PRÁCTICA 7

Orozco Barrientos Ana Raquel

ANÁLISIS Y DISEÑO DE ALGORITMOS

ESCUELA SUPERIOR DE CÓMPUTO

3CV4

Introducción:

El objetivo de esta práctica es elaborar un programa que realice lo siguiente:

1. Cargue un archivo de texto que tenga al menos 50 palabras.
2. Genere el diccionario con las frecuencias de cada símbolo o letra del archivo.
3. Construye el árbol de Huffman para ese diccionario.
4. Genera los códigos de Huffman a partir del árbol.
5. Comprime el contenido del archivo.
6. Calcula la tasa de compresión, es decir determina el tamaño en bits del archivo original y el tamaño en bits del archivo codificado.
7. Almacena el archivo codificado en un nuevo archivo de texto llamado codificado.txt
8. Carga el archivo codificado.txt y averigua la forma de decodificarlo empleando el árbol Huffman generado en el paso 3.

Código:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_TREE_HT 256
#define SYMBOLS 256

typedef struct HuffmanNode {
    char data;
    unsigned freq;
    struct HuffmanNode *left, *right;
} HuffmanNode;

typedef struct {
    unsigned size;
    int array[SYMBOLS];
} Code;

typedef struct MinHeap {
    unsigned size;
    unsigned capacity;
    HuffmanNode** array;
} MinHeap;

HuffmanNode* newNode(char data, unsigned freq) {
    HuffmanNode* temp = (HuffmanNode*) malloc(sizeof(HuffmanNode));
    temp->left = temp->right = NULL;
    temp->data = data;
```

```
    temp->freq = freq;
    return temp;
}

MinHeap* createMinHeap(unsigned capacity) {
    MinHeap* minHeap = (MinHeap*) malloc(sizeof(MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (HuffmanNode**) malloc(minHeap->capacity *
sizeof(HuffmanNode*));
    return minHeap;
}

void swapNode(HuffmanNode** a, HuffmanNode** b) {
    HuffmanNode* t = *a;
    *a = *b;
    *b = t;
}

void minHeapify(MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    if (left < minHeap->size && minHeap->array[left]->freq < minHeap-
>array[smallest]->freq)
        smallest = left;
    if (right < minHeap->size && minHeap->array[right]->freq < minHeap-
>array[smallest]->freq)
        smallest = right;
    if (smallest != idx) {
        swapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

int isSizeOne(MinHeap* minHeap) {
    return (minHeap->size == 1);
}

HuffmanNode* extractMin(MinHeap* minHeap) {
    HuffmanNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[--minHeap->size];
    minHeapify(minHeap, 0);
    return temp;
}
```

```
void insertMinHeap(MinHeap* minHeap, HuffmanNode* node) {
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && node->freq < minHeap->array[(i - 1)/2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1)/2];
        i = (i - 1)/2;
    }
    minHeap->array[i] = node;
}

void buildMinHeap(MinHeap* minHeap) {
    int n = minHeap->size - 1;
    for (int i = (n - 1)/2; i >= 0; --i)
        minHeapify(minHeap, i);
}

int isLeaf(HuffmanNode* root) {
    return !(root->left) && !(root->right);
}

MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
    MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

HuffmanNode* buildHuffmanTree(char data[], int freq[], int size) {
    HuffmanNode *left, *right, *top;
    MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);
    while (!isSizeOne(minHeap)) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }
    return extractMin(minHeap);
}

void storeCodes(HuffmanNode* root, int arr[], int top, Code codes[]) {
```

```
    if (root->left) {
        arr[top] = 0;
        storeCodes(root->left, arr, top + 1, codes);
    }
    if (root->right) {
        arr[top] = 1;
        storeCodes(root->right, arr, top + 1, codes);
    }
    if (isLeaf(root)) {
        codes[(unsigned char)root->data].size = top;
        memcpy(codes[(unsigned char)root->data].array, arr, top *
sizeof(int));
    }
}

void printCodes(Code codes[]) {
    for (int i = 0; i < SYMBOLS; ++i) {
        if (codes[i].size) {
            printf("%c: ", i);
            for (int j = 0; j < codes[i].size; ++j)
                printf("%d", codes[i].array[j]);
            printf("\n");
        }
    }
}

void compressFile(const char* inputFile, const char* outputFile, Code
codes[], int* originalBits, int* compressedBits) {
    FILE* in = fopen(inputFile, "r");
    FILE* out = fopen(outputFile, "w");
    int ch;
    *originalBits = 0;
    *compressedBits = 0;
    while ((ch = fgetc(in)) != EOF) {
        *originalBits += 8;
        for (int i = 0; i < codes[ch].size; ++i) {
            fprintf(out, "%d", codes[ch].array[i]);
            (*compressedBits)++;
        }
    }
    fclose(in);
    fclose(out);
}

void decompressFile(HuffmanNode* root, const char* inputFile) {
```

```
    FILE* in = fopen(inputFile, "r");
    int ch;
    HuffmanNode* current = root;
    printf("Texto decodificado:\n");
    while ((ch = fgetc(in)) != EOF) {
        if (ch == '0') current = current->left;
        else if (ch == '1') current = current->right;
        if (isLeaf(current)) {
            printf("%c", current->data);
            current = root;
        }
    }
    fclose(in);
}

int main() {
    char filename[] = "entrada.txt";
    int freq[SYMBOLS] = {0};
    char text[10000];

    FILE* file = fopen(filename, "r");
    if (!file) {
        printf("No se pudo abrir el archivo.\n");
        return 1;
    }

    int index = 0;
    int ch;
    while ((ch = fgetc(file)) != EOF) {
        text[index++] = ch;
        freq[(unsigned char)ch]++;
    }
    text[index] = '\0';
    fclose(file);

    char data[SYMBOLS];
    int filteredFreq[SYMBOLS];
    int size = 0;
    for (int i = 0; i < SYMBOLS; ++i) {
        if (freq[i]) {
            data[size] = (char)i;
            filteredFreq[size] = freq[i];
            size++;
        }
    }
```

```
    HuffmanNode* root = buildHuffmanTree(data, filteredFreq, size);
    Code codes[SYMBOLS] = {0};
    int arr[MAX_TREE_HT];
    storeCodes(root, arr, 0, codes);

    printf("Códigos de Huffman:\n");
    printCodes(codes);

    int originalBits = 0, compressedBits = 0;
    compressFile("entrada.txt", "codificado.txt", codes, &originalBits,
&compressedBits);

    printf("\nTasa de compresion:\n");
    printf("Tamano original: %d bits\n", originalBits);
    printf("Tamano comprimido: %d bits\n", compressedBits);
    printf("Relación de compresion: %.2f%%\n", 100.0 * compressedBits /
originalBits);

    decompressFile(root, "codificado.txt");

    return 0;
}
```

Pruebas del código:

Resultado en la terminal:

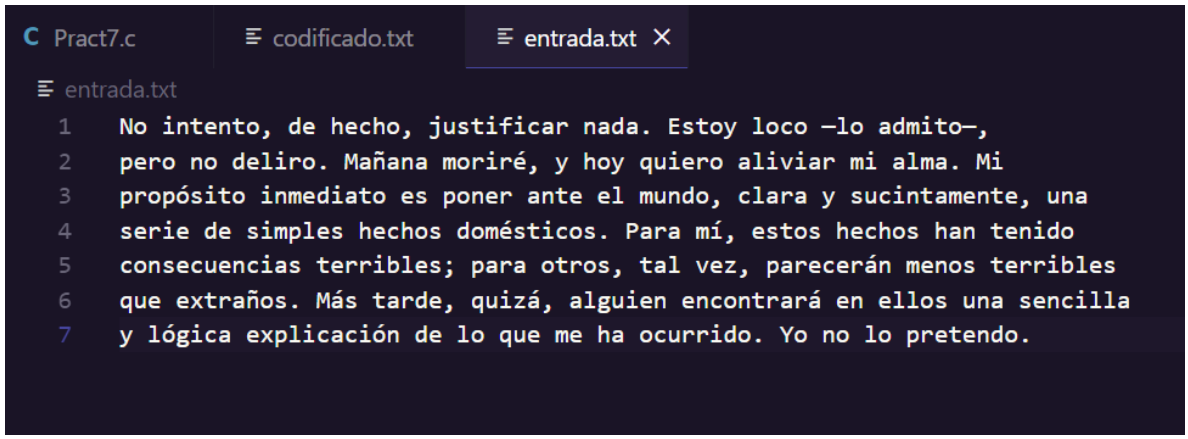
```
C:\Users\woody\OneDrive\Desktop\ADA\Codigos_prac>a
C├│digos de Huffman:
 : 110
,: 111101
.: 010011
;: 100100000
E: 011110011
M: 0100101
N: 100101110
P: 100100001
Y: 100100010
a: 1010
b: 01111000
c: 10110
d: 01110
e: 000
f: 100101111
g: 10010110
h: 100111
i: 1000
j: 100100011
l: 10111
m: 01000
n: 0101
o: 1110
p: 100110
q: 0111101
r: 0110
s: 0010
t: 11111
u: 00111
v: 11110000
x: 11110001
y: 1111001
```

La descompresión del archivo se muestra en la consola:

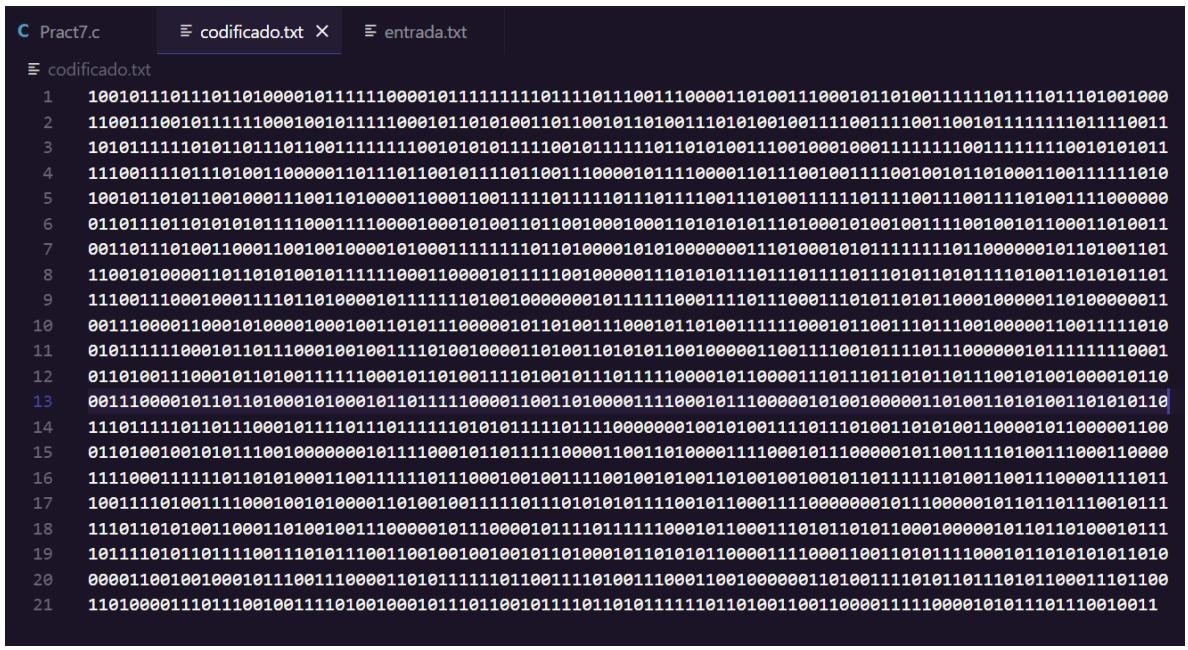
```
y: 1111001
z: 10010100
Ç: 10010101
ö: 01111100
í: 1001001
®: 01111101
¡: 011110010
▒: 01111110
│: 0100100
├: 00110
Ô: 01111111

Tasa de compresion:
Tamano original: 3936 bits
Tamano comprimido: 2204 bits
Relaci├│n de compresion: 56.00%
Texto decodificado:
No intento, de hecho, justificar nada. Estoy loco ÔÇölo admitoÔÇö, pero no deliro. Ma▒ana morir├®, y hoy quiero aliviar mi alma. Mi prop├│sito inmediato es poner ante el mundo,
clara y sucintamente, una serie de simples hechos dom├®sticos. Para m├¡, estos hechos han tenido consecuencias terribles; para otros, tal vez, parecer├ín menos terribles que extr
a▒os. M├ís tarde, quiz├í, alguien encontrar├í en ellos una sencilla y l├│gica explicaci├│n de lo que me ha ocurrido. Yo no lo pretendo.
C:\Users\woody\OneDrive\Desktop\ADA\Codigos_prac>
```

Contenido del archivo de texto de entrada:



Contenido del archivo de texto codificado, después de la ejecución del código:



Enlace Github:

- ★ **https://github.com/Annie-707/Analisis-y-Diseno-de-Algoritmos**