INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

Practica: Algoritmo de Huffman

Los últimos:

Torres Trejo Victor Federico
Sotelo Padrón Lara Leilani
Sánchez Flores Guillermo

PROFESOR

Edgardo Adrián Franco Martínez

ASIGNATURA

Análisis y Diseño de Algoritmos

16 de noviembre de 2021

Índice general

Capítulo 1

Planteamiento del problema

1.1. Objetivo

Implementar un algoritmo de codificación voraz ideado por David Huffman, este algoritmo permite encontrar un codigo binario eficiente para un tipo de información en un bajo orden de complejidad

1.2. Definición del problema

- Implementar el algoritmo de codificación de Huffman para codificar archivos de cualquier tipo bajo Lenguaje C.
  - Implementar codificación voraz de Huffman
  - Implementar el algoritmo de codificación
- Medir y comprobar las ventajas de tamaño de los archivos una vez realizadas diferentes codificaciones de archivos.
- Medir los tiempos de ejecución de las implementaciones (codificador y decodificador).
- Analizar y determinar una cota de la complejidad de la codificación y decodificación

Capítulo 2

Actividades y pruebas

2.1. Entorno experimental

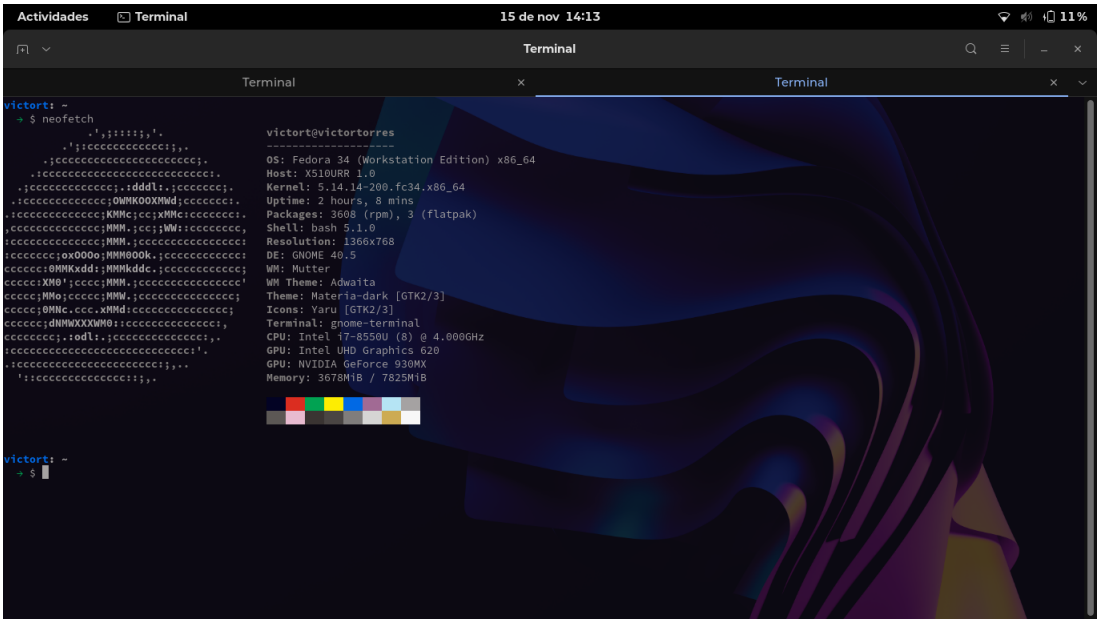

Figura 2.1

2.2. Verificación de la solución

2.2.1. Foto original

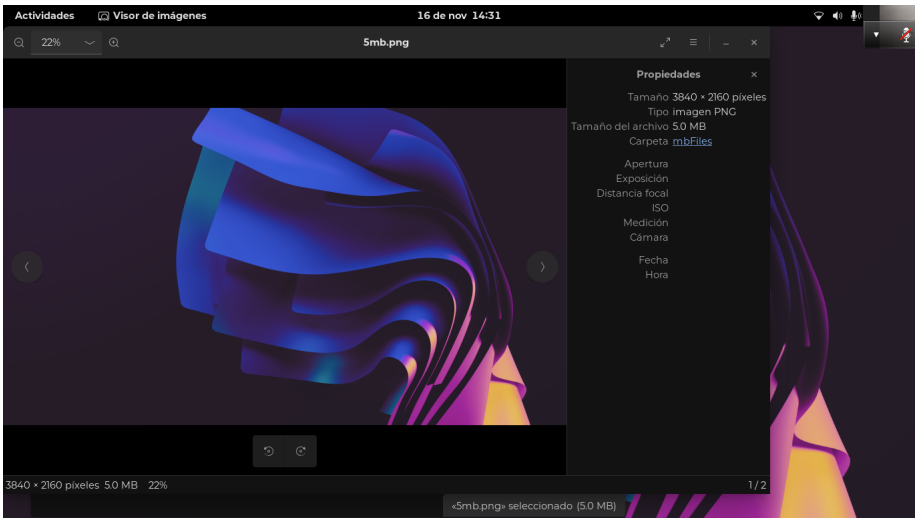

Figura 2.2

2.2.2. Ejecución del algoritmo

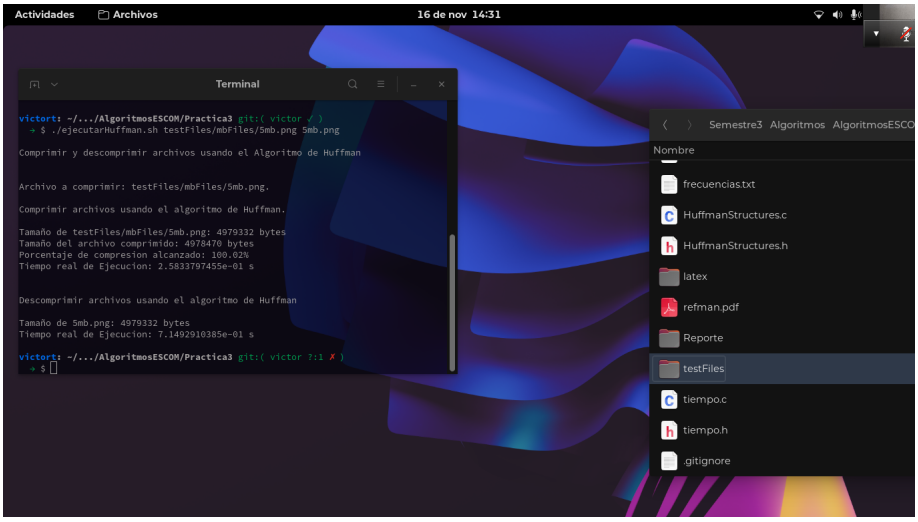

Figura 2.3

2.2.3. Foto final

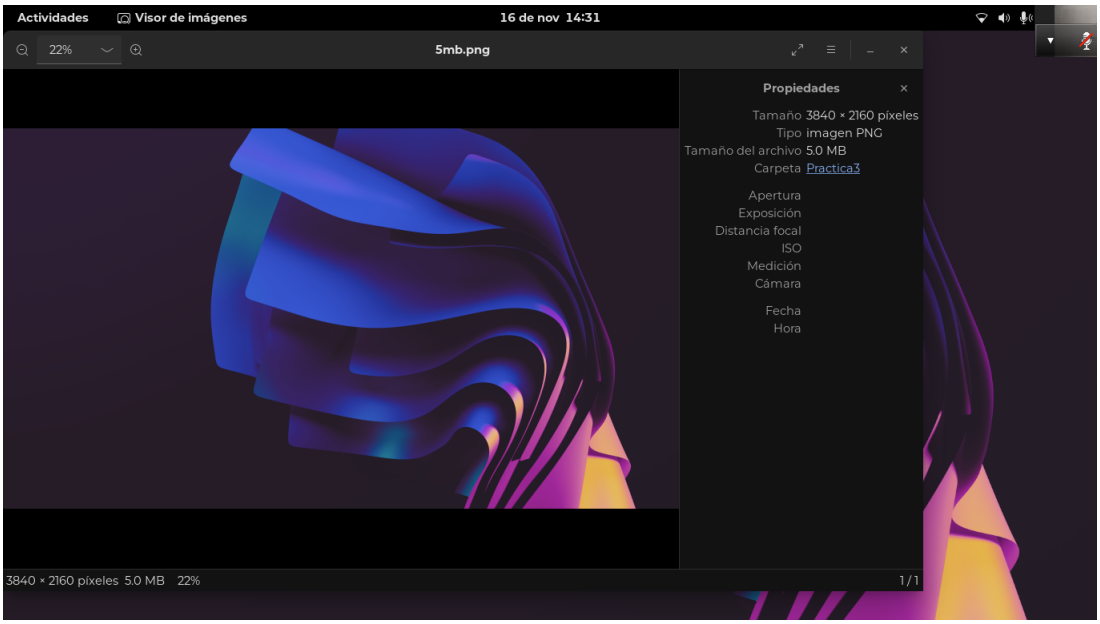

Figura 2.4

2.3. Descripción de la solución

2.3.1. Descripción de la odificación

Para este codigo lo primero que se hizo fue leer los bytes del archivo original, con ayuda de la función fread se logro, para guardar la frecuencia de repetición de cada byte, se le hacia un enmascaramiento del byte a su valor en entero, esto sabíamos que iba a ser de $0 - 255$ bytes, por lo que pudimos aplicar hash, ya que conocemos que el valor hash de cada byte, será único, por lo que podemos hacer uso de esta estructura, así entonces en su valor correspondiente aumentamos la frecuencia en 1 cada vez que detectamos el byte y guardamos que byte es en la posición correspondiente del arreglo. Al final guardamos también una cadena con todos los bytes leídos, esto para después poder hacer la codificación correspondiente.
Ya después insertamos en un arreglo de arboles de Huffman, todos los bytes cuya frecuencia hayan sido mayor a 0, ya que esto significa que aparecieron en el archivo original, así mismo, cada vez que guardamos un byte en un nodo, formamos dicho nodo en una cola de prioridad usando montículos, de esta manera el montículo ordena cada nodo de acuerdo a su frecuencia, una vez guardado esto, unimos todos los arboles de cada byte aparecido en un árbol de Huffman principal, esto se hace sacando el primer elemento de la cola de prioridad, sacando el segundo y re-insertarlos en el montículo pero ahora con los dos nodos unidos, este ciclo se repite hasta que el tamaño del montículo sea 1, indicando que ya tenemos el árbol de Huffman principal.
Una vez que tenemos el árbol, leemos las codificaciones correspondientes de los bits apoyándonos del recorrido in-order, si nos vamos a la izquierda aumentamos un 0, y si nos vamos a la derecha un 1 a una cadena de bits, usando corrimientos hasta que por fin llegamos un nodo sin algún hijo, después de ahí, preguntamos si es un nodo hoja, porque solo los nodos hoja, son aquellos que traen un bit y su frecuencia valido, los demás solo traen la suma de frecuencias de dos nodos hijos, llegado a este punto, aplicamos igual hash con el enmascaramiento del byte a entero, y guardamos la cadena de bits que llevábamos hasta el momento con corrimientos y añadiendo 0 o 1 si nos íbamos a la izquierda o a la derecha, guardamos su byte correspondiente y guardamos el tamaño de los bits correspondiente, esto es porque si bien guardamos la cadena de bits correspondiente, no podemos guardar solo esos bits, y como las guardamos en un entero, añade 0 para formar sus 32 bits, entonces guardando el tamaño de los bits, le decimos, solo queremos los últimos n bits.
Para escribir el codigo binario correspondiente, igual haremos uso de corrimientos, leemos de acuerdo a la cadena de bytes originalmente leída, el byte en el que vamos, y de eso obtenemos los bits que le corresponden de la codificación junto con su tamaño, dependiendo del tamaño, tenemos 3 casos:

- si su tamaño de bits es de 8, solo le hacemos un enmascaramiento a esos 8 bits, para que se eliminen los bits innecesarios y que no nos importan y solo tomemos los 8, y después escribimos en el dat correspondiente.

- si es mayor a 8, entonces guardamos los últimos tamaño de bits $-$ 8 bits que no podemos escribir pero que tenemos que escribir la siguiente vez, le hacemos un corrimiento a bits del mismo tamaño de los bits que guardamos para eliminarlos y el enmascaramiento mencionado en el primer caso de los primeros 8 bits que si podemos escribir. Los bits

no escritos y su tamaño se almacenan en una variable temporal.

- si es menor a 8, significa que no podemos escribir, por lo que guardamos los bits y su tamaño en una variable temporal

Si hay byte por escribir lo hacemos, si no es así, no escribimos nada en el dat.
En la siguiente iteración después de obtener los bits y su tamaño del byte original leído, verificamos si guardamos algo en variables temporales, esto sucede en 2 de los 3 casos descritos. Si hay algo, entonces esos bits, van primero, ya que es lo primero que tenemos que escribir, así que con corrimiento los mandamos al principio y después los bits del byte en el que vamos, restablecemos estas variables auxiliares, indicando que ya las usamos, dejándolas vacías por si se vuelven a ocupar.
Con el fin de evitar desbordamientos en las variables temporales, tenemos un ciclo while, donde vemos si a pesar que ya escribimos una vez, tenemos en la variable temporal 8 o mas bits, si es así, entonces escribiremos los primeros 8, guardamos el exceso y se lo asignamos de nuevo a la variable temporal, siendo ese exceso el valor de la variable temporal. Esto se repetirá hasta que el tamaño de los bits de la variable temporal sea menor a 8.

Al final de escribir todos los bytes en su codificación, preguntamos si hay bits que no escribimos, si es así, los escribimos y guardamos cuantos bits extras escribimos, ya que si teníamos x bits, necesitamos de $8 - x$ bits extra para poder escribirlo en el dat, en el ultimo byte del dat, escribimos precisamente cuantos bits extra no correspondientes a la codificación, escribimos, esto para poder leerlo y descomprimirlo bien.
Como salida extra escribimos en un txt la tabla de frecuencias de aparición de cada byte junto con el tamaño del archivo original.

2.3.2. Descripción de la decodificación

Como primer paso, leemos la tabla de frecuencias, esto con el fin de rearmar el árbol, las frecuencias de bytes, el armado del árbol de Huffman y la obtención de la codificación de cada byte, se hacen de la misma manera descrita en la codificación. Una vez con el árbol, leemos todos los bytes del dat que generemos en la compresión, y ya con todo esto, escribimos de vuelta el archivo original. Primero, del ultimo byte escrito en el dat, lo leemos, ya que ese byte nos indica cuantos bits extra escribimos. Ahora recorremos cada byte leído excepto el ultimo.
Esto se hace que a cada byte leído, recorremos el árbol como nos va indicando, si en el byte en el primer bit hay un 1, se va a la derecha y si lee un 0, a la izquierda, esto lo hace preguntando si el nodo actual es nodo hoja, si no lo es, checa que la posición del bit leído no se haya desbordado, ya que sus limites son de 7 a 0, y consulta que bit hay en esa posición del byte que esta leyendo actualmente, como ya mencionamos hacia donde se va dependiendo si es 1 o 0. Si el nodo es hoja, significa que llegamos a un byte correspondiente, así que retorna el byte a escribir en el archivo original, junto con la posición de bit donde se quedo.
Ya que tenemos que byte debemos escribir, lo hacemos, aumentamos nuestro tamaño de archivo original y verificamos que la posición en bits no se haya desbordado, después de la primera iteración, debemos cuidar que no escribamos mas bytes de los que tenía el archivo original, por ello tenemos ese conteo y ademas cuidamos que no escribamos los bits de mas,

esto es, si estamos en el penúltimo byte leído y ademas, nuestra posición en bits, corresponde al complemento de los bits extra, entonces ya acabamos de escribir.

2.3.3. Descripción de la cola de prioridad

Usamos un montículo o montón para la cola de prioridad, donde sus funciones son las siguientes:

- Insert, insertará nodos a el montón.
- Bottom up lo que hará es recorrer el montón de arriba hacia abajo
- Top bottom es ordenara de abajo hacia arriba y buscará el de mayor prioridad y lo colocará hasta arriba ya usado el de mayor prioridad le hará un pop y volvemos a hablar a top bottom para que busque al de mayor prioridad y lo coloque hasta arriba
- Pop eliminara el primer elemento, el de menor frecuencia de todos los nodos en la cola

2.4. Análisis de complejidad

2.4.1. Análisis de la codificación de Huffman

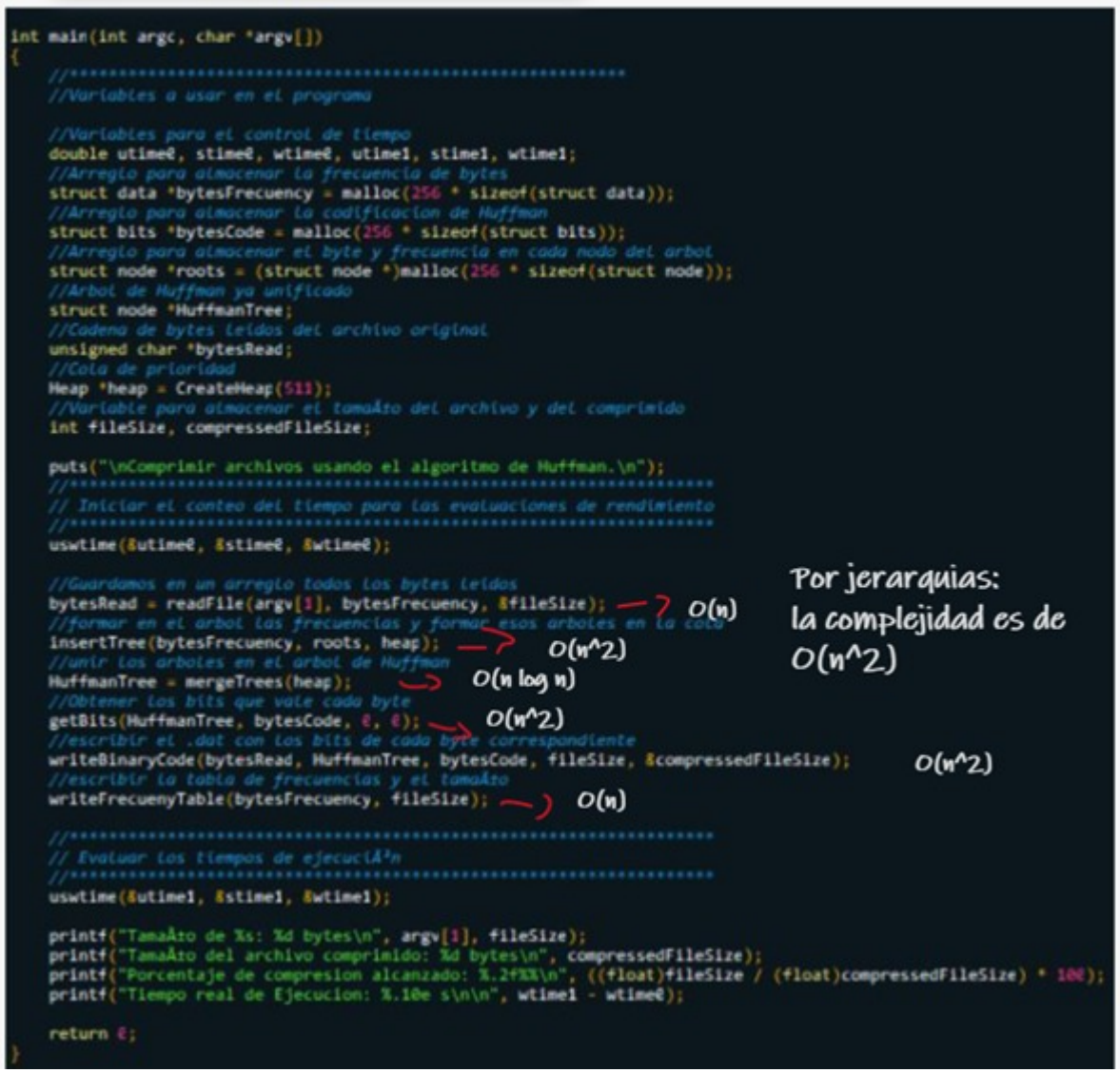

Figura 2.5

Figura 2.6

Figura 2.7

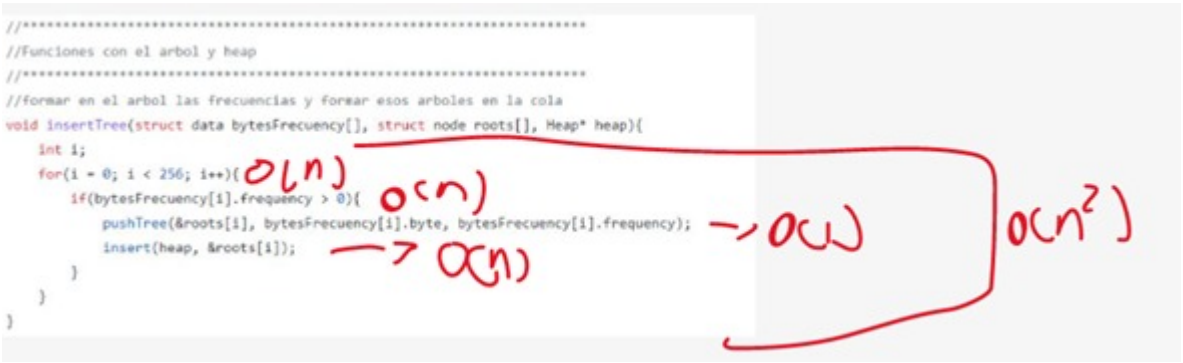

Figura 2.8

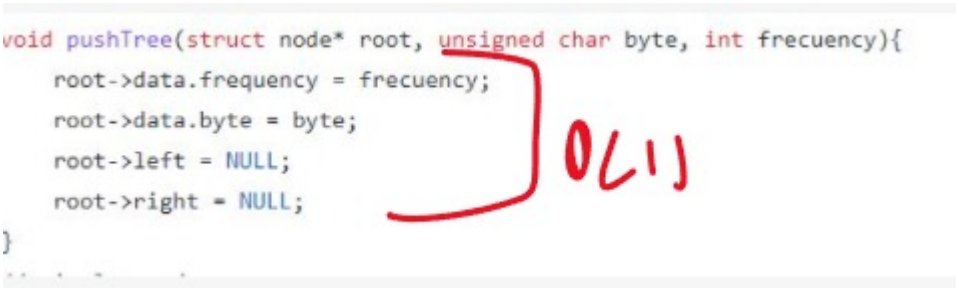

Figura 2.9

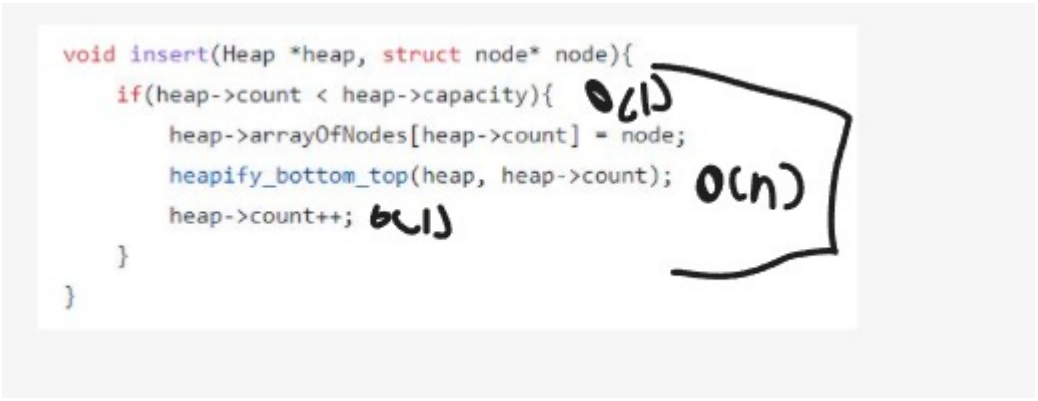

Figura 2.10

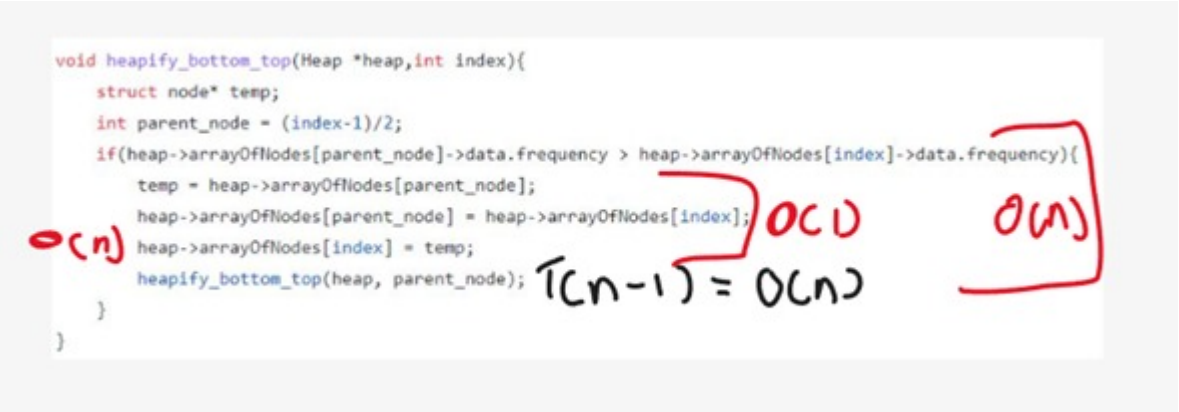

Figura 2.11

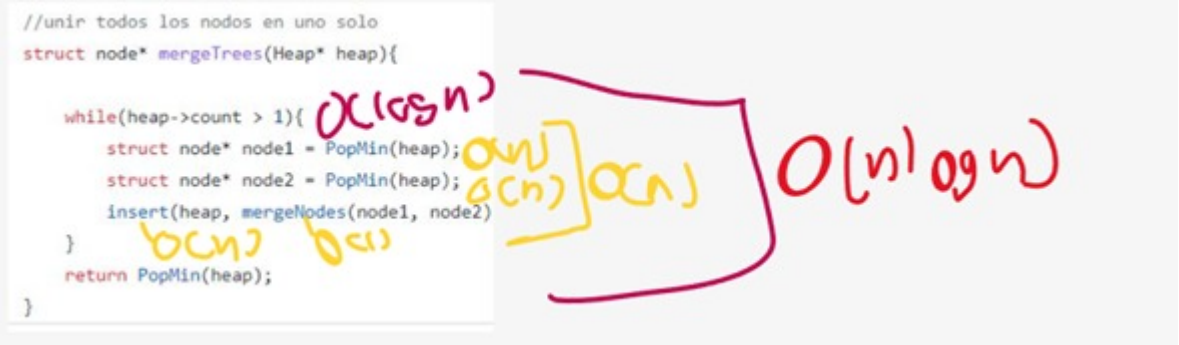

Figura 2.12

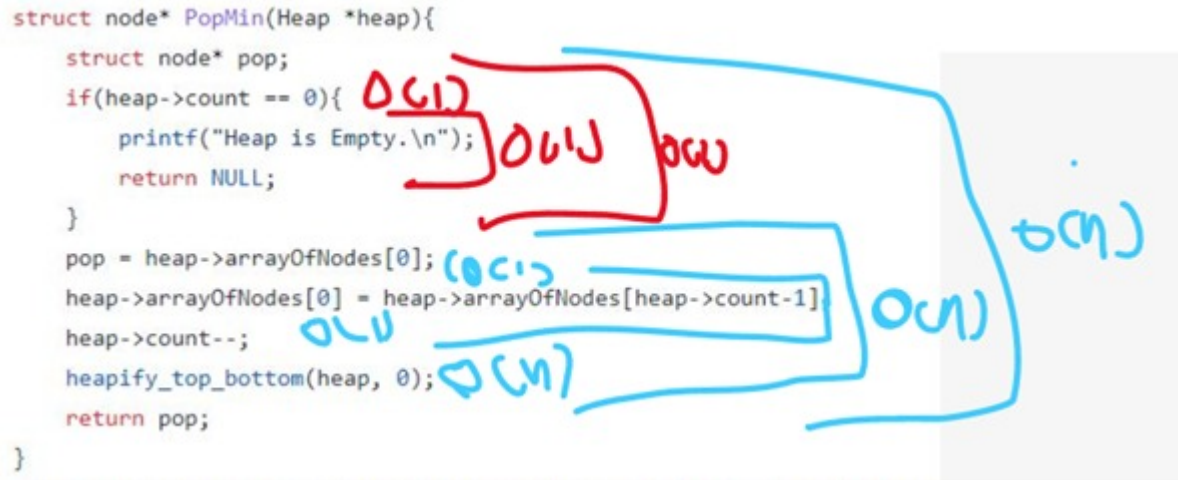

Figura 2.13

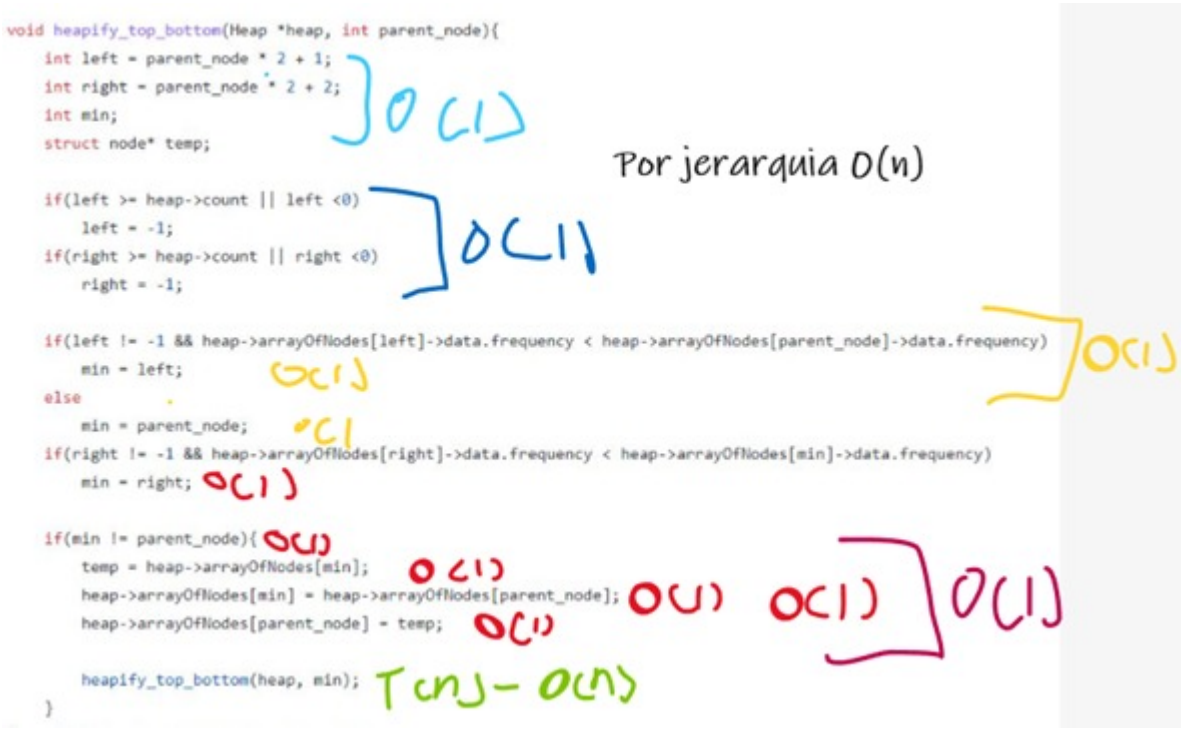

Figura 2.14

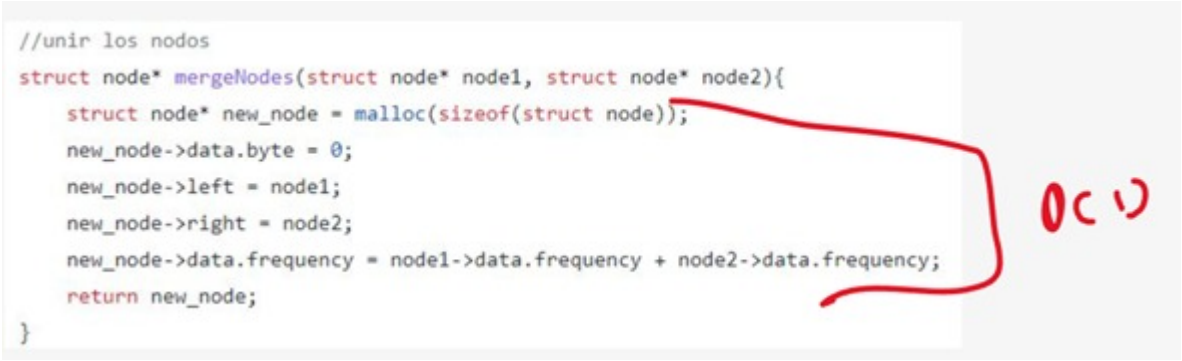

Figura 2.15

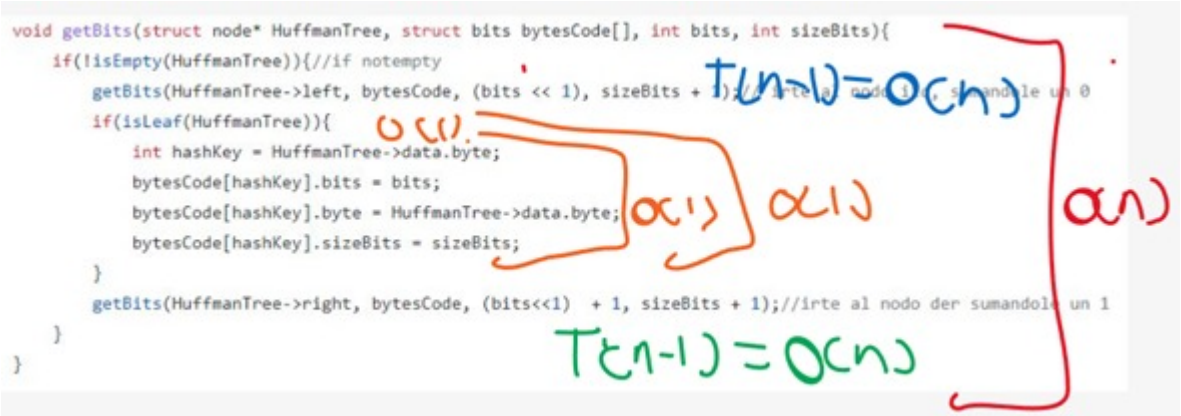

Figura 2.16

Figura 2.17

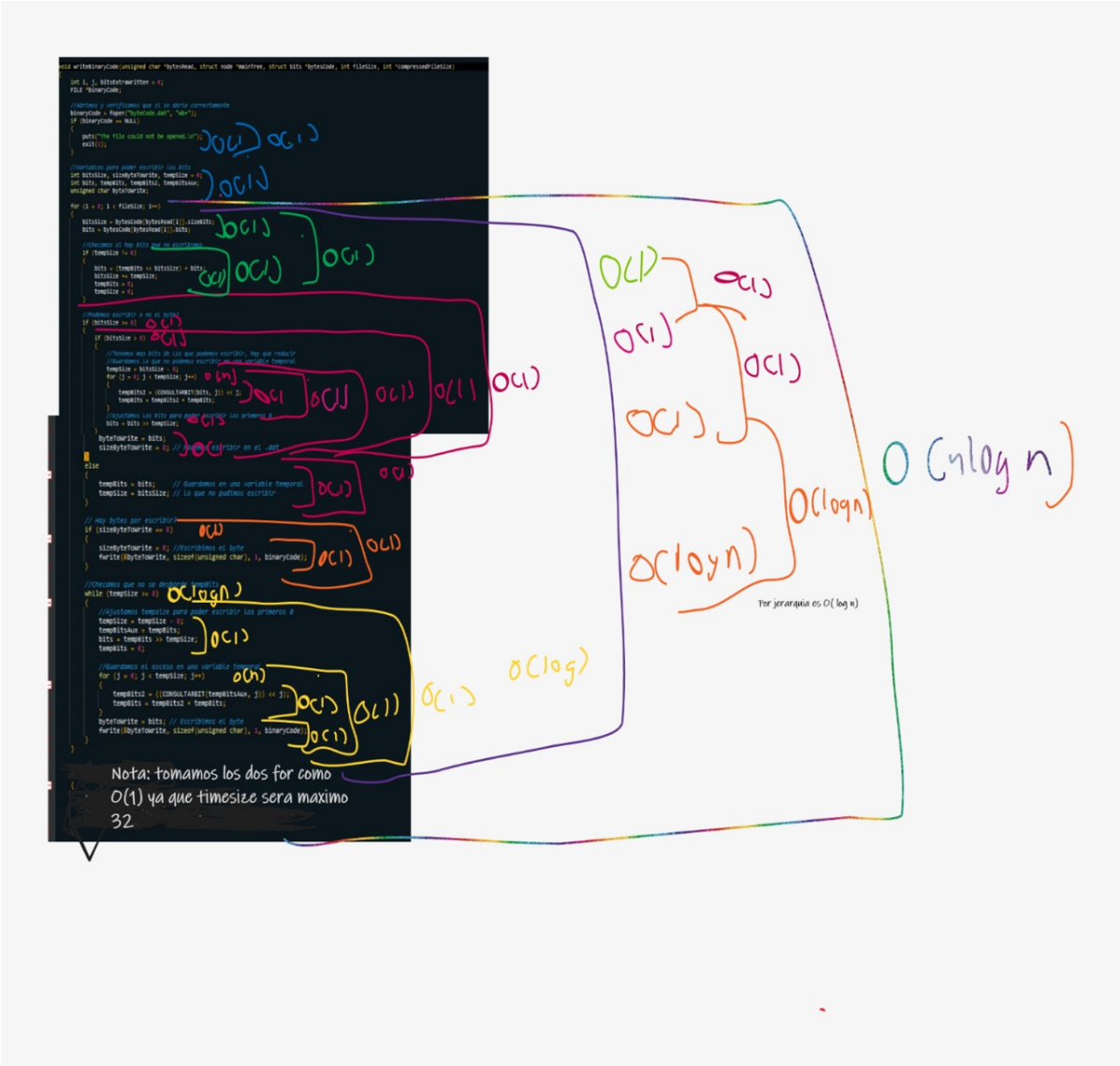
Nota: tomamos los dos for como
O(1) ya que timesize sera maximo
32
O(1)
O(log n)
Por jerarquia es O(log n)
O (nlog n)

Figura 2.18

2.4.2. Análisis de la decodificación de Huffman

Figura 2.19

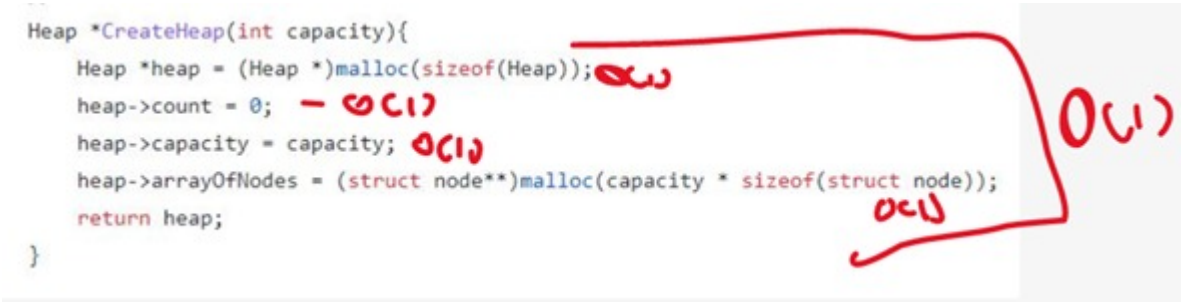

Figura 2.20

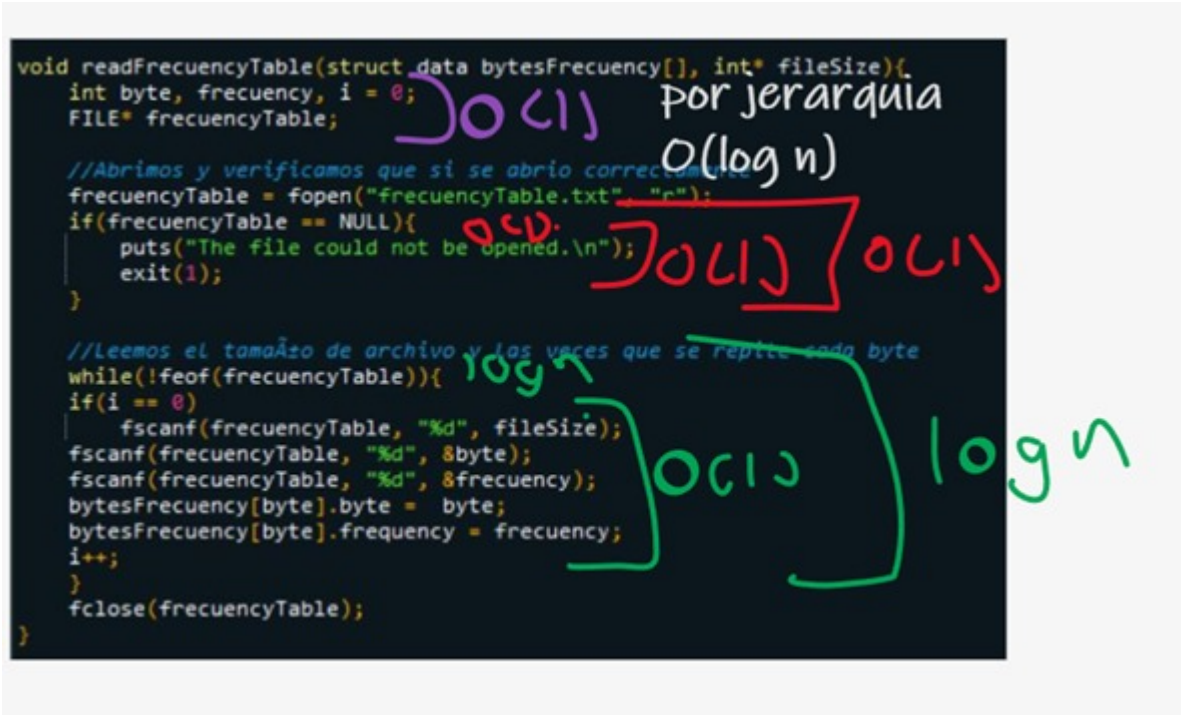

Figura 2.21

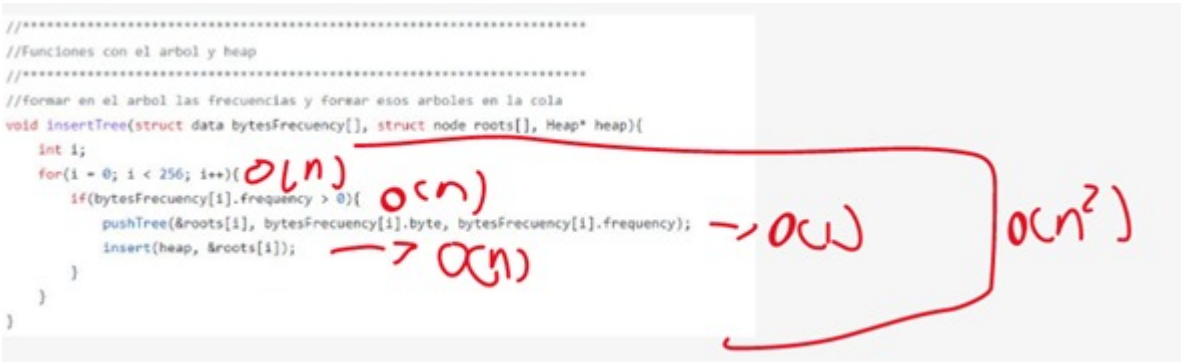

Figura 2.22

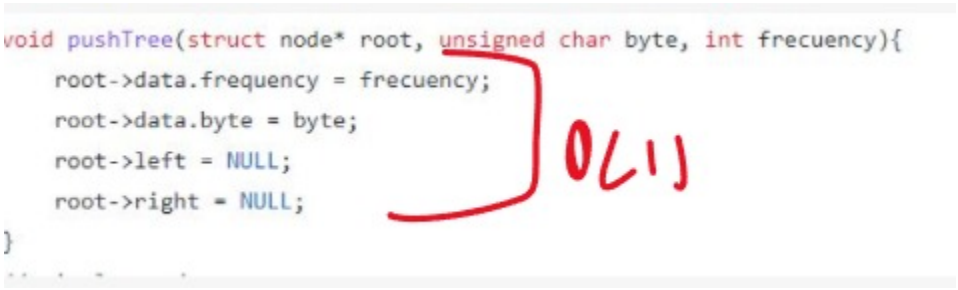

Figura 2.23

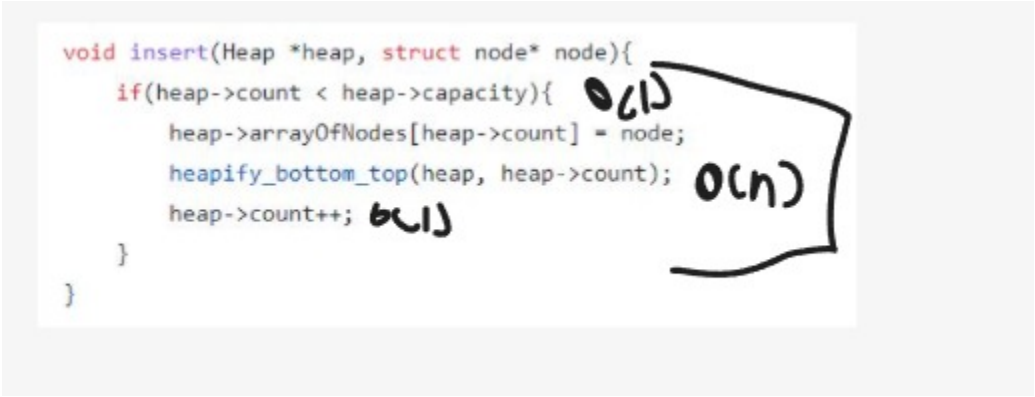

Figura 2.24

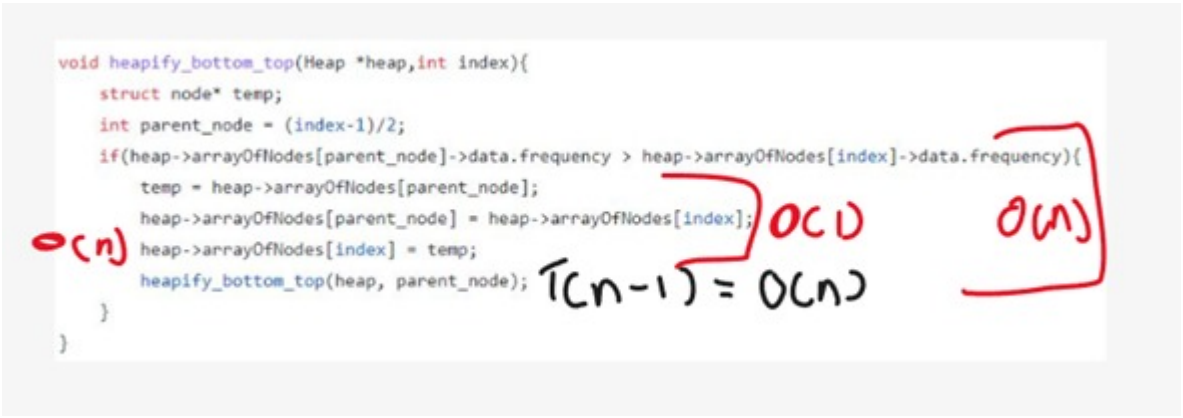

Figura 2.25

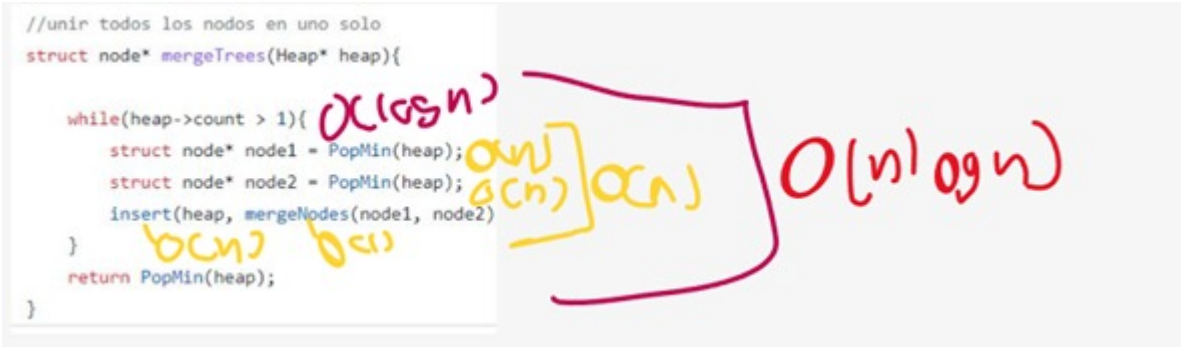

Figura 2.26

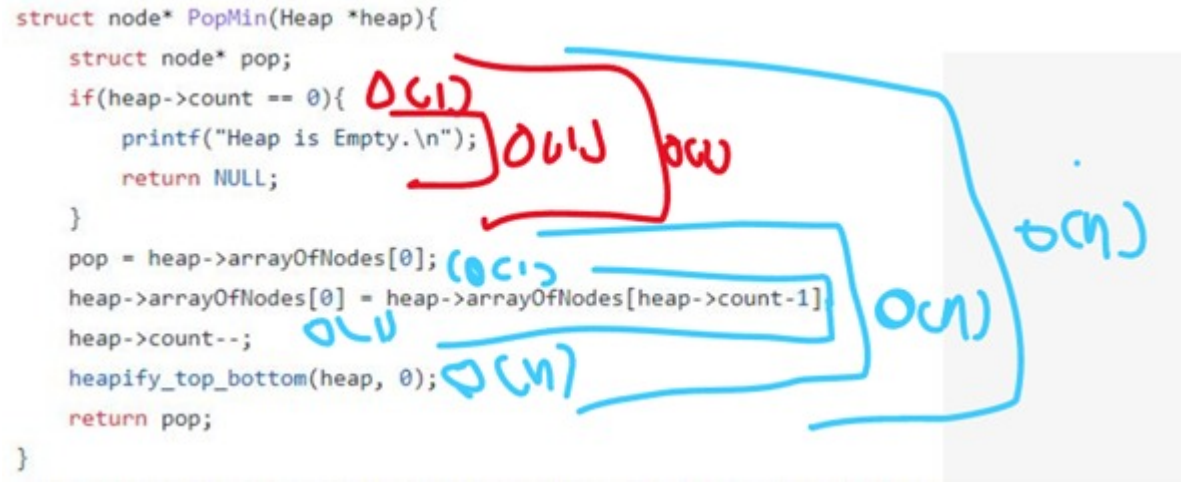

Figura 2.27

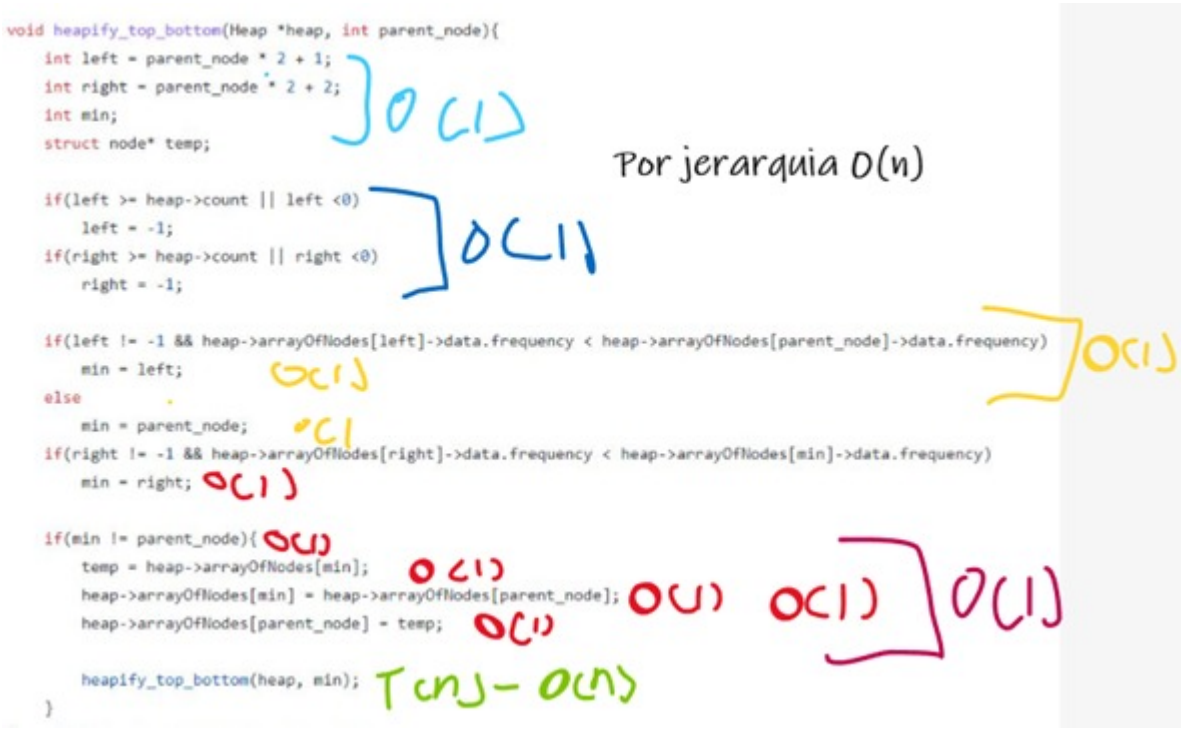

Figura 2.28

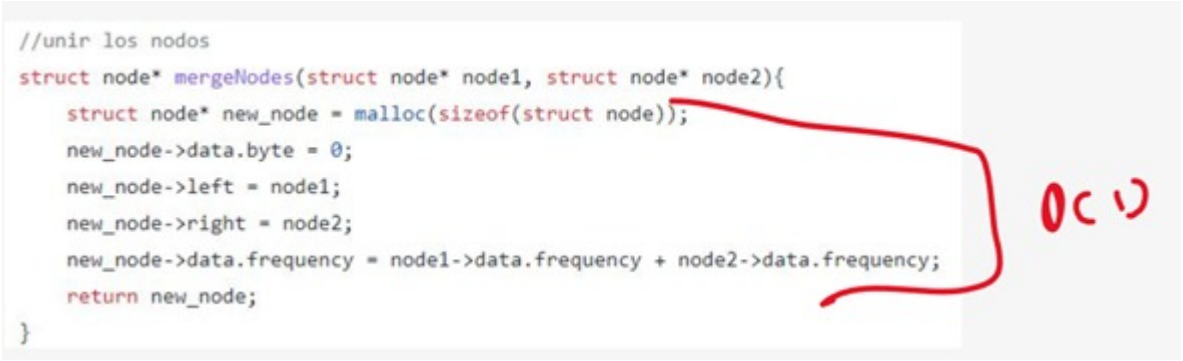

Figura 2.29

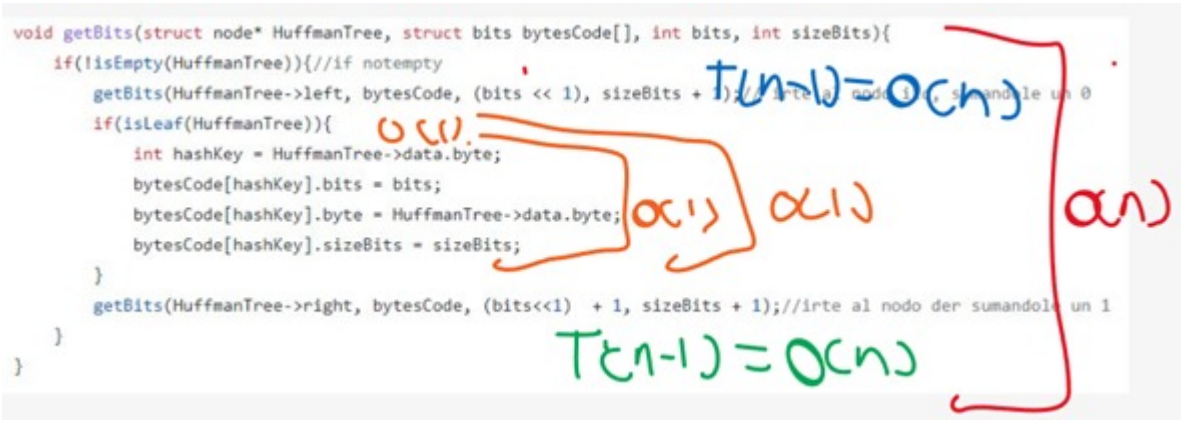

Figura 2.30

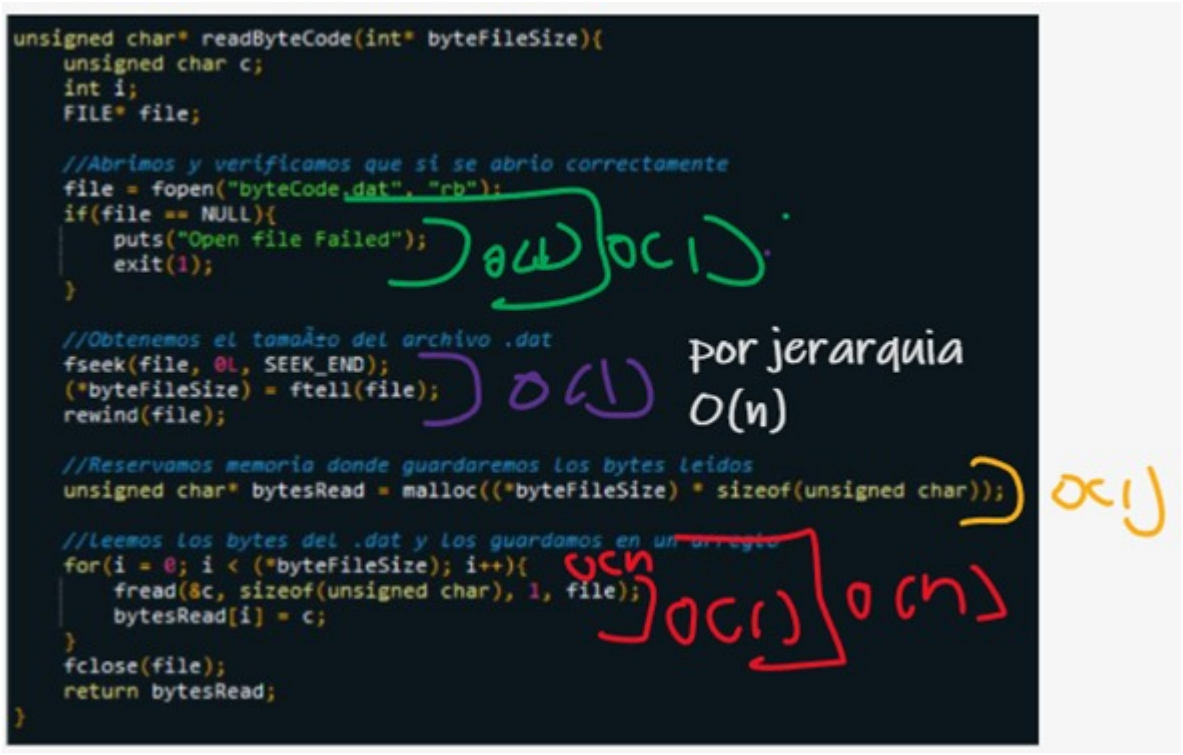

Figura 2.31

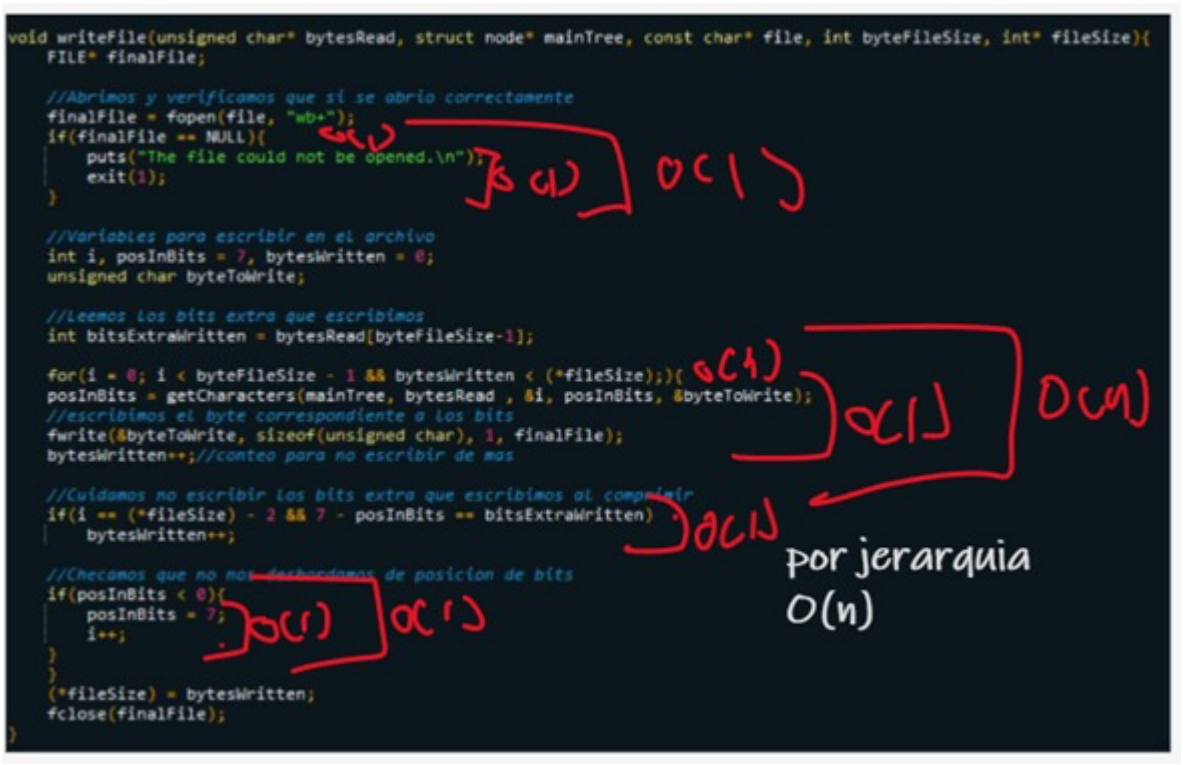

Figura 2.32

Figura 2.33

2.5. Porcentajes de compresion

Tipo de archivo	% de compresión alcanzado
Txt	194.7 %
BMP	678 %
PNG	100 %
PDF	100 %
PPT	110.69 %
ZIP	100 %
RAR	102 %

2.6. Tiempos de ejecución

Tamaño	Tipo	tiempo comprimiendo	tiempo descomprimiendo
5 bytes	txt	$2.3698806763 \times 10^{-4}$	$1.0299682617 \times 10^{-4}$
15 bytes	txt	$1.9407272339 \times 10^{-4}$	$8.7022781372 \times 10^{-5}$
50 bytes	txt	$1.6713142395 \times 10^{-4}$	$1.0800361633 \times 10^{-4}$
700 bytes	rar	$2.3603439331 \times 10^{-4}$	$2.7418136597 \times 10^{-4}$
900 bytes	rar	$2.9397010803 \times 10^{-4}$	$3.4594535828 \times 10^{-4}$
1 kb	png	$2.6917457581 \times 10^{-4}$	$2.5701522827 \times 10^{-4}$
35.9 kb	txt	$2.1967887878 \times 10^{-3}$	$3.6909580231 \times 10^{-3}$
103 kb	pdf	$5.7950019836 \times 10^{-3}$	$1.5046119690 \times 10^{-2}$
115.5 kb	zip	$5.8871030807 \times 10^{-2}$	$4.9322843552 \times 10^{-2}$
142.8 kb	pdf	$8.2550048828 \times 10^{-3}$	$2.3307085037 \times 10^{-2}$
248.3 kb	ppt	$1.2196063995 \times 10^{-2}$	$2.9481887817 \times 10^{-2}$
512.6 kb	png	$2.1743059158 \times 10^{-2}$	$6.8802833557 \times 10^{-2}$
1 mb	txt	$4.3466091156 \times 10^{-2}$	$7.3101997375 \times 10^{-2}$
2.1 mb	txt	$7.6452016830 \times 10^{-2}$	$1.4342093468 \times 10^{-1}$
5 mb	png	$2.8373908997 \times 10^{-1}$	$6.7059397697 \times 10^{-1}$
10.7 mb	zip	$7.6861405373 \times 10^{-1}$	1.4399669170
11.6 mb	bmp	$9.0315604210 \times 10^{-1}$	$3.249371051 \times 10^{-1}$
105.3 mb	pdf	4.6418728828	$1.3878057003 \times 10^{1}$
615 mb	zip	$2.0849997044 \times 10^{1}$	$8.0337167025 \times 10^{2}$
1 gb	txt	$6.8174697876 \times 10^{1}$	$1.2259612106 \times 10^{2}$

2.7. Pruebas experimentales

2.7.1. Compresión del archivo

$f(t) = 0.047t^2 + 60.95t$

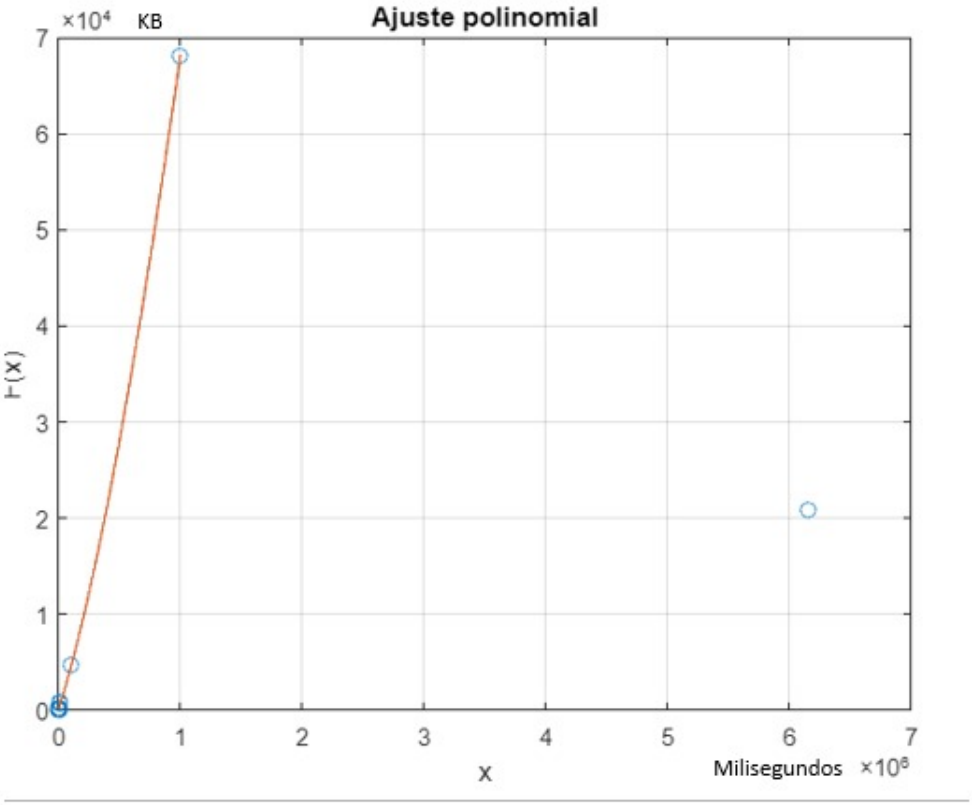

Figura 2.34: Codificación de Huffman

2.7.2. Descompresión del archivo

$f(t) = -0.153t^2 + 216.5914t$

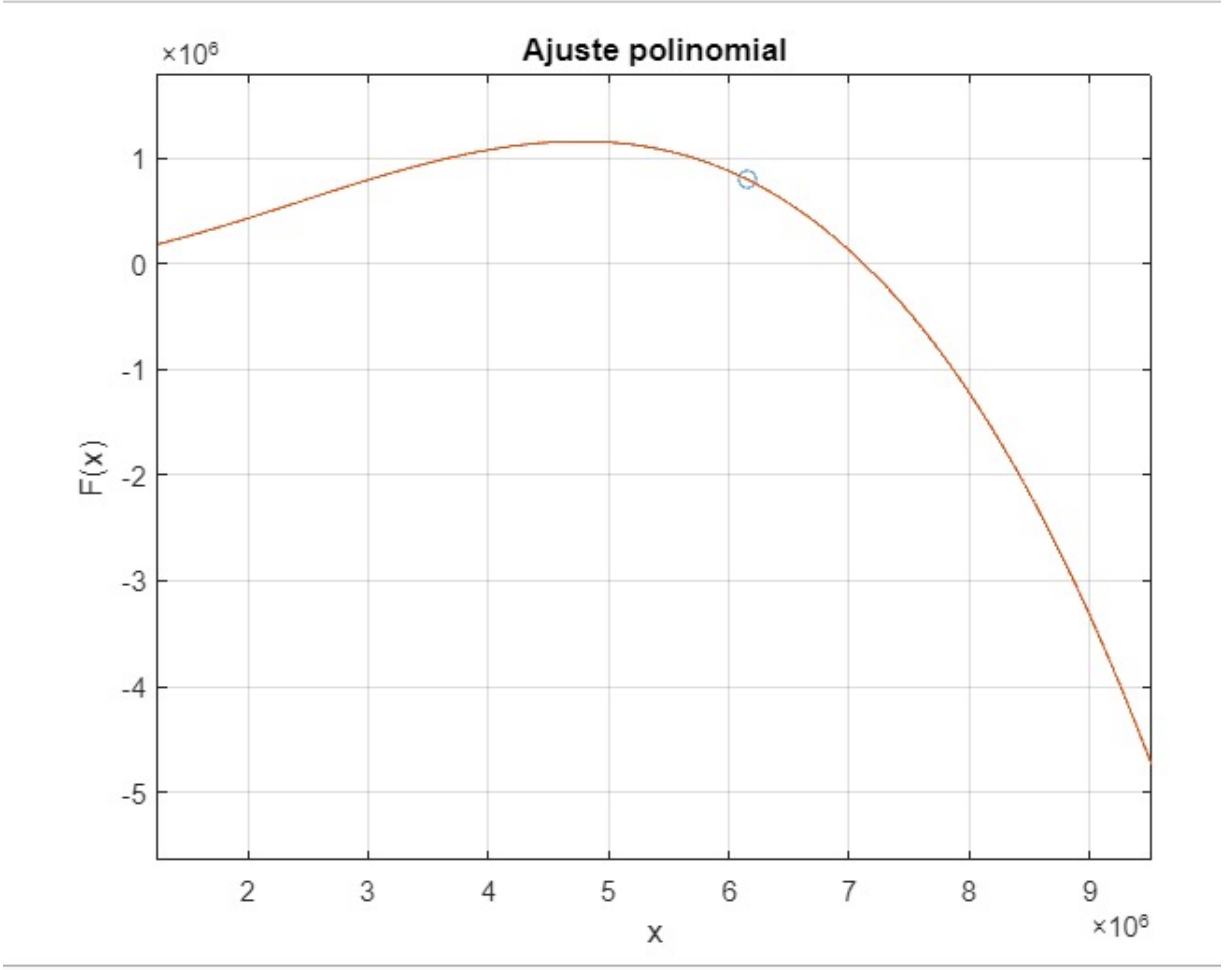

Figura 2.35: Decodificación de Huffman

2.8. Cuestionario

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

2. ¿Los tiempos de codificación o decodificación del archivo son muy grandes?
   No, a menos que se pase el gB, el algoritmo es rápido relativamente.

3. ¿Cual es la proporción temporal que guardan la codificación vs la decodificación?
   Por lo general, la decodificación es mas tardada que la codificación.

4. ¿Ocurrieron perdidas de la información al codificar los archivos?
   No.

5. ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?
   Sí, los pdf o zip son comprimidos en si, es difícil sacarle mas compresión, mientras que los txt o bmp son algoritmos que generalmente se comprimen para enviarse, por lo que era lógico que pudiéramos comprimir dichos archivos de una manera eficiente.

6. ¿Qué características deberá tener un archivo de imagen para codificarse en menor espacio?
   Deben contener colores repetidos en gran medida, para que se lean como bytes repetidos y el algoritmo genere una codificación mas eficiente.

7. ¿Qué características deberá tener un archivo de texto para tener una codificación en menor espacio?
   Que tenga mucho texto similar, de manera ideal, un archivo con el mismo símbolo a lo largo del mismo.

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

   - Al enviar txt muy grandes que tardan años en enviarse por internet.
   - Para comprimir bmp sin perder su calidad ni información ya que estos archivos son muy grandes de tamaño por lo general

9. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?
   Sí, scripts para compilar y ejecutar los archivos.

10. ¿Qué recomendaciones darían a nuevos equipos para realizar esta practica?
    Entender y modularizar el algoritmo lo mas posible, desarrollar las estructuras por separado para después solo cambiar sus campos para aceptar bytes enmascarados, desarrollar pruebas de escritorio ademas de entender corrimientos y operaciones con bits de datos primitivos.

Capítulo 3

Anexo

3.1. Codificación

3.1.1. Header

```
/**
 * @file
 * @brief Este programa comprime un archivo usando el algoritmo de Huffman
 *
 * @author Victor Torres
 * @author Leilani Sotelo
 * @author Guillermo Sanchez
 * @version 1.0
 */
//compilacion: gcc -lm tiempo.c HuffmanStructures.c CompressHuffman.c -o CompressHuffman
//ejecución ./CompressHuffman nombreDelArchivo
/**
 * Lee los bytes del archivo de entrada y lo guarda en una cadena.
 * Esta funcion lee un archivo de entrada, calcula el tamaño del
 * archivo, lee los bytes del archivo y los guarda en una cadena
 * ademas registra la frecuencia de aparicion de cada byte
 * aplicando el concepto de Hash, donde la llave es el valor
 * en decimal del byte que leimos
 *
 * @param fileToOpen nombre del archivo a abrir
 * @param bytesFrecuency arreglo donde esta la frecuencia de repeticion de bits
 * @param fileSize direccion de memoria de la variable del tamaño del archivo
 * @return el apuntador de la cadena de los bytes leidos
 */
unsigned char *readFile(const char *fileToOpen, struct data bytesFrecuency[],
                        int *fileSize);
/**
 *
```

```
 * Escribe la tabla de frecuencias y el tamaño del archivo leido.
 * Escribe en un txt, el tamaño original del archivo ademas de
 * el valor en decimal de cada bit que aparecio seguida de las
 * veces que aparecio solo si la frecuencia de aparicion es
 * mayor a 0, esto es, que el byte esta en el archivo original
 *
 * @param symbolFrecuency arreglo donde esta la frecuencia de repeticion de bits
 * @param fileSize direccion de memoria de la variable del tamaño del archivo
 */
void writeFrecuenyTable(struct data bytesFrecuency[], int fileSize);
/**
 * Escribe la codificacion de Huffman de cada byte correspondiente.
 * En un archivo llamado byteCode.dat, escribe el equivalente de
 * la codificacion de Huffman de los bytes leidos del archivo original,
 * esto lo hace mediante corrimientos de bits y con una bandera llamada
 * sizeByteToWrite que nos indica si tenemos los 8 bits para escribir
 * en el .dat, si no la guarda y mediante corrimeintos la suma con la
 * codificacion del siguiente byte, en el ultimo byte del .dat, escribe
 * cuantos bits extra escribimos que no corresponden a la codificacion
 * de Huffman
 *
 * @param bytesRead el apuntador al inicio de la cadena de los bytes leidos
 * @param mainTree el arbol de Huffman donde unimos todos los bytes y su frecuencia
 * @param frecuencyTable un arreglo donde temenos la codificacion de Huffman de los byte
 * @param fileSize el tamaño del archivo original
 */
void writeBinaryCode(unsigned char *bytesRead, struct node *mainTree,
                     struct bits *bytesCode, int fileSize,
                     int *compressedFileSize);
```

3.1.2. Codigo

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "HuffmanStructures.h"
#include "CompressHuffman.h"
#include "tiempo.h"

unsigned char *readFile(const char *fileToOpen, struct data bytesFrecuency[],
                        int *fileSize) {
  int i;
  FILE *file;
  unsigned char c;
```

```
  // Abrimos y verificamos que si se abrio correctamente
  file = fopen(fileToOpen, "rb");
  if (file == NULL) {
    puts("The file could not be opened.\n");
    exit(1);
  }

  // Obtenemos el tamaño del archivo.
  fseek(file, 0L, SEEK_END);
  (*fileSize) = ftell(file);
  rewind(file);

  // Reservamos memoria para la cadena de bytes leidos
  unsigned char *bytesRead = malloc((*fileSize) * sizeof(unsigned char));

  // Guardamos los bytes leidos y su frecuencia con su valor en decimal
  for (i = 0; i < (*fileSize); i++) {
    fread(&c, sizeof(unsigned char), 1, file);
    bytesRead[i] = c;
    bytesFrecuency[c].byte = c;     // Enmascaramiento
    bytesFrecuency[c].frequency++; // frecuencia de apariciones
  }
  fclose(file);
  return bytesRead;
}

void writeFrecuenyTable(struct data bytesFrecuency[], int fileSize)
{
    int i;
    FILE *frecuencyTable;

    //Abrimos y verificamos que si se abrio correctamente
    frecuencyTable = fopen("frecuencias.txt", "wt+");
    if (frecuencyTable == NULL)
    {
        puts("The file could not be opened.\n");
        exit(1);
    }

    //Escribimos el tamaño original al inicio del txt
    fprintf(frecuencyTable, "%d\n", fileSize);
    for (i = 0; i < 256; i++)
      if (bytesFrecuency[i].frequency > 0)
        fprintf(frecuencyTable, "%d %d\n", bytesFrecuency[i].byte,
```

```
                bytesFrecuency[i].frequency);
    // Si el simbolo aparecio en el archivo, escribelo

    fclose(frecuencyTable);
}

void writeBinaryCode(unsigned char *bytesRead, struct node *mainTree,
                     struct bits *bytesCode, int fileSize,
                     int *compressedFileSize) {
  int i, j, bitsExtraWritten = 0;
  FILE *binaryCode;

  // Abrimos y verificamos que si se abrio correctamente
  binaryCode = fopen("codificacion.dat", "wb+");
  if (binaryCode == NULL) {
    puts("The file could not be opened.\n");
    exit(1);
  }

  // Variables para poder escribir los bits
  int bitsSize, sizeByteToWrite, tempSize = 0;
  int bits, tempBits, tempBits2, tempBitsAux;
  unsigned char byteToWrite;

  for (i = 0; i < fileSize; i++) {
    bitsSize = bytesCode[bytesRead[i]].sizeBits;
    bits = bytesCode[bytesRead[i]].bits;

    // Checamos si hay bits que no escribimos
    if (tempSize != 0) {
      bits = (tempBits << bitsSize) + bits;
      bitsSize += tempSize;
      tempBits = 0;
      tempSize = 0;
    }

    // Podemos escribir o no el byte?
    if (bitsSize >= 8) {
      if (bitsSize > 8) {
        // Tenemos mas bits de los que podemos escribir, hay que reducir
        // Guardamos lo que no podemos escribir en una variable temporal
        tempSize = bitsSize - 8;
        for (j = 0; j < tempSize; j++) {
          tempBits2 = (CONSULTARBIT(bits, j)) << j;
          tempBits = tempBits2 + tempBits;
```

```
      }
      // ajustamos los bits para poder escribir los primeros 8
      bits = bits >> tempSize;
    }
    byteToWrite = bits;
    sizeByteToWrite = 8; // Podemos escribir en el .dat
  } else {
    tempBits = bits;     // Guardamos en una variable temporal
    tempSize = bitsSize; // lo que no pudimos escribir
  }

  // Hay bytes por escribir?
  if (sizeByteToWrite == 8) {
    sizeByteToWrite = 0; // Escribimos el byte
    fwrite(&byteToWrite, sizeof(unsigned char), 1, binaryCode);
  }

  // Checamos que no se desborde tempBits
  while (tempSize >= 8) {
    // Ajustamos tempsize para poder escribir los primeros 8
    tempSize = tempSize - 8;
    tempBitsAux = tempBits;
    bits = tempBits >> tempSize;
    tempBits = 0;

    // Guardamos el exceso en una variable temporal
    for (j = 0; j < tempSize; j++) {
      tempBits2 = ((CONSULTARBIT(tempBitsAux, j)) << j);
      tempBits = tempBits2 + tempBits;
    }
    byteToWrite = bits; // Escribimos el byte
    fwrite(&byteToWrite, sizeof(unsigned char), 1, binaryCode);
  }
}

// Faltaron bits por escribir?
if (tempBits != 0) {
  tempBits = (tempBits << (8 - tempSize));
  byteToWrite = tempBits;
  fwrite(&byteToWrite, sizeof(unsigned char), 1, binaryCode);
  bitsExtraWritten = 8 - tempSize; // Bits extra escritos
}

// En el ultimo byte, indica cuantos bits extra escribimos
byteToWrite = bitsExtraWritten;
```

```
  fwrite(&byteToWrite, sizeof(unsigned char), 1, binaryCode);

  // Obtenemos el tamaño del archivo comprimido
  fseek(binaryCode, 0L, SEEK_END);
  (*compressedFileSize) = ftell(binaryCode);
  rewind(binaryCode);
  // Cerrar el archivo
  fclose(binaryCode);
}

int main(int argc, char *argv[])
{
    //*********************************************************
    //Variables a usar en el programa

    //Variables para el control de tiempo
    double utime0, stime0, wtime0, utime1, stime1, wtime1;
    //Arreglo para almacenar la frecuencia de bytes
    struct data *bytesFrecuency = malloc(256 * sizeof(struct data));
    //Arreglo para almacenar la codificacion de Huffman
    struct bits *bytesCode = malloc(256 * sizeof(struct bits));
    //Arreglo para almacenar el byte y frecuencia en cada nodo del arbol
    struct node *roots = (struct node *)malloc(256 * sizeof(struct node));
    //Arbol de Huffman ya unificado
    struct node *HuffmanTree;
    //Cadena de bytes leidos del archivo original
    unsigned char *bytesRead;
    //Cola de prioridad
    Heap *heap = CreateHeap(511);
    //Variable para almacenar el tamaño del archivo y del comprimido
    int fileSize, compressedFileSize;

    puts("\nComprimir archivos usando el algoritmo de Huffman.\n");
    //*************************************************************
    // Iniciar el conteo del tiempo para las evaluaciones de rendimiento
    //*************************************************************
    uswtime(&utime0, &stime0, &wtime0);

    //Guardamos en un arreglo todos los bytes leidos
    bytesRead = readFile(argv[1], bytesFrecuency, &fileSize);
    //formar en el arbol las frecuencias y formar esos arboles en la cola
    insertTree(bytesFrecuency, roots, heap);
    //unir los arboles en el arbol de Huffman
    HuffmanTree = mergeTrees(heap);
    //Obtener los bits que vale cada byte
```

```
    getBits(HuffmanTree, bytesCode, 0, 0);
    //escribir el .dat con los bits de cada byte correspondiente
    writeBinaryCode(bytesRead, HuffmanTree, bytesCode, fileSize, &compressedFileSize);
    //escribir la tabla de frecuencias y el tamaño
    writeFrecuenyTable(bytesFrecuency, fileSize);

    //*****************************************************************
    // Evaluar los tiempos de ejecución
    //*****************************************************************
    uswtime(&utime1, &stime1, &wtime1);

    printf("Tamaño de %s: %d bytes\n", argv[1], fileSize);
    printf("Tamaño del archivo comprimido: %d bytes\n", compressedFileSize);
    printf("Porcentaje de compresion alcanzado: %.2f%%\n",
           ((float)fileSize / (float)compressedFileSize) * 100);
    printf("Tiempo real de Ejecucion: %.10e s\n\n", wtime1 - wtime0);

    return 0;
}
```

3.2. Decodificación

3.2.1. Header

```
/**
 * @file
 * @brief Este programa descomprime un archivo usando el algoritmo de Huffman
 *
 * @author Victor Torres
 * @author Leilani Sotelo
 * @author Guillermo Sanchez
 * @version 1.0
 */
// compilacion: gcc -lm tiempo.c HuffmanStructures.c DecompressHuffman.c -o
// DecompressHuffman
// ejecución: ./DecompressHuffman nombreDelArchivo
/**
 *
 * Lee y guarda la tabla de frecuencias. A partir de la tabla de frecuencia
 * generada al comprimir el archivo, leemos las veces que se se repitio
 * cada bit, ademas al principio del archivo leemos el tamaño original del
 * archivo, las frecuencias las guardamos en un arreglo, con su respectivo
 * byte aplicando hash, donde la llave es el byte.
 *
```

```
 * @param bytesFrecuency arreglo donde esta la frecuencia de repeticion de bits
 * @param fileSize direccion de memoria de la variable del tamaño del archivo
 */
void readFrecuencyTable(struct data bytesFrecuency[], int *fileSize);

/**
 *
 * Lee el archivo comprimido. Obtenemos el tamaño del archivo comprimido
 * y los bytes que lo componen, esto lo guardamos en una cadena.
 *
 * @param fileSize direccion de memoria de la variable del tamaño del archivo
 * @param byteFileSize direccion de memoria de la variable del tamaño del
 * archivo comprimido
 * @return el arreglo donde estan los bytes leidos
 */
unsigned char *readByteCode(int *fileSize, int *byteFileSize);

/**
 *
 * Escribimos el archivo original a partir de la decodificacion de los bytes.
 * Vamos recorriendo bit a bit de los bytes leidos en el archivo comprimido,
 * y recorremos el arbol de Huffman, en base a esto nos devuelve tanto la
 * la posicion en bits de 0 a 7 en la que se quedo, como en la variable
 * byteToWrite, el byte correspondiente a cada cadena de bits, una ves que
 * tenemos esto, escribimos en el archivo el byte correspondiente.
 *
 * @param bytesRead el apuntador al inicio de la cadena de los bytes leidos
 * @param mainTree el arbol de Huffman donde unimos todos los bytes y su frecuencia
 * @param file el nombre del archivo en el que escribiremos
 * @param byteFileSize el tamaño del archivo comprimido
 * @param fileSize apuuntador de la variable del tamaño del archivo que escribiremos
 */
void writeFile(unsigned char *bytesRead, struct node *mainTree,
               const char *file, int byteFileSize, int *fileSize);
```

3.2.2. Codigo

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "HuffmanStructures.h"
#include "DecompressHuffman.h"
#include "tiempo.h"
```

```
void readFrecuencyTable(struct data bytesFrecuency[], int* fileSize){
    int byte, frecuency, i = 0;
    FILE* frecuencyTable;

    //Abrimos y verificamos que si se abrio correctamente
    frecuencyTable = fopen("frecuencias.txt", "r");
    if(frecuencyTable == NULL){
        puts("The file could not be opened.\n");
        exit(1);
    }

    //Leemos el tamaño de archivo y las veces que se repite cada byte
    while(!feof(frecuencyTable)){
        if(i == 0)
            fscanf(frecuencyTable, "%d", fileSize);
        fscanf(frecuencyTable, "%d", &byte);
        fscanf(frecuencyTable, "%d", &frecuency);
        bytesFrecuency[byte].byte =  byte;
        bytesFrecuency[byte].frequency = frecuency;
        i++;
    }
    fclose(frecuencyTable);
}

unsigned char* readByteCode(int* byteFileSize){
    unsigned char c;
    int i;
    FILE* file;

    //Abrimos y verificamos que si se abrio correctamente
    file = fopen("codificacion.dat", "rb");
    if(file == NULL){
        puts("Open file Failed");
        exit(1);
    }

    //Obtenemos el tamaño del archivo .dat
    fseek(file, 0L, SEEK_END);
    (*byteFileSize) = ftell(file);
    rewind(file);

    //Reservamos memoria donde guardaremos los bytes leidos
    unsigned char* bytesRead = malloc((*byteFileSize) * sizeof(unsigned char));

    //leemos los bytes del .dat y los guardamos en un arreglo
```

```
    for(i = 0; i < (*byteFileSize); i++){
        fread(&c, sizeof(unsigned char), 1, file);
        bytesRead[i] = c;
    }
    fclose(file);
    return bytesRead;
}

void writeFile(unsigned char *bytesRead, struct node *mainTree,
               const char *file, int byteFileSize, int *fileSize) {
  FILE *finalFile;

  // Abrimos y verificamos que si se abrio correctamente
  finalFile = fopen(file, "wb+");
  if (finalFile == NULL) {
    puts("The file could not be opened.\n");
    exit(1);
  }

  // Variables para escribir en el archivo
  int i, posInBits = 7, bytesWritten = 0;
  unsigned char byteToWrite;

  // Leemos los bits extra que escribimos
  int bitsExtraWritten = bytesRead[byteFileSize - 1];

  for (i = 0; i < byteFileSize - 1 && bytesWritten < (*fileSize);) {
    posInBits = getCharacters(mainTree, bytesRead, &i, posInBits, &byteToWrite);
    // escribimos el byte correspondiente a los bits
    fwrite(&byteToWrite, sizeof(unsigned char), 1, finalFile);
    bytesWritten++; // conteo para no escribir de mas

    // Cuidamos no escribir los bits extra que escribimos al comprimir
    if (i == (*fileSize) - 2 && 7 - posInBits == bitsExtraWritten)
      bytesWritten++;

    // Checamos que no nos desbordamos de posicion de bits
    if (posInBits < 0) {
      posInBits = 7;
      i++;
    }
  }
  (*fileSize) = bytesWritten;
  fclose(finalFile);
}
```

```
int main(int argc, char* argv[]){
    //*****************************************************
    //Variables a usar en el programa

     //Variables para el control de tiempo
    double utime0, stime0, wtime0, utime1, stime1, wtime1;
    //Arreglo para almacenar la frecuencia de bytes
    struct data* bytesFrecuency = malloc(256 * sizeof(struct data));
    //Arreglo para almacenar la codificacion de Huffman
    struct bits* bytesCode = malloc(256* sizeof(struct bits));
    //Arreglo para almacenar el byte y frecuencia en cada nodo del arbol
    struct node* roots = (struct node*)malloc(256 * sizeof(struct node));
    //Arbol de Huffman ya unificado
    struct node* HuffmanTree;
    //Cadena de bytes leidos del archivo original
    unsigned char* bytesRead;
    //Cola de prioridad
    Heap* heap = CreateHeap(511);
    //Variables para almacenar los tamaños de archivos
    int fileSize = 0, byteFileSize = 0;


    puts("\nDescomprimir archivos usando el algoritmo de Huffman\n");
    //*****************************************************************
    // Iniciar el conteo del tiempo para las evaluaciones de rendimiento
    //*****************************************************************
    uswtime(&utime0, &stime0, &wtime0);

    //leer la tabla de frecuencias
    readFrecuencyTable(bytesFrecuency, &fileSize);
    //hacer el arbol con la tabla de frecuencias
    insertTree(bytesFrecuency, roots, heap);
    //unir los arboles en uno solo
    HuffmanTree = mergeTrees(heap);
    //Obtener el codigo de bits de cada byte
    getBits(HuffmanTree, bytesCode, 0, 0);
    //leer el .dat generado en la compresion
    bytesRead = readByteCode(&byteFileSize);
    //sustituir la cadena de .dat por su valor de arbol
    writeFile(bytesRead, HuffmanTree, argv[1], byteFileSize, &fileSize);

    //*****************************************************************
    // Evaluar los tiempos de ejecución
    //*****************************************************************
```

```
    uswtime(&utime1, &stime1, &wtime1);

    printf("Tamaño de %s: %d bytes\n", argv[1], fileSize);
    printf("Tiempo real de Ejecucion: %.10e s\n\n", wtime1 - wtime0);

    return 0;
}
```

3.3. Estructuras de Huffman

3.3.1. Header

```
/**
 * @file
 * @brief Las esctructuras y funciones usadas en el Algoritmo de Huffman
 *
 * @author Victor Torres
 * @author Leilani Sotelo
 * @author Guillermo Sanchez
 * @version 1.0
 */

//*************************************************************
//Estructuras utilizadas en el programa

/** @struct bits
 * @brief Almacena el byte correspondiente, su tamaño y sus bits de la
 * codificacion de Huffman
 * @var bits::byte
 * 'byte' contiene el valor decimal del byte leido
 * @var bits::bits
 * 'bits' contiene el valor en entero de los bits correspondientes a la
 * codifiacion
 * @var bits::sizeBits
 * 'sizeBits' contiene el tamaño de los bits correspondientes a la codifcacion
 */

struct bits
{
    unsigned char byte;
    int sizeBits;
    int bits;
};
```

```
/** @struct data
 * @brief Almacena un byte y las veces que aparecio en el archivo
 * @var data::byte
 * 'byte' contiene el valor decimal del byte leido
 * @var data::frecuency
 * 'frecuency' contiene las veces que aparecio el byte en el archivo leido
 */
struct data
{
    int frequency;
    int byte;
};

/** @struct node
 * @brief Estructura del arbol de Huffman
 * @var node::left
 * 'left' contiene el nodo hijo izquierdo
 * @var node::right
 * 'right' contiene el nodo hijo derecho
 * @var node::data
 * 'data' contiene el byte correspondiente y su frecuencia
 */
struct node
{
    struct node *left;
    struct node *right;
    struct data data;
};

/** @struct Heap
 * @brief Almacena un arreglo de nodos
 * @var node::arrayOfNodes
 * 'arrayOfNodes' contiene un arreglo de nodos
 * @var Heap::count
 * 'count' contiene los nodos existentes en el arreglo
 * @var Heap::capacity
 * 'capacity' contiene la capacidad maxima del arreglo
 */
struct Heap
{
    struct node **arrayOfNodes;
    int count;
    int capacity;
};
//Alias de Heap
```

```
typedef struct Heap Heap;

//************************************************************
//Funciones con el arbol de Huffman

/**
 * Nos dice si nodo en el arbol de Huffman es vacio o no.
 *
 * @param root el nodo a comparar
 * @return 1 si el arbol es vacio
 * @return 0 si el arbol no es vacio
 */
int isEmpty(struct node *root);
/**
 * Nos dice si nodo en el que estamos es un nodo hoja.
 *
 * @param root el nodo a analizar
 * @return 1 si el nodo es hoja
 * @return 0 si el nodo no es hoja
 */
int isLeaf(struct node *root);
/**
 * Inserta dado un byte y una frecuencia, las inserta en el nodo del arbol
 *
 * @param root el arbol de Huffman donde se insertara
 * @param byte el byte correspondiente de su frecuencia
 * @param frecuency la frecuencia correspondiente del byte
 */
void pushTree(struct node *root, unsigned char byte, int frecuency);
/**
 * Une dos nodos en un nodo ancestro comun
 *
 * @param node1 el primer nodo a unir que estara del lado izquierdo
 * @param node2 el segundo nodo a unir que estara del lado derecho
 * @return el nodo comun que tiene como hijos a ambos nodos
 */
struct node *mergeNodes(struct node *node1, struct node *node2);
/**
 * Obtiene la codificaciond de Huffman correspondiente a los bytes.
 * La guarda en un arreglo de struct junto con el tamaño de bits
 * correspondiente. La codificacion se obtiene recorriendo el arbol
 * inorder y cada vez registramos un 0 o un 1 si nos vamos a la
 * izquierda o derecha, al final si es nodo hoja, guardamos lo que
 * llevamos
 *
```

```
 * @param HuffmanTree el arbol de Huffman
 * @param bytesCode el arreglo donde se guarda el byte, su codificacion y tamaño
 * @param bits los bits con los que inicia el nodo que por defecto es 0
 * @param bitsSize el tamaño de los bits que por defecto es 0
 */
void getBits(struct node *HuffmanTree, struct bits bytesCode[], int bits,
             int bitsSize);
/**
 * Obtiene el byte correspondiente a la codificacion de Huffman. Recorre
 * el arbol analizando si es 0 o 1 el bit en el que esta para irse
 * a la izquierda o a la derecha, cuando detecta que ya ha llegado a un
 * nodo hoja, regresa en byteToWrite el valor del byte dado la cadena
 * de bits
 *
 * @param HuffmanTree el arbol de Huffman
 * @param cadena la cadena de bytes leidos del archivo comprimido
 * @param posInString la posicion del arreglo de la cadena
 * @param posInBits la posicion de bits en la que vamos en relacion a un byte
 * @param byteToWrite la variable donde guardaremos el byte correspondiente
 * @return la posicion en bits en donde nos quedamos en relacion a un byte
 */
int getCharacters(struct node *HuffmanTree, unsigned char *cadena,
                  int *posInString, int posInBits, unsigned char *byteToWrite);

//*************************************************************
//Funciones con heap

//Crea un espacio en memoria para el heap
/**
 * Crea una estructura Heap en memoria dinamica dada su
 * capacidad, la cual es un arreglo de nodos.
 *
 * @param capacity es la capacidad que queremos que tenga heap
 * @return el apuntador a la estructura heap creada
*/
Heap *CreateHeap(int capacity);
//Inserta elementos en el heap
/**
 * Inserta Nodos en la estructura de datos, indicando el Heap donde se desea
 * hacer la incersion, seguido del nodo que se desea insertar, simempre y cuando
 * la capacidad de Heap no haya sido exedida.
 *
 * @param heap es el monticulo donde vamos a insertar el nodo
 * @param node es el nodo a insertar dentro del monticulo
*/
```

```
void insert(Heap *heap, struct node *node);
/**
 * Ordena los elementos del arreglo de nodos, haciendo que el elemento mas
 * pequeño sea la rama principal de nuestro arbol.
 *
 * @param Heap es el monticulo al que vamos a ordenar su arreglo de nodos
 * @param index es el indice en este caso el numero de elementos que hay que ordenar
*/
void heapify_bottom_top(Heap *heap, int index);
/**
 * Ordena los elementos del arreglo de nodos, haciendo que el elemento mas
 * pequeño sea la rama principal de nuestro arbol,este lo usamos al momento de
 * retirar el elemento principal de nuestro Heap, debido a que el ordenamiento
 * lo realiza a partir de nuestra raiz hacia los elementos hijos del arbol.
 *
 * @param Heap es el monticulo al que vamos a ordenar su arreglo de nodos
 * @param parent_node es el indice del elemento padre de nuestro nodo para
realizar
 *
*/
void heapify_top_bottom(Heap *heap, int parent_node);
//Elimina el primer elemento de la cola de prioridad

/**
 * Elmina el elemento mas pequeño de una cola de prioridad
 *
 * @param Heap Es la estructura heap
 * @return nodo eliminado
*/
struct node *PopMin(Heap *heap);

//*****************************************************************
//Funciones con el arbol de huffman y heap

/**
 * Inserta en la cola de prioridad los nodos. EL numero de nodos
 * que inserta son los bytes distintos que aparecieron en el archivo
 *
 * @param bytesFrecuency el arreglo de struct donde viene el byte y su frecuencia
 * @param roots el arreglo de nodos que trae todos los bytes y su frecuencia
 * @param heap la cola de prioridad
 */
void insertTree(struct data bytesFrecuency[], struct node roots[], Heap *heap);
/**
 * Une todos los nodos hoja en un solo, llamado Arbol de Huffman.
```

```
 * Usando la cola de prioridad saca los dos nodos de menor frecuencia
 * suma sus dos frecuencia y los une en un nodo padre, haciendo esto
 * hasta que solo quede un nodo padre el cual contiene a todos los nodos
 * de los bytes leidos
 *
 * @param heap la cola de prioridad con los nodos de los bytes leidos
 * @return el Arbol de Huffman
 */
struct node *mergeTrees(Heap *heap);

//*********************************************************************
//Macros para trabajas con bits
#define PESOBIT(bpos) 1 << bpos
#define CONSULTARBIT(var, bpos) (*(unsigned *)&var & PESOBIT(bpos)) ? 1 : 0
```

3.3.2. Codigo

```
#include <stdio.h>
#include <stdlib.h>
#include "HuffmanStructures.h"

//*********************************************************************
//Operaciones con el arbol de Huffman.
//*********************************************************************
//Es vacio el arbol
int isEmpty(struct node *root) { return root == NULL; }
//Es un nodo hoja
int isLeaf(struct node *root) { return root->left == NULL && root->right == NULL; }

//Insertar elementos en el arbol huffman
void pushTree(struct node *root, unsigned char byte, int frecuency)
{
    root->data.frequency = frecuency;
    root->data.byte = byte;
    root->left = NULL;
    root->right = NULL;
}
//unir los nodos
struct node *mergeNodes(struct node *node1, struct node *node2)
{
    struct node *new_node = malloc(sizeof(struct node));
    new_node->data.byte = 0;
    new_node->left = node1;
    new_node->right = node2;
    new_node->data.frequency = node1->data.frequency + node2->data.frequency;
```

```
    return new_node;
}
//obtener la codificacion de los caracteres
void getBits(struct node *HuffmanTree, struct bits bytesCode[], int bits,
             int sizeBits) {
  if (!isEmpty(HuffmanTree)) { // if notempty
    getBits(HuffmanTree->left, bytesCode, (bits << 1),
            sizeBits + 1); // irte al nodo izq, sumandole un 0
    if (isLeaf(HuffmanTree)) {
      int hashKey = HuffmanTree->data.byte;
      bytesCode[hashKey].bits = bits;
      bytesCode[hashKey].byte = HuffmanTree->data.byte;
      bytesCode[hashKey].sizeBits = sizeBits;
    }
    getBits(HuffmanTree->right, bytesCode, (bits << 1) + 1,
            sizeBits + 1); // irte al nodo der sumandole un 1
  }
}

int getCharacters(struct node *HuffmanTree, unsigned char *cadena,
                  int *posInString, int posInBits, unsigned char *byteToWrite) {
  if (isLeaf(HuffmanTree)) {
    *byteToWrite = HuffmanTree->data.byte;
    return posInBits;
  } else {
    if (posInBits < 0) {
      (*posInString)++;
      posInBits = 7;
    }
    if (((int)CONSULTARBIT(cadena[(*posInString)], (posInBits))) == 0)
      return getCharacters(HuffmanTree->left, cadena, posInString,
                           (posInBits)-1, byteToWrite);
    else
      return getCharacters(HuffmanTree->right, cadena, posInString,
                           posInBits - 1, byteToWrite);
  }
}

//*****************************************************************
//Funciones de heap.
//*****************************************************************
Heap *CreateHeap(int capacity)
{
    Heap *heap = (Heap *)malloc(sizeof(Heap));
    heap->count = 0;
```

```
    heap->capacity = capacity;
    heap->arrayOfNodes = (struct node **)malloc(capacity * sizeof(struct node));
    return heap;
}

void insert(Heap *heap, struct node *node)
{
    if (heap->count < heap->capacity)
    {
        heap->arrayOfNodes[heap->count] = node;
        heapify_bottom_top(heap, heap->count);
        heap->count++;
    }
}

void heapify_bottom_top(Heap *heap, int index)
{
    struct node *temp;
    int parent_node = (index - 1) / 2;
    if (heap->arrayOfNodes[parent_node]->data.frequency >
        heap->arrayOfNodes[index]->data.frequency) {
      temp = heap->arrayOfNodes[parent_node];
      heap->arrayOfNodes[parent_node] = heap->arrayOfNodes[index];
      heap->arrayOfNodes[index] = temp;
      heapify_bottom_top(heap, parent_node);
    }
}

void heapify_top_bottom(Heap *heap, int parent_node)
{
    int left = parent_node * 2 + 1;
    int right = parent_node * 2 + 2;
    int min;
    struct node *temp;

    if (left >= heap->count || left < 0)
        left = -1;
    if (right >= heap->count || right < 0)
        right = -1;

    if (left != -1 && heap->arrayOfNodes[left]->data.frequency <
                          heap->arrayOfNodes[parent_node]->data.frequency)
      min = left;
    else
      min = parent_node;
```

```
    if (right != -1 && heap->arrayOfNodes[right]->data.frequency <
                            heap->arrayOfNodes[min]->data.frequency)
      min = right;

    if (min != parent_node)
    {
        temp = heap->arrayOfNodes[min];
        heap->arrayOfNodes[min] = heap->arrayOfNodes[parent_node];
        heap->arrayOfNodes[parent_node] = temp;

        heapify_top_bottom(heap, min);
    }
}

struct node *PopMin(Heap *heap)
{
    struct node *pop;
    if (heap->count == 0)
    {
        printf("Heap is Empty.\n");
        return NULL;
    }
    pop = heap->arrayOfNodes[0];
    heap->arrayOfNodes[0] = heap->arrayOfNodes[heap->count - 1];
    heap->count--;
    heapify_top_bottom(heap, 0);
    return pop;
}
//**********************************************************************
//Funciones con el arbol y heap
//**********************************************************************
//formar en el arbol las frecuencias y formar esos arboles en la cola
void insertTree(struct data bytesFrecuency[], struct node roots[], Heap *heap)
{
    int i;
    for (i = 0; i < 256; i++) {
      if (bytesFrecuency[i].frequency > 0) {
        pushTree(&roots[i], bytesFrecuency[i].byte,
                 bytesFrecuency[i].frequency);
        insert(heap, &roots[i]);
      }
    }
}
//unir todos los nodos en uno solo
struct node *mergeTrees(Heap *heap)
```

```
{

    while (heap->count > 1)
    {
        struct node *node1 = PopMin(heap);
        struct node *node2 = PopMin(heap);
        insert(heap, mergeNodes(node1, node2));
    }
    return PopMin(heap);
}
```

3.4. Tiempo

3.4.1. Header

```
//*****************************************************************
// TIEMPO.H
//*****************************************************************
//*****************************************************************
// M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
// Curso: Análisis de algoritmos
//(C) Enero 2013
// ESCOM-IPN
// Ejemplo de medición de tiempo en C y recepción de parametros en C bajo UNIX
// Compilación de la libreria: "gcc -c tiempo.c " (Generación del código objeto)
//*****************************************************************

//*****************************************************************
// uswtime (Declaración)
//*****************************************************************
// Descripción: Función que almacena en las variables referenciadas
// el tiempo de CPU, de E/S y Total actual del proceso actual.
//
// Recibe: Variables de tipo doble para almacenar los tiempos actuales
// Devuelve:
//*****************************************************************
void uswtime(double *usertime, double *systime, double *walltime);
/*
Modo de Empleo:
La función uswtime se puede emplear para medir los tiempos de ejecución de
determinados segmentos de código en nuestros programas. De forma esquemática, el
empleo de esta función constaría de los siguientes pasos:

    1.- Invocar a uswtime para fijar el instante a partir del cual se va a medir
```

```
el tiempo.

            uswtime(&utime0, &stime0, &wtime0);

    2.- Ejecutar el código cuyo tiempo de ejecución se desea medir.
    3.- Invocar a uswtime para establecer el instante en el cual finaliza la
medición del tiempo de ejecución.

            uswtime(&utime1, &stime1, &wtime1);

    4.- Calcular los tiempos de ejecución como la diferencia entre la primera y
segunda invocación a uswtime:

            real:   wtime1 - wtime0
            user:   utime1 - utime0
            sys :   stime1 - stime0

            El porcentaje de tiempo dedicado a la ejecución de ese segmento de
código vendría dado por la relación CPU/Wall:

        CPU/Wall = (user + sys) / real x 100 %*/
```

3.4.2. Codigo

```
//*****************************************************************
//TIEMPO.C
//*****************************************************************
//*****************************************************************
//M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
//Curso: Análisis de algoritmos
//(C) Enero 2013
//ESCOM-IPN
//Ejemplo de medición de tiempo en C y recepción de parametros en C bajo UNIX
//Compilación de la libreria: "gcc -c tiempo.c " (Generación del código objeto)
//*****************************************************************

//*****************************************************************
// Librerias incluidas
//*****************************************************************
#include "tiempo.h"
#include <sys/resource.h>
#include <sys/time.h>

//*****************************************************************
// uswtime (Definición)
```

```
//**************************************************************
// Descripción: Función que almacena en las variables referenciadas
// el tiempo de CPU, de E/S y Total actual del proceso actual.
//
// Recibe: Variables de tipo doble para almacenar los tiempos actuales
// Devuelve:
//**************************************************************#include
//<stdio.h>
void uswtime(double *usertime, double *systime, double *walltime) {
  double mega = 1.0e-6;
  struct rusage buffer;
  struct timeval tp;
  struct timezone tzp;
  getrusage(RUSAGE_SELF, &buffer);
  gettimeofday(&tp, &tzp);
  *usertime = (double)buffer.ru_utime.tv_sec + 1.0e-6 * buffer.ru_utime.tv_usec;
  *systime = (double)buffer.ru_stime.tv_sec + 1.0e-6 * buffer.ru_stime.tv_usec;
  *walltime = (double)tp.tv_sec + 1.0e-6 * tp.tv_usec;
}

/*En Unix, se dispone de temporizadores ejecutables (en concreto time) que nos
proporcionan medidas de los tiempos de ejecución de programas. Estos
temporizadores nos proporcionan tres medidas de tiempo:

    * real:  Tiempo real que se ha tardado desde que se lanzó el programa a
ejecutarse hasta que el programa finalizó y proporcionó los resultados.
    * user: Tiempo que la CPU se ha dedicado exclusivamente a la computación del
programa.
    * sys:         Tiempo que la CPU se ha dedicado a dar servicio al sistema
operativo por necesidades del programa (por ejemplo para llamadas al sistema
para efectuar I/O).

El tiempo real también suele recibir el nombre de elapsed time o wall time.
Algunos temporizadores también proporcionan el porcentaje de tiempo que la CPU
se ha dedicado al programa. Este porcentaje viene dado por la relación entre el
tiempo de CPU (user + sys) y el tiempo real, y da una idea de lo cargado que se
hallaba el sistema en el momento de la ejecución del programa.

El grave inconveniente de los temporizadores ejecutables es que no son capaces
de proporcionar medidas de tiempo de ejecución de segmentos de código. Para
ello, hemos de invocar en nuestros propios programas  a un conjunto de
temporizadores disponibles en la mayor parte de las librerías de C de Unix, que
serán los que nos proporcionen medidas sobre los tiempos de ejecución de trozos
discretos de código.
```

```
En nuestras prácticas vamos a emplear una función que actúe de temporizador y
que nos proporcione los tiempos de CPU (user, sys) y el tiempo real. En
concreto, vamos a emplear el procedimiento uswtime listado a continuación.

Este procedimiento en realidad invoca a dos funciones de Unix: getrusage y
gettimeofday. La primera de ellas nos proporciona el tiempo de CPU, tanto de
usuario como de sistema, mientras que la segunda nos proporciona el tiempo real
(wall time). Estas dos funciones son las que disponen de mayor resolución de
todos los temporizadores disponibles en Unix.

Modo de Empleo:

La función uswtime se puede emplear para medir los tiempos de ejecución de
determinados segmentos de código en nuestros programas. De forma esquemática, el
empleo de esta función constaría de los siguientes pasos:

    1.- Invocar a uswtime para fijar el instante a partir del cual se va a medir
el tiempo.

            uswtime(&utime0, &stime0, &wtime0);

    2.- Ejecutar el código cuyo tiempo de ejecución se desea medir.
    3.- Invocar a uswtime para establecer el instante en el cual finaliza la
medición del tiempo de ejecución.

            uswtime(&utime1, &stime1, &wtime1);

    4.- Calcular los tiempos de ejecución como la diferencia entre la primera y
segunda invocación a uswtime:

            real:   wtime1 - wtime0
            user:   utime1 - utime0
            sys :   stime1 - stime0

            El porcentaje de tiempo dedicado a la ejecución de ese segmento de
código vendría dado por la relación CPU/Wall:

        CPU/Wall = (user + sys) / real x 100 %*/
```