CENTRO UNIVERSITÁRIO NORTE DO ESPÍRITO SANTO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

ANTONIELLY BERGAMI RIBEIRO
WILLIAN PACHECO SILVA

RELATÓRIO: ALGORITMO DE HUFFMAN

SÃO MATEUS
2021

ANTONIELLY BERGAMI RIBEIRO
WILLIAN PACHECO SILVA

RELATÓRIO: ALGORITMO DE HUFFMAN

Relatório apresentado à Disciplina de Estrutura de dados II dos cursos de bacharelado em Ciência e Engenharia da Computação 2020/2 do Centro Universitário Norte do Espírito Santo, como requisito parcial para avaliação .
Orientador: Luciana Lee.

SÃO MATEUS
2021

RESUMO

Este relatório faz parte da avaliação parcial da disciplina de Estruturas de Dados II dos cursos de Bacharelado em Ciência e Engenharia da Computação do Centro Universitário Norte do Espírito Santo.

Sua finalidade é apresentar os resultados da implementação do algoritmo de Huffman que foi usado para comprimir um arquivo de texto em um arquivo binário e depois descomprimir o arquivo binário para outro arquivo de texto.

Para isso ocorrer, foram implementadas várias funções como, por exemplo, as funções de criação e manipulação de um heap e funções para comprimir e descomprimir o arquivo. Também foram implementadas uma série de funções auxiliares para gerar a árvore de Huffman, liberar a memória alocada durante o programa e a função de imprimir a árvore. Cada função que foi implementada será explicada neste relatório, assim como o seu funcionamento.

SUMÁRIO

1 INTRODUÇÃO

O algoritmo de huffman foi desenvolvido em 1952 e trata-se de um método utilizado para compactar arquivos de texto. Neste algoritmo é utilizada a quantidade de ocorrências de um caractere no arquivo a ser comprimido para que assim possa ser gerado um código para cada caractere, de forma que os caracteres que mais se repetem terão códigos que ocupam menos bits.

Para que seja possível atribuir o código aos caracteres é criada a chamada “árvore de Huffman” baseada na ocorrência dos símbolos, então as folhas dessas árvores representam os símbolos e estão associadas com a ocorrência de cada um deles. Os nós que não são folhas, armazenam a soma das ocorrências dos dois filhos que ele possui. Com base nessa árvore é possível partir da raiz e ir até o nó folha contabilizando o caminho que foi percorrido. Nesse percurso, foi convencionado o bit 0 para quando se desloca para o filho à esquerda e o bit 1 para quando se desloca para o filho à direita. Dessa forma, ao percorrer a árvore da raiz até a folha será gerado um código binário que é distinto para cada caminho e que pode ser usado para representar a folha em que o caminho termina. Por meio desse algoritmo, como os nós mais próximos da raiz possuem um maior número de repetições, é possível notar que eles precisarão de menos bits para serem representados, o que ocasiona em uma boa economia de espaço e uma compressão no arquivo.

Nesse relatório contém os resultados encontrados após a implementação do algoritmo com suas funções e tratamentos.

2 OBJETIVOS

Esse trabalho tem como objetivo apresentar conhecimentos adquiridos em sala de aula da disciplina de estruturas de dados II, e também por meio de pesquisas, sendo aplicados na implementação do algoritmo de Huffman, apresentando os métodos utilizados para que a atividade proposta seja realizada.

3 METODOLOGIA

Os resultados aqui apresentados foram obtidos através da implementação do algoritmo de huffman desenvolvido pelos alunos através da linguagem de programação C.

3.1 LINGUAGEM C

C é uma linguagem de programação compilada de propósito geral, estruturada, imperativa, procedural, padronizada pela Organização Internacional para Padronização (ISO), criada em 1972 por Dennis Ritchie na empresa AT&T Bell Labs para desenvolvimento do sistema operacional Unix (originalmente escrito em Assembly). C é uma das linguagens de programação mais populares e existem poucas arquiteturas para as quais não existem compiladores para C. C tem influenciado muitas outras linguagens de programação (por exemplo, a linguagem Java), mais notavelmente C++, que originalmente começou como uma extensão para C.

4 APRESENTAÇÃO DOS RESULTADOS E DISCUSSÕES

Para alcançarmos o objetivo proposto, que é a implementação do algoritmo de Huffman, alguns arquivos foram implementados, são eles: Huffman.h, que é o arquivo cabeçalho, Huffman.c, onde contém todas as implementações das funções, main.c, que é o arquivo do cliente, e o Makefile. Esses arquivos serão explicados com mais detalhes a seguir.

4.1 Huffman.h

O arquivo cabeçalho serve para que o cliente compreenda cada função: o que elas fazem, quais os parâmetros, qual o retorno dela e etc. Nesse arquivo além do cabeçalho de todas as funções necessárias para a execução do programa, também foram incluídas todas as bibliotecas necessárias para uma perfeita execução. Esse arquivo posteriormente será uma biblioteca incluída no arquivo Huffman.c e no arquivo main.c. Na imagem abaixo é possível observar como esse arquivo foi implementado.

```c
// Integrantes:
// Antonielly Bergami Ribeiro (2018205200)
// Willian Pacheco Silva (2019107753)

// Este arquivo se trata de uma biblioteca para o algoritmo de Huffman.
#ifndef __HUFFMAN_H__
#define __HUFFMAN_H__

//Inclusão das bibliotecas necessárias.
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

//Definição da estrutura onde irá conter as informações do nó da árvore
typedef struct huffman
{
    unsigned char caractere;           //armazena o caractere
    int frequencia;                    //armazena a frequência do caractere
    int tipoFilho;                     //armazena se o nó é filho a direita (0) ou a esquerda (1) do pai dele
    struct huffman *pai;               //aponta para o nó pai
    struct huffman *subarvoreDireita;  //aponta para o filho da direita
    struct huffman *subarvoreEsquerda; //aponta para o filho da esquerda
} Huffman;

//Definição da estrutura que vai ser gravada no arquivo dentro do heap
typedef struct huffmanArquivo
{
    unsigned char caractere;
    int frequencia;
} HuffmanArquivo;

//Definição da estrutura do cabeçalho que será gravada no aquivo compactado
typedef struct cabecalho
{
    unsigned char tamanhoHeap;
    int tamanhoBinario;
} Cabecalho;

/*Função utilizada para criação de um nó da árvore de Huffman. Recebe como parametro o caractere e a
frequência dele no arquivo de entrada. O retorno dessa função será o nó já com campos atribuidos.*/
Huffman *criaNo(unsigned char caractere, int frequencia);

/*Função auxiliar utilizada para retornar a posição do pai de um nó do heap em um vetor. Recebe como
parametro a posição do nó atual no heap*/
int Pai(int i);
```

IMAGEM 1.1: Huffman.h

```c
/*Função auxiliar utilizada para retornar a posição do filho à esquerda de um nó do heap em um vetor.
Recebe como parametro a posição do nó atual no heap.*/
int FilhoEsquerdo(int i);

/*Função auxiliar utilizada para retornar a posição do filho à direita de um nó do heap em um vetor.
Recebe como parametro a posição do nó atual no heap.*/
int FilhoDireito(int i);

/*Função utilizada para restabelecer as propriedades do heap mínimo. Recebe como parâmetro um heap com os
nós da árvore, o tamanho do heap e uma variavel de controle da posição. Essa função não possui retorno. */
void MinHeapify(Huffman **vetor, int i, int tamanho);

/*Função utilizada para retirar o menor elemento de um heap. Essa função recebe como parâmetro um heap e
um ponteiro para a variável que armazena o tamanho dele. O retorno será o menor elemento do heap*/
Huffman *RetiraMinimo(Huffman **a, int *tamanho);

/*Função é utilizada para posicionar a chave de um heap. Recebe como parametro o heap, a posição da chave e
a nova chave que será inserida nessa posição.*/
void DecrementaChave(Huffman **a, int i, Huffman *chave);

/*Função utilizada para inserir uma chave no heap. Recebe como parametro o heap, a nova chave a ser inserida
e um ponteiro para a variável que armazena o tamanho do heap. Essa função não possui retorno.*/
void InserirHeapMinimo(Huffman **a, Huffman *chave, int *tamanho);

/*Função utilizada para inserir os elementos do arquivo no heap. Recebe como parâmetro a tabela hash que
relaciona cada caractere com a quantidade de repetições dele e o número de letras distintas que há no
arquivo de entrada. Depois disso, ela cria um heap com essas informações e retorna o heap criado. Vale
ressaltar que esse heap é um heap de ponteiros para os nós da árvore.*/
Huffman **criaHeapMin(int *repeticoes, int letrasDistintas);

/*Função utilizada para copiar as informações do heap, que contém os ponteiros para os nós da árvore, para
um vetor que posteriormente possa ser armazenado no arquivo. O retorno dessa função será o vetor com as
informações de cada chave e sua respectiva frequência.*/
HuffmanArquivo *criaHeapEstatico(Huffman **heapMin, int tamanhoHeap);

/*Função utilizada para criar uma tabela hash relacionando o caractere com um ponteiro que aponta para o nó
que guarda esse caractere na árvore. Ela recebe como parâmetro o heap e uma variável contendo o tamanho do
heap. Com base nesse heap, a tabela hash é criada e o retorno será a própria tabela hash.*/
Huffman **criaHashHeap(Huffman **heap, int tamanhoHeap);

/*Função utilizada para gerar a árvore de Huffman. Essa função recebe como parâmetro o heap mínimo que
armazena os nós da árvore e uma variável com o tamanho do heap. O retorno dessa função será um ponteiro
para o nó raiz da árvore.*/
Huffman *criaArvoreHuffman(Huffman **heapMin, int tamanhoHeap);
```

IMAGEM 1.2: Huffman.h

```c
/*Função utilizada para criar uma tabela hash que relaciona o caractere com o número de repetições dele
no arquivo. Ela recebe como parâmetro o nome do arquivo, um ponteiro para a variável que armazena a
quantidade de letras distintas e outro ponteiro para a variável que armazena a quantidade de caracteres
presentes no arquivo de texto. O retorno dessa função será a tabela hash. */
int *criaVetorRepeticoes(char *nomeArquivo, int *letrasDistintas, int *quantidadeCaracteres);

/*Função utilizada para preencher o byte através da árvore. Recebe como parâmetro um ponteiro para o nó
folha de uma árvore que contém o caractere, um vetor de bytes (char) que armazena o código binário, um
ponteiro que aponta para a váriavel que controla a posição do vetor, um ponteiro para a variável que
controla o número de bits ocupados daquele byte, um ponteiro que aponta para a variável que controla o
tamanho do código binário e uma variável que armazena a quantidade de letras distintas presentes no arquivo.*/
void codificaChar(Huffman *arvore, char *codBin, int *posicaoVetor, int *numeroBits, int *tamanhoBinario, int letrasDistintas);

/*Função utilizada para converter os caracteres do arquivo para o código binário. Recebe como parâmetro o
nome de um arquivo a ser lido, a tabela hash que associa cada caractere com o nó dele da árvore, uma variável
que contém a quantidade de cada caracteres presentes no arquivo, um ponteiro que aponta para a variável que
contém o tamanho do código binário e uma variável que contém a quantidade de caracteres distintos do arquivo.
O retorno dessa função será a codificação dos caracteres do arquivo em um vetor de bytes.*/
char *criaBinario(char *nomeArquivo, Huffman **hashHeap, int numeroLetras, int *tamanhoBinario, int letrasDistintas);

/*Função utilizada para comprimir o arquivo. Recebe como parâmetro o nome do arquivo a ser comprimido, a
estrutura do cabeçalho, o heap com as informações do caractere e o código binário que foi gerado a partir
da conversão do texto do arquivo para a sequência de bits.*/
void comprimir(char *nomeArquivo, Cabecalho cab, HuffmanArquivo *heapEstatico, char *codigoBinario);

/*Função utilizada para descomprimir um arquivo. Recebe como parametro o nome do arquivo compactado e
um nome do arquivo para onde ele será descompactado. */
void descomprimir(char *arquivoCompactado, char *arquivoDestino);
```

IMAGEM 1.3: Huffman.h

```
120
121  /*Função utilizada para liberar a mamória alocada para uma árvore. Recebe como parametro
122  um ponteiro para o nó raiz de uma árvore.*/
123  void liberaArvore(Huffman *a);
124
125  /*Função utilizada para imprimir a sequência binária que representa um caractere com base na árvore de
126  Huffman. Recebe como parametro um ponteiro para o nó raiz da árvore e uma váriavel de controle da quantidade
127  de letras distintas.*/
128  void imprimeBinarioCaractere(Huffman *arvore, int letrasDistintas);
129
130  #endif
```

IMAGEM 1.4: Huffman.h

Vale ressaltar que o arquivo foi bem comentado com descrições do funcionamento de cada função para que isso possa orientar o cliente. No arquivo Huffman.h também foram implementadas as estruturas utilizadas no programa. Elas serão explicadas nas seções seguintes.

4.1.1 Estrutura Huffman

A estrutura “Huffman” é a estrutura que contém as informações referentes ao nó da árvore de Huffman. Essa estrutura foi implementada da seguinte forma:

```
//Definição da estrutura onde irá conter as informações do nó da árvore
typedef struct huffman
{
    unsigned char caractere;           //armazena o caractere
    int frequencia;                    //armazena a frequência do caractere
    int tipoFilho;                     //armazena se o nó é filho a direita (0) ou a esquerda (1) do pai dele
    struct huffman *pai;               //aponta para o nó pai
    struct huffman *subarvoreDireita;  //aponta para o filho da direita
    struct huffman *subarvoreEsquerda; //aponta para o filho da esquerda
} Huffman;
```

IMAGEM 2: Estrutura “Huffman”

Como é possível observar, essa estrutura possui alguns campos, são eles:

unsigned char caractere: Esse campo irá armazenar o caractere. Vale lembrar que ela é definida como unsigned char para trabalhar no intervalo que vai de 0 a 255. Isso é importante pois não permite que o char possua valor negativo, o que traria problemas nas tabelas hash.

int frequencia: Esse campo é o responsável por armazenar a frequência em que o caractere aparece no arquivo.

int tipoFilho: Esse campo armazena se o nó é filho à direita (0) ou à esquerda (1). Isso é importante para que seja possível gerar o código binário a partir da árvore posteriormente.

struct huffman *pai: Campo do tipo ponteiro para Huffman que irá apontar para o pai do nó atual.

struct huffman *subarvoreDireita: Campo do tipo ponteiro para Huffman que irá apontar para o nó filho da direita do nó atual.

struct huffman *subarvoreEsquerda: Campo do tipo ponteiro para Huffman que irá apontar para o nó filho da esquerda do nó atual.

Importante dizer que um dado do tipo “ponteiro para huffman” quer dizer que essa variável irá apontar para uma estrutura do tipo huffman que foi alocada na memória previamente, nesse caso, o nó pai ou o nó filho da direita ou da esquerda.

4.1.2 Estrutura HuffmanArquivo

A estrutura “HuffmanArquivo” contém as informações que futuramente serão gravadas em um arquivo dentro do heap mínimo. Essa estrutura foi implementada da seguinte forma:

```
//Definição da estrutura que vai ser gravada no arquivo dentro do heap
typedef struct huffmanArquivo
{
    unsigned char caractere;
    int frequencia;
} HuffmanArquivo;
```

IMAGEM 3: Estrutura “HuffmanArquivo”

É possível perceber que essa estrutura possui dois campos: unsigned char caractere e int frequencia. O que esse campo irá armazenar é análogo à estrutura Huffman, ou seja, irá armazenar o caractere e a frequência com que ele se repete no arquivo de texto que foi lido.

Essa estrutura foi criada principalmente para que seja possível armazenar no arquivo binário apenas o necessário e evitar o uso desnecessário de espaço, já que o intuito do algoritmo é gerar um arquivo comprimido que seja do menor tamanho possível. Como é possível perceber, a estrutura possui apenas um char e um int, e não possui os outros campos que estão relacionados à árvore. Isso implica em uma redução no espaço necessário para armazenar esse tipo de dado no arquivo.

4.1.3 Estrutura Cabeçalho

A estrutura “cabeçalho” é uma estrutura que irá armazenar as informações referentes ao cabeçalho que, posteriormente, será gravado no arquivo para que seja possível obter algumas informações importantes para a leitura do mesmo e para o processo de descompactação. Ela foi implementada da seguinte forma:

```
//Definição da estrutura do cabeçalho que será gravada no aquivo compactado
typedef struct cabecalho
{
    unsigned char tamanhoHeap;
    int tamanhoBinario;
} Cabecalho;
```

IMAGEM 4: Estrutura “Cabecalho”

Nota-se que nessa estrutura os campos existentes são: unsigned char tamanhoHeap e int tamanhoBinario. Esses campos que vão armazenar a

quantidade de elementos do heap mínimo que se encontra no arquivo e a quantidade de bits que foram necessários para o código binário, respectivamente.

4.2 Huffman.c

O arquivo Huffman.c é o arquivo onde foram implementadas todas as funções necessárias para a execução do programa. Nele, foi incluída a biblioteca “huffman.h” e foram feitas as implementações das funções relativas ao algoritmo de Huffman. Na sequência, cada função presente nesse arquivo será explicada.

```c
//Inclusão da biblioteca
#include "Huffman.h"
```

IMAGEM 5: Inclusão da biblioteca “Huffman.h”

4.2.1 Cria nó

A função criaNo é a responsável por alocar a memória para um nó da árvore de Huffman. Ela recebe como parâmetro um caractere e a frequência desse caractere.

Inicialmente, é feita a alocação de memória para um novo nó. Caso não ocorra nenhum erro de alocação de memória, o nó tem seus campos inicializados para que por fim a função possa retornar esse nó.

A implementação dessa função ficou da seguinte forma:

```c
//Função utilizada para criar um novo nó da árvore
Huffman *criaNo(unsigned char caractere, int frequencia)
{
    //aloca memória para um novo nó
    Huffman *novo = (Huffman *)malloc(sizeof(Huffman));
    if (!novo) //caso ocorra algum erro na alocação
    {
        printf("Erro na alocacao de memoria.");
        exit(1);
    }
    //caso contrário, inicializa o nó
    novo->caractere = caractere;
    novo->frequencia = frequencia;
    novo->tipoFilho = 0;

    novo->pai = NULL;
    novo->subarvoreDireita = NULL;
    novo->subarvoreEsquerda = NULL;

    return novo; //retorna o nó criado e inicializado.
}
```

IMAGEM 6: Implementação da função “criaNo”

4.2.2 Pai

A função “Pai” é uma função auxiliar utilizada apenas para retornar a posição do pai do nó de um heap. Ela recebe como parâmetro a posição do nó atual e

retorna a posição do pai dele no heap. A implementação dessa função se encontra abaixo.

```c
/*Função auxiliar utilizada para retornar a posição
do pai de um nó do heap em um vetor*/
int Pai(int i)
{
    return i / 2;
}
```

IMAGEM 7: Implementação da função “Pai”

4.2.3 Filho esquerdo

A função “filhoEsquerdo” é uma função auxiliar utilizada apenas para retornar a posição do filho esquerdo do nó de um heap. Ela recebe como parâmetro a posição do nó atual e retorna a posição do filho à esquerda dele no heap. A implementação dessa função se encontra abaixo.

```c
/*Função auxiliar utilizada para retornar a posição do
filho à esquerda de um nó do heap em um vetor*/
int FilhoEsquerdo(int i)
{
    return 2 * i;
}
```

IMAGEM 8: Implementação da função “filhoEsquerdo”

4.2.4 Filho direito

A função “filhoDireito” é uma função auxiliar utilizada apenas para retornar a posição do filho direito do nó de um heap. Ela recebe como parâmetro a posição do nó atual e retorna a posição do filho à direita dele no heap. A implementação dessa função se encontra abaixo.

```c
/*Função auxiliar utilizada para retornar a posição do
filho à direita de um nó do heap em um vetor*/
int FilhoDireito(int i)
{
    return 2 * i + 1;
}
```

IMAGEM 9: Implementação da função “filhoDireito”

4.2.5 Min Heapify

A função “minHeapify” é uma das funções básicas de um heap. Ela é utilizada para que seja possível restabelecer as propriedades de um heap depois de realizar

uma inserção ou remoção de algum elemento. A implementação dessa função se encontra na imagem abaixo.

```
51  //Função utilizada para restabelecer as propriedades do heap minimo
52  void MinHeapify(Huffman **vetor, int i, int tamanho)
53  {
54      int esquerdo, direito, menor;
55      Huffman *aux;
56      esquerdo = FilhoEsquerdo(i);
57      direito = FilhoDireito(i);
58
59      //condicionais para buscar o menor elemento entre o nó atual e os filhos dele
60      if (esquerdo <= tamanho && vetor[esquerdo]->frequencia < vetor[i]->frequencia)
61          menor = esquerdo;
62      else
63          menor = i;
64
65      if (direito <= tamanho && vetor[direito]->frequencia < vetor[menor]->frequencia)
66          menor = direito;
67
68      //troca o elemento que se encontra na posicao menor como o elemento que se encontra na posicao i
69      if (menor != i)
70      {
71          aux = vetor[i];
72          vetor[i] = vetor[menor];
73          vetor[menor] = aux;
74
75          //realiza a recursão para consertar as propriedades de heap nos outros nós
76          MinHeapify(vetor, menor, tamanho);
77      }
78  }
```

IMAGEM 10: Implementação da função “minHeapify”

Nessa função, inicialmente são armazenadas as posições dos filhos à esquerda e à direita do nó atual. Depois disso, são feitas algumas comparações nas linhas 60 a 66 para verificar se o nó que possui a menor chave é o nó atual, ou algum dos filhos dele. Após realizar essas comparações, caso o nó de menor chave seja um dos filhos do nó atual, é feita uma troca entre o nó atual e o filho que possui a menor chave e a função “minHeapify” é chamada recursivamente para que esse processo seja feito em todo o heap.

4.2.6 Retira mínimo

A função “retiraMinimo” é uma função utilizada para retirar o menor elemento de um heap. Essa função recebe como parâmetro um heap e o tamanho dele. Depois disso, é definida uma variável onde o elemento será armazenado (linha 84).

Após isso, a função irá fazer uma verificação: caso o tamanho do heap seja menor que 1 o programa irá apresentar uma mensagem de erro. Caso contrário, o primeiro elemento do heap mínimo será armazenado na variável auxiliar, já que ele é o menor elemento, e o último elemento passará a ser o primeiro (linhas 91-92). Depois disso, o tamanho do heap é decrementado em uma unidade e a função “MinHeapify” é chamada para restabelecer as propriedades do heap após a remoção. Por fim, a função poderá retornar o menor elemento. A implementação dessa função se encontra na imagem abaixo.

```
80  ///Função para retirar o menor elemento do heap
81  Huffman *RetiraMinimo(Huffman **a, int *tamanho)
82  {
83      //variavel auxiliar para armazenar o menor elemento do heap
84      Huffman *min = 0;
85      if (*tamanho < 1) //se o tamanho do heap for menor que 1, apresenta o erro de underflow
86      {
87          printf("Underflow");
88          exit(1);
89      }
90      //troca o primeiro elemento com o último
91      min = a[1];
92      a[1] = a[*tamanho];
93      //decrementa o tamanho do heap
94      *tamanho = *tamanho - 1;
95      //restabelece as propriedades do heap mínimo
96      MinHeapify(a, 1, *tamanho);
97      //retorna o menor elemento do heap
98      return min;
99  }
100
```

IMAGEM 11: Implementação da função “retiraMinimo”

4.2.7 Decrementa chave

Essa função é utilizada para posicionar a chave de um heap. A função inicia fazendo um verificação e caso a nova chave seja maior do que a chave atual do heap uma mensagem de erro será exibida. Caso contrário, a memória da chave do heap é liberada pois ela só foi alocada para que seja possível realizar essa comparação, e a nova chave é incluída (linha 110).

Depois disso, enquanto a posição do heap for maior do que 1 e a frequência do pai do elemento atual for menor do que a frequência do elemento (linha 114-122), o elemento será trocado de posição com o pai dele e o processo irá se repetir para o pai.

```
100
101 //Função utilizada para reposicionar a chave
102 void DecrementaChave(Huffman **a, int i, Huffman *chave)
103 {
104     if (a[i] && chave->frequencia > a[i]->frequencia) //caso a chave armazenada no heap seja maior que a chave atual
105     {
106         printf("A chave é maior do que a chave atual.");
107         exit(1);
108     }
109     free(a[i]);   //libera a memória antiga da chave atual
110     a[i] = chave; //recebe a chave
111
112     /*enquanto a posição do heap for maior do que 1 e a frequencia do pai do elemento atual for menor do que
113     a frequencia dele, o laço é executado*/
114     while (i > 1 && a[Pai(i)]->frequencia > a[i]->frequencia)
115     {
116         //troca o elemento de posição com o pai dele
117         Huffman *aux = a[i];
118         a[i] = a[Pai(i)];
119         a[Pai(i)] = aux;
120         //continua o laço, mas agora com o pai do nó atual
121         i = Pai(i);
122     }
123 }
124
```

IMAGEM 12: Implementação da função “decrementaChave”

4.2.8 Inserir Heap Mínimo

A função “inserirHeapMinimo” é utilizada para inserir um novo elemento no heap. Ela recebe como parâmetro o heap onde será inserido, a chave a ser inserida e um ponteiro que aponta para a variável que armazena o tamanho do heap. A função é iniciada incrementando o tamanho do heap em uma unidade, já que agora

ele irá possuir um elemento a mais. Depois disso, é alocada memória para um novo nó para que seja possível atribuir o valor INT_MAX (+infinito) a frequência do novo espaço livre do heap e depois realizar as comparações necessárias na função “decrementaChave”. Por fim, a função “decrementaChave” é chamada para posicionar a chave no heap.

```
125  //Função para inserir um elemento em um heap
126  void InserirHeapMinimo(Huffman **a, Huffman *chave, int *tamanho)
127  {
128      //incrementa o tamanho do heap
129      *tamanho = *tamanho + 1;
130
131      //aloca memória para o nó na nova posição
132      a[*tamanho] = (Huffman *)malloc(sizeof(Huffman));
133
134      if (!a[*tamanho]) //caso ocorra erro de alocação de memória
135      {
136          printf("Erro de alocacao de memoria.");
137          exit(1);
138      }
139
140      //atribui ao novo espaço vazio no vetor o maior valor possivel
141      a[*tamanho]->frequencia = INT_MAX;
142
143      //chama função DecrementaChave
144      DecrementaChave(a, *tamanho, chave);
145  }
146
```

IMAGEM 13: Implementação da função “inserirHeapMinimo”

4.2.9 Cria Heap Mínimo

Essa função é usada para inserir os elementos do arquivo no heap. Ela recebe como parâmetro a tabela hash que relaciona cada caractere com o número de repetições dele, e uma variável que armazena o número de letras distintas que se encontram no arquivo.

A função é iniciada alocando a memória para o heap. Como o heap irá armazenar os elementos distintos e suas repetições, e o heap começa na posição 1 do vetor e não na posição 0, foi alocada a memória do tamanho “letrasDistintas” + 1 para o heap. Depois disso, um laço é criado para percorrer todos os caracteres da tabela ascii. Caso o caractere esteja presente no arquivo (linha 155), é criado um novo nó com aquele elemento e esse nó é inserido no heap através da função InserirHeapMinimo. Por fim, a função retorna o vetor em que se encontra o heap.

```
147  //Função para inserir os elementos distintos, obtidos do arquivo, no heap
148  Huffman **criaHeapMin(int *repeticoes, int letrasDistintas)
149  {
150      //aloca memória para um vetor de ponteiros do tipo huffman, que será o heap mínimo
151      Huffman **heapMin = (Huffman **)calloc((letrasDistintas + 1), sizeof(Huffman *));
152      int tamanhoFila = 0;
153      for (int i = 0; i < 256; i++) //percorre todas os 256 caracteres da tabela ascii
154      {
155          if (repeticoes[i] > 0) //caso o caractere esteja no txt
156          {
157              //cria um novo nó da árvore com o caractere e o número de repetições dele
158              Huffman *novoNo = criaNo((unsigned char)i, repeticoes[i]);
159              if (!novoNo) //caso ocorra erro de alocação de memória
160              {
161                  printf("Erro de alocacao de memoria");
162                  exit(1);
163              }
164
165              //insere o novo nó criado no heap mínimo
166              InserirHeapMinimo(heapMin, novoNo, &tamanhoFila);
167          }
168      }
169      //retorna o heap criado
170      return heapMin;
171  }
172
```

IMAGEM 14: Implementação da função “criaHeapMin”

4.2.10 Cria Heap Estático

A função “criaHeapEstatico” é a responsável por copiar as informações dos nós da árvore que estão armazenados no heap para um vetor estático de forma que ele possa, posteriormente, ser armazenado no arquivo. Ela recebe como parâmetro o vetor em que está armazenado o heap, e uma variável que contém o tamanho desse heap.

Depois disso, é alocada memória para um vetor que, assim como o heap, possui tamanho “tamanhoHeap” + 1 devido ao fato do heap começar na posição 1 do vetor e não na posição 0. O heap que foi passado como parâmetro é percorrido e o vetor recebe as informações do caractere e da frequência que estão salvas nele.

Por fim, a função poderá retornar o vetor (heap) que agora contém os dados que serão salvos no arquivo.

```
173  /*Função que copia as informações dos nós da árvore armazenados no heap por meio de ponteiros
174  para um vetor estático (que não seja um vetor de ponteiros) que possa ser armazenado no arquivo*/
175  HuffmanArquivo *criaHeapEstatico(Huffman **heapMin, int tamanhoHeap)
176  {
177      //aloca a memória para o vetor que será o heap
178      HuffmanArquivo *heapArquivo = (HuffmanArquivo *)malloc((tamanhoHeap + 1) * sizeof(HuffmanArquivo));
179      for (int i = 1; i <= tamanhoHeap; i++) //percorre todo o heap
180      {
181          //atribui os valores ao heap
182          heapArquivo[i].caractere = heapMin[i]->caractere;
183          heapArquivo[i].frequencia = heapMin[i]->frequencia;
184      }
185      //retorna o heap com os dados dos caracteres pronto para ser inserido no arquivo
186      return heapArquivo;
187  }
188
```

IMAGEM 15: Implementação da função “criaHeapEstatico”.

4.2.11 Cria Hash Heap

A função “criaHashHeap” é utilizada para criar uma tabela hash que relaciona o caractere com um ponteiro que aponta para o nó que guarda aquele caractere na

árvore. Essa função recebe como parâmetro uma variável contendo o tamanho do heap e um vetor em que está contido o heap.

A função inicialmente aloca a memória para um vetor com 256 posições, para que seja possível abranger todos os 256 caracteres da tabela ascii (do 0 ao 255). Com isso, é criado um laço para percorrer o heap e, para cada caractere que se encontra será feita uma associação com um ponteiro que aponta para o nó da árvore em que esse caractere se encontra (linha 199). Por fim, o retorno da função será a tabela hash que contém essa relação. A implementação dessa função se encontra na imagem abaixo.

```
188
189  /*Função utilizada para criar uma tabela hash que relaciona cada caractere com um
190  ponteiro que aponta para o nó que guarda aquele caractere na árvore*/
191  Huffman **criaHashHeap(Huffman **heap, int tamanhoHeap)
192  {
193      //aloca memória para a tabela hash
194      Huffman **hashHeap = (Huffman **)calloc(256, sizeof(Huffman *));
195
196      for (int i = 1; i <= tamanhoHeap; i++) //laço para percorrer o heap
197      {
198          //para cada caractere no heap, armazena na tabela o ponteiro que aponta para o nó dele na árvore
199          hashHeap[heap[i]->caractere] = heap[i];
200      }
201      //retorna a tabela hash
202      return hashHeap;
203  }
204
```

IMAGEM 16: Implementação da função “criaHashHeap”.

4.2.12 Cria Árvore De Huffman

Essa função é utilizada para a criação da árvore utilizada no algoritmo de huffman para gerar o código binário relacionado a cada elemento do arquivo. A função recebe como parâmetro o heap mínimo com os elementos e uma variável com o tamanho do heap.

A função consiste basicamente em uma estrutura de repetição while que repete o mesmo procedimento enquanto o número de elementos no heap for maior do que 1. O procedimento executado dentro do while consiste basicamente em retirar os dois menores elementos do heap e armazená-los em variáveis. Depois disso, um novo nó é criado para unir os dois menores elementos. Para isso, o filho à esquerda do novo nó se torna o primeiro elemento removido do heap, e o pai dele também se torna o novo nó. Além disso, ele é marcado como filho à esquerda (tipoFilho = 0). O mesmo é feito para o segundo elemento removido do heap, ele se torna o filho à direita do novo nó, o novo nó se torna o pai dele e ele é marcado como filho à direita (tipoFilho = 1). Por fim a função será chamada a função “InserirHeapMinimo” para que esse novo nó seja inserido no heap.

Quando o heap tiver apenas um elemento, esse elemento será retornado pois ele é o nó raiz da árvore formada. A imagem abaixo mostra como ficou a implementação dessa função.

```c
Huffman *criaArvoreHuffman(Huffman **heapMin, int tamanhoHeap)
{
    while (tamanhoHeap > 1) //enquanto o tamanho do heap for maior do que 1
    {
        //retira os dois elementos de menor frequência do heap
        Huffman *m1 = RetiraMinimo(heapMin, &tamanhoHeap);
        Huffman *m2 = RetiraMinimo(heapMin, &tamanhoHeap);

        //aloca memória para um novo nó para unir m1 e m2
        Huffman *novoNo = criaNo((unsigned char)0, m1->frequencia + m2->frequencia);

        //realiza os apontamentos
        novoNo->pai = NULL; //garante que o novo nó não terá pai (pois a raiz não tem pai)

        m1->tipoFilho = 0;              //m1 será o filho á esquerda do novo nó (0)
        m1->pai = novoNo;               //atualiza o pai de m1
        novoNo->subarvoreEsquerda = m1; //faz m1 ser o filho à esquerda do novo nó

        m2->tipoFilho = 1;             //m2 será o filho à direita do novo nó (1)
        m2->pai = novoNo;              //atualiza o pai de m2
        novoNo->subarvoreDireita = m2; //faz m2 ser o filho à direita do novo nó

        //Insere o novo nó no heap
        InserirHeapMinimo(heapMin, novoNo, &tamanhoHeap);
    }
    //retorna a raiz da árvore caso o heap só tenha um elemento (a raiz)
    return RetiraMinimo(heapMin, &tamanhoHeap);
}
```

IMAGEM 17: Implementação da função “criaArvoreHuffman”.

4.2.13 Cria Vetor de Repetições

A função “criaVetorRepeticoes” é responsável pela criação da tabela hash que irá relacionar cada caractere com o número de repetições dele no arquivo. Ela vai receber como parâmetro o nome do arquivo, um ponteiro que aponta para a variável que armazena a quantidade de letras distintas e um ponteiro que aponta para a variável que armazena a quantidade de caracteres presentes no arquivo.

Inicialmente essa função aloca a memória para um vetor de 256 inteiros. Esse vetor será a tabela hash que irá relacionar cada caractere da tabela ascii (do 0 ao 255) ao número de repetições dele no arquivo.

Depois disso, o arquivo é aberto. Caso ocorra algum erro ao abrir o arquivo, será exibida uma mensagem de erro e será encerrada a execução do programa. Caso contrário, o arquivo é lido até o fim percorrendo caractere por caractere.

Caso a quantidade de repetições de um caractere seja 0, ou seja ainda não tenha sido encontrado, a variável que armazena a quantidade de letras distintas é incrementada. Depois disso, a quantidade de repetições daquele caractere é incrementada. Além disso, a quantidade de caracteres no arquivo também é incrementada. Por fim, o arquivo é fechado e a função retorna a tabela hash. A implementação dessa função se encontra na imagem abaixo.

```
235  //Função utilizada para criar uma tabela hash que relaciona o caractere com o número de repetições dele no arquivo
236  int *criaVetorRepeticoes(char *nomeArquivo, int *letrasDistintas, int *quantidadeCaracteres)
237  {
238      char ch;
239      //aloca a memória para a tabela hash com os 256 caracteres da tabela ascii
240      int *repeticoes = (int *)calloc(256, sizeof(int));
241
242      //leitura do arquivo
243      FILE *fp = fopen(nomeArquivo, "r");
244      if (!fp) //caso não encontre o arquivo
245      {
246          printf("\nErro ao abrir o arquivo. Esse arquivo nao foi encontrado.\n");
247          exit(1);
248      }
249      while ((ch = fgetc(fp)) != EOF) //lê o arquivo caractere por caractere
250      {
251          unsigned char c = ch;      //utilização do unsigned para garantir que os caracteres tenham apenas códigos positivos
252          if (repeticoes[c] == 0)    //caso esse caractere ainda não tenha sido encontrado no arquivo
253              (*letrasDistintas)++;  //incrementa o número de caracteres distintos
254          repeticoes[c]++;           //incrementa o número de repetições daquele caractere na tabela hash
255          (*quantidadeCaracteres)++; //incrementa o contador da quantidade de caracteres do arquivo
256      }
257      fclose(fp);        //fecha o arquivo
258      return repeticoes; //retorna a tabela hash
259  }
260
```

IMAGEM 18: Implementação da função “criaVetorRepeticoes”.

4.2.14 Codifica Char

A função “codificaChar” é a função responsável por, através da árvore de huffman, preencher um byte. Essa função vai receber como parâmetro um ponteiro para o nó folha de uma árvore que contém o caractere que será codificado, uma variável que armazena o código binário em forma de um vetor de bytes, uma váriavel que controla a posição do vetor (em qual byte) está se inserindo, uma variável que contabiliza o número de bits que já foram ocupados daquele byte, outra variável que incrementa o tamanho do código binário e uma variável que contém a quantidade de letras distintas presentes no arquivo.

Essa função é uma função recursiva que percorre a árvore do nó folha até a raiz (linhas 264 e 267). Quando ela chega na raiz, a recursão começa a retornar para a folha e é nesse momento que o código binário é gerado. Inicialmente é verificado se o byte já está cheio. Caso ele esteja, desloca-se para a próxima posição do vetor (próximo byte) e a variável que controla o número de bits ocupados daquele byte é reiniciada para 0 (linhas 270 e 271). Depois disso, um bit é deslocado para a esquerda para que se insira o bit 0 na posição mais à direita daquele byte. e outra comparação é feita para saber se esse nó é um filho à direita do pai dele. Caso ele seja, é realizada uma operação para que esse bit 0, que foi inserido anteriormente na última posição do byte, se transforme no bit 1 (linhas 274 e 275). Por fim, o tamanho do código binário e o número de bits ocupados daquele byte são incrementados.

Caso o nó da árvore não possua pai e apenas uma letra distinta, ou seja, caso a árvore seja composta apenas por um nó, é feito o mesmo procedimento mas sem realizar as recursões em que se desloca o ponteiro para o pai do nó. A implementação do código descrito se encontra na imagem abaixo.

```
261  //Função para preencher o byte através da árvore
262  void codificaChar(Huffman *arvore, char *codBin, int *posicaoVetor, int *numeroBits, int *tamanhoBinario, int letrasDistintas)
263  {
264      if (arvore->pai) //enquanto o nó da árvore tiver pai
265      {
266          //chama a função recursivamente percorrendo da folha até a raiz
267          codificaChar(arvore->pai, codBin, posicaoVetor, numeroBits, tamanhoBinario, letrasDistintas);
268          if (*numeroBits == 8) //se o byte encher
269          {
270              (*posicaoVetor)++; //desloca para a próxima posição do vetor
271              *numeroBits = 0;   //começa o processo novamente no bit 0
272          }
273          codBin[*posicaoVetor] = codBin[*posicaoVetor] << 1;    //desloca os bits para a esquerda
274          if (arvore->tipoFilho)                                 //se for um filho à direita
275              codBin[*posicaoVetor] = codBin[*posicaoVetor] ^ 1; //escreve o bit 1 no ultimo bit
276          (*numeroBits)++;                                       //numero de bits é incrementado
277          (*tamanhoBinario)++;                                   //tamanho do binário é incrementado
278      }
279      else if (!arvore->pai && letrasDistintas == 1) //caso a árvore só tenha um nó que é a própria raiz
280      {
281          if (*numeroBits == 8) //caso o byte esteja cheio
282          {
283              (*posicaoVetor)++; //desloca para a próxima posição do vetor
284              *numeroBits = 0;   //rertorna para o primeiro bit
285          }
286          codBin[*posicaoVetor] = codBin[*posicaoVetor] << 1;    //desloca os bits para a esquerda
287          if (arvore->tipoFilho)                                 //se for um filho à direita
288              codBin[*posicaoVetor] = codBin[*posicaoVetor] ^ 1; //escreve o bit 1 no ultimo bit
289          (*numeroBits)++;                                       //incrementa o numero de bits
290          (*tamanhoBinario)++;                                   //incrementa o tamanho do binário
291      }
292  }
```

IMAGEM 19: Implementação da função “codificaChar”.

4.2.15 Cria Binário

A função “criaBinario” é utilizada para ler os caracteres do arquivo e realizar as conversões deles para código binário. Essa função recebe como parâmetro o nome de um arquivo a ser lido, um heap mínimo, uma variável que contém o número de letras presentes no arquivo, um ponteiro para a variável que armazena o tamanho do arquivo binário, e uma variável com a quantidade de letras distintas.

Inicialmente, o arquivo é aberto em modo leitura e ocorre então um tratamento para o caso de ocorrer algum erro ao abrir o arquivo (linhas 299 e 300). Depois de aberto, o arquivo é lido caractere por caractere (linha 308). Ao ler o arquivo, é realizada a conversão do caractere lido para unsigned char para evitar que ao ler, venha um caractere com código negativo, e isso cause algum problema nas tabelas hash (linha 311). Depois disso, um ponteiro aponta para a folha em que se encontra aquele caractere por meio da tabela hash (linha 313), e é chamada a função “codificaChar” para inserir os bits correspondentes àquele caractere no vetor de bytes. Por fim, o arquivo é fechado e o retorno é o código binário do caractere.

```
294  //Função para converter os caracteres do arquivo para o código binário
295  char *criaBinario(char *nomeArquivo, Huffman **hashHeap, int numeroLetras, int *tamanhoBinario, int letrasDistintas)
296  {
297      char ch;
298      char *codBin = (char *)calloc(numeroLetras, sizeof(char));
299      FILE *fp = fopen(nomeArquivo, "r"); //abre o arquivo para leitura
300      if (!fp)                            //caso ocorra erro para abrir o arquivo
301      {
302          printf("Erro ao abrir o arquivo.");
303          exit(1);
304      }
305
306      int bitsOcupados = 0;
307      int posicaoVetor = 0;
308      while ((ch = fgetc(fp)) != EOF) //lê o arquivo caractere por caractere
309      {
310          //utilizou-se o unsigned para garantir que os códigos dos caracteres sejam inteiros
311          unsigned char c = ch;
312          //ponteiro que aponta para a folha que contém aquele caractere
313          Huffman *arvore = hashHeap[c];
314          //acrescenta em codBin o código binário correspondente ao caractere lido
315          codificaChar(arvore, codBin, &posicaoVetor, &bitsOcupados, tamanhoBinario, letrasDistintas);
316      }
317      fclose(fp);    //fecha o arquivo
318      return codBin; //retorna o código binário
319  }
320
```

IMAGEM 20: Implementação da função “criaBinario”.

4.2.16 Comprimir

A função “comprimir” é utilizada para comprimir um arquivo. Ela recebe como parâmetro o nome do arquivo a ser comprimido, a estrutura do cabeçalho, o heap com as informações do caractere e o código binário gerado a partir da leitura do arquivo e conversão dos caracteres para sequências de bits.

A função inicia salvando em uma variável de controle a quantidade de bytes que será necessário para armazenar o código binário, caso a divisão não seja exata, um byte extra será necessário.

O arquivo então é aberto em formato de escrita binária. A primeira coisa a ser gravada é o cabeçalho, na sequência o heap com as informações dos caracteres e, por fim, o código binário. Após isso, o arquivo pode ser fechado.

```
321  //Função utilizada para comprimir o arquivo
322  void comprimir(char *nomeArquivo, Cabecalho cab, HuffmanArquivo *heapEstatico, char *codigoBinario)
323  {
324
325      //quantidade de bytes necessários para armazenar o binário
326      int numeroBytes = cab.tamanhoBinario / 8;
327      //se a divisão não for exata, é necessário utilizar mais um byte para armazenar o resto dos bits
328      if (cab.tamanhoBinario % 8 > 0)
329      {
330          numeroBytes++;
331      }
332
333      FILE *fp = fopen(nomeArquivo, "wb+");                                  //abre o aquivo para escrever em binário
334      fwrite(&cab, sizeof(Cabecalho), 1, fp);                                //escreve o cabeçalho
335      fwrite(heapEstatico, sizeof(HuffmanArquivo), cab.tamanhoHeap + 1, fp); //escreve o heap
336      fwrite(codigoBinario, sizeof(char), numeroBytes, fp);                  //escreve o código binário
337      fclose(fp);                                                            //fecha o arquivo
338  }
339
```

IMAGEM 21: Implementação da função comprimir.

4.2.17 Descomprimir

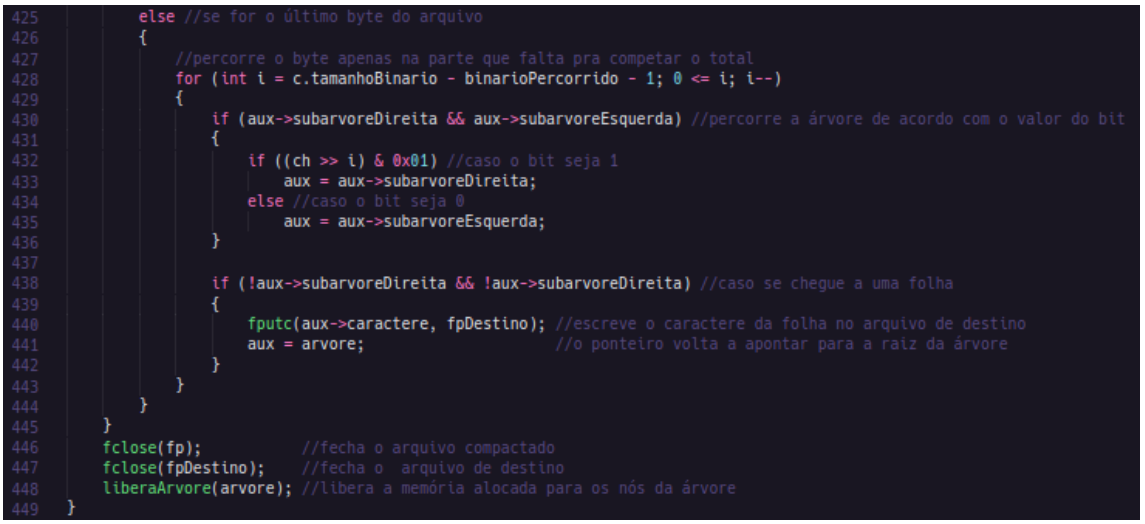
A função “descomprimir” é a função utilizada para realizar a descompressão de um arquivo binário, que foi codificado com este algoritmo, novamente para um arquivo de texto com as mesmas informações que o arquivo original que foi compactado continha. Essa função recebe apenas o nome do arquivo compactado e o nome do arquivo de destino. A implementação dela se encontra na imagem abaixo.

```c
void descomprimir(char *arquivoCompactado, char *arquivoDestino)
{
    Cabecalho c;

    //Abre o arquivo com o modo de leitura binária
    FILE *fp = fopen(arquivoCompactado, "rb");
    if (!fp) //caso ocorra algum problema para abrir o arquivo
    {
        printf("\nErro ao abrir o arquivo. O arquivo %s nao existe.\n", arquivoCompactado);
        exit(1);
    }

    fread(&c, sizeof(Cabecalho), 1, fp); //lê o cabeçalho do arquivo
    fclose(fp);                          //fecha o arquivo

    HuffmanArquivo heapArquivo[c.tamanhoHeap + 1];

    fp = fopen(arquivoCompactado, "rb"); //Abre o arquivo
    if (!fp)                             //caso ocorra algum erro na abertura
    {
        printf("\nErro ao abrir o arquivo. O arquivo %s nao existe.\n", arquivoCompactado);
        exit(1);
    }

    //posiciona o ponteiro do arquivo na posição certa depois do cabeçalho
    fseek(fp, sizeof(Cabecalho), SEEK_SET);
    fread(&heapArquivo, sizeof(HuffmanArquivo), c.tamanhoHeap + 1, fp); //lê o heap do arquivo
    fclose(fp);

    Huffman *heapMin[c.tamanhoHeap + 1];

    /*como o heap do arquivo não era de ponteiros, o heap foi convertido para
    um heap de ponteiros para nó da árvore para poder criar a árvore novamente*/
    for (int i = 1; i <= c.tamanhoHeap; i++)
    {
        heapMin[i] = criaNo(heapArquivo[i].caractere, heapArquivo[i].frequencia);
    }

    Huffman *arvore = criaArvoreHuffman(heapMin, c.tamanhoHeap); //recria a árvore

    int binarioPercorrido = 0;
    unsigned char ch;
    Huffman *aux = arvore;

```

IMAGEM 22.1: Implementação da função “descomprimir”.

```c
    /*abre o arquivo de texto onde serão gravadas as informações do arquivo
    compactado. Caso ele não exista, ele será criado*/
    FILE *fpDestino = fopen(arquivoDestino, "w+");
    if (!fp) //caso ocorra algum erro na abertura
    {
        printf("\nErro ao abrir o arquivo. O arquivo %s nao existe.\n", arquivoDestino);
        exit(1);
    }

    fp = fopen(arquivoCompactado, "rb"); //abre o arquivo compactado
    if (!fp)                             //caso ocorra algum erro
    {
        printf("\nErro ao abrir o arquivo. O arquivo %s nao existe.\n", arquivoCompactado);
        exit(1);
    }
    //posiciona o ponteiro do arquivo na posição certa depois do cabeçalho e do heap
    fseek(fp, sizeof(Cabecalho) + (c.tamanhoHeap + 1) * sizeof(HuffmanArquivo), SEEK_SET);

    while (fread(&ch, sizeof(unsigned char), 1, fp)) //lê o arquivo byte por byte
    {
        if (binarioPercorrido + 8 < c.tamanhoBinario) //se não for o último byte do arquivo
        {
            for (int i = 7; 0 <= i; i--) //percorre todos os bits daquele byte
            {
                if (aux->subarvoreDireita && aux->subarvoreEsquerda) //percorre a árvore dependendo do valor do bit
                {
                    if ((ch >> i) & 0x01) //caso o bit seja 1
                        aux = aux->subarvoreDireita;
                    else //caso o bit seja 0
                        aux = aux->subarvoreEsquerda;
                }

                if (!aux->subarvoreDireita && !aux->subarvoreDireita) //caso se chegue a uma folha
                {
                    fputc(aux->caractere, fpDestino); //escreve o caractere da folha no arquivo de destino
                    aux = arvore;                     //o ponteiro volta a apontar para a raiz da árvore
                }
            }
            binarioPercorrido += 8; //incrementa em um byte a quantidade de bits que já foi percorrida
        }
```

IMAGEM 22.2: Implementação da função “descomprimir”.



```
425         else //se for o último byte do arquivo
426         {
427             //percorre o byte apenas na parte que falta pra competar o total
428             for (int i = c.tamanhoBinario - binarioPercorrido - 1; 0 <= i; i--)
429             {
430                 if (aux->subarvoreDireita && aux->subarvoreEsquerda) //percorre a árvore de acordo com o valor do bit
431                 {
432                     if ((ch >> i) & 0x01) //caso o bit seja 1
433                         aux = aux->subarvoreDireita;
434                     else //caso o bit seja 0
435                         aux = aux->subarvoreEsquerda;
436                 }
437 
438                 if (!aux->subarvoreDireita && !aux->subarvoreDireita) //caso se chegue a uma folha
439                 {
440                     fputc(aux->caractere, fpDestino); //escreve o caractere da folha no arquivo de destino
441                     aux = arvore;                     //o ponteiro volta a apontar para a raiz da árvore
442                 }
443             }
444         }
445     }
446     fclose(fp);           //fecha o arquivo compactado
447     fclose(fpDestino);    //fecha o  arquivo de destino
448     liberaArvore(arvore); //libera a memória alocada para os nós da árvore
449 }
```

IMAGEM 22.3: Implementação da função “descomprimir”.

Nessa função, inicialmente é feita a abertura do arquivo binário para leitura e é feita uma verificação para apresentar uma mensagem de erro e encerrar o programa caso não seja possível abrir o arquivo nas linhas 346 e 347. Depois disso, é feita a leitura do cabeçalho do arquivo e o arquivo é fechado nas linhas 353 e 354. Em seguida, as informações do cabeçalho são utilizadas para alocar a memória para o heap, e o heap é lido na linha 367. Vale ressaltar que o fseek também foi utilizado para que a leitura do heap comece a partir do ponto em que a leitura do cabeçalho começou.

Depois disso, da linha 374 a 379, o heap lido é utilizado para que seja possível montar a árvore novamente. Essa árvore será utilizada posteriormente para decodificar os códigos binários que estão presentes no arquivo.

Logo em seguida, o arquivo binário é aberto novamente e o arquivo de destino também é aberto. Novamente é necessário fazer o uso do fseek na linha 401 para que a leitura do arquivo binário se inicie depois do cabeçalho e do heap.

Com o arquivo aberto, inicia-se a leitura byte por byte na linha 403. Quando se inicia a leitura, já foi criado um ponteiro que está apontando para a raiz da árvore. Esse ponteiro é importante pois, para cada byte que for lido, será percorrido cada bit presente nesse byte (linha 407), e a árvore será percorrida. Caso o bit seja 1, desloca-se para o filho à direita do nó e caso o bit seja 0, desloca-se para o filho à esquerda do nó (linhas 411 a 414). Esse processo é feito até chegar em uma folha.

Quando se chega em uma folha, isso indica que um caractere foi encontrado, então esse caractere é escrito no arquivo de destino e o ponteiro da árvore volta a apontar para a raiz da árvore (linhas 419 e 420). Esse processo é feito para cada byte lido. Entretanto, no último byte é necessário realizar um tratamento diferente pois pode ser que ele não esteja completamente cheio, então é necessário percorrer apenas os bits ocupados dele. O resto do processamento para esse caso é idêntico ao processamento feito para um byte cheio (linhas 425 a 445).

Por fim, os arquivos são fechados e a memória alocada para cada nó da árvore é liberada.

4.2.18 Libera árvore

A função “liberaArvore” é uma função auxiliar utilizada para liberar a memória que foi alocada para cada nó da árvore.

Essa função recebe como parâmetro um ponteiro para o nó raiz da árvore e, através de recursão, percorre toda a árvore liberando a memória de cada nó. A implementação dessa função se encontra na imagem abaixo.

```
472   //Função utilizada para liberar a mamória alocada para uma árvore
473   void liberaArvore(Huffman *a)
474   {
475       if (a) //se a árvore não estiver vazia
476       {
477           liberaArvore(a->subarvoreEsquerda); //percorre a subárvore da esquerda
478           liberaArvore(a->subarvoreDireita);  //percorre a subárvore da direita
479           free(a);                            //libera a memória do nó
480       }
481   }
482
```

IMAGEM 23: Implementação da função “liberaArvore”.

4.2.19 Imprime Binário Caractere

A função “imprimeBinarioCaractere” é utilizada para imprimir a sequência binária de uma caractere através da árvore. Essa função vai receber como parâmetro o ponteiro para o nó folha da árvore que contenha o caractere cujo código binário será impresso e uma variável que armazena a quantidade de letras distintas.

Essa função é uma função recursiva. Por isso, a recursão é utilizada para percorrer a árvore da folha até a raiz. Quando se chega na raiz, a função passa a voltar da raiz até a folha. Nesse processo, é feita a impressão do código binário daquele caractere com base na árvore, sendo que o código binário vai depender apenas do fato do nó ser filho à direita ou à esquerda do pai dele.

Outro caso tratado é quando a árvore só tem apenas um elemento que é a folha e a raiz ao mesmo tempo. Nesse caso, basta imprimir o tipo de filho que ele é (sendo que cada novo nó tem o tipoFilho inicializado como 0 por padrão, logo a raiz também será assim). Essa função não possui retorno e a implementação dela ficou da seguinte forma:

```
496   //Função utilizada para imprimir a sequência binária que representa um caractere através da árvore
497   void imprimeBinarioCaractere(Huffman *arvore, int letrasDistintas)
498   {
499       if (arvore->pai) //se o nó tiver pai
500       {
501           imprimeBinarioCaractere(arvore->pai, letrasDistintas); //chama a função recursivamete para o pai do nó
502           printf("%d", arvore->tipoFilho);                       //imprime o bit
503       }
504       else if (!arvore->pai && letrasDistintas == 1) //caso a árvore não tenha pai e o arquivo tenha apenas uma letra distinta
505       {
506           printf("%d", arvore->tipoFilho); //imprime o bit
507       }
508   }
```

IMAGEM 24: Implementação da função “imprimeBinarioCaractere”.

4.3 main.c

No arquivo cliente, o main.c é onde se encontra a função de menu e a função principal do programa: a main. Esse arquivo também possui a inclusão da biblioteca “Huffman.h”, para que seja possível realizar todas as operações de compactação e descompactação no arquivo, como foi solicitado.

4.3.1 Menu

A função menu é utilizada para que o usuário solicite a operação que deseja realizar. Essa função recebe como parâmetro uma variável que indica a opção escolhida pelo usuário.

As operações a serem realizadas na árvore são escolhidas através desse menu que, no programa feito, está na seguinte ordem:

1 - Comprimir arquivo
2 - Descomprimir arquivo
3 - Sair

O programa é executado e então as opções aparecem na tela para que o usuário escolha a que ele deseja. Depois de escolher, a opção será passada para a função de menu e, a partir de um switch case, será realizada a chamada das funções necessárias para que se realize o que foi solicitado.

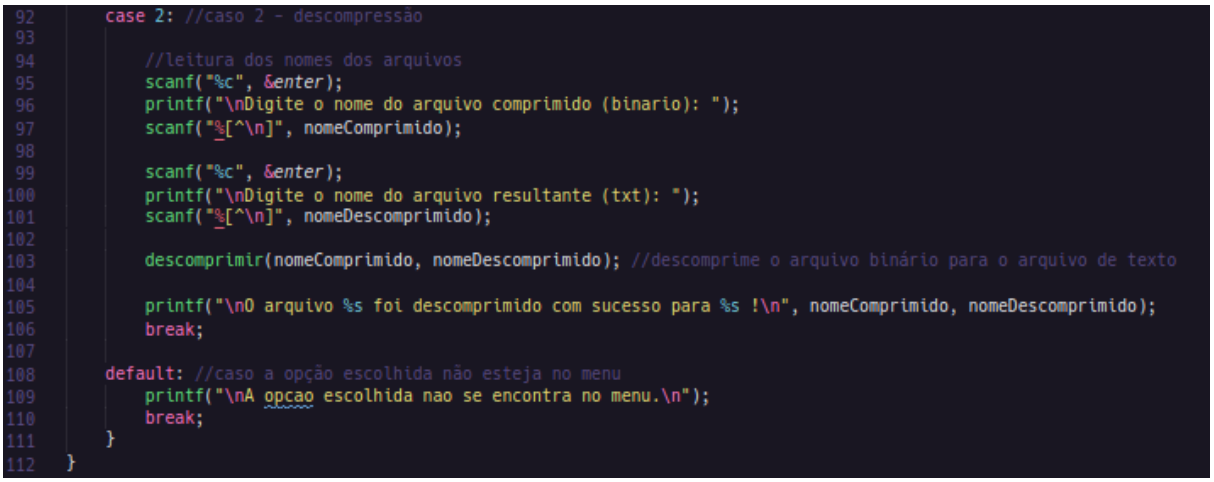
Essa função também trata o caso de ser escolhida uma opção que não existe no menu. Caso isso ocorra, uma mensagem de erro é exibida e é feita a leitura da opção novamente. A imagem abaixo mostra a implementação da função menu:

```c
//Função do menu
void menu(int op)
{
    //Declaração das váriaveis
    char nomeOrigem[100];
    char nomeSaida[100];
    char nomeComprimido[100];
    char nomeDescomprimido[100];
    char enter;
    int letrasDistintas = 0;
    int numeroLetras = 0;
    int tamanhoBinario = 0;

    //switch case para as opções que podem ser escolhidas
    switch (op)
    {
    case 1: //caso 1 - compressão
        //leitura do nome do arquivo que será comprimido
        scanf("%c", &enter);
        printf("\nDigite o nome do arquivo a ser comprimido (txt): ");
        scanf("%[^\n]", nomeOrigem);

        //criação da tabela hash que armazena as repetições dos caracteres
        int *ocorrenciasCaracteres = criaVetorRepeticoes(nomeOrigem, &letrasDistintas, &numeroLetras);

        if (letrasDistintas == 0) //Caso o arquivo não possua letras distintas ele está vazio
        {
            printf("\nO arquivo está vazio!\n");
        }
        else
        {
            //cria o heap mínimo
            Huffman **heapMin = criaHeapMin(ocorrenciasCaracteres, letrasDistintas);
            //cria uma cópia do heap mínimo em forma de vetor estático para mandar para o arquivo
            HuffmanArquivo *heapArquivo = criaHeapEstatico(heapMin, letrasDistintas);
            //cria a tabela hash que relaciona cada caractere com o nó que contém ele na árvore
            Huffman **hashHeap = criaHashHeap(heapMin, letrasDistintas);

            Huffman *arvore = criaArvoreHuffman(heapMin, letrasDistintas); //cria a árvore de Huffman

            printf("\nCódigo de Huffman: \n");
            printf("\nElemento [numero de ocorrencias] : codigo binario\n");
```

IMAGEM 25.1:Implementação da função “menu”.

```c
            for (int i = 0; i < 256; i++) //percorre todos os caracteres da tabela ascii
            {
                if (ocorrenciasCaracteres[i] > 0) //caso o caractere possua alguma ocorrência no arquivo
                {
                    //imprime o caractere
                    printf("%c[%d] : ", i, ocorrenciasCaracteres[i]);
                    //imprime o código binário dele extraído da árvore
                    imprimeBinarioCaractere(hashHeap[i], letrasDistintas);
                    printf("\n");
                }
            }

            //converte todo o texto do arquivo de entrada para binário por meio da árvore de Huffman
            char *codigoBinario = criaBinario(nomeOrigem, hashHeap, numeroLetras, &tamanhoBinario, letrasDistintas);

            printf("\n(O codigo de Huffman e a contagem das ocorrencias dos caracteres foram impressos acima)\n");
            //leitura do nome do arquivo de saída
            scanf("%c", &enter);
            printf("\nDigite o nome do arquivo de saida (binario): ");
            scanf("%[^\n]", nomeSaida);

            Cabecalho c;
            c.tamanhoBinario = tamanhoBinario;
            c.tamanhoHeap = letrasDistintas;

            //salva o cabeçalho, o heap e o binário no arquivo compactado
            comprimir(nomeSaida, c, heapArquivo, codigoBinario);

            printf("\nO arquivo %s foi comprimido com sucesso para %s !\n", nomeOrigem, nomeSaida);

            //libera memória alocada
            free(heapMin);
            free(heapArquivo);
            free(codigoBinario);
            liberaArvore(arvore);
        }
        //liberação de memória
        free(ocorrenciasCaracteres);

        break;
```

IMAGEM 25.2: Implementação da função “menu”.



```
92      case 2: //caso 2 - descompressão
93
94          //leitura dos nomes dos arquivos
95          scanf("%c", &enter);
96          printf("\nDigite o nome do arquivo comprimido (binario): ");
97          scanf("%[^\n]", nomeComprimido);
98
99          scanf("%c", &enter);
100         printf("\nDigite o nome do arquivo resultante (txt): ");
101         scanf("%[^\n]", nomeDescomprimido);
102
103         descomprimir(nomeComprimido, nomeDescomprimido); //descomprime o arquivo binário para o arquivo de texto
104
105         printf("\nO arquivo %s foi descomprimido com sucesso para %s !\n", nomeComprimido, nomeDescomprimido);
106         break;
107
108     default: //caso a opção escolhida não esteja no menu
109         printf("\nA opcao escolhida nao se encontra no menu.\n");
110         break;
111     }
112 }
```

IMAGEM 25.3: Implementação da função “menu”.

Quando a opção escolhida é a opção 1, ou seja, a opção de compactar arquivo, o menu é chamado e com o switch case para realizar a operação correta de acordo com o que foi escolhido. No caso 1 ocorrem as chamadas das funções necessárias para que isso ocorra. Nesse caso, a primeira coisa a se fazer é a leitura do nome do arquivo que será comprimido, com isso, o programa verifica se o arquivo é válido, caso não seja, uma mensagem de erro é apresentada na tela. Depois disso, será gerada a tabela hash de repetições, o heap mínimo e a árvore de Huffman.

Em seguida, será impresso uma listagem contendo o caractere, a quantidade de ocorrências desse caractere e o código binário referente ao caractere.

Depois disso, é gerado o vetor de bytes que será enviado para o arquivo e é solicitado o nome do arquivo de saída para que, assim, a função comprimir possa ser chamada e o arquivo ser comprimido. Por fim, a memória utilizada é liberada.

Caso a opção escolhida seja a opção 2, ou seja, a opção de descompactar um arquivo, a primeira coisa que acontece é a leitura do nome do arquivo binário que será descompactado e o nome do arquivo onde as informações serão gravadas. Com essas informações, a função descomprimir é chamada e uma mensagem de sucesso é apresentada.

Por fim, caso a opção escolhida não esteja no menu, uma mensagem de erro é apresentada.

4.3.2 Main

A função main é a principal função do programa já que ela é a primeira função a ser executada ao iniciar o programa. Nela, existe apenas a definição de uma variável onde será feita a leitura da opção do menu e um laço “do while” onde irá aparecer as opções e será requerido que o usuário selecione uma opção. A partir

disso a função menu será chamada caso a opção não seja a opção de sair. A imagem abaixo mostra a implementação da função main:

```c
//Função principal
int main()
{
    int op = 0;
    //Laço para escolher a opção do menu
    do
    {
        //Opções do menu
        printf("\n--------- Menu ---------\n\n");
        printf("1 - Comprimir arquivo\n");
        printf("2 - Descomprimir arquivo\n");
        printf("3 - Sair\n");

        //Leitura da opção
        printf("\nEscolha a opcao: ");
        scanf("%d", &op);

        if (op != 3)  //Se a opção selecionada não for a de sair
            menu(op); //chamada do menu

    } while (op != 3); //Permanece no menu enquanto não for selecionada a opção de sair

    return 0;
}

```

IMAGEM 26: Implementação da função “main”.

4.4 MAKEFILE

No arquivo “makefile” contém um conjunto de diretivas que serão usadas para a compilação dos arquivos. Ele permite que todo o código seja compilado utilizando apenas um comando: o “make”, tornando assim a compilação do programa mais simples e prática. A implementação do arquivo makefile se encontra na imagem abaixo.

```
all: Huffman

Huffman: main.o Huffman.o
    gcc -o Huffman main.o Huffman.o

main.o: main.c Huffman.h
    gcc -c main.c -Wall

Huffman.o: Huffman.c Huffman.h
    gcc -c Huffman.c -Wall

clean:
    rm Huffman main.o Huffman.o

```

IMAGEM 27: Implementação do arquivo makefile.

5 CONCLUSÃO

Portanto, é possível concluir que o algoritmo de Huffman é um bom modelo a ser seguido no processo de compactação de arquivos. Isso acontece justamente pois esse algoritmo visa representar os elementos com maiores repetições com um número pequeno de bits. Isso faz com que os elementos ocupem muito menos espaço na memória, e o arquivo possa ter o tamanho reduzido. Além disso, foi possível notar que o algoritmo de compactação só reduz o tamanho do arquivo caso ele seja maior do que um determinado número de bytes. Isso acontece pois esse processo de compactação requer que o cabeçalho e o heap sejam inseridos no arquivo compactado para que seja possível descompactá-lo posteriormente sem auxílio de nenhum outro arquivo externo. Como esse cabeçalho e o heap ocupam espaço no arquivo compactado, o processo de compactação só irá funcionar para arquivos um pouco maiores do que o número de bytes ocupados por esses elementos, pois também deve-se levar em conta o número de bytes ocupado pelo código binário gerado para compactar o arquivo de texto que também será inserido no arquivo de texto.

A maior dificuldade encontrada durante a implementação foi no processo de codificar os bits de cada byte utilizando operadores bit a bit para que eles possam ser enviados para o arquivo compactado mas, apesar das dificuldades, foi possível compactar o arquivo de texto para um arquivo binário e descompactar ele novamente para um arquivo de texto. A eficiência da compactação vai depender da quantidade de letras distintas presentes no arquivo e da quantidade de repetições de cada uma no arquivo. Mesmo assim, o algoritmo apresentou uma boa taxa de compactação nos arquivos testados.

6 REFERÊNCIAS

https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman

https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/huffman.html

https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html

https://pt.wikipedia.org/wiki/C_(linguagem_de_programa%C3%A7%C3%A3o)