREARFRECUENCIAS



Windows - Funciones.c

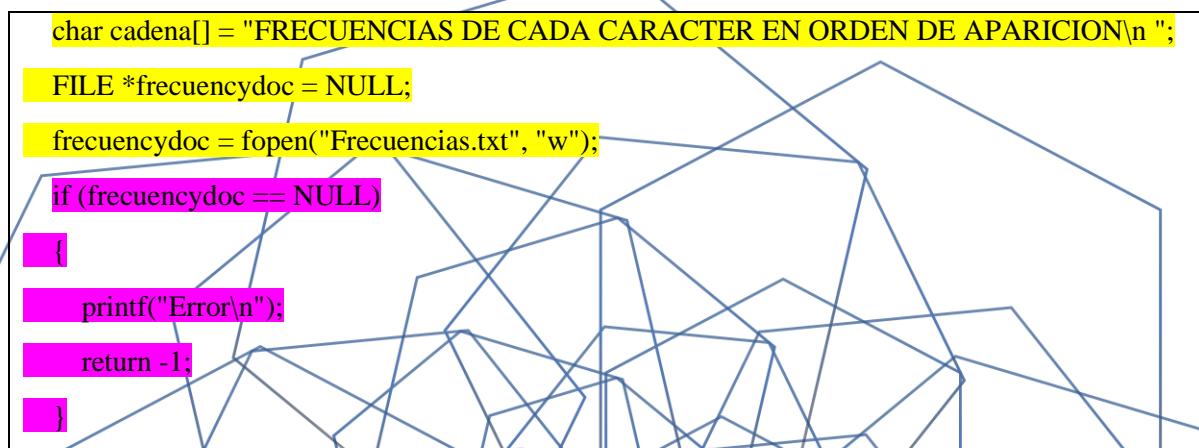
```
1 int crearFrecuenciasTxt(ListaEnlazada lista)
2 {
3     char cadena[] = "FRECUENCIAS DE CADA CARACTER EN ORDEN DE APARICION\n"; //Encabezado para el archivo
4     FILE *frecuencydoc = NULL;
5
6     frecuencydoc = fopen("Frecuencias.txt", "w"); //En caso de que el archivo no exista se crea automaticamente
7     if (frecuencydoc == NULL)
8     {
9         printf("Error\n");
10        return -1;
11    }
12
13    fputs(cadena, frecuencydoc); //Insercion de Encabezado
14    fprintf(frecuencydoc, "Frecuencias\n");
15    fprintf(frecuencydoc, "%d\n", totalDebits);
16    while (lista != NULL) //iremos iterando en toda nuestra lista, mientras el siguiente nodo no este vacio quiere decir que aun tenemos informacion que debemos imprimir en el archivo
17        //Imprimimos la letra y la frecuencia de la letra usando fprintf
18        fprintf(frecuencydoc, "%c - %d\n", lista->inf.c, lista->inf.frec);
19        lista = lista->sig;
20    }
21    fclose(frecuencydoc);
22 }
```

## **DESCRIPCIÓN**

La ejecución de esta función es muy sencilla simplemente mediante una lista se crea el archivo de frecuencias por cada uno de los caracteres, está ya se debe encontrar ordenada para que el posterior uso de esta pueda ser correctamente implementado, para esto nos auxiliamos de las funciones que ya se nos proporcionan para el manejo de archivos, mientras que nuestra lista no se encuentre vacía iremos escribiendo en el archivo de las frecuencias la frecuencia en cuestión así como el carácter al cual corresponde dicha frecuencia y posteriormente nos moveremos al siguiente elemento de la lista.

## ANÁLISIS

Esta función es sencilla, por lo que la analizaremos en un solo bloque:



```
fputs(cadena, frequencydoc); //Insercion de Encabezado  
fprintf(frequencydoc, "Frecuencias:\n" );  
fprintf(frequencydoc, "%d\n", totalDeBits);  
  
while (lista != NULL){  
    fprintf(frequencydoc, "%c - %d\n", lista->inf.c, lista->inf.frec);  
    lista = lista->sig;  
}  
  
fclose(frequencydoc);
```

Al inicio encontramos algunas **instrucciones constantes** seguida de una **condicional de igual** orden constante en su interior, pero si no se cumple pasamos al bloque que posee el **ciclo for**, que recorre todo el arreglo y por lo tanto, tiene complejidad  $O(n)$ . Finalmente, aplicando regla de la condicional, esta función es de cota  $O(n)$ .

## FUNCIÓN CREARCODIFICACIONDAT

```
Windows - Funciones.c

1 int crearCodificacionDAT(Listaenlazada listaBinaria, FILE *archivo)
2 {
3
4     //Declaramos un variable para escribir el archivo de salida
5     FILE *codificacion = NULL;
6
7     //Declaramos una listaEnlazada auxiliar para dezplazarnos
8     //en la lista sin perder el origen o cabeza
9     Listaenlazada listaAuxiliar = listaBinaria;
10
11    //Declaramos la variable donde guardaremos el caracter
12    //escaneado del archivo original
13    char caracterEscaneado;
14
15    //Variable auxiliar que tendra el caracter de la lista que coincide
16    //con el escaneado (Necesitado para usar el strcmp())
17    char caracterLista;
18
19    // Dar mas espacio al arreglo
20    int posicionBit = 7;
21    int posicionByte = 0;
22    int queryValue;
23
24    int random;
25
26    unsigned char arreglo[1024]; //Inicializamos una arreglo con 1024 posiciones
27
28    //Inicializando todo el arreglo con el valor de 0 por dafault
29    for (int i = 0; i < 1024; i++)
30        arreglo[i] = 0;
31
32    //Abrimos o creamos el archivo donde tendremos la codificacion
33    //en modo escritura binaria
34    codificacion = fopen("codificacion.dat", "wb");
35
36    //Verificamos que el archivo se haya abierto de manera correcta
37    if (codificacion == NULL){
38        printf("Error al crear/abrir codificacion.dat \n");
39        return -1;
40    }
```

```

Windows - Funciones.c
1 //Igualamos el carácter de la lista a la variable de apoyo
2 caracterLista = listaAuxiliar->inf.c;
3
4 //Si encontramos la letra rompemos el ciclo
5 if (cmpchar(caracterLista, caracterEscaneado))
6 {
7     break;
8 }
9 }
10
11 for (int j = 0; j < listaAuxiliar->inf.frec; j++)
12 {
13     queryValue = CONSULTARBIT(arreglo[posicionByte], posicionBit);
14     random = listaAuxiliar->inf.binario[j];
15     if (queryValue != random)
16     {
17         CAMBIA(arreglo[posicionByte], posicionBit);
18     }
19     posicionBit--;
20     totalDeBits++;
21
22     if (posicionBit == -1)
23     {
24         posicionBit = 7;
25         posicionByte++;
26     }
27
28     if (posicionByte == 1024)
29     {
30
31         fwrite(arreglo, sizeof(unsigned char), 1024, codificacion);
32         posicionByte = 0;
33     }
34 }
35 } while (!feof(archivo));
36

```

```

Windows - Funciones.c
1 //Rebobinamos hasta el inicio del archivo donde tenemos nuestros
2 //datos originales
3 rewind(archivo);
4 //Mientras no lleguemos al final del archivo haremos lo siguiente
5 do
6 {
7
8     //Escaneamos el carácter del archivo
9     caracterEscaneado = fgetc(archivo);
10
11    //Escanea el End Of File, si se llega a este valor
12    //se romperá el ciclo
13
14    if (feof(archivo) == 16) //Igualar a 1 en ejecución de Linux
15        break;
16
17    //Volvemos a la cabeza de la lista enlazada
18    listaAuxiliar = listaBinaria;
19
20    //Lo buscamos en la lista enlazada
21    for (listaAuxiliar; listaAuxiliar != NULL; listaAuxiliar = listaAuxiliar->sig)
22    {

```

#### DESCRIPCIÓN:

Declaramos un variable para escribir el archivo de salida, declaramos una listaEnlazada auxiliar para desplazarnos en la lista sin perder el origen o cabeza, Declaramos la variable donde guardaremos el carácter escaneado del archivo original

Inicializamos un arreglo de 1024 bytes en toda posición con el numero 0, realizamos la apertura o bien creación de un archivo en el cual se almacena la codificación generada por el programa, rebobinamos hasta el inicio de nuestro archivo por seguridad y tener garantía de que estamos al inicio de nuestro archivo y mientras no lleguemos al final del archivo realizaremos el proceso de escanear el carácter, en caso de que llegue al terminador el ciclo terminara en caso contrario continuara y realizará la búsqueda del carácter en cuestión dentro de nuestra lista enlazada el cual será almacenado en una de las variables que se crearon,

comparamos los bits de estos dos caracteres gracias a la función que un poco más adelante explicaremos, la cual fue necesaria para realizar la comparación con respecto al nivel binario, operaciones que fueron más sencillas de implementar gracias a aquel programa que fue proporcionado por el profesor de la unidad de aprendizaje, en caso de que encontremos la letra rompemos el ciclo, en el caso contrario continuamos, posterior a esto continuamos con el resto del do-while para la frecuencia del carácter en cuestión realizaremos un ciclo donde iremos realizando la consulta de los bits en donde iremos comparando e incrementando la posición del bit, pues por la forma en que se manejan comenzamos con la posición de este en el número 7 e iremos decrementando hasta el numero 0, para de esta forma garantizar que realmente no siempre estamos cambiando el mismo bit o un bit de forma aleatoria e iremos cambiando conforme necesitemos, cuando hayamos completado los 7 bits y llegado a -1 (pues 0 también es una de las posiciones de los arreglo en lenguaje C), debemos reiniciar el byte nuevamente lo cual quiere decir que debemos comenzar nuevamente desde el bit número 7 y con fines prácticos y didácticos incrementamos la cantidad de bytes que estamos utilizando para la codificación para así ver posteriormente que tanta memoria estamos comprimiendo y si realmente está resultando efectivo nuestro algoritmo o no

```
Windows - Funciones.c

1 if (posicionByte < 1024)
2 {
3     if (posicionBit == 7)
4     {
5         fwrite(arreglo, sizeof(unsigned char), posicionByte, codificacion);
6     }
7     else
8     {
9         fwrite(arreglo, sizeof(unsigned char), posicionByte + 1, codificacion);
10    }
11 }
12
13 printf("La cantidad de bits neecarios es de: %d\n",totalDeBits);
14 printf("La cantidad de Bytes iniciales del archivo es de %d \n", totalDeBytes); //Imprimimos la cantidad de bits necesarios en el caso de compilacion
15 printf("La cantidad de Bytes finales del archivo es de %d \n", posicionByte+1);
16 //Imprimimos la cantidad de bits necesarios en el caso de compilacion
17 printf("El nivel de compresion del archivo es de: %d.2 ", (totalDeBytes / (posicionByte + 1)) * 100);
18
19 fclose(codificacion);
20 }
21
```

**ANÁLISIS:**

Analicemos el inicio de la función:

```

FILE *codificacion = NULL;
Listaenlazada listaAuxiliar = listaBinaria;
char caracterEscaneado;
char caracterLista;
int posicionBit = 7;
int posicionByte = 0;
int queryValue;
int random;
unsigned char arreglo[1024];
for (int i = 0; i < 1024; i++)
    arreglo[i] = 0;
codificacion = fopen("codificacion.dat", "wb");
if (codificacion == NULL){
    printf("Error al crear/abrir codificacion.dat \n");
    return -1;
}
rewind(archivo);

```

Las primeras instrucciones son asignaciones, por lo que son constantes, pasamos a un ciclo for que siempre se iterará 1024 veces, es decir  $O(1024) = O(1)$ , después tenemos una llamada a la función "fopen" seguido de una condicional, dentro de ésta, solo se ejecutan dos instrucciones constantes, sin embargo, en otro caso se tiene una llamada a la función "rewind". Nuestra cota por regla de la suma queda como:

$$O(\text{mayor}(O(1), O(\text{fopen}), O(\text{rewind})))$$

Supongamos que tanto "fopen" como "rewind" son de orden constante.  $O(1)$ .

Pasemos al siguiente bloque de la función, donde tenemos dos ciclos anidados, empezemos por los for internos:

```

for (listaAuxiliar; listaAuxiliar != NULL; listaAuxiliar = listaAuxiliar->sig)
{
    caracterLista = listaAuxiliar->inf.c;
    if (cmpchar(caracterLista, caracterEscaneado))

```

```

    }

    break;

}

}

for (int j = 0; j < listaAuxiliar->inf.frec; j++)

{

    queryValue = CONSULTARBIT(arreglo[posicionByte], posicionBit);

    random = listaAuxiliar->inf.binario[j];

    if (queryValue != random)

    {

        CAMBIA(arreglo[posicionByte], posicionBit);

    }

    posicionBit--;

    totalDeBits++;

    if (posicionBit == -1)

    {

        posicionBit = 7;

        posicionByte++;

    }

    if (posicionByte == 1024)

    {

        fwrite(arreglo, sizeof(unsigned char), 1024, codificacion);

        posicionByte = 0;

    }

}

```

El primer for recorrerá "listaAuxiliar" una posición a la vez hasta que se cumpla la condición interna de este, donde podemos ver que se hace una llamada a la función "cmpchar", que es de orden constante como veremos más adelante, sin embargo, como no sabemos en qué posición del arreglo se cumplirá dicha condición, debemos aplicar regla del peor caso, que es cuando el último elemento del arreglo cumple dicha condición, por lo tanto, es  $O(n)$ .

El segundo for, que itera desde cero hasta la frecuencia del nodo actual, empieza con una cuantas asignaciones ( $O(1)$ ), seguido de unas cuantas condicionales donde en ambos caminos, el costo de sus instrucciones es constante (si suponemos que CONSULTARBIT y CAMBIA son de orden constante); así hasta la última condicional donde en el camino principal encontramos a una llamada a la función "fwrite", por lo que aplicando regla de la suma, el



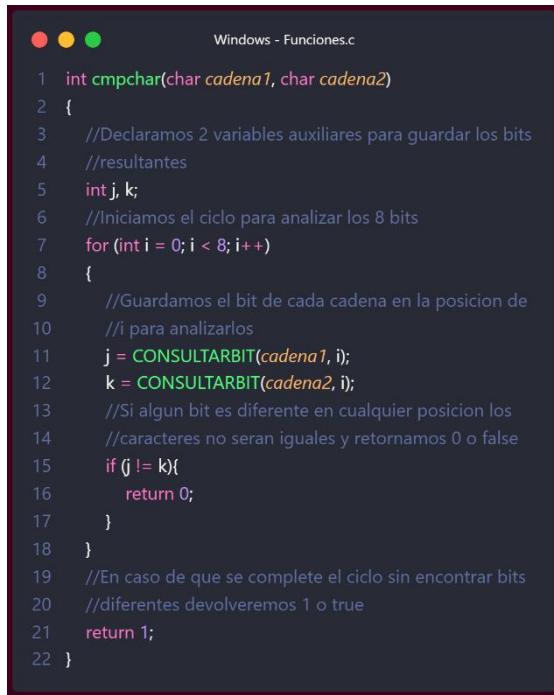
bloque interno de este for es de complejidad:  $O(\max(O(1), O(\text{fwrite}))$ ), nuevamente por comodidad, haremos constante a "fwrite". Finalmente debemos aplicar regla del peor caso para las iteraciones del ciclo for (pues el número de estas variará de acuerdo a la frecuencia del nodo actual), y en una función anterior vimos que podemos aproximar este peor caso a una complejidad lineal, así la cota del segundo for es  $O(n)$ . Como los dos ciclos for están en secuencia, se aplica regla de la suma y este bloque queda con una complejidad final de  $O(n)$ .

Analicemos el do que contiene a estos for:

```
do
{
    caracterEscaneado = fgetc(archivo);
    if (feof(archivo) == 1)
        break;
    listaAuxiliar = listaBinaria;
    // Bloque de complejidad O(n)
} while (!feof(archivo));
```

Al inicio de cada iteración hacemos una llamada a la función "fgetc", la cual saca el byte de la posición actual del apuntador al archivo y posteriormente recorre en uno dicho apuntador, por lo que solo hicimos una asignación y una operación aritmética, así que esta función es de orden constante, luego tenemos una comparación para entornos Linux donde el coste cuando entra es cero (pues rompe el ciclo y no se ejecutan más instrucciones), por esta razón y aplicando regla de la condicional, las complejidad de esta condicional será la del camino donde no se cumplió la condición, y vemos que dicho camino nos conduce al bloque previamente analizado de los dos for; aplicando regla de la suma, el bloque interno del do es de complejidad lineal ( $O(n)$ ). Finalmente vemos que se iterará todo el archivo ( $n$  veces), pues la condición para que se rompa el ciclo es que nos encontramos en el final del archivo, aplicamos regla del producto y la complejidad final de toda esta función es  $O(n^2)$ .

## FUNCIÓN CMPCHAR



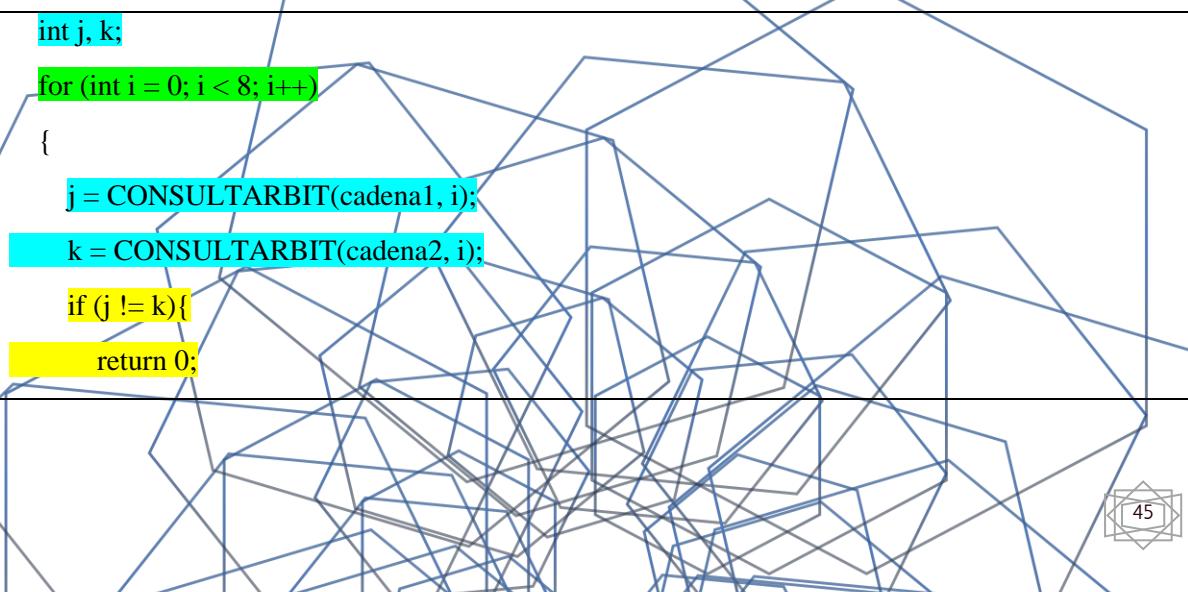
```
Windows - Funciones.c
1 int cmpchar(char cadena1, char cadena2)
2 {
3     //Declaramos 2 variables auxiliares para guardar los bits
4     //resultantes
5     int j, k;
6     //Iniciamos el ciclo para analizar los 8 bits
7     for (int i = 0; i < 8; i++)
8     {
9         //Guardamos el bit de cada cadena en la posicion de
10        //i para analizarlos
11        j = CONSULTARBIT(cadena1, i);
12        k = CONSULTARBIT(cadena2, i);
13        //Si algun bit es diferente en cualquier posicion los
14        //caracteres no seran iguales y retornamos 0 o false
15        if (j != k){
16            return 0;
17        }
18    }
19    //En caso de que se complete el ciclo sin encontrar bits
20    //diferentes devolveremos 1 o true
21    return 1;
22 }
```

## DESCRIPCIÓN

Esta función nos sirve para poder ir comparando los bits de cada uno de los caracteres para ver si es el mismo carácter o es un carácter diferente, esto se realiza mediante la implementación de las funciones que el profesor proporciono con el manejo de bits y un ciclo for que va iterando en cada una de las posiciones del arreglo, como sabemos un car posee 1 byte de memoria, lo cual son 8 bits, es por eso que vamos iterando sobre cada uno de esos bits para poder saber si se va a someter a un cambio o no.

## ANÁLISIS

Analicemos todo el código.

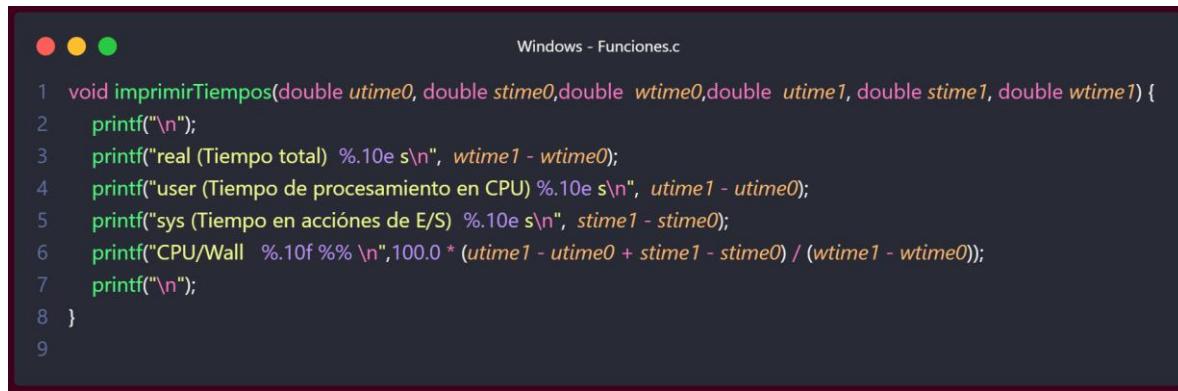


```
    }
}
return 1;
```

La complejidad de esta función radicará en el ciclo for, pues antes de éste y después de él solo encontramos dos instrucciones constantes. Dentro del ciclo for se ejecutan dos instrucciones constantes y después una condicional que puede romper el ciclo, sin embargo, el ciclo solo puede iterarse 8 veces como máximo (pues estamos recorriendo un bit), así que el número de iteraciones posibles va de 1 a 8; aplicando regla del peor caso la complejidad de esta función es  $O(8) = O(1)$ .

### FUNCIÓN IMPRIMIRTIEMPOS

Función únicamente de tipo auxiliar para poder realizar la impresión de los tiempos y poder extraer los datos que se necesitan para poder conocer la cantidad de tiempo que el programa ha demorado, debemos recordar que esta función utiliza la librería de Tiempo que solo se encuentra disponible en el sistema operativo de Linux, por lo cual en las ejecuciones realizadas en Windows no es posible realizar dichas mediciones de tiempo, simplemente la obtención de cantidad de bytes que fueron obtenidos



Windows - Funciones.c

```
1 void imprimirTiempos(double utime0, double stime0, double wtime0, double utime1, double stime1, double wtime1) {
2     printf("\n");
3     printf("real (Tiempo total) %.10e s\n", wtime1 - wtime0);
4     printf("user (Tiempo de procesamiento en CPU) %.10e s\n", utime1 - utime0);
5     printf("sys (Tiempo en acciones de E/S) %.10e s\n", stime1 - stime0);
6     printf("CPU/Wall %.10f %%\n", 100.0 * (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
7     printf("\n");
8 }
```

## DECODIFICACIÓN

La parte de Decodificación reutiliza varias de las funciones que ya hemos empleado para la creación de la codificación por lo cual sería redundante volver a realizar la descripción y análisis de estas (aunque pueden ser encontradas dentro de la parte de anexos en donde tenemos situado y todo el código que permite el funcionamiento del código), por lo que solo se procede a realizar el análisis de aquellas funciones que tuvieron que ser implementadas dentro del archivo de Decodificación las cuales son las siguientes:

```
Windows - FuncionesD.c

1 void invertList(Listaenlazada *lista, Listaenlazada *listalnvertida)
2 {
3     //Creamos un auxiliar para dezplazarnos en la lista original
4     Listaenlazada listaAuxiliar = *lista;
5
6     //Haremos este procedimiento hasta haber concluido con los elementos de la lista
7     while(listaAuxiliar != NULL){
8
9         //En caso de que la lista que tendra los datos invertidos (al derecho) no haya
10        //sido inicializada lo que haremos es iniciarla con los valores del nodo cabeza
11        //de la lista original
12        if (*listalnvertida == NULL){
13
14            //Asignamos memoria para este nodo cabeaa
15            *listalnvertida = (Nodolista *)malloc(sizeof(Nodolista));
16
17            //guardamos los valores de la lista original en la nueva
18            (*listalnvertida)->inf.c = listaAuxiliar->inf.c;
19            (*listalnvertida)->inf.frec = listaAuxiliar->inf.frec;
20            //Declaramos la siguiente posicion igual a NULL
21            (*listalnvertida)->sig = NULL;
22            //Nos dezplazamos en una posicion en la lista auxiliar
23            listaAuxiliar = listaAuxiliar->sig;
24        }
25        else{
26            //En caso de que la lista ya haya sido iniciada haremos el mismo procedimiento
27            //anterior usando un nuevo nodo esta vez
28            Listaenlazada nuevoNodo;
29
30            //Asignamos memoria para este nodo cabeaa
31            nuevoNodo = (Nodolista *)malloc(sizeof(Nodolista));
32
33            //guardamos los valores de la lista original en la nueva
34            nuevoNodo->inf.c = listaAuxiliar->inf.c;
35            nuevoNodo->inf.frec = listaAuxiliar->inf.frec;
36
37            //actualizamos la cabeza de la lista enlazada
38            nuevoNodo->sig = *listalnvertida;
39            *listalnvertida = nuevoNodo;
40
41            //Nos dezplazamos en una posicion en la lista original
42            listaAuxiliar = listaAuxiliar->sig;
43        }
44    }
45 }
```

### DESCRIPCIÓN

Como ya se mencionó estamos utilizando una lista LIFO, last in first out, por lo cual tenemos de cierta forma la lista al revés, lo último que hemos insertado realmente es lo primero a lo que podemos acceder en caso de realizar una impresión lo último que insertamos será lo primero que podamos ver, por lo cual es necesario realizar una inversión de la lista para poder acceder a los elementos de la forma correcta (esta situación al momento de concluir la elaboración del código mediante esta idea, pudo haber sido evitada si se implementaba otro tipo de lista o una lista doblemente enlazada), continuando, realizamos la asignación de memoria así como el almacenamiento de los caracteres y frecuencia en la nueva lista, actualizamos la cabeza de la lista enlazada y nos vamos desplazando, este

proceso se realizara hasta que la lista no este vacía, para poder seguir realizando el ordenamiento de los elementos

## ANÁLISIS

Empecemos con el bloque interno del ciclo do:

```

if (*listaInvertida == NULL){

    *listaInvertida = (Nodolista *)malloc(sizeof(Nodolista));

    (*listaInvertida)->inf.c = listaAuxiliar->inf.c;

    (*listaInvertida)->inf.frec = listaAuxiliar->inf.frec;

    (*listaInvertida)->sig = NULL;

    listaAuxiliar = listaAuxiliar->sig;

}

else{

    Listaenlazada nuevoNodo;

    nuevoNodo = (Nodolista *)malloc(sizeof(Nodolista));

    nuevoNodo->inf.c = listaAuxiliar->inf.c;

    nuevoNodo->inf.frec = listaAuxiliar->inf.frec;

    nuevoNodo->sig = *listaInvertida;

    *listaInvertida = nuevoNodo;

    listaAuxiliar = listaAuxiliar->sig;

}

```

Está compuesto por una condicional, donde en el **primer camino**, se realiza una serie de asignaciones, cosa que de igual manera se realiza en el **segundo**, así, la complejidad del bloque interno es constante  $O(1)$ .

Ahora pasemos al ciclo per se:

```

Listaenlazada listaAuxiliar = *lista;

while(listaAuxiliar != NULL){

    //Bloque de complejidad O(1)

}

```

La condición para que el ciclo pare es que se llegue al final de "listaAuxiliar", y dicha lista se va desplazando en uno con cada iteración (pues en los dos caminos de la condicional anterior encontramos la instrucción: "**listaAuxiliar = listaAuxiliar->sig**"), por lo que al final se habrá iterado por todo el arreglo. Aplicando regla del producto, la complejidad de esta función es:  $O(n)$ .



## FUNCIÓN CREADERDECODIFICACIÓN

```
Windows - FuncionesD.c

1 int crearDecodificacion(Listaenlazada arbolDeHuffman){
2     //Creamos una variable para leer el archivo con la codificacion
3     FILE *codificado = NULL;
4
5     //Creamos una variable para escribir el archivo decode
6     FILE *decode = NULL;
7
8     //Declaramos una variable auxiliar de tipo arbol que nos ayudara
9     //para recorrer el arbol en el orden en que se presentan los bits
10    Listaenlazada arbolAuxiliar = arbolDeHuffman;
11    //Variable para almacenar el byte escaneado
12    unsigned char caracterEscaneado;
13    //Variable que nos determina en que posicion estamos en los bits
14    int posicionBit = 7;
15    //Posicion que nos determina en que posicion del byte de codificado estamos
16    int posicionByte = 0;
17    //variable que tendra el valor escaneado del byte que se haya escaneado
18    //mediante fgetc
19    int valorBinario;
20    bool cambiarByte;
21    //Abrimos el archivo codificado en modo lectura binaria
22    codificado = fopen(nombreArchivoCodificacion, "rb");
23    //Abrimos el archivo decode en modo escritura normal
24    decode = fopen("decodificacion.txt", "w");
25
26    //Comprobamos que el archivo codificado se haya abierto correctamente
27    if(codificado == NULL){
28
29        printf("Error en Codificacion\n");
30        return -1;
31    }
32
33    //comprobamos que el archivo decode se haya abierto correctamente
34    if(decode == NULL){
35
36        printf("Error en Decodificacion\n");
37        return -1;
38    }
39}
```

```
Windows - FuncionesD.c

1 do{
2
3     //Escaneamos el primer caracter dentro de codificado
4     caracterEscaneado = fgetc(codificado);
5
6     //Escanea el End Of File, si se llega a este valor
7     //se rompera el ciclo
8     if (feof(codificado) == 1){
9         fputc(arbolAuxiliar->inf.c, decode);
10        break;
11    }
12
13    //Iniciamos en el nodo raiz del arbol
14    cambiarByte = false;
15
16    //Escaneamos el primer bit del caracter escaneado
17    do{
18
19        if ((arbolAuxiliar->izq == NULL) && (arbolAuxiliar->der == NULL)){
20
21            fputc(arbolAuxiliar->inf.c, decode);
22            arbolAuxiliar = arbolDeHuffman;
23        }
24        else{
25            valorBinario = CONSULTARBIT(caracterEscaneado, posicionBit);
26            if (valorBinario == 0){
27                arbolAuxiliar = arbolAuxiliar->izq;
28
29            }else{
30                arbolAuxiliar = arbolAuxiliar->der;
31            }
32            posicionBit--;
33            if (posicionBit == -1){
34                posicionBit = 7;
35                cambiarByte = true;
36            }
37        }
38    }
39    }while(cambiarByte != true);
40 }while(!feof(codificado));
41 }
```

## DESCRIPCIÓN

El funcionamiento de la parte de decodificación es muy similar a cuando estamos realizando la codificación del archivo, comenzamos creando un archivo que lleve el nombre de decodificación, así como la apertura del archivo que posee las frecuencias y la codificación que previamente ya obtuvimos gracias a nuestro programa anterior.

Procedemos a realizar la obtención de cada uno de los caracteres escaneados del archivo de decodificación.

Para este proceso de decodificación es necesario irnos moviendo nuevamente dentro del árbol de Huffman, ya que este es el que posee los caracteres y la frecuencia que estos poseen en sus nodos hoja, proceso que iremos realizando de una forma iterativa y no recursiva como generalmente lo hacemos, pues aquí la recursión nos afectara para la aplicación de las distintas condiciones que estamos aplicando, nos moveremos conforme realicemos la consulta de bit, pues recordemos que a la izquierda deberíamos tener 0 y a la derecha debemos tener 1, es así como nos vamos moviendo a lo largo de nuestro árbol, cuando por fin hemos llegado a un nodo hoja, es el momento de realizar la escritura de dicho carácter que esta almacenaba dentro de nuestro archivo de decodificación. De igual forma es necesario realizar el reinicio de la posición del bit para poder ir iterando en cada una de las posiciones de forma lógica por lo cual igual debe incrementar el byte, para no siempre estar iterando sobre las mismas posiciones y realizar una sobreescritura de lo que ya hemos obtenido

## ANÁLISIS

Iniciemos con el bloque más interno de la función:

```
caracterEscaneado = fgetc(codificado);
if (feof(codificado) == 1) {
    fputc(arbolAuxiliar->inf.c, decode);
    break;
}
cambiarByte = false;
do{
    if ((arbolAuxiliar->izq == NULL) && (arbolAuxiliar->der == NULL)){
        fputc(arbolAuxiliar->inf.c, decode);
        arbolAuxiliar = arbolDeHuffman;
    }
}
```

```

        else{

            valorBinario = CONSULTARBIT(caracterEscaneado, posicionBit);

            if (valorBinario == 0){

                arbolAuxiliar = arbolAuxiliar->izq;

            }else{

                arbolAuxiliar = arbolAuxiliar->der;

            }

            posicionBit--;

            if (posicionBit == -1){

                posicionBit = 7;

                cambiarByte = true;

            }

        }

    }while(cambiarByte != true);
}

```

Empezamos con una instrucción constante, seguida de una condicional, donde el costo de entrar es constante y si no entra pasará al costo del do anidado, el cual, aunque en un principio parecería recorrer muchas posiciones (casi de forma lineal), en realidad solamente recorre un byte, por lo que su peor caso es  $O(8) = O(1)$ .

Ahora continuemos con el resto de la función:

```

FILE *codificado = NULL;

FILE *decode = NULL;

Listaenlazada arbolAuxiliar = arbolDeHuffman;

unsigned char caracterEscaneado;

int posicionBit = 7;

int posicionByte = 0;

int valorBinario;

bool cambiarByte;

codificado = fopen(nombreArchivoCodificacion, "rb");

```

```

decode = fopen("decodificacion.txt", "w");

if(codificado == NULL){

    printf("Error en Codificacion\n");
    return -1;
}

if(decode == NULL){

    printf("Error en Decodificacion\n");
    return -1;
}

do{

    //Código de complejidad O(n)

}while(!feof(codificado));

```

Las primeras instrucciones son constantes, ya que solo se tratan de declaraciones y asignaciones, luego tenemos un par de condicionales en las cuales si entramos, el costo es constante, porque solo hacen "printf" y un retorno, sin embargo, si ninguna de esta se cumple pasamos al ciclo do, el cual recorre todo el archivo (pues la condición de detenerse es llegar al final de este), y dentro de este ciclo ejecuta operaciones constante, por lo que aplicando regla del producto, la complejidad de este do es  $O(n)$ , y aplicando regla de las condicionales con los if anteriores, la complejidad de esta función es  $O(n)$ .

## PRUEBAS TABLAS COMPARATIVAS Y GRÁFICAS

TABLA DE PORCENTAJE DE COMPRESIÓN DE ARCHIVOS CON DIFERENTES TAMAÑOS DE BYTES.

Nombre archivo	Tamaño de bytes	Porcentaje de compresión
texto1.txt	10	60%
texto2.txt	15977	47.21%
imagen1.bmp	7332626	2.81%
imagen2.bmp	67306434	93.62%
imagen3.bmp	12505782	15.30%
imagen1.png	159383	0%
imagen2.png	105771	0%
image3.png	413920	0.01%
pdf1.pdf	376335	5.97%
pdf2.pdf	1594017	0.54%
pdf3.pdf	619889	0.05%

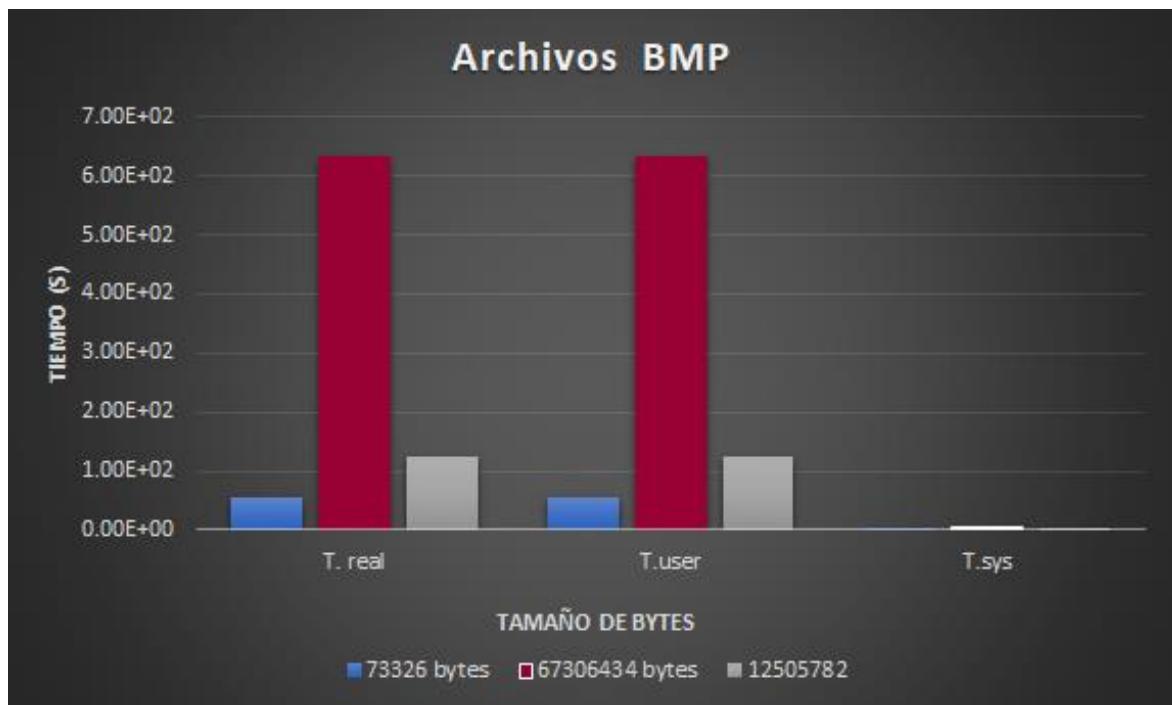
## GRÁFICAS DE TIEMPO DE EJECUCIÓN CODIFICACIÓN

### ARCHIVOS TXT

Tamaño en bytes	Tiempo real	Tiempo usuario	Tiempo sistema
10	5.3596496582e-04 s	0.0000000000e+00 s	4.1100000000e-04 s
15977	2.6309013367e-02 s	2.6309013367e-02 s	0.0000000000e+00 s

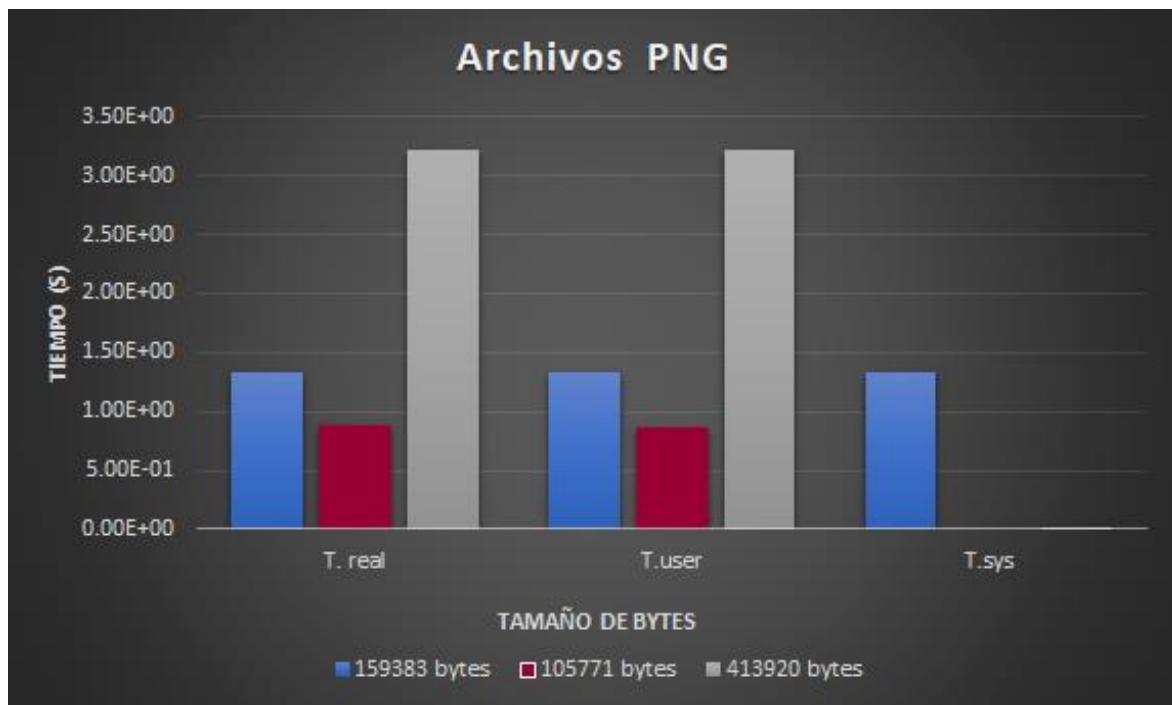
### ARCHIVOS BMP

Tamaño en bytes	Tiempo real	Tiempo usuario	Tiempo sistema
73326	54.805311918	54.635752	0.107956
67306434	630.207094	630.207094	2.011667
12505782	124.95460796	124.618306	0.275956



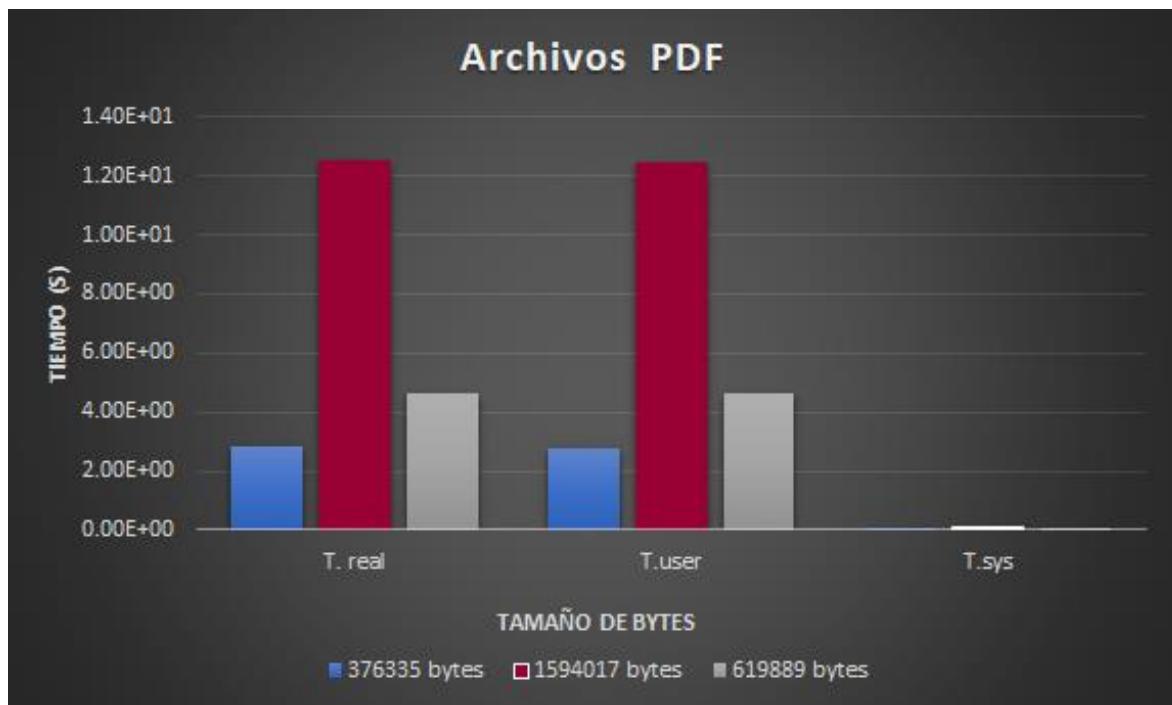
### ARCHIVOS PNG

Tamaño en bytes	Tiempo real	Tiempo usuario	Tiempo sistema
159383	1.3312711716	1.330592	1.330592
105771	0.86451196671	0.858072	0
413920	3.2223680019	3.214092	0.008006



### ARCHIVOS PDF

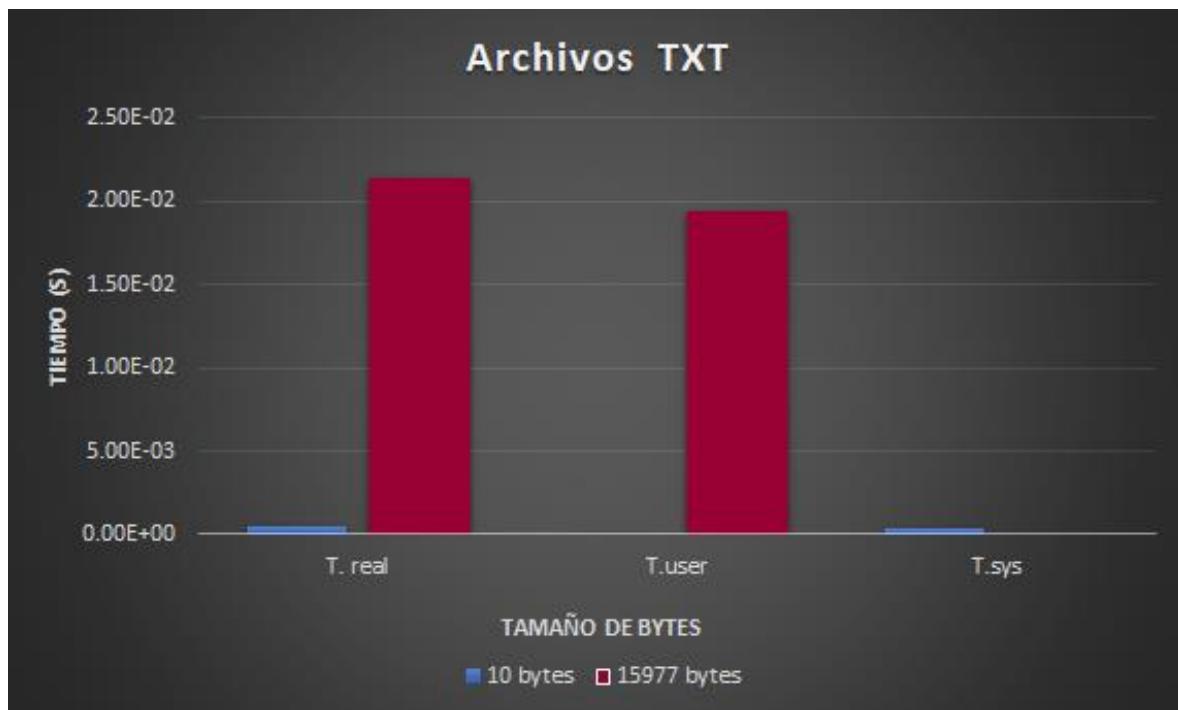
Tamaño en bytes	Tiempo real	Tiempo usuario	Tiempo sistema
376335	2.7989549637	2.741826	0.055999
105771	12.507796049	12.432315	0.031995
413920	4.6576499939	4.64943	0.003999



## DECODIFICACIÓN

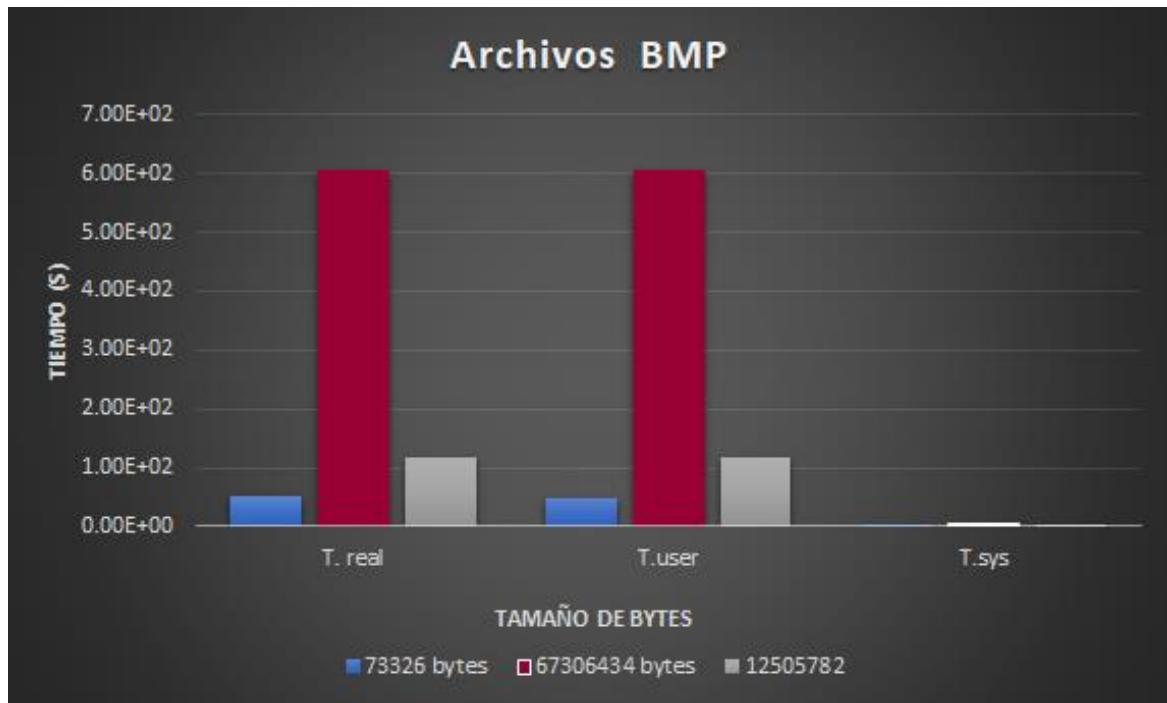
### ARCHIVOS TXT

Tamaño en bytes	Tiempo real	Tiempo usuario	Tiempo sistema
10	5.3596496582e-04 s	0.0000000000e+00 s	4.1100000000e-04 s
15977	2.6309013367e-02 s	2.6309013367e-02 s	0.0000000000e+00 s



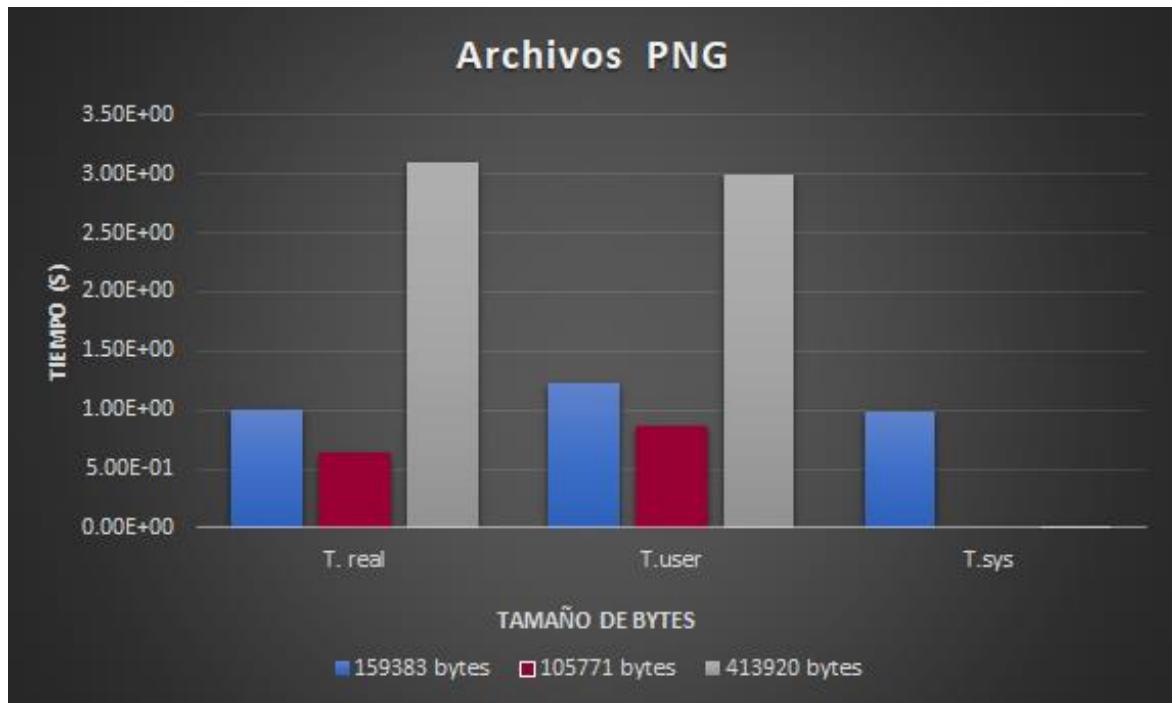
### ARCHIVOS BMP

Tamaño en bytes	Tiempo real	Tiempo usuario	Tiempo sistema
73326	54.805311918	54.635752	0.107956
6730643	630.207094	630.207094	2.011667
1250578	124.95460796	124.618306	0.275956



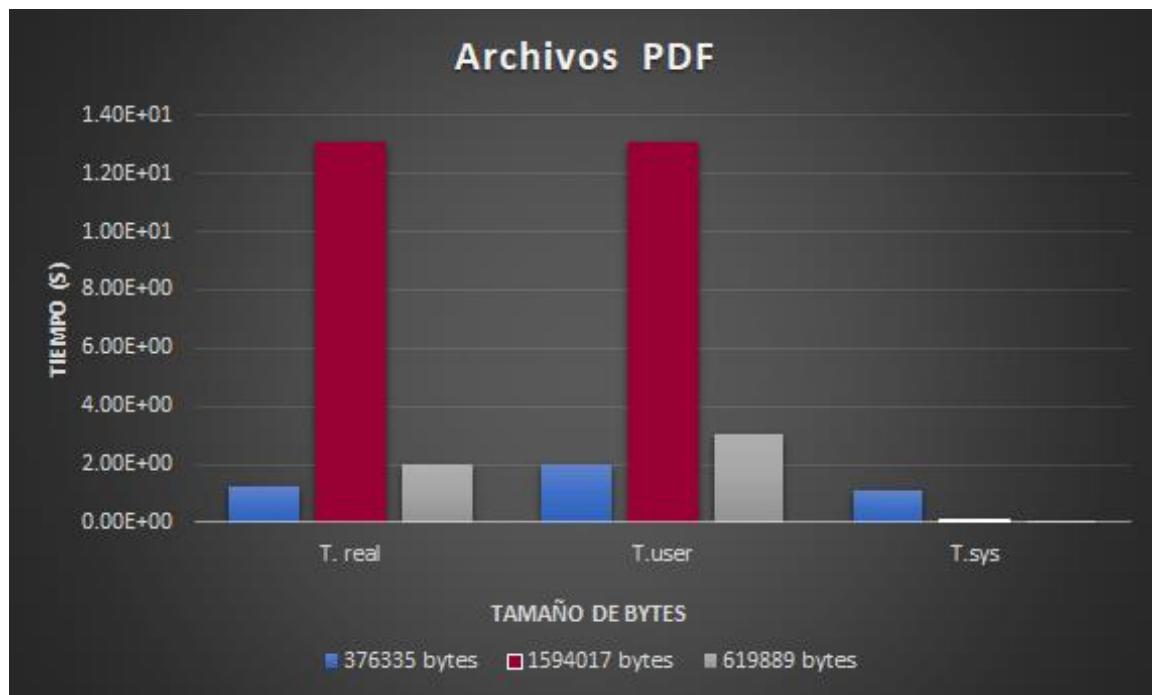
### ARCHIVOS PNG

Tamaño en bytes	Tiempo real	Tiempo usuario	Tiempo sistema
159383	1.002766437	1.123198765	0.982777463553
105771	0.625789001	0.85548373	0
413920	3.091862408	3.000693842	0.007136523093



### ARCHIVOS PDF

Tamaño en bytes	Tiempo real	Tiempo usuario	Tiempo sistema
376335	1.2387319083	2.0019341234	0.054476
1594017	10.019238412	11.01053039993	0.100062662
619889	2.0233943669	3.0720039432	0.0037641345



## APROXIMACIONES DE FUNCION COMPLEJIDAD

### CODIFICACIÓN

Archivos TXT

GRAFICA	APROXIMACION
$F(x)=-0.0192199775533x+0.411$	

Imagen bmp

GRAFICA	APROXIMACION
$F(x)=-0.146245059288x+2.931343873517$	

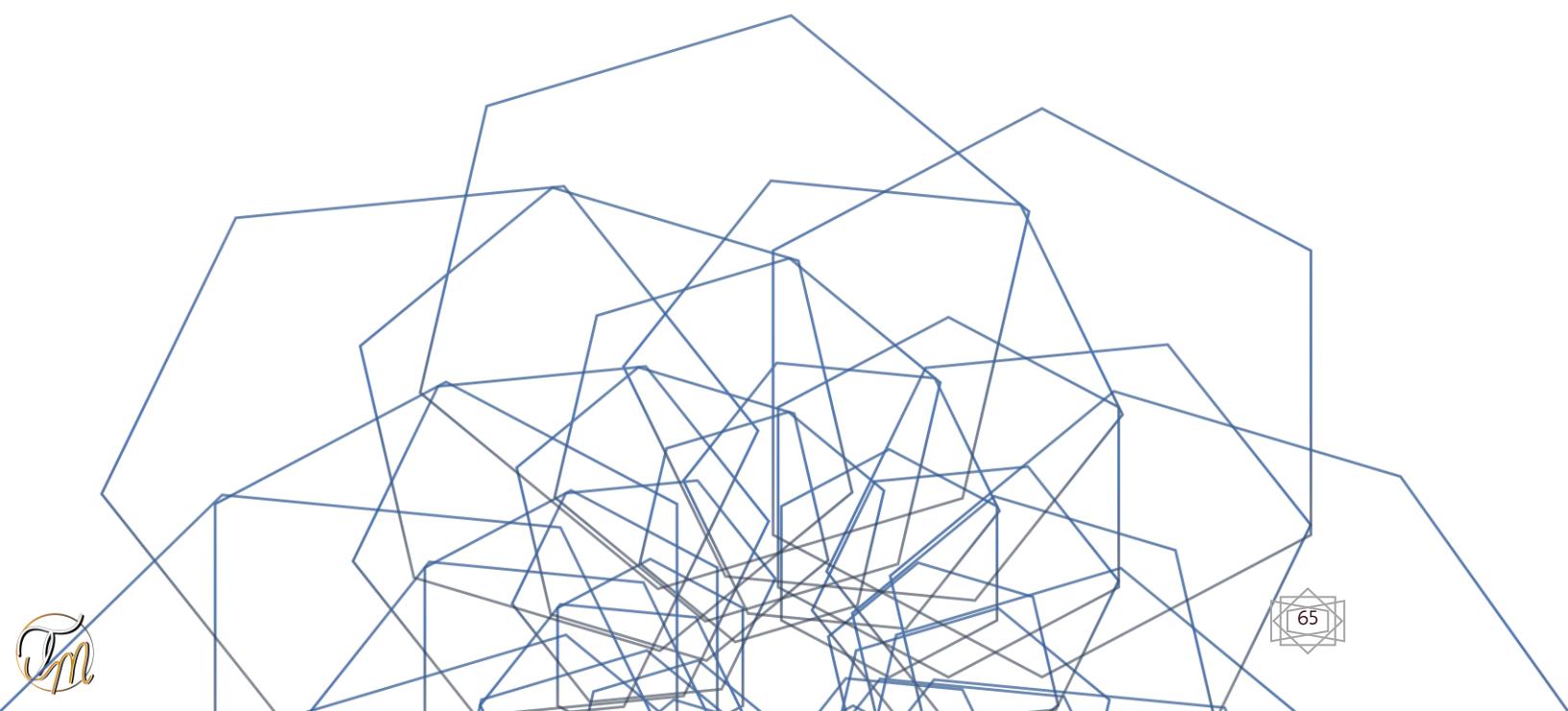
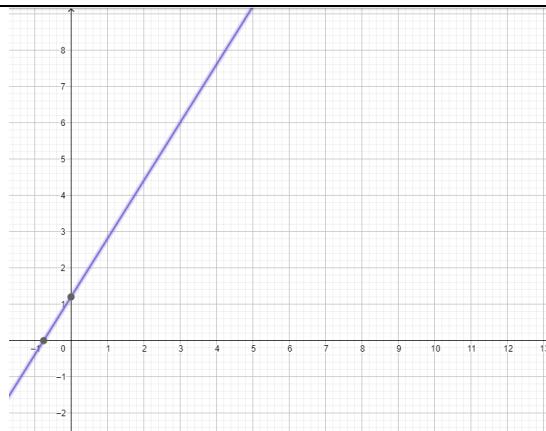
Imagen PNG

GRAFICA	APROXIMACION
$F(x)=4.25531914x-5.65957446808$	

Archivo PDF

GRAFICA	APROXIMACION

$F(x)=1.59999999996x+1.206$



## CUESTIONARIO

### 1. ¿Los niveles de codificación proporcionan una ventaja respecto al tamaño del archivo original en el promedio de los casos?

Si proporcionan una ventaja notoria, pues en la mayoría de las ocasiones los archivos redujeron su tamaño a casi la mitad de lo que ocupaban originalmente, por lo cual podemos observar que realmente hay una ventaja notoria al momento de realizar una compresión de archivos con este tipo de algoritmo, realmente desconocemos si se pudieran llegar a obtener mejores resultados, pues consideramos que la implementación que se le dé al algoritmo influye mucho.

### 2. ¿Los tiempos de codificación o decodificación del archivo son muy grandes?

Depende del tipo de archivo, algunas veces el tiempo no pasa del segundo, pero otras veces sí toman un poco de tiempo para ser procesados.

### 3. ¿Cuál es la proporción temporal que guardan la codificación vs la decodificación?

Proporcionalidad temporal = 20.5009

### 4. ¿Ocurrieron perdidas de información al codificar los archivos?

Debido a la implementación en la parte decodificadora no fue posible poder llegar a un resultado 100% satisfactorio, pues al finalizar la decodificación a partir de un archivo codificado por el programa principal este continua agregando algunas letras extra, esto ocurre debido a la forma en que se están manejando los bits, y hasta el momento no hemos logrado visualizar cual es el caso que no estamos considerando dentro de la programación para poder evitar esta situación, por lo que no hay perdida como tal si no un incremento a la información en cuestión

### 5. ¿El comportamiento experimental de los algoritmos era el esperado?

Sí, ya que, en la mayoría de los resultados de las ejecuciones, en número necesario de bites que se necesitaban, eran menores al número de bites originales.

### 6. ¿Qué características deberá tener un archivo de imagen para codificarse en menor espacio?

El rendimiento de un codificador de imágenes con pérdidas viene determinado por dos factores: la distorsión que introduce en la imagen decodificada y el grado de compresión de la imagen codificada.

Por otro lado, la medida que se utiliza para evaluar el tamaño es el número de bits por píxel ( $bpp$ ), que no es más que el número de bits esperado a la salida del codificador dividido por el número de muestras codificadas. Es usual presentar gráficamente el rendimiento de un codificador mediante una gráfica en la que se representa la evolución del nivel de PSNR para distintos grados de compresión medidos en bits por píxel, o lo que es lo mismo, la relación entre la tasa y la distorsión.

**7. ¿Qué características deberá tener un archivo de texto para tener una codificación en menor espacio?**

Debe ser dividido en líneas y cuyo contenido es exclusivamente texto simple, es decir, sólo hay caracteres alfanuméricos (letras y números), retornos de carro y tabuladores, también pueden ser abiertos e inspeccionados sin necesidad de hacer uso de un software especial diseñado para trabajar con ellos.

**8. Proporcione 3 aplicaciones posibles en problemas de la vida real a la codificación de Huffman.**

- a. Ejemplo 1: Se usa a menudo como un "back-end" para algún otro método de compresión. DEFLATE (algoritmo de PKZIP) y códecs multimedia como JPEG y MP3 tienen un modelo front-end y cuantificación seguido de codificación Huffman.
- b. Ejemplo 2: El código Huffman se usa para convertir códigos de longitud fija en códigos de longitud variable, lo que resulta en una compresión sin pérdidas. Los códigos de longitud variable pueden comprimirse aún más utilizando técnicas JPEG y MPEG para obtener la relación de compresión deseada.
- c. Ejemplo 3: El último de los algoritmos de compresión sin pérdidas más eficientes, Brotli Compression, lanzado por Google, también usa Huffman Coding. Además de eso, Brotli también usa LZ77 y algunos otros algoritmos fundamentales de compresión sin pérdida.

**9. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?**

Si el entorno que se usó en todo momento fue el equipo de cómputo que se especifica dentro del documento, con él se realizaron las pruebas con archivos de distinto tipo, así como con archivos de distinto tamaño y diferente cantidad de texto.

**10. ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?**

Una de las principales recomendaciones que proporcionamos a las personas es que antes de meterse de lleno a la programación investiguen acerca del manejo de apunadores y funciones con argumentos por referencia, así como el uso del *typedef* pues fueron de las principales ventajas que se pudieron implementar a lo largo del algoritmo, pues de otra forma se hubieran que tenido que estar usando apunadores en todo momento y en cierto punto puede llegar a ser confuso.

De igual forma realizar pruebas de forma "real" agarrar lápiz y papel e ir viendo cómo es que funciona el algoritmo, así como una lista de ideas a implementar e irlo haciendo por pasos; de igual forma uno de los aspectos más importantes sería realizar una investigación acerca del algoritmo y realmente comprenderlo.

## CONCLUSIONES

La elaboración de esta práctica realmente presento un reto, pues el manejo de apuntadores y tener en consideración tantas funciones así como las diversas situaciones que cada una de las funciones podía presentar con lo que respecta a la obtención de los caracteres así como la elaboración del árbol de Huffman y las listas que deben contener dichos caracteres presento una situación compleja de abordar, pues existen diversas implementaciones y algunas podrán ser mejor que otras, me encuentro consciente de que esta no es una de las mejores, pero proporciona un resultado bastante satisfactorio de acuerdo a lo solicitado.

Me parece de suma importancia saber cómo es que funciona la memoria dinámica, pues es uno de los principales pilares que deben ser considerados para poder trabajar de una forma un tanto más cómodo y así poder evitar la confusión, aunque esta parte depende del tipo de persona y de la experiencia que tenga con el manejo de apuntadores.

De igual forma me pareció sumamente importante la ventaja que le sacamos a las operaciones binarias, pues nos ahorran mucho trabajo, así como el ahorro de la implementación de un bucle for para siempre estar evaluando sobre los 8 bits cuando solo necesitamos una posición específica, situación que podemos evitar gracias a las funciones que el profesor proporciona y al manejo de los enmascaramientos y operaciones binarias.

-Mora Ayala José Antonio

Dentro de esta práctica lo que mejor resultó un aporte fue en la parte del algoritmo de Huffman, principalmente en la parte de la creación de un árbol binario en el que se etiquetan los nodos hoja con los caracteres, junto a sus frecuencias, y de forma consecutiva se van uniendo cada pareja de nodos que menos frecuencia sumen, pasando a crear un nuevo nodo intermedio etiquetado con dicha suma. Se procede a realizar esta acción hasta que no quedan nodos hoja por unir a ningún nodo superior, y se ha formado el árbol binario. De este modo destacamos la parte de la memoria dinámica, pues puede ir incrementando durante la ejecución del programa.

-Lemus Ruiz Mariana Elizabeth

Personalmente, esta práctica fue uno de los más complicados que he realizados a lo largo de esta materia. Teóricamente, entendí muy bien cómo funciona el algoritmo de Huffman y el objetivo que se espera tener a la hora de ser implementado, aunque siendo sincera, al principio, me costó trabajo entenderlo, pero gracias a la explicación que se realiza en la práctica, poco a poco entendí el resultado que se obtiene con este algoritmo.

Igual observé que a la hora de probar con los diversos archivos que se utilizaron para ejecutar el código, tienen diferentes tiempos de ejecución, y aunque suene muy obvio, siento que es



importante mencionarlo, por ejemplo, no porque sea el mismo archivo pdf, txt, png, etc etc., deben de tener la misma, si no que debemos de observar el tamaño de bytes que dichos archivos usan, y de esta manera, entenderemos los resultados que obtenemos.

-Jeong Jeong Paola

La codificación de Huffman me parece una genialidad, pues, aunque en un principio parezca compleja, si nos damos el tiempo de analizarla veremos que es relativamente sencilla e incluso un tanto obvia, sinceramente pienso que los arboles binarios son mi estructura de datos favorita pues es muy sencilla y tiene múltiples usos como pudimos comprobar en esta práctica. Al analizar las funciones en este reporte pude notar que posee en general una complejidad de orden lineal, una cosa que realmente considero positiva, pues se trata de tanto leer como codificar el archivo, cosa que si intentara yo por mi cuenta, seguramente no conseguiría algo mejor que cuadrático.

López López Oscar Manuel

## CÓDIGO

Para llevar a cabo la ejecución del programa decodificador es necesario mandar como parámetro el archivo .dat que desea ser codificado por lo que se solicita que se ejecute de esta forma específico

```
gcc main.c -o main  
\main texto.dat
```

Lo cual proporcionara como salida 2 tipos de archivos, uno con la lista de frecuencias y el segundo con la codificación

```
● ● ● undefined - Funciones.h  
1 #ifndef Funciones  
2 #define Funciones  
3 #include <stdio.h>  
4  
5 /**  
6 ****  
7 * @file Funciones.h  
8 * @author Mora Ayala Jose Antonio  
9 * @version 2.0  
10 * @date October 30 2021  
11 * @brief Archivo de cabeceras de las funciones que competen al programa principal  
12 ****  
13 */  
14  
15 /* Definiciones que nos sirven para pdoer realizar el manejo de bits conforme sea requerido  
16 Mediante los "bitwise Operators" o bien Operadores binarios.  
17 Recurso proporcionado por:  
18 @Autor : Edgardo Adrian Franco Martinez  
19 */  
20  
21 #define PESOBIT(bpos) 1 << bpos  
22 #define CONSULTARBIT(var, bpos) (*(unsigned *)&var & PESOBIT(bpos)) ? 1 : 0  
23 #define PONE_1(var, bpos) *(unsigned *)&var |= PESOBIT(bpos)  
24 #define PONE_0(var, bpos) *(unsigned *)&var &= ~PESOBIT(bpos)  
25 #define CAMBIA(var, bpos) *(unsigned *)&var ^= PESOBIT(bpos)  
26
```



```
● ● ●
undefined - Funciones.h

1 typedef struct Lista
2 {
3     Nodoinfo inf;
4     struct Lista *sig, *izq, *der;
5 } Nodolista;
6
7 typedef Nodolista *Listaenlazada;
8 //Sin el typedef
9 // Nodolista *lista
10
11 Nodolista **arbol;
12 int tam;
13 int totalDeBits;
14
15 // *arbol -> Estructura lista SegundoApuntador **-> (Parte dinamica para la memoria)
16 //ListaEnlazada **arbol = ***arbol
17
18
19 //Arreglo para ir almacenando cada uno de los binarios
20 // int auxBin[8];
21 //Variable con el tamanno total de la lista enlazada
22 int tamanoLista;
23
24 //Variable con el tamanno total de bytes del archivo de entrada
25 int totalDeBytes;
26
27 /*
28 /////////////////////////////////Prototipo de funcion fillList///////////////////
29 Prototipo de la función
30 ->Return Value : Void
31 ->Opera con: Necesita de un lista de tipo Listaenlazada pasada por referencia y un caracter
32 /////////////////////////////////Prototipo de funcion fillList///////////////////
33 */
34 void fillList(Listaenlazada *lista, char c);
35 /*
36 /////////////////////////////////Prototipo de funcion printList/////////////////
37 Prototipo de la funcion :
38 ->Return Value : Void
39 ->Opera con: Lista enlazada
40 /////////////////////////////////Prototipo de funcion printList/////////////////
41 */
42 void printList(Listaenlazada lista);
43 /*
44 /////////////////////////////////Prototipo de funcion orderList/////////////////
45 Prototipo de la funcion :
46 ->Return Value : Void
47 ->Opera con: Lista enlazada pasa por referencia con el objetivo de poder ordenar los valores que encuentre dentro de esa lista
48 y poder seguir siendo usada de esa forma en operaciones posteriores
49 /////////////////////////////////Prototipo de funcion orderList/////////////////
50 */
51 void orderList(Listaenlazada *lista);
52 /*
53 /////////////////////////////////Prototipo de funcion huffmanTree/////////////////
54 Prototipo de la funcion : huffmanTree
55 ->Return Value : Void
56 ->Opera con: nada
57 ->Dado que durante el programa nuestro arbol de huffman fue creado de una manera global no es necesario que reciba
58 uno de ese tipo y dado que hemos estado trabajando con memoria dinamica no es necesario que mandemos como
59 parametro alguna lista
60 /////////////////////////////////Prototipo de funcion huffmanTree/////////////////
61 */
62
```



```
undefined - Funciones.h

1 void huffmanTree();
2 /*
3 //////////////////////////////////////////////////Prototipo de funcion copyList////////////////////////////////////////////////
4 Prototipo de la función copyList
5 ->Return Value : Void
6 ->Opera con: Lista enlazada
7 ->Dada una lista que no este vacia la función se encargara de proceder a copiar los valores que encuentre dentro de esa lista
8 en el arbol de Huffman al cual podemos acceder dado que fue declarado de forma global
9 //////////////////////////////////////////////////Prototipo de funcion copyList////////////////////////////////////////////////
10 */
11 void copyList(Listaenlazada Lista);
12 /*
13 //////////////////////////////////////////////////Prototipo de funcion frecuenciasTXT////////////////////////////////////////////////
14 Prototipo de la función frecuenciasTXT
15 ->Return Value : Int (en caso de error debemos retornar un valor)
16 ->Opera con: Lista enlazada
17 -> Dado que solo nos interesa conocer los valores dentro de la lista y no modificarlos solo se pasa como parametro
18 //////////////////////////////////////////////////Prototipo de funcion frecuenciasTxt////////////////////////////////////////////////
19 */
20 int crearFrecuenciasTxt(Listaenlazada lista);
21 /*
22 //////////////////////////////////////////////////Prototipo de funcion binaryRoute////////////////////////////////////////////////
23 Prototipo de la función Ruta Binaria
24 ->Return Value : Void
25 ->Opera con: Listaenlazada *arbollImpresion, int pos, int bit, int izq, int der, int *auxBin
26 -> Realiza el recorrido en postorden de nuestro arbol de Huffman para poder ir obteniendo las codificaciones correspondientes asi
27 como el caracter en cuestion y la frecuencia
28 //////////////////////////////////////////////////Prototipo de funcion binaryRoute////////////////////////////////////////////////
29 */
30 void binaryRoute(Listaenlazada *arbollImpresion, int pos, int bit, int izq, int der, int *auxBin);
31 /*
32 //////////////////////////////////////////////////Prototipo de funcion createBinaryList////////////////////////////////////////////////
33 Prototipo de la función crear Lista Binaria
34 ->Return Value : Void
35 ->Opera con: Lista enlazada y una listaenlazada pasada por referencia la cual almacenara el recorrido binario
36 //////////////////////////////////////////////////Prototipo de funcion createBinaryList////////////////////////////////////////////////
37 */
38 void createBinaryList(Listaenlazada arbollImpresion, Listaenlazada *listaBinaria);
39 /*
40 //////////////////////////////////////////////////Prototipo de funcion crearCodificacionDAT////////////////////////////////////////////////
41 Prototipo de la función
42 ->Return Value : int
43 ->Opera con: Lista binaria, archivo
44 //////////////////////////////////////////////////Prototipo de funcion crearCodificacionDAT////////////////////////////////////////////////
45 */
46
```

```
undefined - Funciones.h

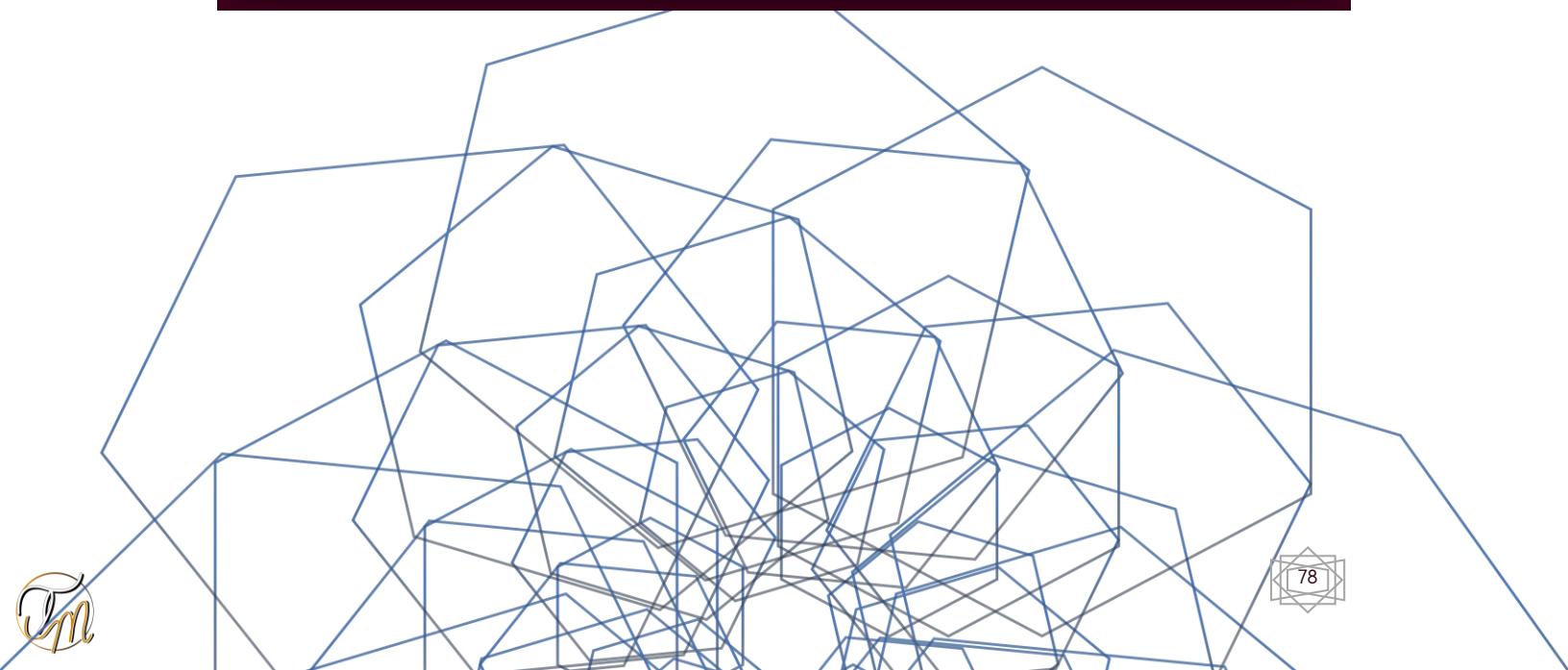
1 int crearCodificacionDAT(Listaenlazada listaBinaria, FILE *archivo);
2 /*
3 //////////////////////////////////////////////////////////////////Prototipo de funcion cmpchar////////////////////////////////////////////////////////////////
4 Prototiplo de la función
5 ->Return Value : int
6 ->Opera con: 2 cadenas dadas las cuales quieran ser sometidas a una comparacion en cunato a valores binarios
7 //////////////////////////////////////////////////////////////////Prototipo de funcion cmpchar////////////////////////////////////////////////////////////////
8 */
9 int cmpchar(char cadena1, char cadena2);
10
11 /*
12 //////////////////////////////////////////////////////////////////Prototipo de funcion imprimirTiempos////////////////////////////////////////////////////////////////
13 Prototiplo de la función
14 ->Return Value : Void
15 ->Opera con: Lista enlazada
16 //////////////////////////////////////////////////////////////////Prototipo de funcion imprimirTiempos////////////////////////////////////////////////////////////////
17 */
18 void imprimirTiempos(double utime0, double stime0, double wtime0, double utime1, double stime1, double wtime1);
19
20
21 #endif //Funciones.h
22
```



# FUNCIONES.C

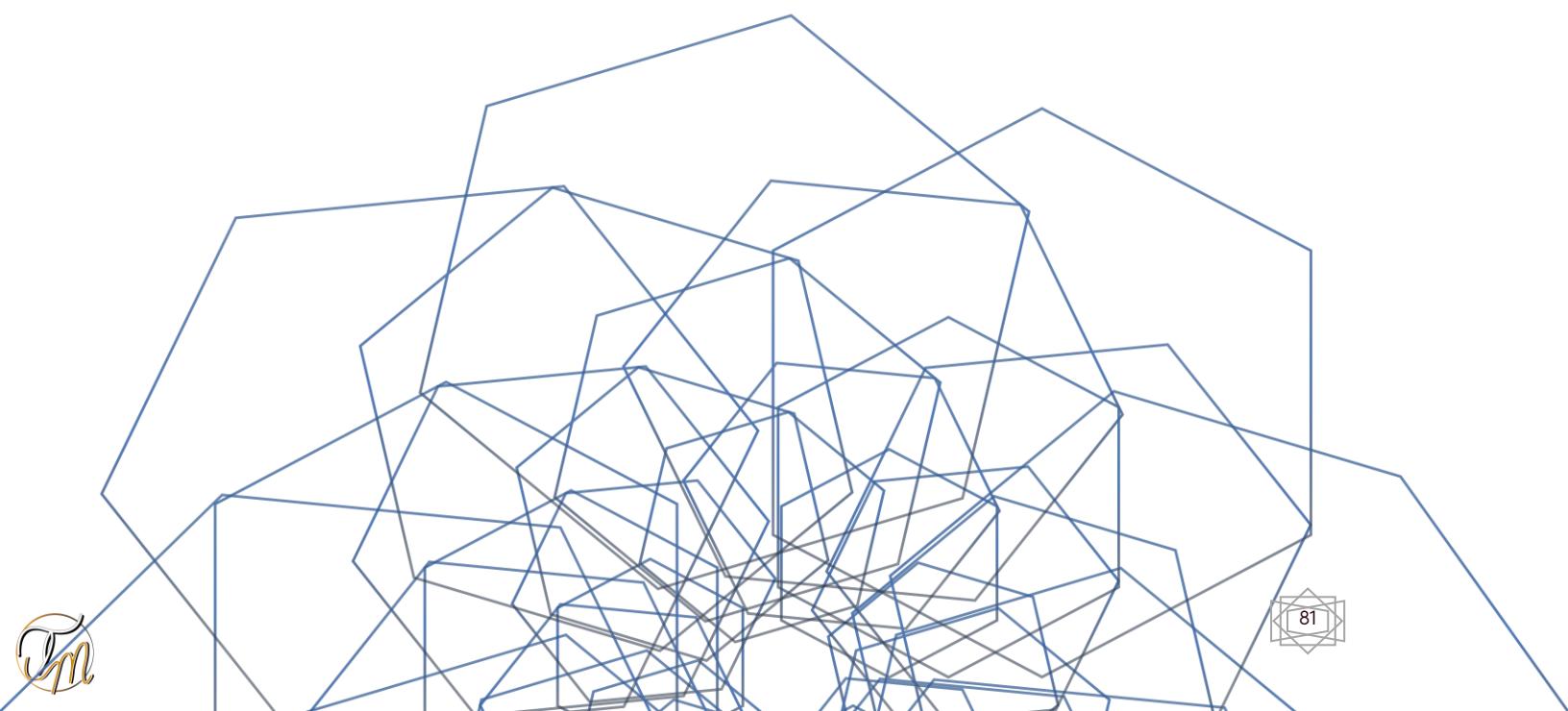
















## MAIN.C

```
undefined - idea.c

1  /**
2   ****
3  * @file Main.c
4  * @author Mora Ayala Jose Antonio
5  * @version 2.0
6  * @date October 29 2021
7  * @brief Archivo que contiene la sección principal del programa de Algoritmo de Huffman
8  ****
9 */
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <stdbool.h>
14 #include "./Funciones.c"
15 #include "./Tiempo/tiempo.h"
16
17
18 int main(int argc, char const *argv[])
19 {
20     double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
21     FILE *archivo; //Archivo con el cual trabajaremos el algoritmo
22     Listaenlazada lista = NULL; //Lista enlazada para ir almacenando cada carácter (también nos sirve como la estructura del árbol)
23     Listaenlazada BinaryList = NULL; //Lista para ir almacenando los binarios
24     int auxBin[8]; //Arreglo para ir almacenando los binarios de cada carácter
25     char listaCaracteres[1024]; //Arreglo para ir almacenando la cantidad de caracteres que nos encontramos en el archivo
26     int readBytes; //Cantidad de bytes leídos
27     char c; //Carácter para poder ir realizando la obtención y comparación de cada uno de los caracteres en el archivo
28
29     if (argc != 2){
30         printf("Ingresa el nombre del archivo\n", argv[0]);
31         exit(0);
32     }
33     else{
34         archivo = fopen(argv[1], "rb");
35     }
36
37     do
38     {
39         //Guardamos en bytes leídos el tamaño de bytes escaneados
40         readBytes = fread(listaCaracteres, sizeof(char), 1024, archivo);
41
42         //Guardamos el total de bytes leído cada vez para obtener el total
43         totalDeBytes = totalDeBytes + readBytes;
44         //Haremos esto hasta que lleguemos al final del archivo
45     } while (!feof(archivo));
46
47     printf("El total de bytes que se encontraron en el archivo son %d\n", totalDeBytes);
48
49     // es necesario hacer uso de la función rewind incluida en el manejo de archivos, dado que recorrimos todo el archivo y debemos volver
50     // al principio para poder comenzar nuevamente con nuestro procedimiento
51     rewind(archivo);
52
53     // * lista -> Nodolista;
54
55     /*
56     Realizando la obtención de cada uno de los caracteres que nos encontramos dentro del archivo,
57     situación que sucede hasta que hayamos llegado al final del archivo
58     IMPORTANTE: se debe considerar que la función fgetc al final agrega un salto de línea por lo cual
59     es sumamente necesaria la consideración de poner que el carácter debe ser distinto de menos 1, ya
60     que de no ser así modificaría el resultado de nuestro árbol de Huffman y por lo tanto las ocurrencias
61     */
62     while (!feof(archivo))
63     {
64         c = fgetc(archivo);
65         if (c != -1)
66         {
67             fillList(&lista, c);
68         }
69     }
70 }
```

```
undefined - idea.c

1 printf("Nombre del archivo: %s - Tamanno en bytes: %d\n", argv[1], totalDeBytes);
2
3 // uswtime(&utime0, &stime0, &wtime0); //Inicio de contador de ordenamiento de la lista
4
5 orderList(&lista); //Ya que hemos llenando la lista procedemos a ordenarla
6
7 // uswtime(&utime1, &stime1, &wtime1); //Fin del ordenamiento e la lista
8 // imprimirTiempos(utime0, stime0, wtime0,utime1, stime1, wtime1);
9
10 printList(lista); //Realizmamos la impresion de la lista para asegurarnos que todo esta saliendo como deberia
11
12 printf("Copiando Lista\n");
13
14 copyList(lista); //Copiamos la lista dentro del arbol de Huffman el cual fue declarado de forma global
15
16 printf("Lista Copiada\n");
17
18 printList(lista);
19
20 huffmanTree0; //Procedemos a la cracion del arbol de Huffman
21
22 Listaenlazada arbolimp = arbol[0];
23
24 binaryRoute(&arbolimp, -1, 0, 0, 0,auxBin); //Realizacion de un recorrido en postorden con respecto al arbol para poder realizar la obtencion de cada codigo
```



undefined - idea.c

```
1 createBinaryList(arbolimp, &BinaryList); //Guardamos en nuestra lista de valores binarios
2
3 crearCodificacionDAT(BinaryList, archivo); //Realizamos la creacion de la codificacion
4     if (crearFrecuenciasTxt(lista) == -1)
5     {
6         return -1;
7     } //Creamos el archivo frecuencias.txt
8
9 fclose(archivo);
10
11 //Liberamos la memoria de la cabeza de la lista enlazada
12 free(lista);
13
14 //liberamos la memoria de nuestro arbol de huffman de impresion
15 free(arbolimp);
16
17 //Liberamos la memoria para nuestra lista enlazada con nuestros
18 //datos en binario
19 free(BinaryList);
20
21 //Imprimios este mensaje para saber que el programa se haya
22 //ejecutado u terminado sin ningun problema
23 printf("Fin\n");
24 return 0;
25 }
```

## ARCHIVO DECODIFICACION

### FUNCIONESD.H



#### undefined - FuncionesD.h

```
1  #ifndef FuncionesD
2  #define FuncionesD
3
4  #define PESOBIT(bpos) 1 << bpos
5  #define CONSULTARBIT(var, bpos) (*(unsigned *)&var & PESOBIT(bpos)) ? 1 : 0
6  #define PONE_1(var, bpos) *(unsigned *)&var |= PESOBIT(bpos)
7  #define PONE_0(var, bpos) *(unsigned *)&var &= ~(PESOBIT(bpos))
8  #define CAMBIA(var, bpos) *(unsigned *)&var ^= PESOBIT(bpos)
9
10 typedef struct informacion
11 {
12     unsigned char c;
13     int freq;
14     int binario[16];
15 } Nodoinfo;
16
17
18 typedef struct Lista
19 {
20     Nodoinfo inf;
21     struct Lista *sig, *izq, *der;
22 } Nodolista;
23
24 typedef Nodolista *Listaenlazada;
25 //Sin el typedef
26 // Nodolista *lista
27 Nodolista **arbol;
28 int tam;
```



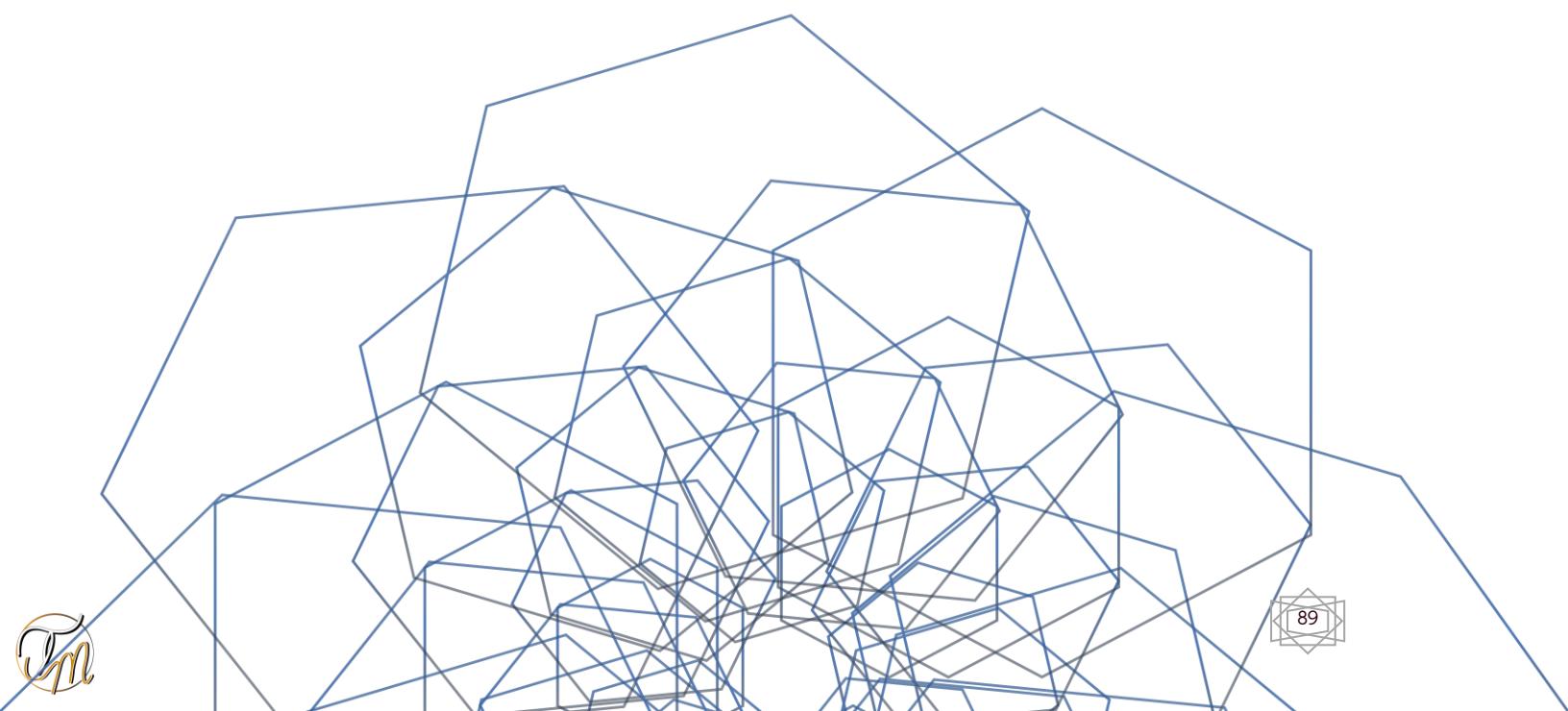
undefined - FuncionesD.h

```
1 char *nombreArchivoFrecuencia;
2
3 //Variable que tendra el nombre del archivo de codificacion
4 char *nombreArchivoCodificacion;
5
6 //Declaramos variables que almacenaran el tipo de archivo y el nombre
7 //que tendra el archivo decode
8 char *cadenaPrueba;
9
10 //Declaramos la variable que tendra el valor escaneado en modo texto
11 char *cadenaValor;
12
13 //Variable que tendra el valor del numero de bits de tipo entero
14 int valorBits;
15
16 //Declaramos variable para el tamano de la lista enlazada
17 int tamanoLista;
```



undefined - FuncionesD.h





# FUNCIONESD.c



undefined - FuncionesD.c

```
1 //Comenzamos a escanear los valores hasta terminar con el archivo
2 do{
3
4     //Si el archivo llega al final rompemos el ciclo
5     if (feof(frecuenciasTXT) == 1) break;
6
7     //En el caso de que la lista no haya sido creada aun con un nodo
8     //la inicializamos con la primera letra del archivo y su frecuencia
9     if (*lista == NULL){
10
11         //asignamos memoria para ese nodo
12         *lista = (Nodolista *)malloc(sizeof(Nodolista));
13
14         //escaneamos la letra del archivo de frecuencias y la almacenamos
15         //en nuestra variable letra de la struct
16         fgets(escaneados,5,frecuenciasTXT);
17         (*lista)->inf.c = escaneados[0];
18
19         //escanemos el valor numerico que corresponde a la frecuencia
20         //como estamos escaneando texto procedemos a convertir la cadena
21         //a entero mediante atoi
22         fgets(escaneados,10,frecuenciasTXT);
23         (*lista)->inf.frec = atoi(escaneados);
24
25         //Declaramos la posicion siguiente a NULL
26         (*lista)->sig = NULL;
27
28         //Aumentamos en 1 el tamano de la lista para usar la variable en futuras
29         //funciones que lo requieran
30         tamanoLista++;
31     }
32     else{
33
34         //En caso de que la lista ya tenga nodos creados lo que procedemos a hacer
35         //es crear un nuevo nodo que contendra los datos de la lista
36         Listaenlazada nuevoNodo;
37         nuevoNodo = (Nodolista *)malloc(sizeof(Nodolista));
38
39         //Escaneamos la letra siguiente del archivo de frecuencias y la almacenamos
40         //dentro del la variable letra
41         fgets(escaneados,5,frecuenciasTXT);
42         nuevoNodo->inf.c = escaneados[0];
43
44         //Posteriormente escaneamos la frecuencia de la letra y de igual manera
45         //mediante atoi convertimos la cadena escaneada en enteros
46         fgets(escaneados,10,frecuenciasTXT);
47         nuevoNodo->inf.frec = atoi(escaneados);
48
49         //actualizamos la cabeza de la lista enlazada
50         nuevoNodo->sig = *lista;
51         *lista = nuevoNodo;
52
53         //Aumentamos en 1 el tamano de la lista para usar la variable en futuras
54         //funciones que lo requieran
55         tamanoLista++;
56     }
57 }while(!feof(frecuenciasTXT));
```



undefined - FuncionesD.c

```
1 //Una vez terminado el ciclo lo que procederemos a hacer es eliminar el primer
2 //elemento de la lista enlazada, debido a que como el archivo detecta a los
3 //saltos de linea como caracteres el ultimo elemento sera un salto de linea y
4 //una frecuencia anterior por ello eliminamos este ultimo nodo agregado
5 *lista = (*lista)->sig;
6
7 //Restamos en 1 el tamano de la lista debido a esta situacion
8 tamanoLista--;
9
10 //Cerramos el archivo de frecuencias que ya no nos sera util a partir de ahora
11 fclose(frecuenciasTXT);
12 }
```



undefined - FuncionesD.c















## undefined - FuncionesD.c

```
1      //Escaneamos el primer caracter dentro de codificado
2      caracterEscaneado = fgetc(codificado);
3
4      //Escanea el End Of File, si se llega a este valor
5      //se rompera el ciclo
6      if (feof(codificado) == 1) {
7          fputc(arbolAuxiliar->inf.c, decode);
8          break;
9      }
10
11     //Iniciamos en el nodo raiz del arbol
12     cambiarByte = false;
13
14     //Escaneamos el primer bit del caracter escaneado
15     do{
16
17         if ((arbolAuxiliar->izq == NULL) && (arbolAuxiliar->der == NULL)){
18
19             fputc(arbolAuxiliar->inf.c, decode);
20             arbolAuxiliar = arbolDeHuffman;
21         }
22         else{
23             valorBinario = CONSULTARBIT(caracterEscaneado, posicionBit);
24             if (valorBinario == 0){
25                 arbolAuxiliar = arbolAuxiliar->izq;
26
27             }else{
28                 arbolAuxiliar = arbolAuxiliar->der;
29             }
30             posicionBit--;
31             if (posicionBit == -1){
32                 posicionBit = 7;
33                 cambiarByte = true;
34             }
35
36         }
37     }while(cambiarByte != true);
38 }while(!feof(codificado));
39 }
```

**DECODIFICADOR.C**

undefined - Decodificador.c

```

1  /*
2  ****
3  * @file Main.c
4  * @author Mora Ayala Jose Antonio
5  * @version 2.0
6  * @date October 29 2020
7  * @brief Archivo que contiene la sección principal del programa de Algoritmo de Huffman
8  ****
9 */
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <stdbool.h>
14 #include "./FuncionesD.c"
15
16
17 int main(int argc, char const *argv[])
18 {
19     //Variable de la lista enlazada que tendra las letras y frecuencias
20     // Variable de una lista con estructura listaenlazada como la que hemos manejado para la parte
21     // de decodificación la cual tendra en si las letras y las frecuencias en el orden
22     // que habíamos manejado
23     Listaenlazada lista = NULL;
24
25     /* Dado que hemos manejado la creaciond e la lista mediante la actualizacion de la cabecera es necesario que
26     la lista se someta a un proceso de inversion, pues basicamente la forma en que accedemos a los elementos sera
27     lo ultimo en haber sido insertado es lo primero a lo cual tendremos acceso */
28     Listaenlazada listalinvertida = NULL;
29
30     /* Para la ejecucion del programa es necesario recibir dos argumentos como maximo los cuales tiene que ver con un archivo codificado
31     asi como la lista de frecuencias que le fue generada para poder proceder a la realización de este programa de decodificacion */
32     if(argc!=3){
33
34         printf("Debes ingresar los nombres de los archivos primero el de Frecuencias.txt y posteriormente el archivo a decodificar %s\n", argv[0]);
35         exit(0);
36     }else{
37         nombreArchivoFrecuencia = argv[1];
38         nombreArchivoCodificacion = argv[2];
39     }
40
41     // Procedemos a llenar la lista
42     if(fillList(&lista) == -1) exit(0);
43
44     invertList(&lista ,&listalinvertida); // Dado que tenemos una lista ya llena pasamos como parametro esa lista y la segunda lista en la cual almacenaremos
45     // los datos en el orden en el cual nos interesa obtenerlos para poder realizar la insercion de los bits de la forma correcta conforme vayamos leyendo
46     // los bits del archivo codificado
47     printf("Lista normal\n");
48     printList(lista);
49     printf("Lista invertida\n");
50     printList(listalinvertida);
51
52     copyList(listalinvertida); //Realizamos la copia de los valores de la lista que tenemos en orden logico para nosotros conforme deseamos obtener
53     // los caracteres asi como su codificacion en el arbol de Huffman
54     huffmanTree(); //Creamos el arbol de Huffman
55     Listaenlazada arbolImp = arbol[0]; //Asignamos el arbol en la posicion 0 a una variable de tipo listaenlazada, ya que esta posee en su estructura tambien la
56     // de la estructura de un arbol Binario
57     if (crearDecodificacion(arbolImp) == -1) exit(0); /* Ya que hemos realizado todo el procedimiento podemos proceder a realizar la creacion del archivo de
58     texto en cuestion */
59     return 0;
60 }
61
62

```

## BIBLIOGRAFÍA

# BIBLIOGRAFÍA

- [1] C. d. l. p. Wikimedia., «Wikipedia,» Wikipedia, 21 Octubre 2004. [En línea]. Available: [https://es.wikipedia.org/wiki/Codificaci%C3%B3n\\_Huffman](https://es.wikipedia.org/wiki/Codificaci%C3%B3n_Huffman). [Último acceso: 06 11 2021].
- [2] CCM Benchmark, «CCM,» Group Figario, 12 09 2009. [En línea]. [Último acceso: 07 11 2021].
- [3] EcuRed, «EcuRed,» -, 21 04 2007. [En línea]. Available: [https://www.ecured.cu/%C3%81rbol\\_binario](https://www.ecured.cu/%C3%81rbol_binario). [Último acceso: 2021 11 06].
- [4] G. Brassard y P. Bratley, Fundamentos de Algoritmia., Madrid: PrenticeHall, 1997.
- [5] UNAM, «Instituto de Matemáticas UNAM,» 21 Febrero 2009. [En línea]. Available: <https://paginas.matem.unam.mx/cprieto/biografias-de-matematicos-a-e/198-euclides>. [Último acceso: 17 09 2021].
- [6] E. A. F. Martinez, «EaFranco.Eakdemy,» [En línea]. Available: <https://eafranco.eakdemy.com/mod/scorm/player.php>. [Último acceso: 07 Octubre 2021].
- [7] R. R. Rodríguez, «UTM.mx,» 4 Febrero 2005. [En línea]. Available: <https://www.utm.mx/~rruiz/cursos/ED/material/ABB.pdf>. [Último acceso: 7 Noviembre 2021].
- [8] M. Ing. Bruno López Takeyas, «ITNuevloLaredo,» 7 Junio 2006. [En línea]. Available: <http://www.itnuevolaredo.edu.mx/takeyas/Apuntes/Estructura%20de%20Datos/Apuntes/07-ABB.pdf>. [Último acceso: 06 Noviembre 2021].
- [9] GeeksforGeeks, «GeeksforGeeks,» GeeksforGeeks, 7 Julio 2021. [En línea]. Available: <https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>. [Último acceso: 6 Noviembre 2021].
- [10] TutorialsPoint, «Data Structure and Algorithms - AVL Trees,» 5 Agosto 2008. [En línea]. Available: [https://www.tutorialspoint.com/data\\_structures\\_algorithms/avl\\_tree\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm). [Último acceso: 5 Noviembre 2021].