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

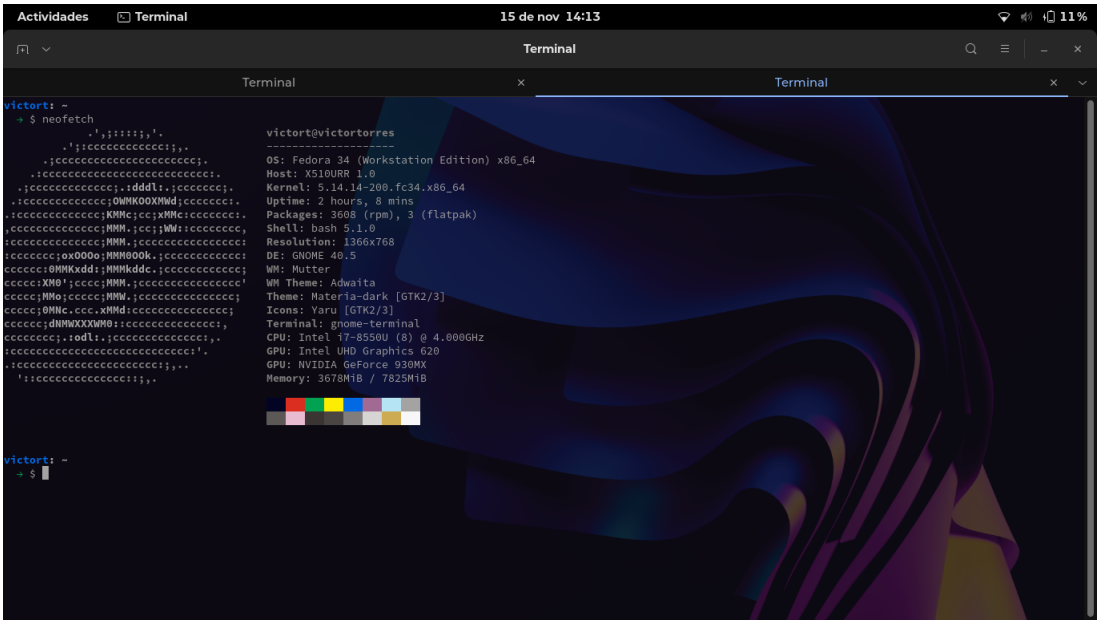


Figura 2.1

2.2. Descripción de la solución

2.2.1. 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.

2.2.2. 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.

2.2.3. 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

2.3. Análisis de complejidad

2.3.1. Análisis de la codificación de Huffman

```c
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 tamaÃ±o 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 tamaÃ±o
    writeFrecuenyTable(bytesFrecuency, fileSize);

    //*****************************************************************
    // Evaluar los tiempos de ejecuciÃ³n
    //*****************************************************************
    uswtime(&utime1, &stime1, &wtime1);

    printf("TamaÃ±o de %s: %d bytes\n", argv[1], fileSize);
    printf("TamaÃ±o 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;
}
```

O(n)

O(n^2)

O(n log n)

O(n^2)

O(n^2)

O(n)

Por jerarquias:
la complejidad es de
O(n^2)

Figura 2.2

Figura 2.3

Figura 2.4

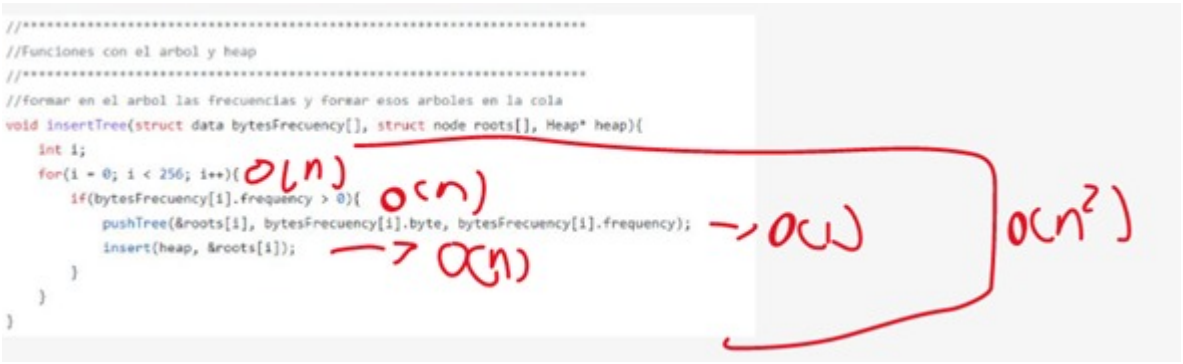


Figura 2.5

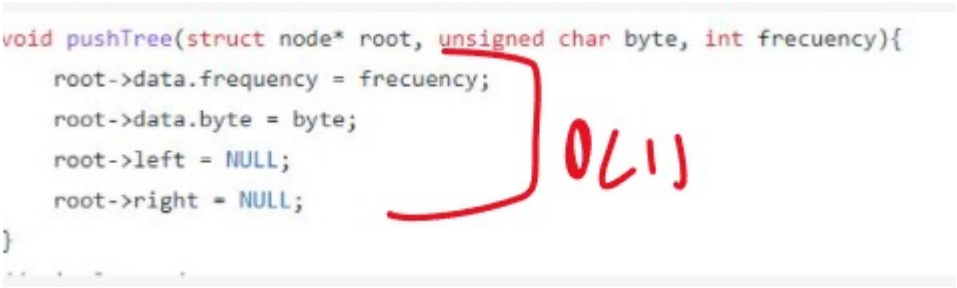


Figura 2.6

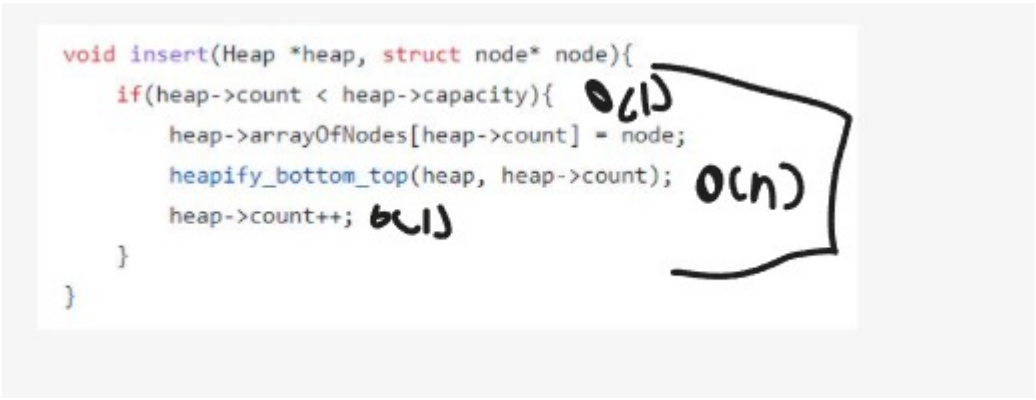


Figura 2.7

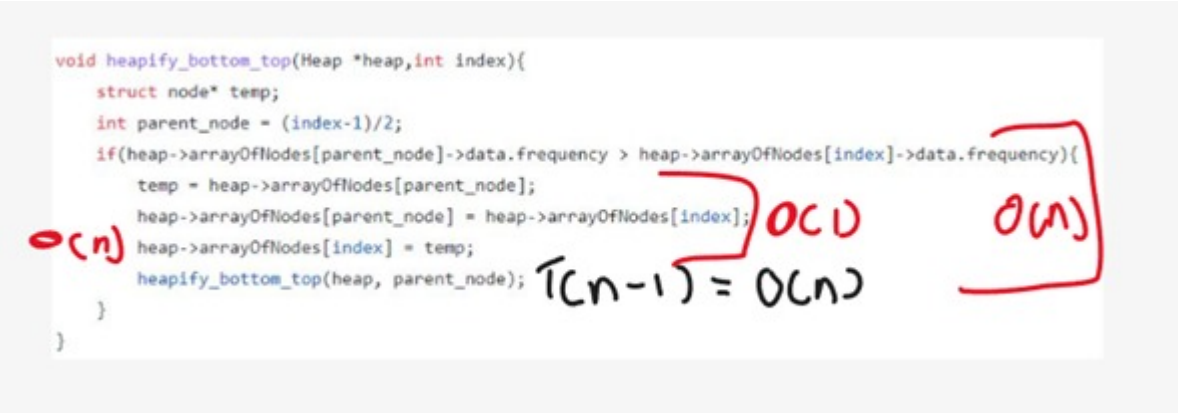


Figura 2.8

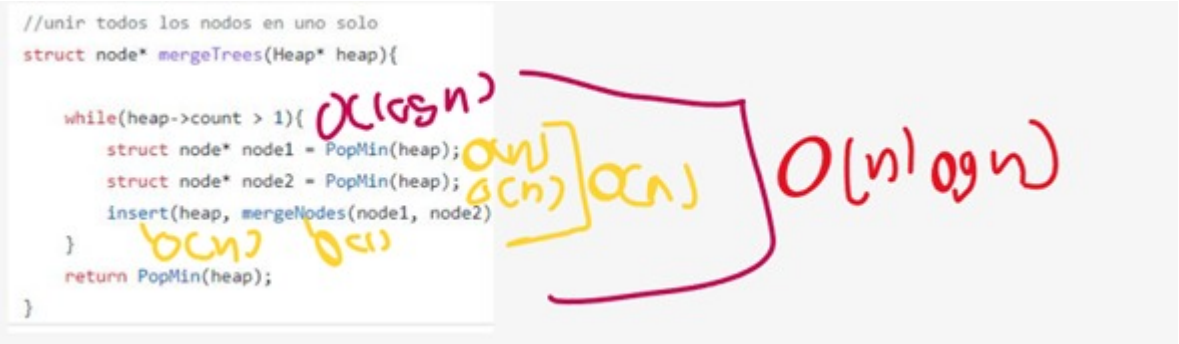


Figura 2.9

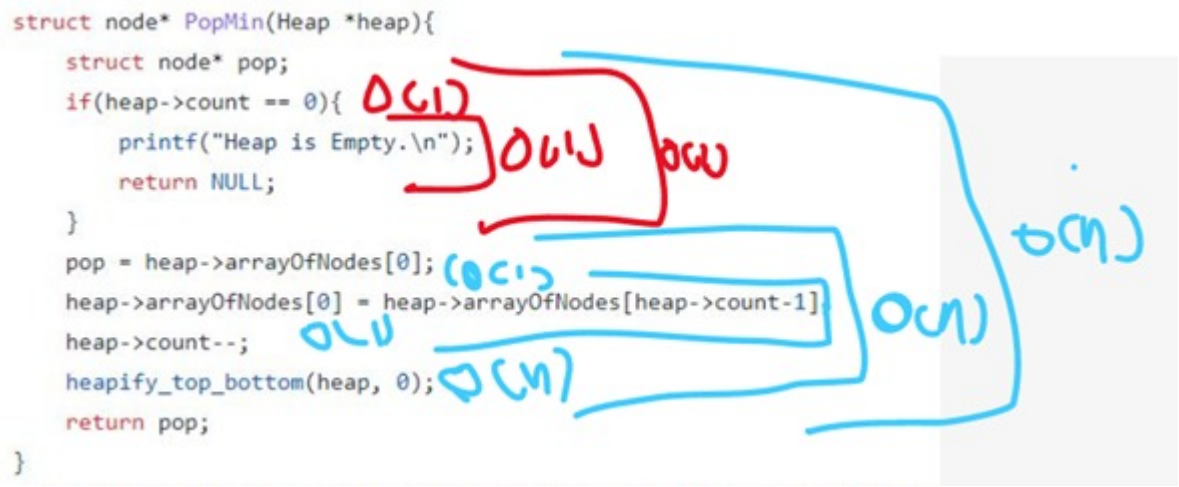


Figura 2.10

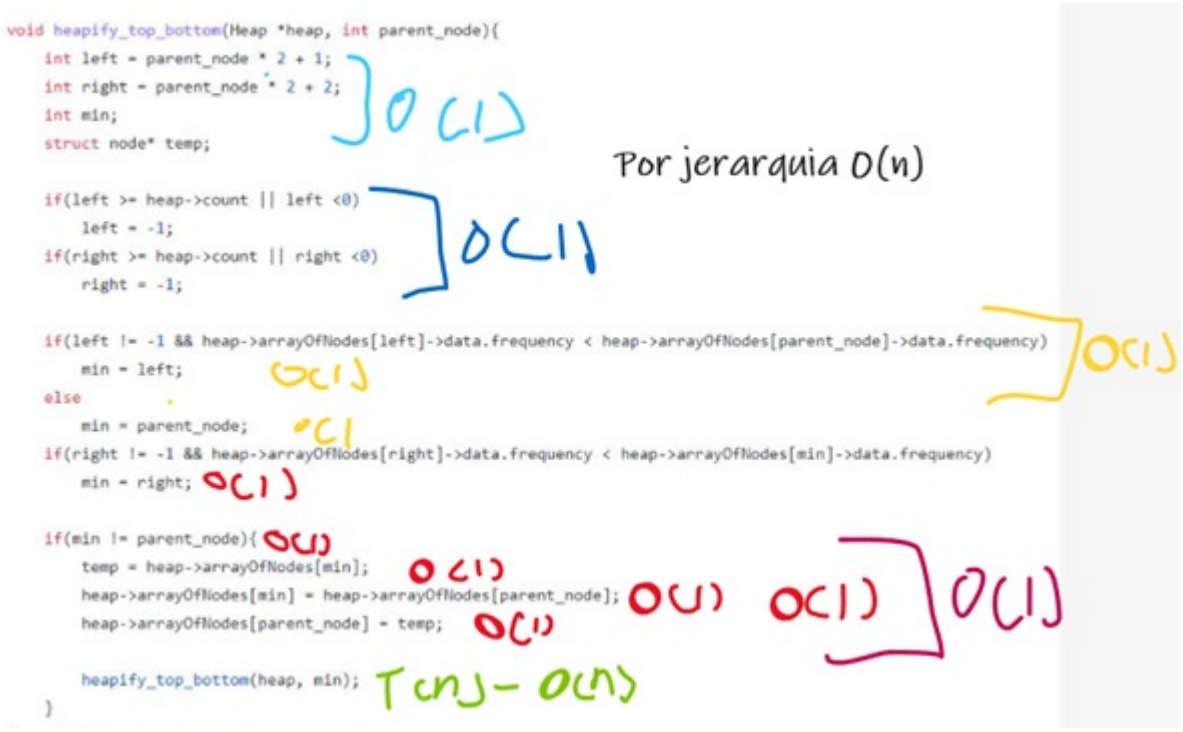


Figura 2.11

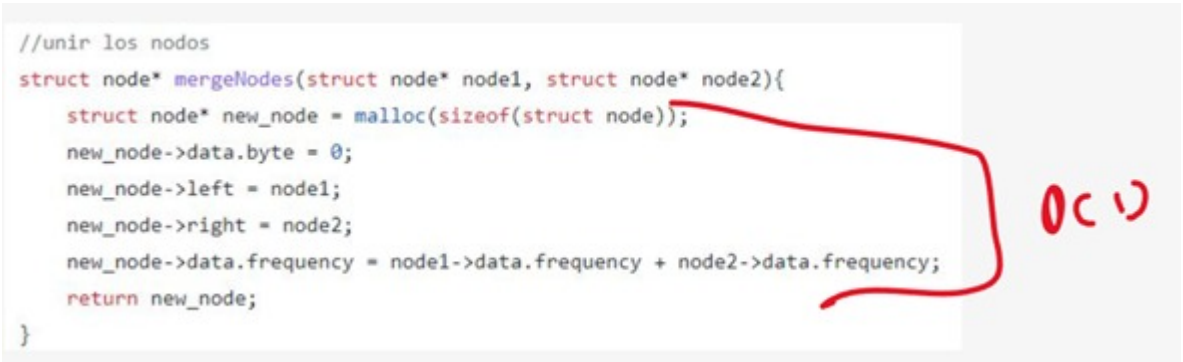


Figura 2.12

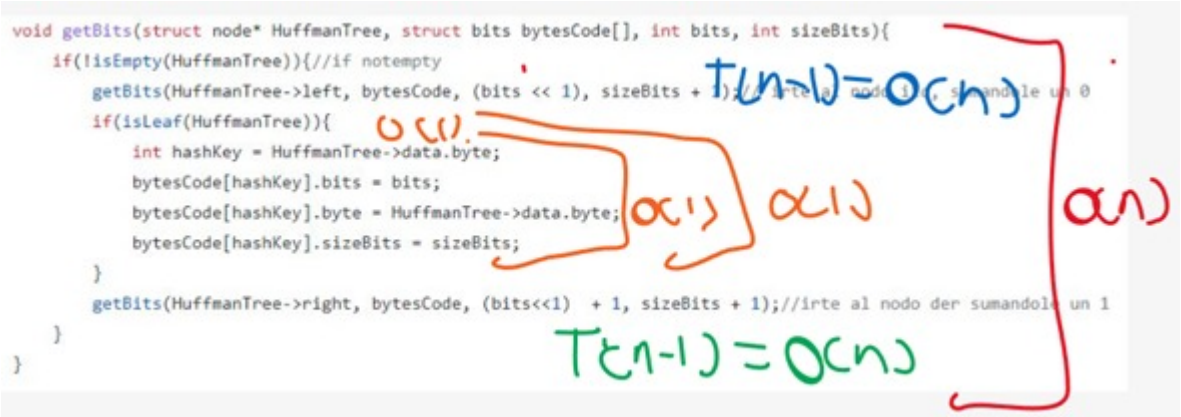


Figura 2.13

Figura 2.14

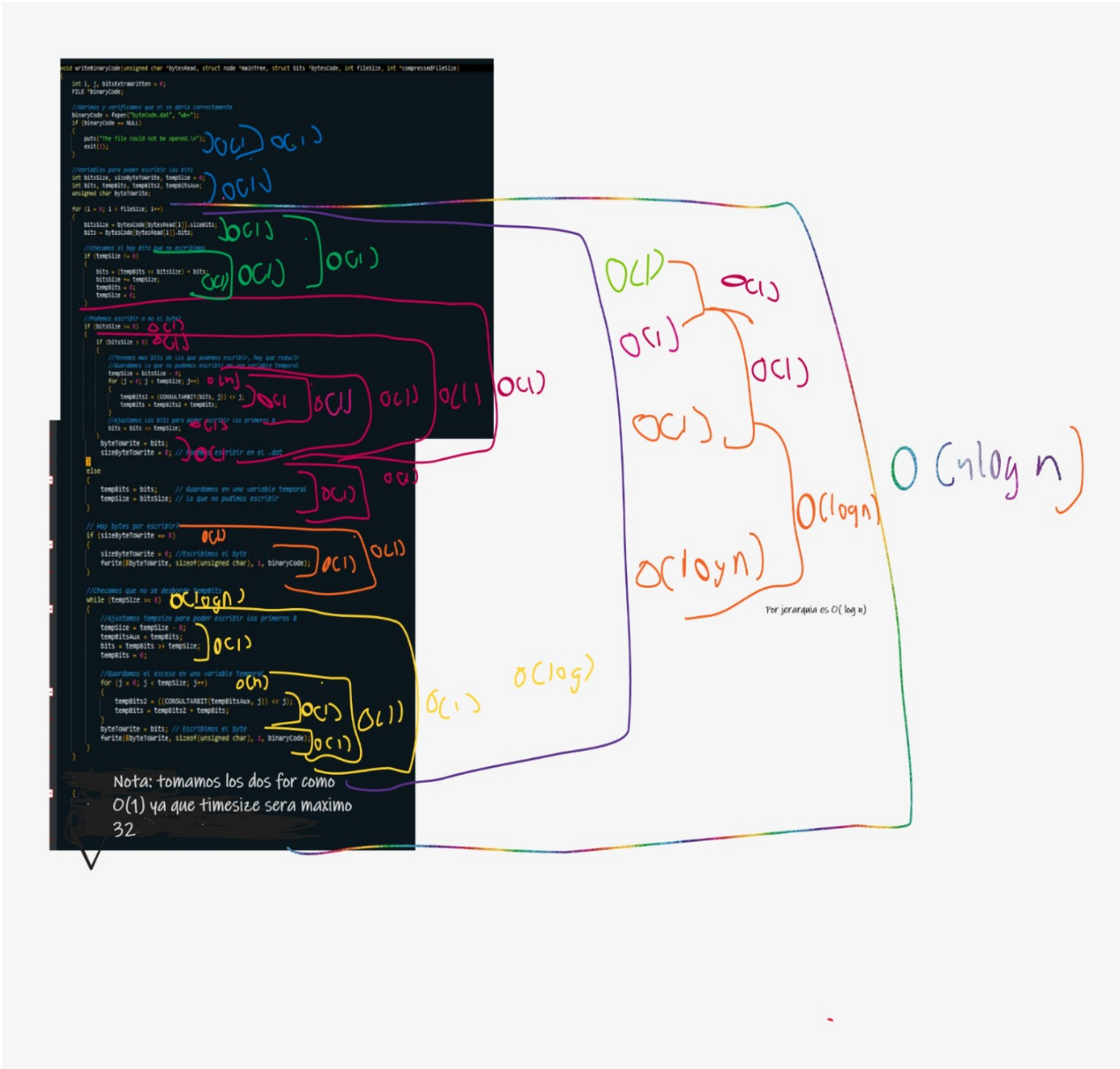

O(n log n)
Por jerarquia es O(log n)
Nota: tomamos los dos for como
O(1) ya que timesize sera maximo
32

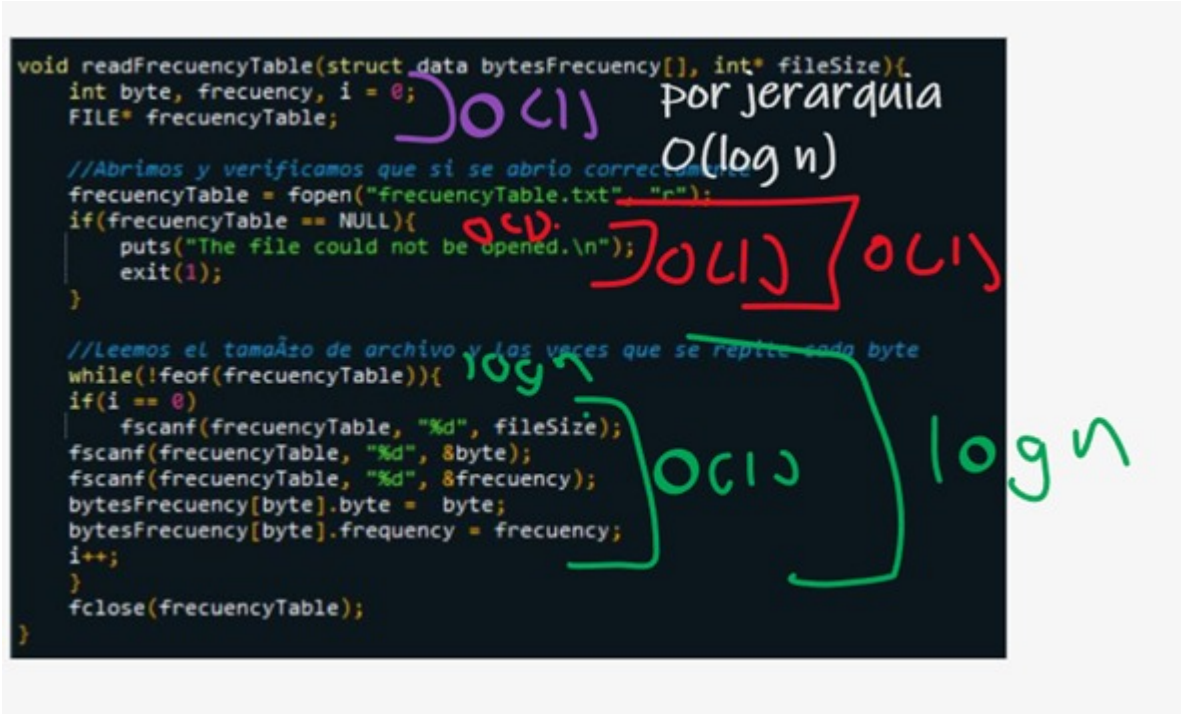
Figura 2.15

2.3.2. Análisis de la decodificación de Huffman

Figura 2.16

```
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;
}
```

Figura 2.17



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

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

    //Leemos el tamaÃ±o 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);
}
```

Figura 2.18

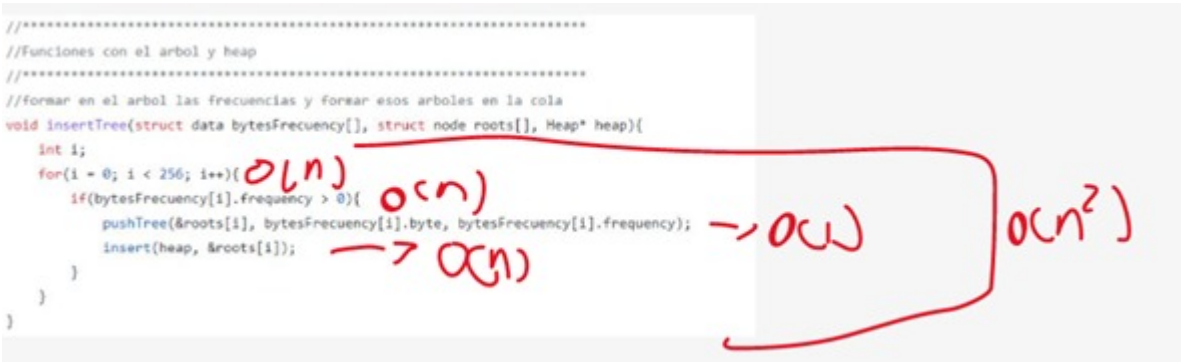


Figura 2.19

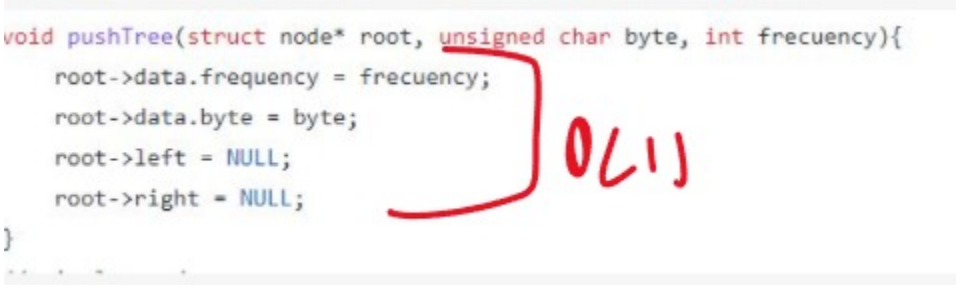


Figura 2.20

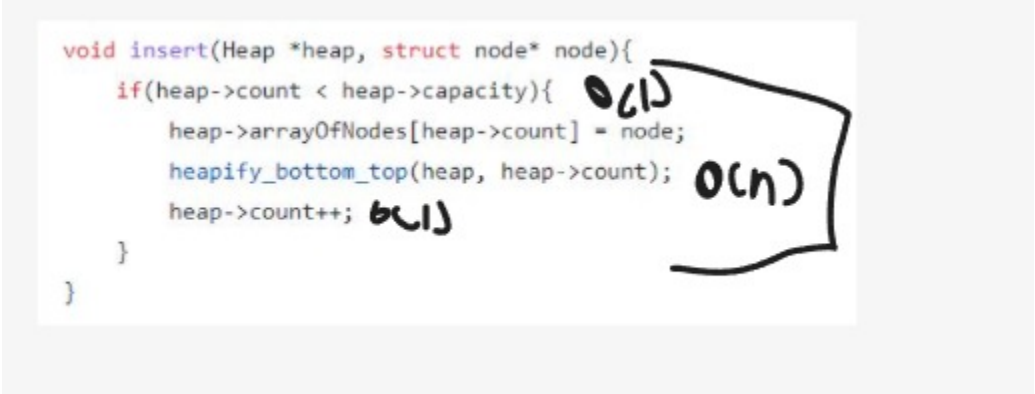


Figura 2.21

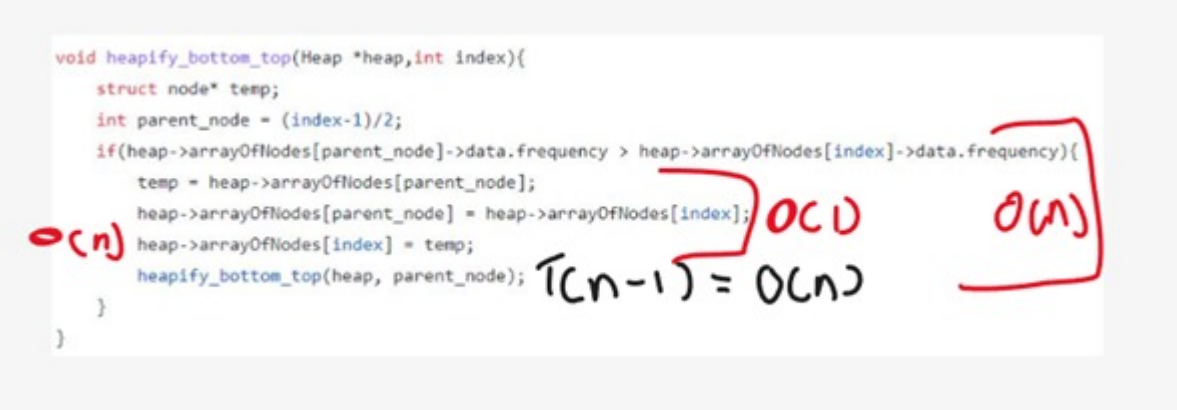


Figura 2.22

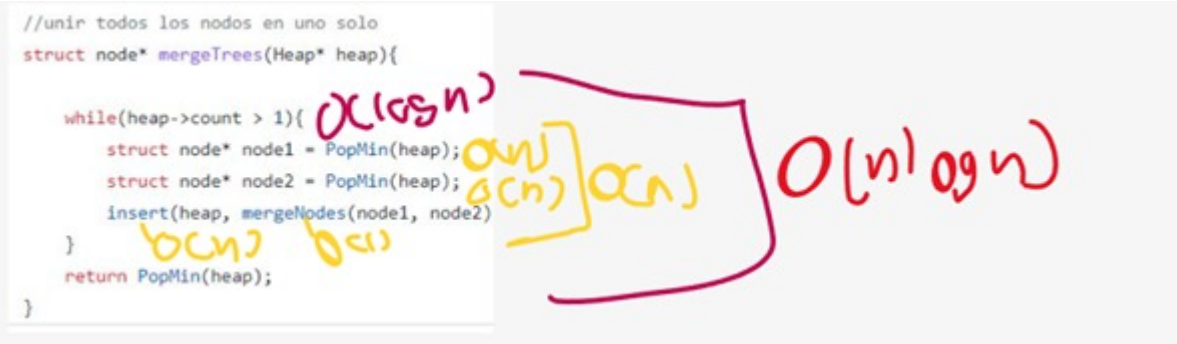


Figura 2.23

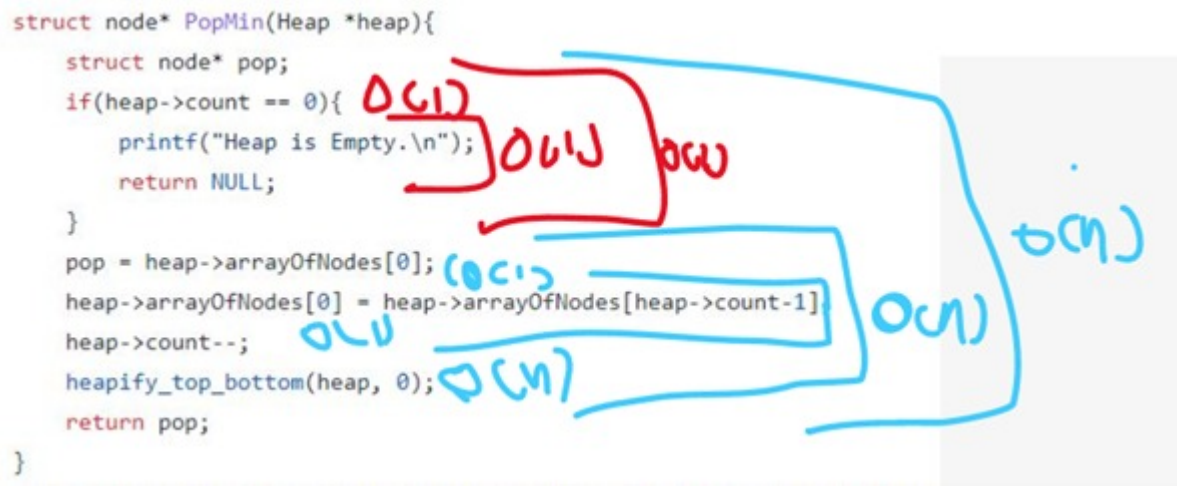


Figura 2.24

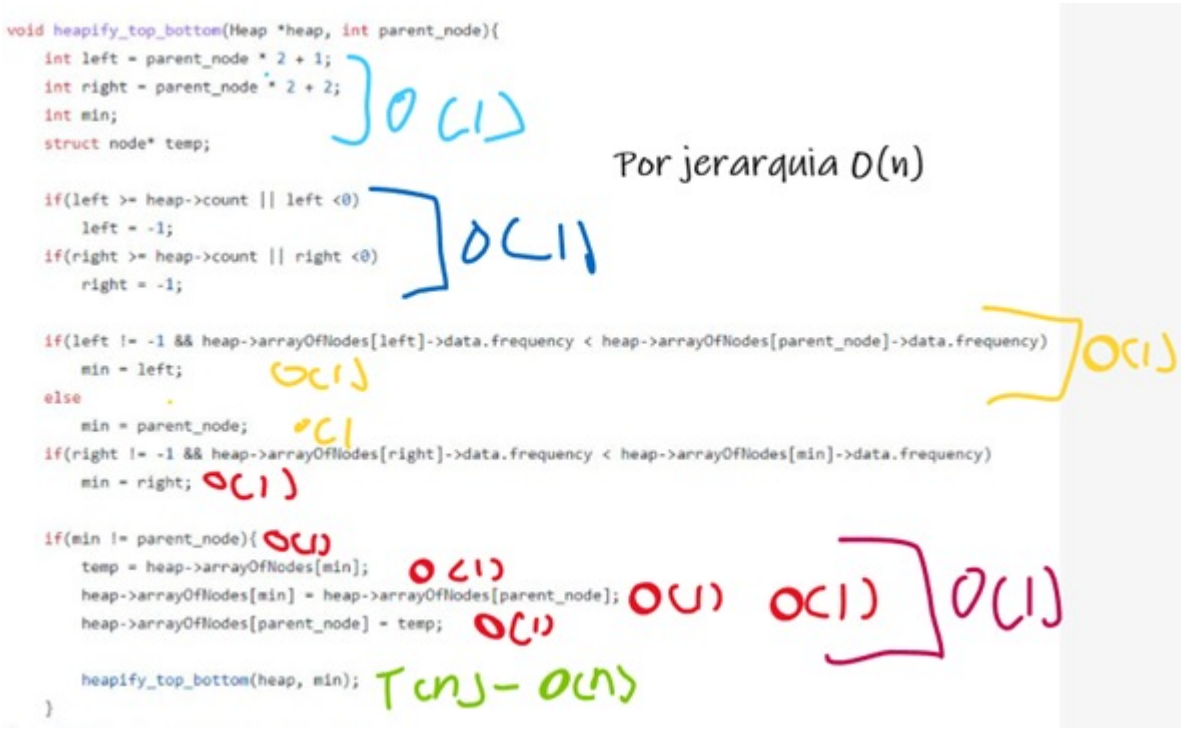


Figura 2.25

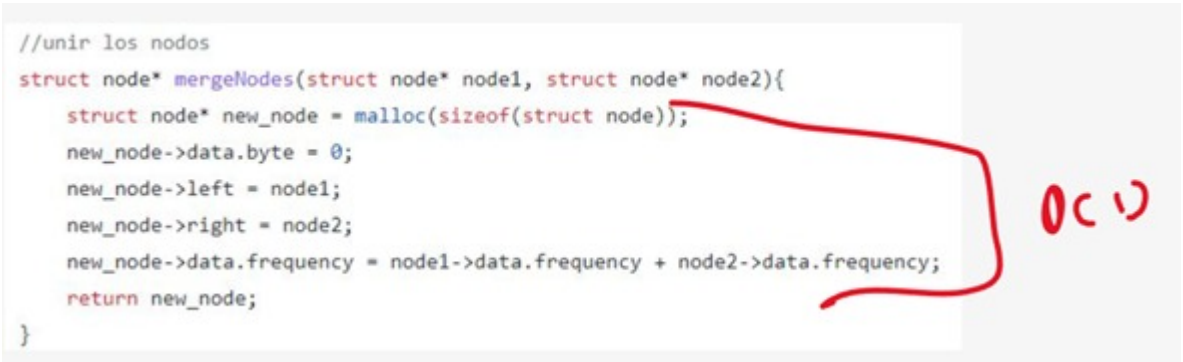


Figura 2.26

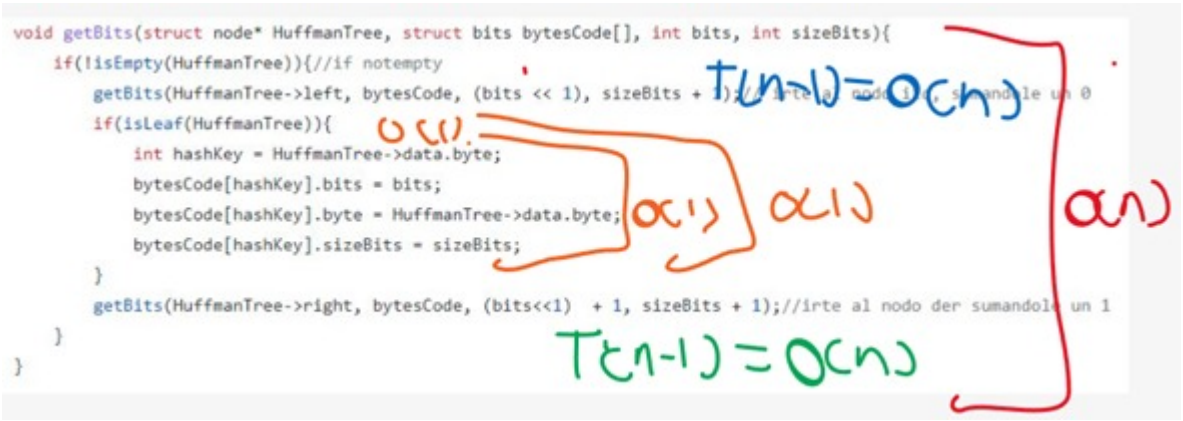


Figura 2.27

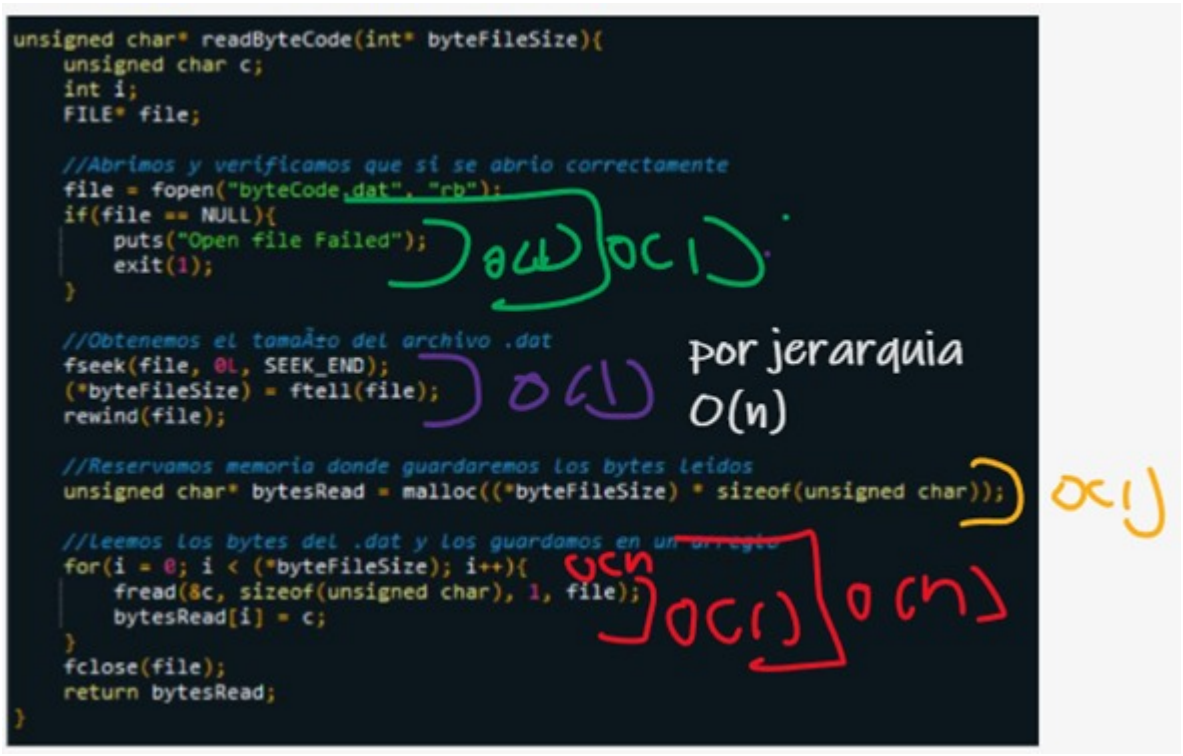


Figura 2.28

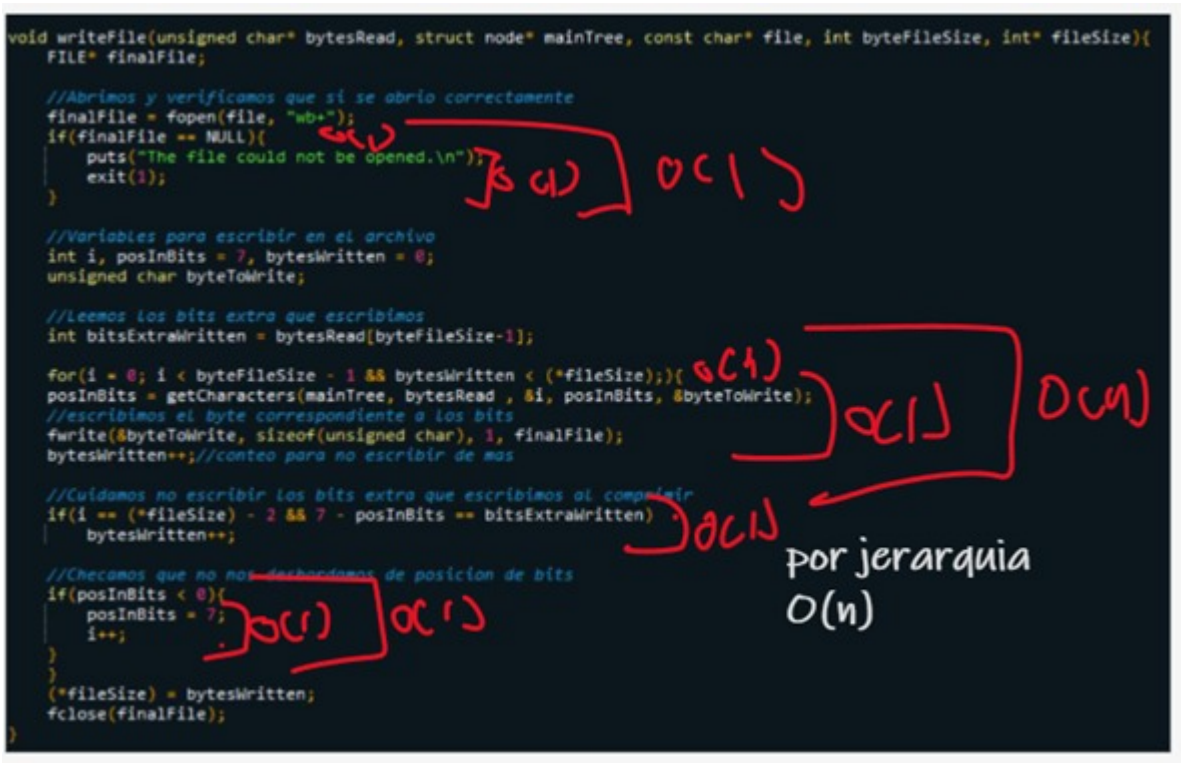


Figura 2.29

Figura 2.30

2.4. 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 %

2.5. 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	$6.8174697876 \times 10^{1}$	$1.2259612106 \times 10^{2}$

2.6. Cuestionario

Capítulo 3

Anexo