



Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
Campus Tecnológico Central de Cartago

[IC2001] Estructuras de Datos

Profesor: Ing. Víctor Garro Abarca

Manual Técnico

Compresor/Descompresor

José Daniel Araya Ortega c.2022209303

Jocsan Adriel Pérez Coto c.2022437948

Entrega: 09/11/2022

II Semestre

Contenido

Estructuras de datos	3
Nodos	5
Relación entre estructuras	7
Algoritmos/Lógica	8
Main	15

Estructuras de datos

Árboles binarios: Un árbol binario es un conjunto finito de nodos que consta de un nodo raíz que tiene dos subárboles binarios denominados **subárbol izquierdo** y **subárbol derecho**.

El árbol binario es una estructura de datos muy útil cuando el tamaño de la estructura no se conoce, se necesita acceder a sus elementos ordenadamente.

En sí un árbol binario es una colección de objetos, cada uno de los cuales contiene datos o una referencia a su subárbol derecho.

Un ejemplo de cómo se ve un árbol binario:

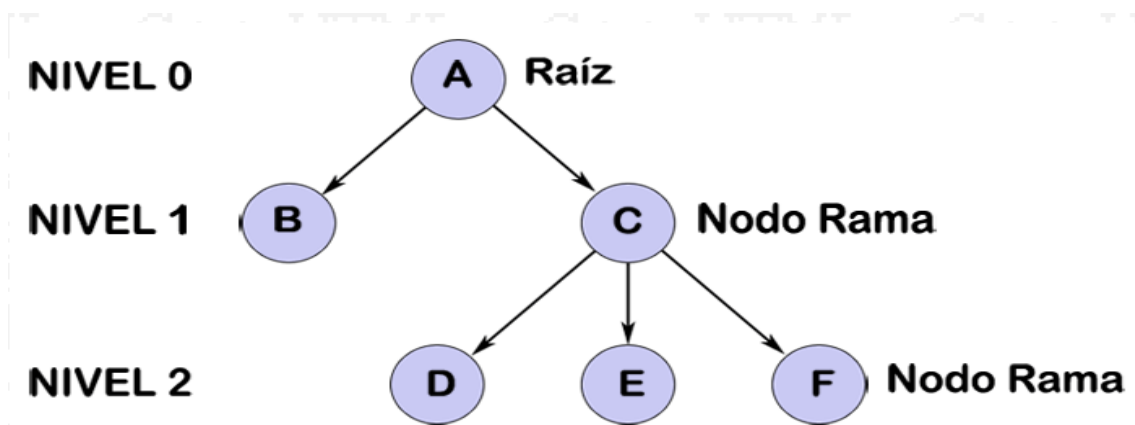


Fig 1. Árbol binario. Tomado de: http://aniei.org.mx/paginas/uam/CursoPoo/curso_poo_12.html

Algoritmo de compresión de codificación de Huffman

La codificación Huffman es un algoritmo para realizar la compresión de datos y forma la idea básica detrás de la compresión de archivos. Esta publicación habla sobre la codificación de longitud fija y de longitud variable, los códigos decodificables únicos, las reglas de prefijo y la construcción del árbol de Huffman.

Codificación de Huffman

La técnica funciona creando un árbol binario de nodos. Un nodo puede ser un nodo hoja o un nodo interno. Inicialmente, todos los nodos son nodos hoja, que contienen el carácter en sí, el peso (frecuencia de aparición) del carácter. Los nodos internos contienen peso de carácter y enlaces a dos nodos secundarios. Como convención común, un poco 0 representa seguir al hijo izquierdo, y un poco 1 representa seguir al hijo correcto. Un árbol terminado tiene n nodos de hojas y $n-1$ nodos internos. Se recomienda que Huffman Tree descarte los caracteres no utilizados en el texto para producir las longitudes de código óptimas.

Un ejemplo del árbol de Huffman:

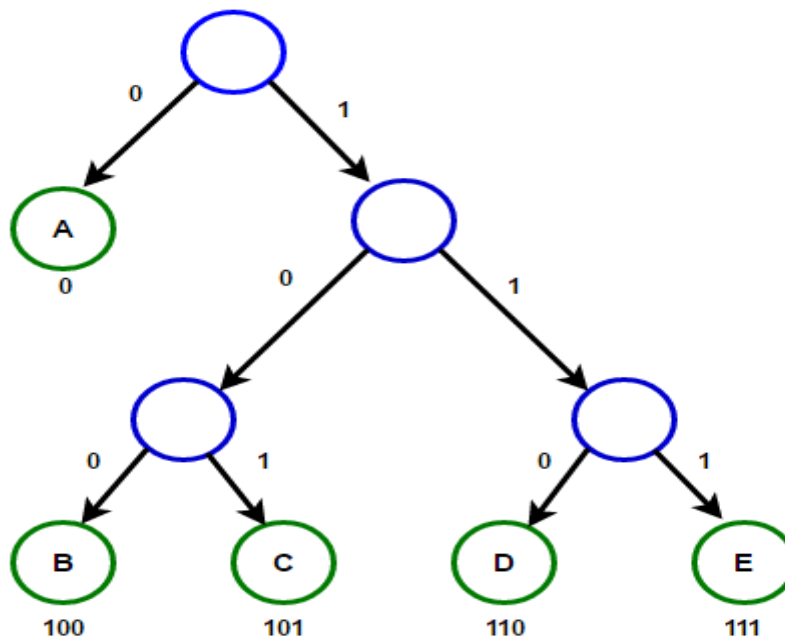
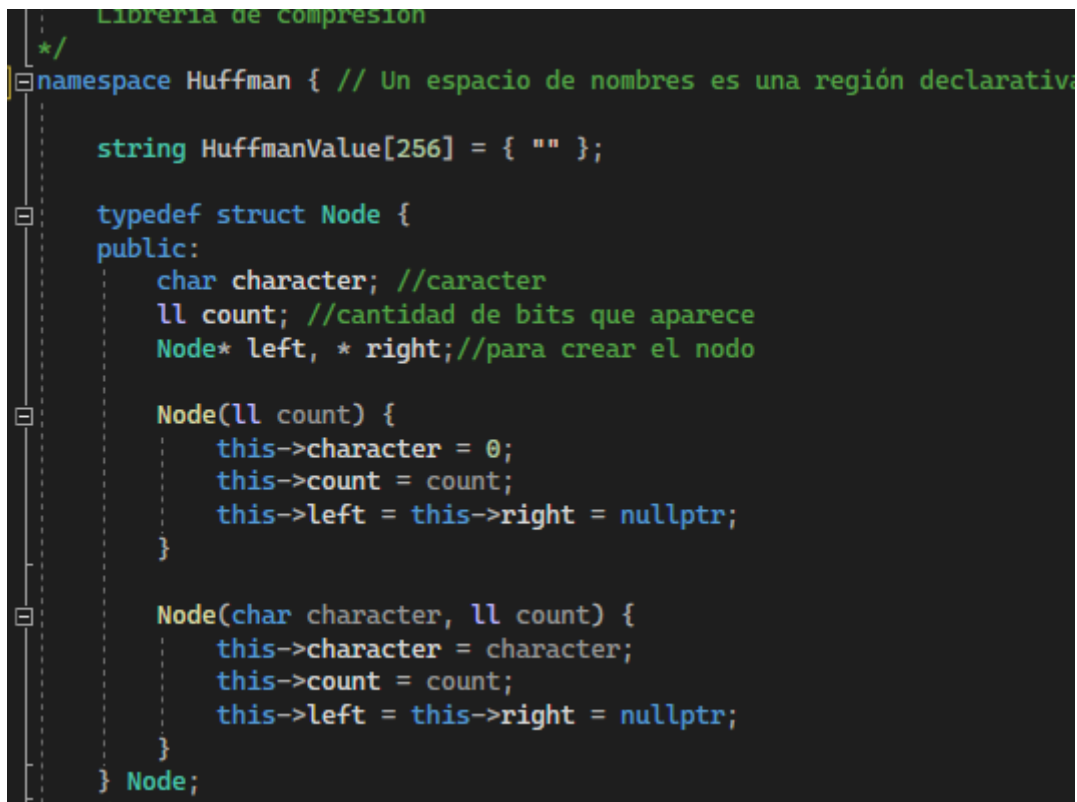


Fig 2 Árbol de Huffman. Tomado de: <https://www.techiedelight.com/es/huffman-coding/>

namespace

namespace en C++ es la parte declarativa donde se declara el alcance de los identificadores como las funciones, el nombre de los tipos, las clases, las variables, etc. El código generalmente tiene múltiples bibliotecas, y el espacio de nombres ayuda a evitar la ambigüedad que puede ocurrir cuando dos identificadores tienen el mismo nombre.



```
Libreria de compresion
*/
namespace Huffman { // Un espacio de nombres es una región declarativa

    string HuffmanValue[256] = { "" };

    typedef struct Node {
    public:
        char character; //caracter
        ll count; //cantidad de bits que aparece
        Node* left, * right; //para crear el nodo

        Node(ll count) {
            this->character = 0;
            this->count = count;
            this->left = this->right = nullptr;
        }

        Node(char character, ll count) {
            this->character = character;
            this->count = count;
            this->left = this->right = nullptr;
        }
    } Node;
}
```

Nodos

Para el proyecto se utilizará un tipo de nodo para la creación del método Huffman

Node: Nodo de un arbol, el cual tiene el carácter a almacenar, cantidad de bits, y nodos hijos

Luego cuenta con funciones para contar las apariciones del carácter y luego moverse, ya sea izquierda o derecha.

```

typedef struct Node {
public:
    char character; //caracter
    ll count; //cantidad de bits que aparece
    Node* left, * right; //para crear el nodo

    Node(ll count) {
        this->character = 0;
        this->count = count;
        this->left = this->right = nullptr;
    }

    Node(char character, ll count) {
        this->character = character;
        this->count = count;
        this->left = this->right = nullptr;
    }
} Node;

```

Relación entre estructuras

En este proyecto se utilizará el árbol de Huffman que toma su base en los árboles binarios para poder ordenar y generará códigos para los caracteres a comprimir. Estos los árboles de Huffman y los binarios presentan una misma estructura, lo que los diferencia es que el árbol de Huffman se utiliza con un propósito específico y por tanto tiene funciones especiales que recorren la ruta desde la raíz hasta un nodo generando unas secuencias de ceros y unos.

Este trabajo solo utiliza principalmente árboles de Huffman, estos son usados como recipientes, para comprimir se generan y recorren para descubrir el código para cada carácter; donde para descomprimir se replica el árbol anterior y en base a la secuencia de bits se llega al nodo final, de la capa más inferior y se descubre cuál es el carácter que corresponde a esa secuencia.

Algoritmos/Lógica

Para este proyecto se utilizará un tipo de nodo para analizar los caracteres y namespace para usar las funciones dentro de esos espacios.

Namespace: Huffman, va a almacenar el nodo necesario y sus funciones para contar los caracteres.

Node: nodo que contiene el carácter, las apariciones en bits y los nodos izquierda y derecha.

También contiene las funciones necesarias para contar las apariciones en bits de cada carácter que se analice.

```
namespace Huffman {  
  
    string HuffmanValue[256] = { "" };  
  
    typedef struct Node {  
    public:  
        char character;  
        ll count;  
        Node* left, * right;  
  
        Node(ll count) {  
            this->character = 0;  
            this->count = count;  
            this->left = this->right = nullptr;  
        }  
  
        Node(char character, ll count) {  
            this->character = character;  
            this->count = count;  
            this->left = this->right = nullptr;  
        }  
    } Node;  
}
```


Namespace: Utility, contiene algunas funciones necesarias para el correcto funcionamiento del compresor y descompresor.

```
namespace Utility {
```

GetFileSize: Función que se encarga de obtener lo que pesa el archivo enviado.
(ubicada en Utility).

```
ll GetFileSize(const char* filename) {  
    FILE* p_file = fopen(_FileName: filename, _Mode: "rb");  
    fseek(_Stream: p_file, _Offset: 0, _Origin: SEEK_END);  
    ll size = _ftelli64(_Stream: p_file);  
    fclose(_Stream: p_file);  
    return size;  
}
```

Namespace: CompressUtility, va a contener las funciones necesarias para el compresor.

```
namespace CompressUtility {
```

Combine: Función para combinar dos nodos (almacenada en CompressUtility).

```
Node* Combine(Node* a, Node* b) {  
    Node* parent = new Node((a ? a->count : 0) + (b ? b->count : 0));  
    parent->left = b;  
    parent->right = a;  
    return parent;  
}
```

sortBysec: Función para ordenar los nodos (almacenada en CompressUtility).

```
bool sortBysec(Node* a, Node* b) {  
    return (a->count > b->count);  
}
```

ParseFile: Función para para contar caracteres el archivo y devuelve los caracteres contados en bits. (almacenada en CompressUtility).

```
map <char, ll> ParseFile(const char* filename, ll Filesize) {  
    register FILE* ptr = fopen(_FileName: filename, _Mode: "rb");  
  
    if (ptr == NULL) {  
        perror(_ErrorMessage: "Error: File not found:");  
        exit(_Code: -1);  
    }  
    register unsigned char ch;  
    register ll size = 0, filesize = Filesize;  
    vector<ll>Store(256, 0);  
  
    while (size != filesize) {  
        ch = fgetc(_Stream: ptr);  
        ++Store[ch];  
        ++size;  
    }  
    map <char, ll> store;  
    for (int i = 0; i < 256; ++i)  
        if (Store[i])  
            store[i] = Store[i];  
    fclose(_Stream: ptr);  
    return store;  
}
```

SortByCharacterCount: Función para ordenar un array con los caracteres de menor a mayor, menor cantidad de bits a mayor cantidad de bits. (almacenada en CompressUtility).

```
vector <Node*> SortByCharacterCount(const map <char, ll >& value) {  
    vector < Node* > store;  
    auto std::map<char...const_iterator it = begin(_Cont: value);  
    for (; it != end(_Cont: value); ++it)  
        store.push_back(_Val: new Node(it->first, it->second));  
    sort(_First: begin(& _Cont: store), _Last: end(& _Cont: store), _Pred: sortbysec);  
  
    return store;  
}
```

GenerateHeader: Función para generar la cabecera del archivo con Código Huffman (mínimo: 1 byte, máximo: 255bytes) (almacenada en CompressUtility).

```
string GenerateHeader(char padding) {
    string header = "";
    // UniqueCharacter start from -1 {0 means 1, 1 means 2, to conserve memory}
    unsigned char UniqueCharacter = 255;

    for (int i = 0; i < 256; ++i) {
        if (HuffmanValue[i].size()) {
            header.push_back(_Ch: i);
            header.push_back(_Ch: HuffmanValue[i].size());
            header += HuffmanValue[i];
            ++UniqueCharacter;
        }
    }
    char value = UniqueCharacter;

    return value + header + (char)padding;
}
```

GenerateHuffmanTree: Función para generar el arbol de huffman durante la compresión del archivo, se envía un array con los caracteres contados. (almacenada en CompressUtility).

```
Node* GenerateHuffmanTree(const map <char, ll>& value) {
    vector < Node* > store = SortByCharacterCount(value);
    Node* one, * two, * parent;
    sort(_First: begin(& _Cont: store), _Last: end(& _Cont: store), _Pred: sortbysec);
    if (store.size() == 1) {
        return Combine(store.back(), b: nullptr);
    }
    while (store.size() > 2) {
        one = *(end(& _Cont: store) - 1); two = *(end(& _Cont: store) - 2);
        parent = Combine(a: one, b: two);
        store.pop_back(); store.pop_back();
        store.push_back(_Val: parent);

        vector <Node*> ::iterator it1 = end(& _Cont: store) - 2;
        while ((*it1)->count < parent->count && it1 != begin(& _Cont: store))
            --it1;
        sort(_First: it1, _Last: end(& _Cont: store), _Pred: sortbysec);
    }
    one = *(end(& _Cont: store) - 1); two = *(end(& _Cont: store) - 2);
    parent = Combine(a: one, b: two);
    return parent;
}
```

StoreHuffmanValue: Función que Almacena los valores Huffman para cada carácter de la cadena y devuelve el tamaño del archivo resultante (sin la cabecera), se envía un nodo con el arbol de huffman y un buffer tipo string (almacenada en CompressUtility).

```
ll StoreHuffmanValue(Node* root, string& value) {
    ll temp = 0;
    if (root) {
        value.push_back(_Ch: '0');
        temp = StoreHuffmanValue(root->left, &: value);
        value.pop_back();
        if (!root->left && !root->right) {
            HuffmanValue[(unsigned char)root->character] = value;
            temp += value.size() * root->count;
        }
        value.push_back(_Ch: '1');
        temp += StoreHuffmanValue(root->right, &: value);
        value.pop_back();
    }
    return temp;
}
```

Compress: Función que se encarga de la compresión del archivo, se envía el nombre del archivo, el tamaño del archivo y lo que va a pesar el archivo (almacenada en CompressUtility).

```
void Compress(const char* filename, const ll Filesize, const ll PredictedFileSize) {
    const char padding = (8 - ((PredictedFileSize & (7))) & (7));
    string header = GenerateHeader(padding);
    int header_i = 0;
    const int h_length = header.size();
    cout << "Padding size: " << (int)padding << endl;
    FILE* iptr = fopen(_FileName: filename, _Mode: "rb"), * optr = fopen(_FileName: (string(filename) + ".abiz").c_str(), _Mode: "wb");

    while (header_i < h_length) {
        fputc(_Character: header[header_i], _Stream: optr);
        ++header_i;
    }

    if (!iptr) {
        perror(_ErrorMessage: "Error: File not found: ");
        exit(_Code: -1);
    }

    unsigned char ch, fch = 0;
    char counter = 7;
    ll size = 0, i;
    while (size != Filesize) {
        ch = fgetc(_Stream: iptr);
        i = 0;
        while (HuffmanValue[ch][i] != '\0') {
            fch = fch | ((HuffmanValue[ch][i] - '0') << counter);
            --counter;
            if (counter == -1) {
                fputc(_Character: fch, _Stream: optr);
                counter = 7;
                fch = 0;
            }
            ++i;
        }
        ++size;
        if (((size * 100 / Filesize)) > ((size - 1) * 100 / Filesize))
            printf(_Format: "\r%d%% completed ", (size * 100 / Filesize));
    }
    if (fch)
        fputc(_Character: fch, _Stream: optr);
    printf(_Format: "\n");
    fclose(_Stream: iptr);
    fclose(_Stream: optr);
}
```

Namespace: DecompressUtility, va a contener las funciones necesarias el descompresor.

```
namespace DecompressUtility {
```

GenerateHuffmanTree: Función que se encarga de crear un arbol de huffman durante la descompresión del archivo, se envía un nodo nuevo, los caracteres del archivo y un char para almacenar. (almacenada en DecompressUtility).

```
void GenerateHuffmanTree(Node* const root, const string& codes, const unsigned char ch) {
    Node* traverse = root;
    int i = 0;
    while (codes[i] != '\0') {
        if (codes[i] == '0') {
            if (!traverse->left)
                traverse->left = new Node(0);
            traverse = traverse->left;
        }
        else {
            if (!traverse->right)
                traverse->right = new Node(0);
            traverse = traverse->right;
        }
        ++i;
    }
    traverse->character = ch;
}
```

DecodeHeader: Función para decodificar la cabecera, se envía la cabecera del archivo y se devuelve el nodo, con la cabecera y el total del tamaño (almacenada en DecompressUtility).

```
pair<Node*, pair<unsigned char, int> >DecodeHeader(FILE* iptr) {
    Node* root = new Node(0);
    int charactercount, buffer, total_length = 1;
    register char ch, len;
    charactercount = fgetc(_Stream: iptr);
    string codes;
    ++charactercount;
    while (charactercount) {
        ch = fgetc(_Stream: iptr);
        codes = "";
        buffer = 0;
        len = fgetc(_Stream: iptr);
        buffer = len;

        while (buffer > codes.size())
            codes += fgetc(_Stream: iptr);
        // character (1byte) + length(1byte) + huffmancode(n bytes where n is length of huffmancode)
        total_length += codes.size() + 2;

        GenerateHuffmanTree(root, codes, ch);
        --charactercount;
    }
    unsigned char padding = fgetc(_Stream: iptr);
    ++total_length;
    return { root, {padding, total_length} };
}
```

Decompress: Función que se encarga de la descompresión actual del archivo, se envía el nombre del archivo y lo que pesa este, al final crea el archivo descomprimido.

```
void Decompress(const char* filename, const ll Filesize) {
    string fl = string(filename);
    FILE* iptr = fopen(_FileName: fl.c_str(), _Mode: "rb");
    FILE* optr = fopen(_FileName: string("output" + fl.substr(_Off: 0, _Count: fl.length() - 5)).c_str(), _Mode: "wb");

    if (iptr == NULL) {
        perror(_ErrorMessage: "Error: File not found");
        exit(_Code: -1);
    }

    pair<Node*, pair<unsigned char, int> >HeaderMetadata = DecodeHeader(iptr);
    Node* const root = HeaderMetadata.first;
    const auto const unsigned char padding = HeaderMetadata.second.first;
    const auto const int headersize = HeaderMetadata.second.second;

    char ch, counter = 7;
    ll size = 0;
    const ll filesize = Filesize - headersize;
    Node* traverse = root;
    ch = fgetc(_Stream: iptr);
    while (size != filesize) {
        while (counter >= 0) {
            traverse = ch & (1 << counter) ? traverse->right : traverse->left;
            ch ^= (1 << counter);
            --counter;
            if (!traverse->left && !traverse->right) {
                fputc(_Character: traverse->character, _Stream: optr);
                if (size == filesize - 1 && padding == counter + 1)
                    break;
                traverse = root;
            }
        }
        ++size;
        counter = 7;
        if (((size * 100 / filesize)) > ((size - 1) * 100 / filesize))
            printf(_Format: "\r%lld%% completed, size: %lld bytes ", (size * 100 / filesize), size);
        ch = fgetc(_Stream: iptr);
    }
    fclose(_Stream: iptr);
    fclose(_Stream: optr);
}
```

Main

Para esta parte solo se enfocará en las partes más importantes del main, en este caso es solamente un pequeño main donde va a recibir los argumentos necesarios a la hora de correr el archivo

```
int main(int argc, char* argv[]) {  
    if (argc != 3) {  
        printf(_Format: "Usage:\n (a.exe|./a.out) (-c FileToBeCompressed | -dc FileToBeDecompressed)");  
        exit(_Code: -1);  
    }  
    const char* option = argv[1], * filename = argv[2];  
    printf(_Format: "%s\n", filename);  
  
    time_point <system_clock> start, end;  
    duration <double> time;  
    ll filesize, predfilesize;  
    if (string(option) == "-c") {  
        filesize = Utility::GetFileSize(filename);  
        auto std::map<char..., long long> mapper = CompressUtility::ParseFile(filename, filesize);  
        Node* const root = CompressUtility::GenerateHuffmanTree(value: mapper);  
        string buf = "";  
        predfilesize = CompressUtility::StoreHuffmanValue(root, & value: buf);  
        printf(_Format: "Original File: %lld bytes\n", filesize);  
        printf(_Format: "Compressed File Size (without header): %lld bytes\n", (predfilesize + 7) >> 3);  
  
        start = system_clock::now();  
        CompressUtility::Compress(filename, filesize, PredictedFileSize: predfilesize);  
        end = system_clock::now();  
  
        time = (end - start);  
        cout << "Compression Time: " << time.count() << "s" << endl;  
    }  
    else if (string(option) == "-dc") {  
        filesize = Utility::GetFileSize(filename);  
        start = system_clock::now();  
        DecompressUtility::Decompress(filename, filesize);  
        end = system_clock::now();  
  
        time = (end - start);  
        cout << "\nDecompression Time: " << time.count() << "s" << endl;  
    }  
    else  
        cout << "\nInvalid Option... Exiting\n";  
    return 0;  
}
```

Para recibir los argumentos solamente vamos a las propiedades del proyecto y escribimos lo que se quiera realizar. En el caso de que se envié: -c, vamos a entrar a la opción de comprimir y esta se va a encargar de llamar a todas las funciones necesarias para comprimir el archivo.

Si se envía: -dc, entra a la opción de descomprimir y de igual forma llama a todas las funciones necesarias para descomprimir el archivo.