



Universidade Federal do Espírito Santo
Centro Universitário Norte do Espírito Santo

Estrutura de Dados II

Avaliação 4 - Algoritmo de Huffman

Arthur de Andrade Ferreira
Gabriel Alves Matos

São Mateus
2021

Sumário

1	Estrutura de um nó	1
1.1	Características	1
1.2	Implementação em C	1
2	Estrutura de cabeçalho	1
2.1	Características	1
2.2	Implementação em C	2
3	Criação de nó	2
3.1	Funcionamento	2
3.2	Implementação em C	2
4	Busca ordenada na lista pela frequência	3
4.1	Funcionamento	3
4.2	Implementação em C	4
5	Inserção ordenada na lista pela frequência	4
5.1	Funcionamento	4
5.2	Implementação em C	5
6	Busca na lista por caractere	5
6.1	Funcionamento	5
6.2	Implementação em C	6
7	Inserção ordenada na lista por caractere	6
7.1	Funcionamento	6
7.2	Implementação em C	7
8	Retirar nó com frequência mínima da lista	8

8.1	Funcionamento	8
8.2	Implementação em C	9
9	Função do Algoritmo de Huffman	9
9.1	Funcionamento	9
9.2	Implementação em C	10
10	Encontrar o nó folha de um caractere	10
10.1	Funcionamento	10
10.2	Implementação em C	11
11	Obtenção do código de Huffman para um caractere	12
11.1	Funcionamento	12
11.2	Implementação em C	12
12	Gerar uma string com o conteúdo de um byte	13
12.1	Funcionamento	13
12.2	Implementação em C	14
13	Liberação da memória alocada para a árvore	14
13.1	Funcionamento	14
13.2	Implementação em C	14
14	Compressão de um arquivo de texto	15
14.1	Funcionamento	15
14.2	Implementação em C	17
15	Descompressão de um arquivo binário	20
15.1	Funcionamento	20
15.2	Implementação em C	21


16 Função principal	23
16.1 Funcionamento	23

1 Estrutura de um nó

1.1 Características

A struct “No” é utilizada tanto como um nó da árvore quanto um nó da lista encadeada (lista caractere-frequência). Ela possui campos para as subárvores, um campo para o nó “pai”, um campo “caractere” para o caractere do nó, um campo “freq” para o número de ocorrências do caractere no arquivo e um campo “prox” utilizado na lista encadeada.

1.2 Implementação em C

A screenshot of a code editor with a dark background and light-colored text. The code defines a C struct named 'no'. It includes a typedef for the struct, followed by the struct definition with fields: char caractere, unsigned freq, struct no *prox, and struct no *esq, *dir, *pai. The struct is named 'No' at the end.

```
1 typedef struct no {
2     char caractere;
3     unsigned freq;
4     struct no *prox;
5     struct no *esq, *dir, *pai;
6 } No;
```

2 Estrutura de cabeçalho

2.1 Características

A struct “Cabecalho” é utilizada para armazenar no arquivo comprimido informações necessárias para a descompressão. Ela possui um campo do tipo unsigned int “excedentes” para guardar a quantidade de bits excedentes no último byte gravado no arquivo binário, um campo unsigned int “tamanhoLista” para guardar o número de caracteres distintos existe na lista caractere-frequência (que também é gravada no arquivo) e, por último, outro campo do tipo unsigned int “numBytes” que guarda a quantidade de bytes que foram gravados no arquivo binário.

2.2 Implementação em C

```
1 typedef struct cabecalho {  
2     unsigned excedentes;  
3     unsigned tamanhoLista;  
4     unsigned numBytes;  
5 } Cabecalho;
```

3 Criação de nó

3.1 Funcionamento

A função `criaNo` recebe um valor `char` a ser atribuído ao campo “caractere” do nó a ser criado e retorna o nó criado. Primeiramente é realizada a alocação da memória necessária para um nó e, caso não tenha sido efetuada a alocação com sucesso, então é impressa uma mensagem de erro e o programa é encerrado sem êxito. Posteriormente é atribuído ao campo “caractere” do novo nó o char “caractere” recebido como parâmetro, é atribuído `NULL` aos campos “prox”, “esq”, “dir” e “pai”. O campo “freq” é inicializado como 1 (frequência mínima de um caractere no arquivo). Por fim é retornado o novo nó criado.

3.2 Implementação em C

```
1 No *criaNo(char caractere) {  
2     No *novo = (No*) malloc(sizeof(No)); // Cria o novo nó.  
3  
4     // Caso a alocação tenha sido mal sucedida o programa é finalizado sem êxito.  
5     if (novo == NULL) {  
6         printf("ERRO: problemas com a alocação de memória.\n");  
7         exit(1);  
8     }
```

```
9
10 // Atribui o caractere recebido por parâmetro ao campo caractere do nó e atribui
11 // NULL aos campos prox, esq, dir e pai.
12 novo->caractere = caractere;
13 novo->prox = NULL;
14 novo->esq = NULL;
15 novo->dir = NULL;
16 novo->pai = NULL;
17
18 // Atribui 1 ao campo freq (frequência).
19 novo->freq = 1;
20 return novo; // Retorno do novo nó.
21 }
```

4 Busca ordenada na lista pela frequência

4.1 Funcionamento

A função “buscaListaOrd” é responsável por fazer a busca ordenada de um nó na lista pela frequência do mesmo. Esta função recebe uma lista “L”, um unsigned int “frequencia” que representará a frequência buscada e a referência de um ponteiro “pred” que representará o nó antecessor do nó buscado nesta lista. A função “buscaListaOrd” retorna o nó encontrado, o nó com a menor frequência maior que a recebida via parâmetro ou NULL caso a frequência buscada seja maior que a frequência de todos os nós da lista.

Primeiramente é criado um ponteiro auxiliar “aux” que recebe o primeiro nó da lista, e “pred” recebe o nó cabeça dessa lista. Dessa forma, é feito um laço while que será responsável por percorrer toda a lista, e dentro do laço caso seja encontrado um nó com frequência menor ou igual à buscada o laço é finalizado. Caso contrário, o ponteiro “pred” é atualizado para apontar para “aux” e “aux” aponta para o próximo nó da lista. Por fim, é retornado o nó apontado por “aux”.

4.2 Implementação em C

```
1 No *buscaListaOrd(No *L, unsigned frequencia, No **pred) {
2     // Criação de um ponteiro auxiliar que recebe o primeiro nó da lista
3     // (desconsiderando o nó cabeça), e pred recebe o nó cabeça.
4     No *aux = L->prox;
5     (*pred) = L;
6     // Laço responsável por percorrer toda a lista.
7     while (aux != NULL) {
8         // Se foi encontrado um nó com frequência menor ou igual à recebida via parâmetro
9         // então o laço é finalizado.
10        if (frequencia <= aux->freq) break;
11        (*pred) = aux; // Atualização do predecessor para receber o nó atual.
12    }
13    aux = aux->prox; // Atualização de aux para apontar para o próximo nó.
14 }
15 // Retorna o nó encontrado (ou NULL, caso a frequência seja maior que de todos nós da lista).
16 return aux;
17 }
```

5 Inserção ordenada na lista pela frequência

5.1 Funcionamento

A função “insereFreqListaOrd” é responsável por inserir um nó interno da árvore de Huffman na lista caractere-frequência de forma ordenada pela frequência e fazer os nós mínimos da lista recebidos via parâmetro serem filhos deste novo nó. Esta função recebe uma lista “L”, um unsigned int “frequencia” que representará a frequência buscada, um nó “x” e um nó “y”. Não há retorno de valores.

Primeiramente é criado um novo nó utilizando a função “criaNo” e é feito com que ele receba o retorno dessa função para um caractere “nulo” (“\0”). Em seguida, o campo “freq” que recebe o parâmetro “frequencia”, e o novo nó recebe como filhos à esquerda e à direita “x” e “y”, respectivamente (também é atribuído ao campo “pai” dos nós “x” e “y” o novo nó). É criado um ponteiro “pred” para guardar o nó predecessor do nó com frequência imediatamente maior que a frequência recebida via parâmetro, que é obtido pelo retorno da função “buscaListaOrd” e atribuído a um novo ponteiro “aux”. Por fim, o nó é inserido entre “pred” e o nó “aux” (o campo “prox” do nó novo receberá “aux” e o campo “prox” de “pred” receberá novo), garantindo a inserção ordenada na lista.

5.2 Implementação em C

```

1 void insereFreqListaOrd(No *L, unsigned frequencia, No *x, No *y) {
2     // Cria um nó novo fazendo com que ele receba o retorno da função criaNo para
3     // um caractere "nulo".
4     No *novo = criaNo('\0');
5
6     // Atribui ao campo freq do novo nó o valor de frequência recebido via parâmetro
7     // e faz x e y serem filhos à esquerda e à direita do novo nó, respectivamente.
8     novo->freq = frequencia;
9     novo->esq = x;
10    novo->dir = y;
11
12    // Faz o pai de x e y ser novo nó.
13    x->pai = novo;
14    y->pai = novo;
15
16    // Cria um ponteiro pred e inicializa ele como NULL (predecessor de aux).
17    No *pred = NULL;
18    // Faz com que aux receba o nó que será buscado a partir da frequência recebida
19    // como parâmetro utilizando a função buscaListaOrd.
20    No *aux = buscaListaOrd(L, frequencia, &pred);
21
22    // São efetuados os apontamentos para a inserção ordenada, o nó é inserido entre
23    // pred e o "nó" (pode ser NULL) resultado da busca.
24    novo->prox = aux;
25    pred->prox = novo;
26 }

```

6 Busca na lista por caractere


6.1 Funcionamento

A função “buscaCaractere” é responsável por buscar na lista caractere-frequência um nó com o campo “caractere” coincidente a um caractere dado. Esta função recebe uma lista “L”, um caractere do tipo char a ser buscado na lista “caractere” e a referência de um ponteiro “pred” que apontará para o nó antecessor do nó buscado. A função retorna o nó com campo “caractere” igual ao caractere recebido via parâmetro ou NULL caso não existe nó com campo “caractere” igual ao caractere buscado.

Primeiramente é criado um ponteiro auxiliar “aux” que recebe o primeiro nó da lista (desconsiderando o nó cabeça). Em seguida, é feito um laço while que será responsável por percorrer toda a lista, sendo que dentro do laço, caso o nó atual exista e seu campo

“caractere” seja igual ao caractere buscado então este é retornado, caso contrário o ponteiro “pred” é atualizado para apontar para “aux” e “aux” aponta para o próximo nó da lista. Por fim, se o nó não for encontrado, é retornado “NULL”.

6.2 Implementação em C



```
1 No *buscaCaractere(No *L, char caractere, No **pred) {
2     // Criação de ponteiro auxiliar que aponta para o primeiro nó da lista.
3     No *aux = L->prox;
4
5     // Laço responsável por percorrer toda a lista.
6     while(aux) {
7         // Se o caractere do nó atual é igual ao buscado, o nó é retornado.
8         if (aux->caractere == caractere) return aux;
9         (*pred) = aux; // Atualiza o ponteiro predecessor para apontar para o nó atual.
10
11         aux = aux->prox; // Atualização de aux para apontar para o próximo nó.
12     }
13
14     // Se o nó não for retornado, é retornado NULL.
15     return NULL;
16 }
```

7 Inserção ordenada na lista por caractere

7.1 Funcionamento

A função “insereOrdLista” é responsável por inserir um nó na lista caractere-frequência com o campo “caractere” igual ao caractere dado ou, caso já exista um nó com campo “caractere” igual a caractere, a função é responsável por atualizar o campo de frequência “freq” deste nó, reorganizando a lista de forma a manter a ordenação, se necessário. Esta função recebe uma lista “L” e um caractere do tipo char “caractere” a ser inserido na lista. Não há retorno de valores.

Primeiramente são criados dois ponteiros auxiliares, um que recebe o resultado da busca pelo caractere utilizando a função “buscaCaractere” e outro para seu predecessor “predResultado” obtido também através da função “buscaCaractere”. Caso o resultado da busca seja diferente de NULL (existe um nó com campo “caractere” igual ao caractere recebido via parâmetro), então é acrescido em uma unidade o valor da frequência do caractere (campo “freq”

do nó) e a lista é reorganizada caso necessário, como segue. São criados 2 ponteiros auxiliares, um para a posição para a qual o nó será “movido” (o nó com menor frequência maior que a frequência do caractere de interesse) “aux” e outro para o predecessor deste nó “predAux”, sendo que “aux” recebe inicialmente o nó sucessor do nó encontrado na busca e “predAux” recebe NULL. Em seguida, é utilizado um laço “while” para encontrar o nó com menor frequência maior que a frequência do nó encontrado pela busca, percorrendo a lista e atualizando “predAux” para apontar para o nó e “aux” para apontar para o próximo nó da lista até que “aux” tenha a frequência maior que a frequência de “resultadoBusca” ou seja igual a NULL (fim da lista). Após o laço é verificado se “predAux” é diferente de NULL e, caso verdadeiro, o nó apontado por “resultadoBusca” é inserido entre o nó apontado por “predAux” e o nó apontado por “aux” (garantindo a ordenação da lista). Caso “predAux” seja igual a NULL após o laço não é necessário realizar nenhum reapontamento, uma vez que, se seu valor é NULL então não houve iteração do laço, ou seja, o nó apontado por “resultadoBusca” já está na posição correta na lista encadeada.

Caso o resultado da busca seja NULL (não existe nó na lista com campo “caractere” igual ao caractere recebido via parâmetro) basta inserir um novo nó no início da lista com o novo caractere, visto que a sua frequência será a menor possível na lista (frequência 1). Sendo assim, é criado um nó com caractere igual ao caractere recebido via parâmetro e caso a lista seja vazia (não existe nó sucessor ao nó cabeça) então o novo nó se torna sucessor do nó cabeça (o ponteiro “prox” do nó cabeça passa a apontar para o novo nó), caso contrário o novo nó é inserido no início da lista. Por fim, é acrescentado em uma unidade o campo de frequência “freq” do nó cabeça, pois este é utilizado para determinar o número de nós na lista.

7.2 Implementação em C

```
1 void insereCaractereListaOrd(No *L, char caractere) {
2     // São criados dois ponteiros auxiliares, um para o resultado da busca pelo caractere
3     // e outro para seu predecessor.
4     No *predResultado = L;
5     No *resultadoBusca = buscaCaractere(L, caractere, &predResultado);
6
7     /**
8      * Caso já exista um nó com o caractere recebido por parâmetro, sua posição na
9      * lista é ajustada caso seja necessário. Caso contrário, basta inserir um novo
10     * nó no início da lista com o novo caractere (a frequência mínima é 1).
11     */
```

```

12  if (resultadoBusca) {
13      resultadoBusca->freq += 1;
14
15      /**
16       * Criação de dois ponteiros auxiliares: um para a posição para a qual o nó
17       * será movido (nó com frequência menor que o em questão), caso haja necessidade
18       * (sua frequência foi alterada) e outro para o predecessor deste nó.
19       */
20      No *aux = resultadoBusca->prox;
21      No *predAux = NULL;
22
23      // Laço para encontrar a posição para a qual o nó deve ser movido.
24      while (aux && resultadoBusca->freq > aux->freq) {
25          predAux = aux;
26          aux = aux->prox;
27      }
28
29      // Houve iteração do laço (predAux foi atualizado), portanto é necessário alterar
30      // a posição do nó na lista. Caso contrário, o nó já está em sua posição correta.
31      if (predAux) {
32          predResultado->prox = resultadoBusca->prox;
33          resultadoBusca->prox = aux;
34          predAux->prox = resultadoBusca;
35      }
36
37      } else {
38          // É inserido no início da lista um nó com o caractere informado via parâmetro.
39          No *novo = criaNo(caractere); // Criação de novo nó.
40
41          // Verifica a existência de um nó além do nó cabeça na árvore e caso não exista
42          // o novo nó é inserido como primeiro nó.
43          if (!(L->prox)) {
44              L->prox = novo;
45          } else { // Se existir, o novo nó é inserido no início da lista.
46              novo->prox = L->prox;
47              L->prox = novo;
48          }
49
50          L->freq += 1; // É atualizado o número de nós na lista (campo "freq" do nó cabeça).
51      }
52  }

```

8 Retirar nó com frequência mínima da lista

8.1 Funcionamento

A função “retiraMin” é responsável por retirar da lista caractere-frequência o nó com menor frequência (nó mínimo). Esta função recebe uma lista “L”. Primeiramente, é criado um ponteiro auxiliar “aux” que recebe o conteúdo do campo “prox” do nó cabeça, ou seja, o primeiro nó da lista. Caso exista este primeiro nó além do nó cabeça, ele é retirado da lista fazendo o ponteiro “prox” do nó cabeça apontar para o nó sucessor do primeiro da lista. Por fim é retornado o nó mínimo retirado da lista.

8.2 Implementação em C

```
1  No *retiraMin(No *L) {
2      // Criação de nó auxiliar que aponta para o primeiro nó da lista.
3      No *aux = L->prox;
4
5      // Se existe um nó além do nó cabeça, este é retirado da lista, uma vez que ele
6      // será o nó com menor frequência.
7      if (aux != NULL) {
8          L->prox = aux->prox;
9      }
10
11     // Retorna o nó mínimo que foi retirado.
12     return aux;
13 }
14
```

9 Função do Algoritmo de Huffman

9.1 Funcionamento

A função “huffman” é responsável por gerar a árvore com os códigos de Huffman para os caracteres presentes no arquivo. Ela recebe como parâmetro uma lista encadeada que é organizada por ordem crescente de frequência para cada caractere. A função retorna o nó raiz da árvore gerada.

Primeiramente são criados ponteiros auxiliares “x” e “y” para guardar os nós mínimos da lista, uma variável inteira “soma” para guardar a soma das frequências dos caracteres dos nós mínimos “x” e “y” e uma variável do tipo inteiro “tamanho” que guarda o número de caracteres distintos existentes na lista (ou tamanho da lista).

Caso “tamanho” seja igual a 1, então a lista (e consequentemente o arquivo) possui somente um caractere (que pode estar repetido) sendo que nesse caso basta criar um novo nó interno, atribuir o único nó da lista como filho esquerdo deste nó criado (o campo “freq” do nó interno será igual ao campo “freq” do nó da lista) e por fim é retornar o nó interno criado como raiz da árvore. Assim, asseguramos que o código de Huffman para o caractere será 0.

Por outro lado, se a lista não tem somente um nó, é utilizado um laço “for” que vai de 0 a tamanho-1, repetindo o processo de retirar o mínimo da lista utilizando a função “retiraMin”

e atribuir a “x”, retirar novamente o mínimo utilizando a função “retiraMin” e atribuir “ay” e utilizar a função “insereOrdFila” para inserir na lista um novo nó interno com filho à esquerda igual a “x”, filho à direita igual a “y” e campo de frequência “freq” igual à soma das frequências dos nós “x” e “y”. Este processo constrói a árvore de Huffman de baixo para cima e deixa na lista somente 1 nó. O nó que resta na lista é retornado após o laço, utilizando novamente a função “retiraMin”, pois será a raiz da árvore gerada.

9.2 Implementação em C

```

1  No *huffman(No *lista) {
2  No *x, *y; // Ponteiros auxiliares para guardar o nó mínimo da lista.
3  int soma = 0; // Auxiliar para guardar a soma das frequências de x e y.
4  int tamanho = lista->freq; // Auxiliar para guardar a quantidade de nós na lista.
5
6  // Tratativa que aborda o caso de o arquivo possuir apenas um caractere (repetido uma ou n vezes).
7  if (tamanho == 1) {
8      // É criado um nó para a raiz da árvore e o único nó da lista se torna seu filho à esquerda.
9      // Assim garantimos que o código do caractere será 0.
10     No *raiz = criaNo('\0');
11     raiz->freq = lista->prox->freq;
12     lista->prox->pai = raiz;
13     raiz->esq = lista->prox;
14
15     // Retorna a raiz da árvore de Huffman gerada.
16     return raiz;
17 }
18
19 // Laço para gerar a árvore de huffman.
20 for(int i=0; i<tamanho-1; i++) {
21     // São retirados os dois nós mínimos da lista.
22     x = retiraMin(lista);
23     y = retiraMin(lista);
24     // Faz-se a soma das frequências desses dois nós mínimos
25     soma = x->freq + y->freq;
26     // É inserido na lista de forma ordenada um nó interno com frequência igual à soma das frequências
27     // dos nós mínimos retirados, sendo que estes se tornarão seus filhos.
28     insereFreqListaOrd(lista, soma, x, y);
29 }
30 // Retorna o nó que restou na lista, que será a raiz da árvore.
31 return retiraMin(lista);
32 }

```

10 Encontrar o nó folha de um caractere

10.1 Funcionamento

A função “encontraFolha” é responsável por encontrar o nó folha na árvore de Huffman correspondente a um dado caractere. Ela recebe como parâmetros um ponteiro que aponta

para a raiz da árvore “arvore”, a referência de um ponteiro que irá apontar para o nó folha buscado (caso o caractere exista na árvore) “noFolha”, um char com o caractere buscado “caractere” e um ponteiro de inteiro utilizado para determinar se o nó folha foi ou não encontrado “encontrado”.

Primeiramente é verificado o valor de “encontrado” e caso esse seja 0 (caractere não encontrado) o fluxo continua, caso contrário o nó foi encontrado então a pilha de recursão é encerrada.

Para o caso de não ter sido encontrado o nó folha, caso o nó atual tenha filho à esquerda a função é chamada recursivamente para a subárvore à esquerda, caso o nó atual tenha filho à direita a função é chamada recursivamente para a subárvore à direita e caso o nó não tenha filhos então foi encontrado um nó folha. Uma vez que foi encontrado um nó folha é verificado se o caractere guardado neste coincide com o caractere buscado e, caso a verificação seja verdadeira, é atribuído 1 à variável “encontrado” e também é atribuído o nó atual ao ponteiro “noFolha”.

10.2 Implementação em C

```
1 void encontraFolha(No *arvore, No **noFolha, char caractere, int *encontrado) {
2     // Nó folha não encontrado.
3     if (*encontrado == 0) {
4         // Procura na subárvore à esquerda.
5         if (arvore->esq) {
6             encontraFolha(arvore->esq, noFolha, caractere, encontrado);
7         }
8
9         // Procura na subárvore à direita.
10        if (arvore->dir) {
11            encontraFolha(arvore->dir, noFolha, caractere, encontrado);
12        }
13
14        // Foi encontrado um nó folha.
15        if (!(arvore->esq) && !(arvore->dir)) {
16            // O nó folha encontrado é o buscado (o campo "caractere" é igual ao recebido via parâmetro).
17            if (arvore->caractere == caractere) {
18                // A variável de controle recebe 1 (foi encontrado o nó folha).
19                *encontrado = 1;
20                // Atribuímos o nó atual ao nó da referência recebida via parâmetro para o nó folha.
21                *noFolha = arvore;
22            }
23        }
24    }
25 }
```

11 Obtenção do código de Huffman para um caractere

11.1 Funcionamento

A função “geraCodigo” é responsável por gerar o código de Huffman para um dado caractere em formato de string. Esta recebe como parâmetros o nó folha na árvore correspondente ao caractere “noFolha”, uma string “codigo” para guardar o código do caractere e um ponteiro do tipo inteiro que indicará a posição em que deve ser inserido 0 ou 1 “posicao”. Não há retorno de valores.

Primeiramente é verificado se o nó atual tem pai e, caso a verificação seja falsa, a função é encerrada pois foi atingida a raiz da árvore e a string com o código está completa.

Por outro lado, caso o nó atual tenha pai, é chamada a função recursivamente para o seu pai. Na volta da chamada recursiva, caso o nó atual seja filho à direita de seu pai a string “codigo” recebe o caractere 0 na posição guardada por “posicao” e caso o nó atual seja filho à esquerda de seu pai, a string “codigo” recebe o caractere 1 na posição guardada por “posicao”. Após essa atualização da string, é acrescido em uma unidade o valor guardado por “posicao” e, por fim, caso o nó atual não tenha filhos (a volta da chamada recursiva chegou no nó folha) a string na posição guardada por “posicao” recebe o caractere ‘\0’, que indica o fim da string. No fim deste processo recursivo, a string “codigo” conterá o código de Huffman para o caractere do nó folha passado via parâmetro.

11.2 Implementação em C

```
1 void geraCodigo(No *noFolha, char *codigo, int *posicao) {
2     // Percorre da folha à raiz para encontrar o código binário do
3     // caractere correspondente ao nó folha passado via parâmetro
4     if (noFolha->pai) {
5         geraCodigo(noFolha->pai, codigo, posicao);
6
7         // Se o nó é filho à esquerda de seu pai a string é concatenada com '0'.
8         if (noFolha == noFolha->pai->esq) {
9             codigo[*posicao] = '0';
10        // Se o nó é filho à direita de seu pai a string é concatenada com '1'.
11        } else {
12            codigo[*posicao] = '1';
13        }
```



```
14 // Atualiza a variável posicao.
15 (*posicao) += 1;
16
17 // Se não houver nó nem à esquerda nem à direita a string é concatenada com '\0'
18 // indicando o fim do código que estava sendo gerado.
19 if (!(noFolha->esq) && !(noFolha->dir)) {
20     codigo[*posicao] = '\0';
21 }
22
23 } else {
24     return; // O código já foi encontrado, a função é encerrada.
25 }
26 }
```

12 Gerar uma string com o conteúdo de um byte

12.1 Funcionamento

A função “geraStringDeBits” é responsável por gerar uma string com o conteúdo de um byte (sequência de bits). Esta função recebe uma variável do tipo unsigned char “byte” que será o byte que foi lido do arquivo binário e uma string “stringGerada” que guardará a sequência de bits do byte recebido via parâmetro “byte”.

Primeiramente é criada uma variável para controlar a posição da string que deve receber 1 ou 0. A seguir é realizado um laço “for” para percorrer o tamanho do byte, preenchendo a string “stringGerada” seguindo procedimento: o byte tem seus bits deslocados para a direita “j” vezes (para que o bit de interesse na iteração corrente seja movido para a extremidade direita do byte) e é realizada uma comparação “e” bit a bit com o byte que possui somente o bit 1 em sua extremidade direita (0x01), o que resultará em um byte com todos os bits iguais a 0 caso o bit de interesse seja 0 e um byte com somente 1 bit igual a 1 na extremidade direita do byte. Após a comparação é somado o char ‘0’ para que o resultado seja um char e este char é atribuído à posição “i” de “stringGerada” e a variável de controle “i” é acrescida em uma unidade. Ao final do processo, a string “stringGerada” está preenchida com a sequência de bits do unsigned char “byte” recebido via parâmetro.

12.2 Implementação em C

```
1 void geraStringDeBits(unsigned char byte, char *stringGerada) {
2     // Variável auxiliar para a posição da string.
3     int i = 0;
4
5     /**
6      * É realizado um laço "for" que irá ser decrementado para percorrer o tamanho de um byte.
7      * Usando operadores bitwise, é armazenado o bit em questão em formato de char na respectiva
8      * posição da string "stringGerada".
9      */
10    for (int j = 7; 0 ≤ j; j--){
11        stringGerada[i] = ((byte >> j) & (0x01)) + '0';
12        i++;
13    }
14 }
```

13 Liberação da memória alocada para a árvore

13.1 Funcionamento

A função “liberaArvore” é responsável por liberar a memória alocada para os nós de uma árvore. Esta função recebe como parâmetro o nó raiz da árvore “arvore”. Não há retorno de valores.

Se “arvore” for diferente de NULL (nó atual diferente de NULL) a função será chamada recursivamente para liberar primeiramente os nós das duas subárvores (à esquerda e à direita) e após o retorno da chamada recursiva é liberada a memória alocada para o nó atual utilizando a função “free”. Após o processo recursivo, terá sido liberada toda a memória alocada para os nós da árvore.

13.2 Implementação em C

```
1 void liberaArvore(No *arvore) {
2     // Se o nó é diferente de NULL, é necessário liberar a memória alocada para ele
3     // e suas subárvores.
4     if (arvore) {
5
6         /**
7          * Chama a função recursivamente para as subárvores à esquerda e à direita,
```

```
8      * para liberar primeiro a memória alocada para estas antes de liberar a memória
9      * alocada para o nó atual.
10     */
11     liberaArvore(arvore->esq);
12     liberaArvore(arvore->dir);
13     // Libera a memória alocada para o nó atual.
14     free(arvore);
15 }
16 }
```

14 Compressão de um arquivo de texto

14.1 Funcionamento

A função “comprime” é responsável por criar um arquivo binário contendo a compressão de um arquivo de texto em linguagem natural utilizando o algoritmo de Huffman. Esta função recebe como parâmetros os nomes dos arquivos de entrada “arquivoEntrada” (arquivo de texto) e de saída “arquivoSaida” (arquivo binário) e caso ocorra um erro na abertura a função é encerrada e é impressa uma mensagem de erro. Primeiramente são criadas 3 variáveis para controle do tempo gasto na compressão. Então é aberto o arquivo de entrada (arquivo de texto) e efetuada a inserção de cada caractere presente neste através de um laço que utiliza a função “insereOrdLista” em duas listas, uma que será utilizada para construir a árvore de Huffman “L” e outra para imprimir os caracteres com suas respectivas ocorrências e códigos gerados “G” (a lista utilizada pela função de Huffman fica inutilizável), sendo que após essa passada no arquivo de entrada o mesmo é “rebobinado” utilizando a função “rewind”. Posteriormente é efetuada a movimentação do ponteiro do stream de saída com uma quantidade igual ao tamanho ocupado pelo cabeçalho a partir do início do arquivo. É utilizado um laço para inserir no arquivo de saída os nós da lista caractere-frequência. Dando continuidade, é criada a árvore a partir do algoritmo de Huffman (utilizando a função huffman, passando como parâmetro a lista caractere-frequência) e atribuída a um ponteiro “arvore”. Criamos o cabeçalho e atribuímos ao campo “freq” a quantidade de nós na lista caractere-frequência.

Agora começa o processo de compressão efetiva do arquivo de texto gravando os bytes no arquivo binário. São criadas variáveis auxiliares para o char que será utilizado para gravação “armazenar”, um inteiro “posicao” utilizado para gerar a string do código de Huffman para cada caractere, uma variável inteira “bitsOcupados” para representar a quantidade de bits “ocupados” no char que será gravado, um nó auxiliar “noFolha” para guardar o nó folha

dos nós da lista caractere-frequência na árvore de Huffman, uma variável inteira “numBytes” para guardar a quantidade de bytes gravados no arquivo de saída, uma variável inteira “numExcedentes” para guardar a quantidade de bits excedentes no último byte gravado e uma variável “encontrado” utilizada pela função “encontraFolha”. Após a declaração das variáveis necessárias começa a gravação dos bytes através de um laço que percorre o arquivo de entrada armazenando na variável “caractereAtual” cada caractere contido no arquivo, uma vez que os caracteres são lidos através da função “fgetc”. Dentro do laço atribuímos 0 às variáveis “posicao” e “encontrado” e é criada uma string para armazenar o código do caractere corrente, é utilizada a função “encontraFolha” para obter a folha correspondente ao caractere na árvore de Huffman e logo após é utilizada a função “geraCodigo” passando o nó folha encontrado para gerar o código de Huffman para o caractere. Ainda no laço que lê todos os caracteres, possuímos outro laço que percorre o código gerado e, dentro do mesmo, caso a posição atual na string seja 1 então é deslocado 1 bit para a esquerda no char “armazenar” e inserido um bit 1 na direita, caso contrário é realizado um deslocamento à esquerda apenas (será preenchido com 0 na direita). Ainda dentro do laço que percorre o código, é incrementado o número de bits ocupados e, caso ele se iguale a 8 (o byte foi totalmente ocupado), é utilizada a função “fwrite” para escrever no arquivo binário o char “armazenar” e são atualizadas as variáveis “armazenar” para 0, “bitsOcupados” para 0 e “numBytes” para seu valor mais um. Desta forma é finalizado o laço que percorre o código e também o laço de leitura dos caracteres do arquivo de entrada.

Logo após o laço que percorre o arquivo é verificado se sobraram bits excedentes e caso seja verdadeira a verificação, a variável char “armazenar” tem seus bits deslocados para a esquerda um quantidade de numExcedentes, sendo que é armazenado no campo “excedentes” do cabeçalho o número de bits excedentes, o número de bytes é acrescido em 1 e, finalmente, o byte incompleto (com bits excedentes) é escrito no arquivo de saída. Após isso o arquivo de saída é rebobinado e é escrito no início deste o cabeçalho que foi preenchido até o momento.

Por fim, é utilizado um laço para percorrer a lista caractere-frequência para imprimir os caracteres presentes no arquivo com suas respectivas frequências e também seu código de Huffman. Outro laço é utilizado para liberar a memória alocada para a segunda lista “G”. Os arquivos de entrada e saída são fechados e a memória alocada para os nós da árvore é liberada, além do nó cabeça da lista “L” que ficou inutilizada após a montagem da árvore de Huffman.

14.2 Implementação em C

```

1 void comprime(char *arquivoEntrada, char *arquivoSaida) {
2     // Bloco de variáveis que serão responsáveis por mostrar o tempo que a compressão leva.
3     clock_t inicio, final;
4     double tempoGasto;
5     inicio = clock();
6
7     // Listas caractere-frequência.
8     No *L = (No*) calloc (1, sizeof(No));
9     No *G = (No*) calloc (1, sizeof(No));
10
11     // Atribuição das aberturas dos arquivos aos ponteiros "entrada" e "saida", respectivamente.
12     FILE *entrada = fopen(arquivoEntrada, "r");
13     FILE *saida = fopen(arquivoSaida, "wb");
14
15     // Erro na abertura do arquivo de entrada.
16     if (!entrada) {
17         printf("Arquivo de entrada inexistente!\n\n");
18         return;
19     }
20
21     // Erro na abertura do arquivo de saída.
22     if (!saida) {
23         printf("Arquivo de saída inexistente!\n\n");
24         return;
25     }
26
27     char caractereAtual = '\0'; // Auxiliar para ler os caracteres do arquivo de texto.
28
29     // Laço que percorre o arquivo de entrada preenchendo as lista caractere-frequência.
30     while((caractereAtual=fgetc(entrada)) != EOF) {
31         insereCaractereListaOrd(L, caractereAtual);
32         insereCaractereListaOrd(G, caractereAtual);
33     }
34
35     // "Rebobinação" do arquivo de entrada.
36     rewind(entrada);
37
38     // O ponteiro do stream 'saida' é movido para a posição posterior ao tamanho da
39     // struct Cabecalho.
40     fseek(saida, sizeof(Cabecalho), SEEK_SET);
41
42     // Ponteiro auxiliar para percorrer a lista caractere-frequência que recebe inicialmente
43     // o primeiro nó da lista.
44     No *auxiliarListaFrequencias = L->prox;
45
46     // Percorre toda a lista L gravando cada nó no arquivo de saída.
47     while(auxiliarListaFrequencias) {
48         // Escreve o nó apontado por "auxiliarListaFrequencias" no arquivo de saída.
49         fwrite(auxiliarListaFrequencias, sizeof(No), 1, saida);
50         // Atualiza "auxiliarListaFrequencias" para apontar para o próximo nó da lista.
51         auxiliarListaFrequencias = auxiliarListaFrequencias->prox;
52     }
53
54     // Criação da árvore de Huffman para a lista caractere-frequência montada.
55     No *arvore = huffman(L);
56
57     // Criação do cabeçalho.
58     Cabecalho cabecalho;
59     cabecalho.tamanhoLista = L->freq;
60     cabecalho.excedentes = 0;
61     cabecalho.numBytes = 0;

```

```

62
63 /**
64  * Criação de variáveis que serão importantes para controlar a posição utilizada
65  * na função "geraCodigo", o byte que será armazenado no arquivo de saída e a
66  * quantidade de bits ocupados no byte que será gravado.
67  */
68 unsigned char armazenar = 0;
69 int posicao = 0;
70 int bitsOcupados = 0;
71
72 /**
73  * Criação de variáveis que serão importantes para controlar o número de bytes
74  * gravados no arquivo de saída, o número de bits excedentes no último byte gravado
75  * e para verificar se o nó folha do caractere foi encontrado.
76  */
77 int numBytes = 0;
78 int numExcedentes = 0;
79 int encontrado = 0;
80
81 // Ponteiro auxiliar que aponta para o nó folha na árvore de Huffman para o
82 // caractere de interesse.
83 No *noFolha = NULL;
84
85 // Laço que percorre o arquivo de entrada e é responsável por fazer sua compressão.
86 while((caractereAtual=fgetc(entrada)) != EOF) {
87     posicao = 0; // Posição deve zerar para gerar a string do novo caractere.
88     encontrado = 0; // O nó folha do caractere ainda não foi encontrado.
89     char codigo[1024]; // String que guardará o código de Huffman gerado para o caractere.
90
91     /**
92      * É encontrado o nó folha na árvore para o caractere atual e gerada a string
93      * com o código de Huffman para o mesmo utilizando as funções "encontraFolha"
94      * e "geraCodigo", respectivamente.
95      */
96     encontraFolha(arvore, &noFolha, caractereAtual, &encontrado);
97     geraCodigo(noFolha, codigo, &posicao);
98
99     // Laço para percorrer a string com o código de Huffman do caractere.
100    for (char *i = codigo; *i; i++) {
101        /**
102         * Caso a posição atual contenha o caractere 1 é inserido um bit 1 na
103         * extremidade direita do byte. Caso contrário um bit 0 é inserido na
104         * extremidade direita do byte.
105         */
106        if (*i == '1') {
107            armazenar = (armazenar << 1) + 1;
108        } else if (*i == '0') {
109            armazenar = armazenar << 1;
110        }
111
112        bitsOcupados++; // É acrescida em um unidade a quantidade de bits ocupados no byte.
113
114        // Caso um byte tenha sido completamente ocupado ele é escrito no arquivo de saída.
115        if (bitsOcupados == 8) {
116            // Escreve o conteúdo de "armazenar" no arquivo de saída.
117            fwrite(&armazenar, sizeof(unsigned char), 1, saida);
118            // São zeradas as variáveis de quantidade de bits ocupados (bitsOcupados) e do
119            // byte a ser escrito (armazenar).
120            armazenar = 0;
121            bitsOcupados = 0;
122
123            // É acrescida em uma unidade a quantidade de bytes gravados no arquivo de saída.
124            numBytes++;
125        }
126    }
127 }

```

```

128
129 // Caso tenha sobrado bits não escritos no arquivo de saída, é escrito um byt
130 // com bits excedentes.
131 if (bitsOcupados != 0) {
132     // Os bits que interessam são movidos para a extremidade esquerda do byte.
133     numExcedentes = 8 - bitsOcupados;
134     armazenar = armazenar << numExcedentes;
135
136     // É armazenado no cabeçalho a nova quantidade de bits excedentes.
137     cabecalho.excedentes = numExcedentes;
138     // É acrescida em uma unidade a quantidade de bytes gravados no arquivo de saída.
139     numBytes++;
140
141     // Escreve o conteúdo de "armazenar" no arquivo de saída.
142     fwrite(&armazenar, sizeof(char), 1, saida);
143 }
144
145 // O campo "numBytes" recebe a variável "numBytes" que foi devidamente preenchida.
146 cabecalho.numBytes = numBytes;
147
148 // "Rebobina" o arquivo de saída.
149 rewind(saida);
150
151 // Escreve o conteúdo do cabeçalho no arquivo de saída.
152 fwrite(&cabecalho, sizeof(Cabecalho), 1, saida);
153
154 // É encerrada a contagem do tempo de execução.
155 final = clock();
156 tempoGasto = (double)(final - inicio) / CLOCKS_PER_SEC;
157
158 auxiliarListaFrequencias = G->prox;
159 char codigo[1024] = "";
160 encontrado = 0;
161
162 // É utilizado um laço para percorrer a lista G mostrando na tela cada caractere
163 // do arquivo de entrada com seu número de ocorrências e seu código de Huffman.
164 printf("\tCodigo de Huffman gerado:\n\n");
165 while(auxiliarListaFrequencias) {
166     encontrado = 0;
167     strcpy(codigo, "");
168     posicao = 0;
169     printf("%c [%d] : ", auxiliarListaFrequencias->caractere, auxiliarListaFrequencias->freq);
170
171     encontraFolha(arvore, &noFolha, auxiliarListaFrequencias->caractere, &encontrado);
172     geraCodigo(noFolha, codigo, &posicao);
173
174     printf("%s\n", codigo);
175     auxiliarListaFrequencias = auxiliarListaFrequencias->prox;
176 }
177
178 // É utilizado um laço para liberar a memória alocada para os nós da lista G.
179 No *liberar = NULL;
180 auxiliarListaFrequencias = G;
181 while (auxiliarListaFrequencias->prox) {
182     liberar = auxiliarListaFrequencias;
183     auxiliarListaFrequencias = auxiliarListaFrequencias->prox;
184     free(liberar);
185 }
186
187 fclose(saida); // Fecha o arquivo de saída.
188 fclose(entrada); // Fecha o arquivo de entrada.
189 free(L); // Limpa a memória alocada para o nó cabeça da lista L.
190 liberaArvore(arvore); // Limpa a memória alocada para a árvore gerada pelo código de Huffman.
191
192 printf("\nTempo gasto na compressao: %gs\n\n", tempoGasto);
193 }

```

15 Descompressão de um arquivo binário

15.1 Funcionamento

A função “descomprime” é responsável por criar um arquivo de texto contendo a descompressão de um arquivo binário o algoritmo de Huffman. Esta função recebe como parâmetros os nomes dos arquivos de entrada “arquivoEntrada” (arquivo binário) e de saída “arquivoSaida” (arquivo de texto). Primeiramente é criada uma variável de cabeçalho “cabecalho” e a lista caractere-frequência “listaFrequencias” juntamente com um ponteiro auxiliar para percorrê-la. Logo em seguida são abertos os arquivos de entrada e de saída e caso ocorra um erro na abertura a função é encerrada e é impressa uma mensagem de erro. Primeiramente é lido do arquivo de entrada o cabeçalho armazenado e atribuído à variável “cabecalho” e o tamanho da lista é atribuído ao campo “freq” do nó cabeça da lista caractere-frequência. Seguindo, é utilizado um laço para ler os nós da lista caractere-frequência armazenados no arquivo comprimido fazendo o encadeamento desta com a lista “listaFrequencias” e esta é passada via parâmetro para a função “huffman”, atribuindo assim a um nó “arvore” o retorno da mesma.

Agora começa o processo de descompressão efetiva gravando no arquivo de saída os caracteres correspondentes aos códigos armazenados no arquivo binário. São criadas variáveis auxiliares para o byte lido do arquivo binário “byte”, para o código do byte em string “string”, para o nó atual na árvore “noAtual” (a árvore de Huffman será percorrida), para o bit atual do byte lido “bitAtual” do tipo char, para o número de bits que devem ser lidos do byte “limite” (à priori 8, que é a quantidade de bits em 1 byte) do tipo unsigned int e para controlar o byte atual (inicialmente 1) “byteAtual” do tipo unsigned int. Após a declaração das variáveis necessárias começa a gravação dos caracteres no arquivo de texto. É utilizado um laço que percorre todo o arquivo binário, lendo byte por byte e armazenando o resultado da leitura na variável “byte” e tendo lido o byte é utilizada a função “geraStringDeBits” para obter a representação em formato de string do byte (string com 0 e 1) e essa string é atribuída à variável “codigoByte”. Ainda dentro do laço é verificado se a contagem “byteAtual” é igual ao número de bytes guardado no cabeçalho e, caso a verificação seja verdadeira, a variável “limite” é atualizada recebendo o valor de 8 subtraído pelo número de bits excedentes (também guardado no cabeçalho). Continuando no laço, é realizado outro laço que percorre a string gerada para o byte “codigoByte” da posição 0 ao valor da variável “limite” (assegura-se que não serão descompressos os bits excedentes), sendo que dentro deste laço fazemos a variável “bitAtual” receber a posição “i” do contador do laço. Ainda ao percorrer a string código do byte verifica-se o valor da variável “bitAtual” é 1 ou 0, sendo que se for 1 a variável “noAtual”

recebe seu filho à direita e se for 0 recebe seu filho à esquerda. Por fim, continuando no laço é verificado se o nó atual não tem filhos, ou seja, é um nó folha, e caso a verificação seja verdadeira é escrito no arquivo de texto o caractere guardado pelo nó folha e a variável “noAtual” recebe a raiz da árvore para continuar a descompressão normalmente na próxima iteração do laço.

Após o processo de descompressão, são fechados os arquivos de entrada e saída, é liberada a memória para a árvore de Huffman gerada e também para o nó cabeça da lista “listaFrequencias”, encerrando assim a função de descompressão.

15.2 Implementação em C

```
1 void descomprime(char *arquivoEntrada, char *arquivoSaida) {
2     // Bloco de variáveis que serão responsáveis por mostrar o tempo que a compressão leva.
3     clock_t inicio, final;
4     double tempoGasto;
5     inicio = clock();
6
7     // Criação do cabeçalho, da lista caractere-frequência e de um ponteiro auxiliar
8     // para percorrê-la.
9     Cabecalho cabecalho;
10    No *listaFrequencias = (No *) calloc(1, sizeof(No));
11    No *auxiliarLista = listaFrequencias;
12
13    // Atribuição das aberturas dos arquivos aos ponteiros "entrada" e "saida", respectivamente.
14    FILE *entrada = fopen(arquivoEntrada, "rb");
15    FILE *saida = fopen(arquivoSaida, "w");
16
17    // Erro na abertura do arquivo de entrada.
18    if (!entrada) {
19        printf("Arquivo de entrada inexistente!\n\n");
20        return;
21    }
22
23    // Erro na abertura do arquivo de saída.
24    if (!saida) {
25        printf("Arquivo de saída inexistente!\n\n");
26        return;
27    }
28
29    // Atribuição da leitura do cabeçalho gravado no arquivo binário à variável "cabecalho".
30    fread(&cabecalho, sizeof(Cabecalho), 1, entrada);
31
32    // Atualiza a quantidade de nós na lista de caractere-frequências com o valor
33    // guardado no cabeçalho.
34    listaFrequencias->freq = cabecalho.tamanhoLista;
35
36    // Criação de ponteiro auxiliar para receber os nós guardados no arquivo comprimido.
37    No *aux = NULL;
```

```

38 // Laço responsável por ler os nós guardados no arquivo comprimido e realizar o
39 // encadeamento desses nós com a lista caractere-frequência local (listaFrequencias).
40 for (int i=0; i<cabecalho.tamanhoLista; i++) {
41     aux = (No *) malloc(sizeof(No));
42     fread(aux, sizeof(No), 1, entrada); // Leitura de nó armazenado no arquivo binário.
43
44     // Encadeamento do nó lido à lista "listaFrequencias".
45     auxiliarLista->prox = aux;
46     // Atualização do ponteiro auxiliar da lista "listaFrequencias".
47     auxiliarLista = auxiliarLista->prox;
48 }
49
50 // Criação da árvore de Huffman para a lista lida do arquivo binário.
51 No *arvore = huffman(listaFrequencias);
52
53 // Criação de variáveis auxiliares para guardar o byte lido do arquivo binário
54 // "byte" e o código em string do byte "codigoByte".
55 unsigned char byte = 0;
56 char codigoByte[9] = "";
57
58 /**
59  * Criação de variáveis para controlar qual é o bit atual do byte lido "bitAtual",
60  * a quantidade de bits que devem ser lidos do código do byte "limite" e o número
61  * do byte lido comparado à quantidade de bytes gravados "byteAtual".
62  */
63 char bitAtual = '\0';
64 unsigned limite = 8; // Inicializada com 8 (quantidade de bits em um byte).
65 unsigned byteAtual = 1;
66
67 // Criação de ponteiro auxiliar para percorrer a árvore de Huffman em busca de
68 // um nó folha (que guarda um caractere).
69 No *noAtual = arvore;
70
71 // Laço que percorre o arquivo de entrada e é responsável por fazer sua descompressão.
72 while(fread(&byte, sizeof(unsigned char), 1, entrada)) {
73     // Atribuição da string gerada para a sequência de bits do byte lido à string "codigoByte".
74     geraStringDeBits(byte, codigoByte);
75
76     // Caso o byte atual seja o último gravado no arquivo, o limite é ajustado para
77     // que não sejam descomprimidos os bits excedentes.
78     if (byteAtual == cabecalho.numBytes) limite = 8 - cabecalho.excedentes;
79
80     /**
81      * Laço responsável por percorrer a string da sequência de bits gerada e percorrer
82      * a árvore a depender do conteúdo de cada posição até encontrar um nó folha na
83      * árvore (um caractere foi encontrado).
84      */
85     for(int i=0; i<limite; i++) {
86         // A variável "bitAtual" recebe o char da posição "i" da string "codigoByte".
87         bitAtual = codigoByte[i];
88
89         // Caso o bit atual tenha valor 1 o ponteiro auxiliar "noAtual" recebe seu
90         // filho à direita e caso contrário recebe seu filho à esquerda.
91         noAtual = bitAtual == '1' ? noAtual->dir : noAtual->esq;
92
93         // Caso tenha sido atingido um nó folha (sem filhos) então foi encontrado um
94         // caractere na árvore de Huffman.
95         if (!(noAtual->esq) && !(noAtual->dir)) {
96             // O caractere encontrado é escrito no arquivo de saída.
97             fwrite(&(noAtual->caractere), sizeof(char), 1, saida);
98
99             // O ponteiro auxiliar para percorrer a árvore "noAtual" é atualizado para
100             // apontar para a raiz da árvore novamente.
101             noAtual = arvore;
102         }
103     }
104
105     byteAtual++; // A contagem do byte lido do arquivo comprimido é acrescida em uma unidade.
106 }
107

```

```

108 // É encerrada a contagem do tempo de execução.
109 final = clock();
110 tempoGasto = (double)(final - inicio) / CLOCKS_PER_SEC;
111 printf("Tempo gasto na descompressao: %gs\n\n", tempoGasto);
112
113 fclose(entrada); // Fecha o arquivo de entrada.
114 fclose(saida); // Fecha o arquivo de saída.
115
116 // Libera a memória alocada para o nó cabeça da lista "listaFrequencias".
117 free(listaFrequencias);
118 // É liberada a memória alocada para os nós da árvore de Huffman gerada.
119 liberaArvore(arvore);
120 }

```

16 Função principal

16.1 Funcionamento

A função “main” é responsável por efetuar as chamadas de função necessárias efetivamente. Na função “main” implementada primeiramente são criadas 3 variáveis auxiliares, sendo que uma delas, “escolha”, é uma variável do tipo inteiro utilizada para guardar a escolha do usuário dadas as opções disponíveis, outra é do tipo char de nome “enter” para capturar o “enter” do teclado (pode interferir na leitura do nome do arquivo) e as duas últimas, “arquivoEntrada” e “arquivoSaida”, são utilizadas para guardar os nomes dos arquivos de entrada e saída, respectivamente, que são inseridos pelo usuário.

Após a criação das variáveis auxiliares, é realizado um laço “do while” que primeiramente apresenta o menu para o usuário com as opções “Comprimir arquivo”, “Descomprimir arquivo” e “Sair” e realiza a leitura da escolha do usuário armazenando esta escolha na variável “escolha”. Logo após a escolha do usuário, existe uma estrutura condicional “switch case” que utiliza as funções implementadas para compressão ou descompressão de acordo com o valor da variável “escolha”. Caso o usuário tenha escolhido comprimir um arquivo, são solicitados os nomes dos arquivos de entrada (.txt) e de saída (.bin), respectivamente, e é utilizada a função comprime para realizar a compressão. Por outro lado, caso o usuário tenha escolhido descomprimir um arquivo, são solicitados os nomes dos arquivos de entrada (.bin) e saída (.txt), respectivamente, e é utilizada a função descomprime para realizar a descompressão. Para o caso de o usuário ter escolhido sair nada é feito e caso o usuário tenha inserido uma opção inválida é mostrada uma mensagem informando que a opção é inválida. O laço “do while” é mantido até que o usuário escolha a opção sair, possibilitando que sejam realizados múltiplos processos de compressão/descompressão.