Instituto Politécnico Nacional

Escuela Superior de Cómputo

Estructuras de Datos

Práctica no. 6:

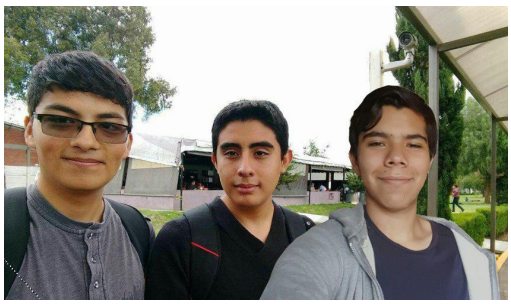
Codificación de Huffman

Profesor: Edgardo Adrián Franco Martínez

Integrantes:

- Calva Hernández José Manuel
- González Núñez Daniel Adrián
- Ruíz López Luis Carlos

Grupo: 1CM7



Introducción

El árbol es una estructura de datos fundamental en informática, muy utilizada en todos sus campos, porque se adapta a la representación natural de informaciones homogéneas organizadas y de una gran comodidad y rapidez de manipulación.

Esta estructura se encuentra en todos los dominios (campos) de la informática, desde la pura algorítmica (métodos de clasificación y búsqueda...) a la compilación (árboles sintácticos para representar las expresiones o producciones posibles de un lenguaje) o incluso los dominios de la inteligencia artificial (árboles de juegos, árboles de decisiones, de resolución, etc.).

Las estructuras tipo árbol se usan principalmente para representar datos con una relación jerárquica entre sus elementos, como son árboles genealógicos, tablas, etc.

Un árbol A es un conjunto finito de uno o más nodos, tales que:

1. Existe un nodo especial denominado RAIZ(v1) del árbol.
2. Los nodos restantes (v2, v3, ..., vn) se dividen en m >= 0 conjuntos disjuntos denominado A1, A2, ..., Am, cada uno de los cuales es, a su vez, un árbol. Estos árboles se llaman subárboles del RAIZ.

La definición de árbol implica una estructura recursiva. Esto es, la definición del árbol se refiere a otros árboles. Un árbol con ningún nodo es un árbol nulo; no tiene raíz.

Existe un tipo de árbol denominado árbol binario que puede ser implementado fácilmente en una computadora.

Un árbol binario es un conjunto finito de cero o más nodos, tales que:

- Existe un nodo denominado raíz del árbol.
- Cada nodo puede tener 0, 1 o 2 subárboles, conocidos como subárbol izquierdo y subárbol derecho. [1]

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

Este algoritmo toma un alfabeto de n símbolos, junto con sus frecuencias de aparición asociadas, y produce un código de Huffman para ese alfabeto y esas frecuencias.

Huffman propuso un algoritmo que obtiene una codificación prefijo óptima.

Para ello construye un árbol binario de códigos de longitud variable de manera

ascendente.

El algoritmo funciona de la siguiente manera:

- Parte de una secuencia inicial en la que los caracteres a codificar están colocados en orden creciente de frecuencia.
- Esta secuencia inicial se va transformando, a base de fusiones, hasta llegar a una secuencia con un único elemento que es el árbol de codificación óptimo.[4]

Planteamiento del problema

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

- Implementar codificación de Huffman
- Implementar el algoritmo de decodificación

Pasos del código de Huffman

1. Se crean varios árboles, uno por cada uno de los símbolos del alfabeto, consistiendo cada uno de los árboles en un nodo sin hijos, y etiquetado cada uno con su símbolo asociado y su frecuencia de aparición.
2. Se toman los dos árboles de menor frecuencia, y se unen creando un nuevo árbol. La etiqueta de la raíz será la suma de las frecuencias de las raíces de los dos árboles que se unen, y cada uno de estos árboles será un hijo del nuevo árbol. También se etiquetan las dos ramas del nuevo árbol: con un 0 la de la izquierda, y con un 1 la de la derecha.
3. Se repite el paso 2 hasta que sólo quede un árbol.

Una vez terminado el árbol se puede ver con facilidad cuál es el código de los símbolos

Decodificación del código de Huffman

1. A partir del árbol de codificación, comenzar a recorrer los caminos según los bits de la codificación. Al llegar a un nodo hoja se toma el valor de esta y coloca en el archivo original.
2. Se repite el paso 1 a partir del bit siguiente de la codificación comenzando un nuevo recorrido a partir de la raíz del árbol de la codificación.
3. La decodificación termina una vez se hayan recorrido todos los bits de la codificación.

Algoritmos

In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute c:freq giving its frequency.

The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |C| leaves and performs a sequence of |C| 1 “merging” operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the freq attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

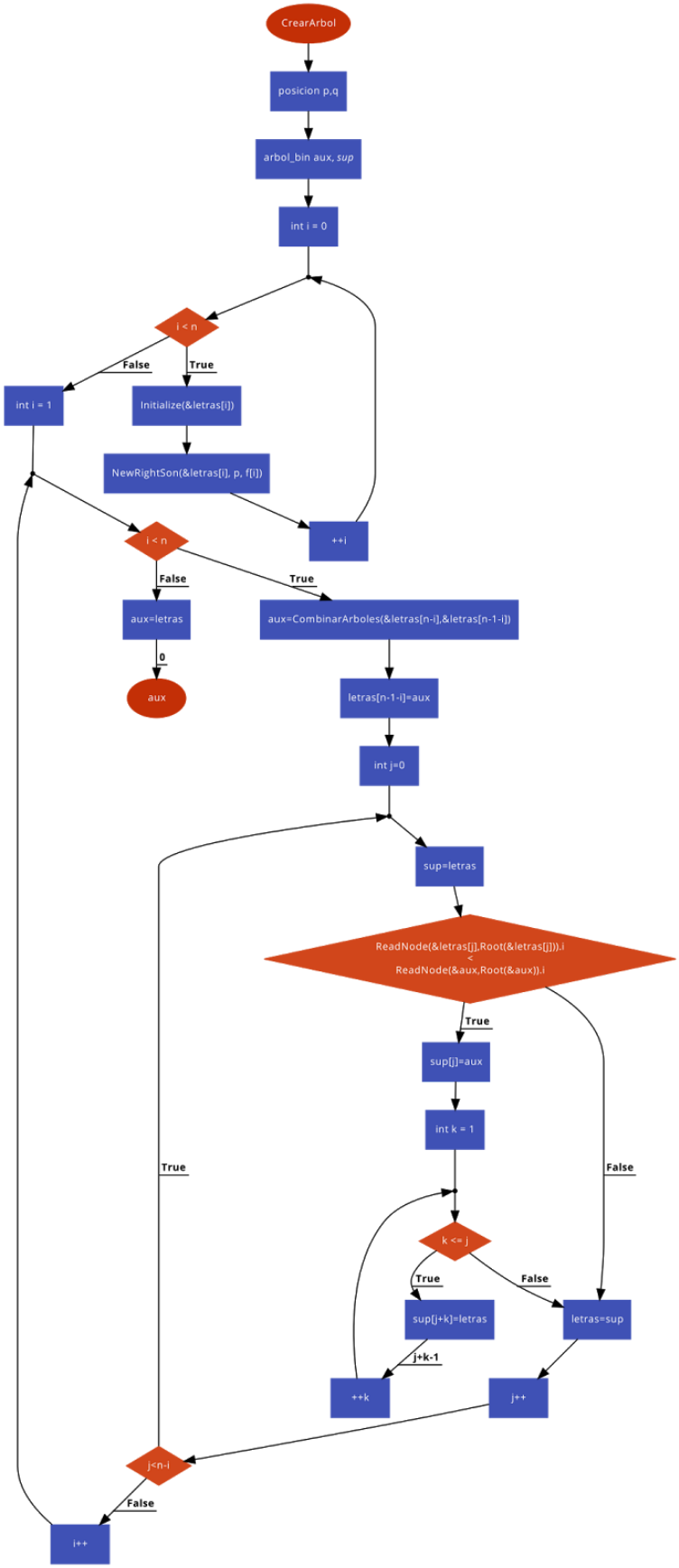
```
HUFFMAN.C
   n←|C|
   Q←|C|
   for i ← 1 to n  1
      allocate a new node z
      z.left ← x ← EXTRACT-MIN(Q)
      z.right ← y ← EXTRACT-MIN(Q)
      z.freq ← x:freq + y:freq
      INSERT(Q,z)
   return EXTRACT-MIN(Q) // return the root of the tree
```

Implementación de los algoritmos

Para la implementación lo primero que se tomó en cuenta es que ya obtuvimos una tabla reducida de frecuencias, es decir, sólo contiene las letras usadas con el número de veces que aparece en el texto, para ello se utilizó un lector de archivo con un arreglo de elementos que contienen la letra y un entero que será su número de frecuencias.

Una vez obtenida esta tabla, procedemos a la parte esencial del programa, el proceso utilizado se basó en una función que nos va juntando dos árboles y suma su frecuencia.

Dado que no vimos colas de prioridad en clase, salvo una muy breve introducción que se quedó a medias, decidimos no usarlas y utilizar algo más simple para poder elegir la posición del árbol en la cola.


CrearArbol
posicion p,q
arbol_bin aux, sup
int i = 0
i < n
False
True
int i = 1
Initialize(&letras[i])
NewRightSon(&letras[i], p, f[i])
++i
i < n
False
True
aux=letras
aux=CombinarArboles(&letras[n-i],&letras[n-1-i])
0
aux
letras[n-1-i]=aux
int j=0
sup=letras
ReadNode(&letras[j],Root(&letras[j])).i
<
ReadNode(&aux,Root(&aux)).i
True
False
sup[j]=aux
int k = 1
k <= j
True
False
sup[j+k]=letras
letras=sup
j+k-1
++k
j++
True
False
j<n-i
False
i++

Funcionamiento (Verificación de la solución, pruebas y resultados de salida *Pantallazos)

```
Archivo que se codificara(con .txt sin espacios):
Leer.txt
Archivo donde se codificara(con .txt sin espacios):
Escribir.txt
ERROR: valorbinario
```

```
Introduce el texto a codificar: Hola mundo

De   hay 1 elementos
De H hay 1 elementos
De a hay 1 elementos
De d hay 1 elementos
De l hay 1 elementos
De m hay 1 elementos
De n hay 1 elementos
De o hay 1 elementos
De u hay 1 elementos
```

```
Introduce el texto a codificar: Hasta luegoi dddd

De   hay 2 elementos
De H hay 1 elementos
De a hay 2 elementos
De d hay 4 elementos
De e hay 1 elementos
De g hay 1 elementos
De i hay 1 elementos
De l hay 1 elementos
De o hay 1 elementos
De s hay 1 elementos
De t hay 1 elementos
De u hay 1 elementos

De H hay 1 elementos
De e hay 1 elementos
De g hay 1 elementos
De i hay 1 elementos
De l hay 1 elementos
De o hay 1 elementos
De s hay 1 elementos
De t hay 1 elementos
De u hay 1 elementos
De   hay 2 elementos
De a hay 2 elementos
De d hay 4 elementos

La frecuencia total es: 17
```

Errores detectados (Si existe algún error detectado, el cuál no fue posible resolver o se desconoce el motivo y solo ocurre con ciertas condiciones es necesario describirlo)

Al momento de comenzar la codificación a nivel de bits, el programa nos genera un error que no supimos descifrar a pesar de que intentamos varios métodos, adicional a ello, la escritura del archivo también nos generaba un error que no supimos contrarrestar.

Posibles mejoras (Describir posibles disminuciones de código en la implementación o otras posibles soluciones)

Concluir la práctica y a partir de ahí, una posible ilustración de cómo se juntan los árboles podría ser una buena idea.

Conclusiones (Por cada integrante del equipo)

Ruiz Lopez Luis Carlos: Esta práctica en especial es muy interesante ya que nos muestra la utilidad de los árboles binarios en el problema de Huffman, esta muy completo y muestra la utilidad de los árboles binarios.

González Núñez Daniel: Sin lugar a duda esta fue la practica mas dificil de todas, ya que combino al TAD más complejo que vimos (TAD Árbol binario) con un mecanismo que usa los bytes de los datos procesados. Esta práctica me interesó y empecé a investigar un poco más a fondo, al final descubrí que el código de Huffman tiene una gran aplicación al momento de compactar archivos de manera eficiente. Es una lastima que no hayamos podido completarla hasta la parte del prendido y apagado de bits.

Calva Hernández José Manuel: De lejos fue la práctica que más nos hizo sufrir, a pesar de que intentamos de varias maneras concluirla satisfactoriamente, desgraciadamente no pudimos hacerlo. Sin embargo, hemos aprendido mucho con esta práctica ya que fue en la que más tuvimos que jugar con el TAD base, además de otras definiciones, para así poder sacar más rendimiento a la hora de implementarlo.

Anexo (Códigos fuente *con colores e instrucciones de compilación)

Guardar el código en .c, junto con archivos del TADArbolBin, y dos archivos txt, uno para leer y otro donde se codificará, en una misma carpeta

Desde *Símbolo del Sistema* ingresar a la ruta donde se guardaron los archivos usando *cd 'ruta de la carpeta'*

Escribir *gcc 'nombre del archivo con el código'.c* y presionar enter para realizar la compilación

Escribir *a* (ejecutable compilado) y presionar enter para ejecutar programa

El programa no es completo nuestro, nuestra parte del código llega hasta el codificador donde se comienza a cifrar en bits, a partir de ahí es en parte de otro equipo dado que nuestra solución no terminaba de funcionar. Decidimos anexarlo para que pruebe el funcionamiento con nuestro proceso de armado del árbol.

I.- Codificador

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "TADArbolBin.c"

#define PESOBIT(bpos) 1<<bpos
#define CONSULTARBIT(var,bpos) (*(unsigned*)&var & PESOBIT(bpos))?1:0
#define PONE_1(var,bpos) *(unsigned*)&var |= PESOBIT(bpos)
#define PONE_0(var,bpos) *(unsigned*)&var &= ~(PESOBIT(bpos))
#define CAMBIA(var,bpos) *(unsigned*)&var ^= PESOBIT(bpos)

elemento* ReducirFrecuencia (elemento* f, int n, int m){
	elemento* g;
	elemento aux;
	g=(elemento*)malloc(n*sizeof(elemento));
```

```
	for (int i = 0; i < 53; ++i)
	{
		if ((f[i].i)!=0)
		{
			g[m]=f[i];
			m++;
		}
	}
	for (int i = 1; i < n; ++i)
	{
		for (int j = n-1; j >= 0; --j)
		{
			if(g[j].i>g[j-1].i)
			{
				aux=g[j];
				g[j]=g[j-1];
				g[j-1]=aux;
			}
		}
	}
	free(f);
	return g;
}

arbol_bin CrearArbol(arbol_bin *letras,elemento *f, int n){
	posicion p,q;
	arbol_bin aux, *sup;
	for (int i = 0; i < n; ++i)
	{
		Initialize(&letras[i]);
```

```
                NewRightSon(&letras[i], p, f[i]);
        }
        for (int i = 1; i < n; i++)//reliza el proceso el numero de letras iniciales menos 1
        {
                aux=CombinarArboles(&letras[n-i],&letras[n-1-i]);
                letras[n-1-i]=aux;
                int j=0;
                do{
                        sup=letras;
                        if (ReadNode(&letras[j],Root(&letras[j])).i<ReadNode(&aux,Root(&aux)).i)
                        {
                                sup[j]=aux;
                                for (int k = 1; k <= j; ++k)
                                {
                                        sup[j+k]=letras[j+k-1];
                                }
                        }
                        letras=sup;
                        j++;
                }while(j<n-i);
        }
        aux=letras[0];
        return aux;
}

void escribir_binario (elemento letra, arbol_bin *arbol, int *binario)
{
        int i,j;
        posicion p;
        for (i = 0; i < 53; i++)
```

```
	{
		binario[i]=-1;
	}
		letra.usado=1;
		p = Search(arbol, letra);
	i=0;
	while(p!=Root(arbol))
	{
		binario[i]=Binario (arbol, p);
		p=Parent(arbol, p);
		i++;
	}
}


int main (void)
{
	int n=0,j,binario[53];
	FILE* archivo,*cifrado;
	char c,arch[50],cifr[50];
	arbol_bin cod;
	arbol_bin* letras;
	elemento* f;
	unsigned char resultado;
	f=(elemento*)malloc(53*sizeof(elemento));
	for (int i = 0; i < 53; i++)
		(f[i].i)=0;
	printf("Archivo que se codificara(con .txt sin espacios): \n");
	scanf("%s",&arch);
	printf("Archivo donde se codificara(con .txt sin espacios): \n");
```

```
scanf("%s",&cifr);
archivo = fopen(arch,"r");
cifrado = fopen(cifr,"w");
do
{
	c=fgetc(archivo);
	if (c>=65&&c<=90)
	{
		(f[c-65].i)++;
		(f[c-65].c)=c;
	}
	else if (c>=97&&c<=122)
	{
		(f[c-97+26].i)++;
		(f[c-97+26].c)=c;
	}
	else if (c==32)
	{
		(f[52].i)++;
		(f[52].c)=c;
	}
}
while(c!=EOF);
fclose(archivo);
for (int i = 0; i < 53; ++i){
	if(f[i].i!=0)
		n++;
}
f=ReducirFrecuencia(f,n,0);
letras=(arbol_bin*)malloc(n*sizeof(arbol_bin));
```

```
cod=CrearArbol(letras,f,n);
archivo = fopen(arch,"r");
c=fgetc(archivo);
while(c!=EOF)
{
	for(int i=0;i<53;i++){
		if(f[i].c==c){
			escribir_binario ((f[i]),&cod, binario);
		}
	}
	for (int k = 52; k >=0; k--)
	{
		if (binario[k]!=-1)
		{
			if (binario[k]==1)//
			{
				PONE_1(resultado,j);
			}
			else if (binario[k]==0)
			{
				PONE_0(resultado,j);
			}
			j++;
			if (j==8)
			{
				printf("%c",resultado);
				fputc(resultado,cifrado);
				j=0;
			}
		}
```

```
                }
        c=fgetc(archivo);
        }
        if (j>0)
        {
                printf("%c",resultado);
                fputc(resultado,cifrado);
                fputs("\n",cifrado);
                fprintf(cifrado, "%d",8-j);
        }
        fputs("\n",cifrado);
        for (int i = 0; i < 53; i++)
        {
                fprintf(cifrado, "%d\n",f[i].i);
        }
        printf("\n");
        fclose(archivo);
        fclose(cifrado);
        free(f);
        free(letras);
        return 0;
}
```

II.- Descodificador

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "TADArbolBin.h"
```

```
#define PESOBIT(bpos) 1<<bpos
#define CONSULTARBIT(var,bpos) (*(unsigned*)&var & PESOBIT(bpos))?1:0
#define PONE_1(var,bpos) *(unsigned*)&var |= PESOBIT(bpos)
#define PONE_0(var,bpos) *(unsigned*)&var &= ~(PESOBIT(bpos))
#define CAMBIA(var,bpos) *(unsigned*)&var ^= PESOBIT(bpos)

arbol_bin generar_arbol (arbol_bin *letras,elemento frecuencias[])
{
        int i,j,k,l,num_letras;
        int min1,min2;
        int a[2];
        arbol_bin aux;
        elemento e;
        posicion p;
        num_letras=0;
        for(i=0;i<53;i++)
        {
                if (frecuencias[i].frec != 0)
                {
                        p=Root(&letras[num_letras]);
                        NewRightSon(&letras[num_letras], p, frecuencias[i]);
                        num_letras++;
                }
        }
        j=num_letras;
        for (k = 0; k < num_letras-1; k++)
        {
                        p=Root(&letras[j-1]);
                        e=ReadNode(&letras[j-1], p);
                        min1=e.frec;
                        min2=min1+1;
                        for (i = 0; i < j; i++)
```

```
{
        p=Root(&letras[i]);
        e=ReadNode(&letras[i], p);
        if (e.frec!=0 && e.usado!=1)
        {
                if (e.frec<min1)
                {
                        min1=e.frec;
                }
                else if (e.frec<=min2 && e.frec>min1)
                {
                        min2=e.frec;
                }
        }
}
l=0;
for (i = 0; i < j; i++)
{
        p=Root(&letras[i]);
        e=ReadNode(&letras[i], p);
        if (e.frec==min1 && e.usado!=1)
        {
                a[l]=i;
                l++;
        }
}
for (i = 0; i < j; i++)
{
        p=Root(&letras[i]);
        e=ReadNode(&letras[i], p);
        if (e.frec==min2 && l<2 && e.usado!=1)
        {
```

```
                                        a[l]=i;
                                        l++;
                                }
                        }
                        aux = unirpadre (&letras[a[0]], &letras[a[1]]);
                        letras[j]=aux;
                        j++;
        }
        return aux;

}
void leer_binario (unsigned char c, int *binario)
{
        int i;
        for (i = 0; i < 8; i++)
        {
                binario[i]=CONSULTARBIT(c,i);
        }
        return;
}

int main (void)
{
        elemento frecuencias[53];
        elemento e;
        arbol_bin final;
        arbol_bin letras[105];
        posicion p;
        int binario[8];
        int i;
        int ignorados;
        unsigned char c,caracter;
```

```
FILE *descifrado,*texto;
for (i = 0; i < 53; i++)
{
	(frecuencias[i].frec)=0;
	(frecuencias[i].caracter)='\0';
	Initialize(&letras[i]);
}
descifrado=fopen("mensaje_descifrado.txt","w");
texto=fopen("mensaje_cifrado.txt","r");
do
{
	c=fgetc(texto);
}
while(c!='\n');
fscanf(texto,"%d\n",&ignorados);
for (i = 0; i < 53; i++)
{
	fscanf(texto,"%d\n",&(frecuencias[i].frec));
	if (i>=0&&i<=25)//mayusculas
	{
		(frecuencias[i].caracter)=i+65;
	}
	else if (i>=26&&i<=51)//minisculas
	{
		(frecuencias[i].caracter)=i+97-26;
	}
	else if (i==52)
	{
		(frecuencias[i].caracter)=32;
	}
}
```

```
final = generar_arbol (letras,frecuencias);
fclose(texto);
texto=fopen("mensaje_cifrado.txt","r");
printf("Mensaje Descifrado: \n");
c=fgetc(texto);
p=Root(&final);
while(c!='\n')
{
	leer_binario (c, binario);
	c=fgetc(texto);
	if (c!='\n')
	{
		for (i = 0; i < 8; i++)
		{
			if(binario[i])
			{
				p=RightSon(&final, p);
			}
			if (!binario[i])
			{
				p=LeftSon(&final, p);
			}
			if(es_hoja (&final,p))				{
				e=ReadNode(&final, p);
				printf("%c",e.caracter);
				fputc(e.caracter,descifrado);
				p=Root(&final);
			}
		}
	}
	else
	{
```

```
                        for (i = 0; i < 8-ignorados; i++)
                        {
                                if(binario[i]==1)
                                {
                                        p=RightSon(&final, p);
                                }
                                if (binario[i]==0)
                                {
                                        p=LeftSon(&final, p);
                                }
                                if(es_hoja (&final,p))
                                {
                                        e=ReadNode(&final, p);
                                        printf("%c",e.caracter);
                                        fputc(e.caracter,descifrado);
                                        p=Root(&final);
                                }
                        }
                }
        }
        printf("\n");
        fclose(texto);
        fclose(descifrado);
        system ("PAUSE");
        return 0;
}
```

Bibliografía (En formato IEEE)

[1]L. Joyanes, *Estructura de Datos en C++*, 1st ed. España: McGraw-Hill España, 2011.

[2]S. Baase and A. Van Gelder, *Computer algorithms*, 1st ed. Delhi: Pearson Education, 2009.

[3]T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to algorithms*, 3rd ed. London, England: The MIT Press, 2009.

[4]Presentación Practica 06: Codificación de Huffman, del profesor Franco, Edgardo para la clase de Estructuras de Datos