Algoritmo de Huffman

2.0

Gerado por Doxygen 1.10.0

1 Índice das Estruturas de Dados 1

1.1 Estruturas de Dados . . . 1

2 Índice dos Arquivos 3

2.1 Lista de Arquivos . . . 3

3 Estruturas 5

3.1 Referência da Estrutura BitHuff . . . 5

3.1.1 Descrição detalhada . . . 5

3.1.2 Campos . . . 5

3.1.2.1 bitH . . . 5

3.1.2.2 size . . . 5

3.2 Referência da Estrutura HEAD . . . 6

3.2.1 Descrição detalhada . . . 6

3.2.2 Campos . . . 6

3.2.2.1 head . . . 6

3.2.2.2 size . . . 6

3.3 Referência da Estrutura NODE . . . 6

3.3.1 Descrição detalhada . . . 7

3.3.2 Campos . . . 7

3.3.2.1 data . . . 7

3.3.2.2 frequency . . . 7

3.3.2.3 left . . . 7

3.3.2.4 next . . . 7

3.3.2.5 right . . . 7

4 Arquivos 9

4.1 Referência do Arquivo inc/decode.h . . . 9

4.1.1 Funções . . . 9

4.1.1.1 find_file_size() . . . 9

4.1.1.2 get_tree_and_trash_size() . . . 10

4.1.1.3 getTree() . . . 10

4.1.1.4 is_seted() . . . 11

4.1.1.5 unzip() . . . 11

4.2 decode.h . . . 12

4.3 Referência do Arquivo inc/encode.h . . . 12

4.3.1 Definições dos tipos . . . 13

4.3.1.1 BitHuff . . . 13

4.3.2 Funções . . . 13

4.3.2.1 buildTable() . . . 13

4.3.2.2 count_frequency() . . . 14

4.3.2.3 getTrashSize() . . . 14

4.3.2.4 getTreeSize() . . . 15

4.3.2.5 init_frequency_table() . . . 15
4.3.2.6 isSetedLong() . . . 16
4.3.2.7 set_bit() . . . 17
4.3.2.8 setBytes() . . . 17
4.3.2.9 setFirstTwoBytes() . . . 18
4.3.2.10 write_header() . . . 18
4.3.2.11 write_tree() . . . 19
4.4 encode.h . . . 19
4.5 Referência do Arquivo inc/list.h . . . 20
4.5.1 Definições dos tipos . . . 21
4.5.1.1 HEAD . . . 21
4.5.1.2 NODE . . . 21
4.5.2 Funções . . . 21
4.5.2.1 create_node() . . . 21
4.5.2.2 file_error_reporter() . . . 21
4.5.2.3 init_struct() . . . 22
4.5.2.4 insert_in_linked_list() . . . 22
4.5.2.5 insert_sorted() . . . 22
4.5.2.6 malloc_error_reporter() . . . 23
4.5.2.7 removeFirst() . . . 23
4.6 list.h . . . 24
4.7 Referência do Arquivo inc/std.h . . . 24
4.7.1 Definições e macros . . . 25
4.7.1.1 INPUT_DIR . . . 25
4.7.1.2 TAM . . . 25
4.7.1.3 UNZIPED_OUT_DIR . . . 25
4.7.1.4 ZIPED_OUT_DIR . . . 25
4.8 std.h . . . 25
4.9 Referência do Arquivo inc/tree.h . . . 25
4.9.1 Funções . . . 26
4.9.1.1 get_data() . . . 26
4.9.1.2 get_frequency() . . . 26
4.9.1.3 huffmanTree() . . . 27
4.9.1.4 is_leaf() . . . 27
4.9.1.5 pre_order_trasversal() . . . 28
4.10 tree.h . . . 28
4.11 Referência do Arquivo src/decode.c . . . 28
4.11.1 Funções . . . 29
4.11.1.1 find_file_size() . . . 29
4.11.1.2 get_tree_and_trash_size() . . . 29
4.11.1.3 getTree() . . . 30
4.11.1.4 is_seted() . . . 30

4.11.1.5 unzip() . . . 31
4.12 Referência do Arquivo src/encode.c . . . 31
4.12.1 Funções . . . 32
4.12.1.1 buildTable() . . . 32
4.12.1.2 count_frequency() . . . 33
4.12.1.3 getTrashSize() . . . 33
4.12.1.4 getTreeSize() . . . 33
4.12.1.5 init_frequency_table() . . . 34
4.12.1.6 isSetedLong() . . . 34
4.12.1.7 set_bit() . . . 35
4.12.1.8 setBytes() . . . 35
4.12.1.9 setFirstTwoBytes() . . . 36
4.12.1.10 write_header() . . . 36
4.12.1.11 write_tree() . . . 37
4.13 Referência do Arquivo src/list.c . . . 37
4.13.1 Funções . . . 38
4.13.1.1 create_node() . . . 38
4.13.1.2 file_error_reporter() . . . 38
4.13.1.3 init_struct() . . . 39
4.13.1.4 insert_in_linked_list() . . . 39
4.13.1.5 insert_sorted() . . . 39
4.13.1.6 malloc_error_reporter() . . . 40
4.13.1.7 removeFirst() . . . 40
4.14 Referência do Arquivo src/main.c . . . 41
4.14.1 Funções . . . 41
4.14.1.1 main() . . . 41
4.15 Referência do Arquivo src/tree.c . . . 42
4.15.1 Funções . . . 43
4.15.1.1 get_data() . . . 43
4.15.1.2 get_frequency() . . . 43
4.15.1.3 huffmanTree() . . . 44
4.15.1.4 is_leaf() . . . 44
4.15.1.5 pre_order_trasversal() . . . 45

Índice Remissivo **47**

Capítulo 1

Índice das Estruturas de Dados

1.1 Estruturas de Dados

Aqui estão as estruturas de dados, uniões e suas respectivas descrições:

BitHuff
: Está estrutura representa o byte codificado 5

HEAD
: Estrutura da lista encadaeada e árvore 6

NODE
: Estrutura de um nó 6

Capítulo 2

Índice dos Arquivos

2.1 Lista de Arquivos

Esta é a lista de todos os arquivos e suas respectivas descrições:

inc/decode.h 9
inc/encode.h 12
inc/list.h 20
inc/std.h 24
inc/tree.h 25
src/decode.c 28
src/encode.c 31
src/list.c 37
src/main.c 41
src/tree.c 42

Capítulo 3

Estruturas

3.1 Referência da Estrutura BitHuff

Está estrutura representa o byte codificado.

`#include <encode.h>`

Campos de Dados

- unsigned long long int bitH
- int size

3.1.1 Descrição detalhada

Está estrutura representa o byte codificado.

3.1.2 Campos

3.1.2.1 bitH

```
unsigned long long int bitH
```

Codificação do byte.

3.1.2.2 size

```
int size
```

tamanho de quantos bits representam a codificação.

A documentação para essa estrutura foi gerada a partir do seguinte arquivo:

- inc/encode.h

3.2 Referência da Estrutura HEAD

Estrutura da lista encadaeada e árvore.

`#include <list.h>`

Campos de Dados

- struct NODE * head
- int size

3.2.1 Descrição detalhada

Estrutura da lista encadaeada e árvore.

Esta estrutura representa tanto a cabeça da lista encadeada quanto a raiz da árvoreas

3.2.2 Campos

3.2.2.1 head

```
struct NODE* head
```

Ponteiro para a cabeça/raiz

3.2.2.2 size

```
int size
```

Tamanho da estrutura

A documentação para essa estrutura foi gerada a partir do seguinte arquivo:

- inc/list.h

3.3 Referência da Estrutura NODE

Estrutura de um nó

`#include <list.h>`

Campos de Dados

- void * data
- void * frequency
- struct NODE * left
- struct NODE * right
- struct NODE * next

3.3.1 Descrição detalhada

Estrutura de um nó

Esta estrutura representa o nó da lista encadeada e árvore

3.3.2 Campos

3.3.2.1 data

```
void* data
```

Ponteiro que armazena o endereço do byte

3.3.2.2 frequency

```
void* frequency
```

Ponteiro que armazena a frequencia do byte

3.3.2.3 left

```
struct NODE* left
```

Filho esquerdo

3.3.2.4 next

```
struct NODE* next
```

Próximo nó da lista encadeada

3.3.2.5 right

```
struct NODE* right
```

Filho direito

A documentação para essa estrutura foi gerada a partir do seguinte arquivo:

- inc/list.h

Capítulo 4

Arquivos

4.1 Referência do Arquivo inc/decode.h

```
#include "list.h"
```

Funções

- unsigned long long int find_file_size (FILE ∗file)

  retorna o tamanho do arquivo compactado
- bool is_seted (uint8_t byte, int i)

  Verifica se o byte está setado na posição i.
- void get_tree_and_trash_size (FILE ∗compressed_file, int ∗trashSize, int ∗treeSize)

  Pega o tamanho da árvore e o tamanho do lixo do cabeçalho.
- NODE ∗ getTree (FILE ∗archive, int ∗treeSize)

  Refaz a árvore de Huffman a partir do cabeçalho.
- void unzip (FILE ∗compressed_file, FILE ∗fileOut, unsigned long long coded_size, NODE ∗treeRoot, int trashSize)

  Descomprimi o arquivo compactado.

4.1.1 Funções

4.1.1.1 find_file_size()

```
unsigned long long int find_file_size (
            FILE * file )
```

retorna o tamanho do arquivo compactado

Parâmetros

file	O arquivo compactado

Retorna

O tamanho do arquivo compactado

```
00007                                                 {
00008
00009     if (file == NULL) {
00010         perror("Erro ao abrir o arquivo");
00011         return 1;
00012     }
00013
00014     unsigned long long int current_postition = ftell(file);
00015
00016     fseek(file, 0, SEEK_END); // Move o indicador de posição para o final do arquivo
00017     unsigned long long int file_size = ftell(file);   // Obtém a posição atual, que é o tamanho do
      arquivo em bytes
00018     fseek(file, current_postition, SEEK_SET); // Move o indicador de posição de volta para o início do
      arquivo
00019
00020     return file_size;
00021 }
```

4.1.1.2 get_tree_and_trash_size()

```
void get_tree_and_trash_size (
              FILE * compressed_file,
              int * trashSize,
              int * treeSize )
```

Pega o tamanho da árvore e o tamanho do lixo do cabeçalho.

Parâmetros

compressed_file	O arquivo compactado
trashSize	Ponteiro para o tamanho do lixo
treeSize	Ponteiro para o tamanho da árvore

```
00023                                                                                         {
00024     uint16_t first_byte = fgetc(compressed_file);
00025     uint16_t second_byte = fgetc(compressed_file);
00026
00027     uint16_t trash_and_tree_size = 0;
00028     trash_and_tree_size |= first_byte;
00029     trash_and_tree_size «= 8;
00030     trash_and_tree_size |= second_byte;
00031
00032     *trashSize = trash_and_tree_size » 13;
00033     trash_and_tree_size «= 3;
00034     *treeSize = trash_and_tree_size » 3;
00035 }
```

4.1.1.3 getTree()

```
NODE * getTree (
              FILE * archive,
              int * treeSize )
```

Refaz a árvore de Huffman a partir do cabeçalho.

Parâmetros

archive	Arquivo compactado
treeSize	Ponteiro para o arquivo compactado

Retorna

A raiz da árvore de Huffman

```
00037                                           {
00038     uint8_t byte;
00039     NODE *huffTree = NULL;
00040 
00041     //tratando caractere especial (sempre folha)
00042     if(*treeSize > 0){
00043         fread(&byte,sizeof(uint8_t),1,archive);
00044         if(byte == '\\'){
00045             (*treeSize)--;
00046             fread(&byte,sizeof(uint8_t),1,archive);
00047             huffTree = create_node();
00048             *(uint8_t*)huffTree->data = byte;
00049             (*treeSize)--;
00050             return huffTree;
00051         }
00052         huffTree = create_node();
00053         *(uint8_t*)huffTree->data = byte;
00054         (*treeSize)--;
00055 
00056         //se for um nó intermediario
00057         if(byte == '*'){
00058             //busca primeiro na esquerda até acha uma folha
00059             huffTree->left = getTree(archive,treeSize);
00060             huffTree->right = getTree(archive,treeSize);
00061         }
00062         return huffTree;
00063     }
00064     return huffTree;
00065 }
```

4.1.1.4 is_seted()

```
bool is_seted (
              uint8_t byte,
              int i )
```

Verifica se o byte está setado na posição i.

Parâmetros

byte	O byte que será avaliado
i	O indice do byte que iremos verificar

Retorna

Se está ou não setado

```
00067                                       {
00068     uint8_t mask = 1; // coloca o bit 1 na posicao i
00069     mask «= i;
00070     return byte & mask; // Se o bit nao estiver setado retorna 0
00071 }
```

4.1.1.5 unzip()

```
void unzip (
              FILE * compressed_file,
              FILE * fileOut,
              unsigned long long coded_size,
              NODE * treeRoot,
              int trashSize )
```

Descomprimi o arquivo compactado.

Parâmetros

compressed_file	O arquivo compactado
FileOut	O arquivo descompactado
coded_size	O tamanho total da codificação
treeRoot	A raiz da árvore de Huffman
trashSize	O tamanho do lixo

```
00073
      {
00074     unsigned long long int index;
00075     uint8_t cmpByte;
00076     NODE* aux = treeRoot;
00077     int j = 7;
00078
00079     for(index = 0; index < coded_size; index++) {
00080         j = 7;
00081         cmpByte = fgetc(compressed_file);
00082
00083         while(j >= 0){
00084             if(is_seted(cmpByte, j))
00085                 treeRoot = treeRoot->right;
00086             else
00087                 treeRoot = treeRoot->left;
00088
00089             if(is_leaf(treeRoot)){
00090                 uint8_t byte = get_data(treeRoot);
00091                 fprintf(fileOut, "%c", byte);
00092                 treeRoot = aux;
00093             }
00094             if ((index == coded_size - 1) && (j == trashSize))
00095                 break;
00096             j--;
00097         }
00098     }
00099 }
```

4.2 decode.h

Ir para a documentação desse arquivo.

```
00001 #ifndef DECODE_H
00002 #define DECODE_H
00003 #include "list.h"
00004
00012 unsigned long long int find_file_size(FILE* file);
00013
00022 bool is_seted(uint8_t byte, int i);
00023
00032 void get_tree_and_trash_size(FILE* compressed_file, int* trashSize, int* treeSize);
00033
00042 NODE *getTree(FILE *archive,int *treeSize);
00043
00054 void unzip(FILE* compressed_file, FILE* fileOut, unsigned long long coded_size, NODE* treeRoot, int
      trashSize);
00055
00056 #endif
```

4.3 Referência do Arquivo inc/encode.h

```
#include "list.h"
```

Estruturas de Dados

- struct BitHuff

  Está estrutura representa o byte codificado.

Definições de Tipos

- typedef struct BitHuff BitHuff

  Está estrutura representa o byte codificado.

Funções

- unsigned int * init_frequency_table ()

  Inicializa a tabela de frequência de bytes.
- void count_frequency (unsigned int *frequency_table, char *file_name)

  Conta a frequencia de cada byte.
- void set_bit (int j, uint8_t *byte)

  Ativa o bit na posição j.
- void write_header (FILE *cmp_file, unsigned short int tree_and_trash_size, NODE *root)

  Escreve o header no arquivo compactado.
- void write_tree (NODE *root, FILE *cmp_file)

  Escreve a árvore de Huffman em pré-ordem no arquivo compactado.
- unsigned short int setFirstTwoBytes (unsigned short int trash_size, unsigned short int tree_size)

  Inicializa os dois primeiros bytes.
- void buildTable (NODE *tree_root, BitHuff table[ ], BitHuff code)

  Constroi o dicionário que mapeia cada byte a sua codificação.
- unsigned short int getTrashSize (unsigned int frequency[ ], BitHuff table[ ])

  Calcula o tamanho do lixo.
- void getTreeSize (NODE *tree_root, unsigned short int *treeSize)

  Calcula o tamanho da árvore contabilizando também o scape.
- bool isSetedLong (unsigned long long int code, int i)

  Verifica se o bit no indice i da codificação de Huffman está ativado.
- void setBytes (FILE *fileIn, FILE *fileOut, BitHuff table[ ])

  Escreve a codificação de Huffman completa no arquivo compactado.

4.3.1 Definições dos tipos

4.3.1.1 BitHuff

```
typedef struct BitHuff BitHuff
```

Está estrutura representa o byte codificado.

4.3.2 Funções

4.3.2.1 buildTable()

```
void buildTable (
            NODE * tree_root,
            BitHuff table[],
            BitHuff code )
```

Constroi o dicionário que mapeia cada byte a sua codificação.

Parâmetros

tree_root	A raiz da árvore de Huffman
table	O dicionário
code	A codificação de Huffman

```
00080                                                                   {
00081     if(is_leaf(tree_node))
00082     {
00083         table[get_data(tree_node)] = code;
00084         return;
00085     }
00086     else
00087     {
00088         code.size++;
00089         code.bitH «= 1;
00090         if(tree_node->left != NULL)
00091         buildTable(tree_node->left, table, code);
00092         code.bitH++;
00093         if(tree_node->right != NULL)
00094         buildTable(tree_node->right, table, code);
00095     }
00096 }
```

4.3.2.2 count_frequency()

```
void count_frequency (
             unsigned int * frequency_table,
             char * file_name )
```

Conta a frequencia de cada byte.

Parâmetros

frequency_table	A tabela de frequência dos bytes
file_name	Nome do camninho do arquivo de entrada (arquivo que será compactado)

```
00016                                                                   {
00017     FILE* file = fopen(file_name, "rb");
00018     file_error_reporter(file);
00019     uint8_t byte;
00020
00021     while(fread(&byte, sizeof(uint8_t), 1, file) == 1){ // Mudar vericacao?
00022         frequency_table[byte]++;
00023     }
00024
00025     fclose(file);
00026 }
```

4.3.2.3 getTrashSize()

```
unsigned short int getTrashSize (
             unsigned int frequency[],
             BitHuff table[] )
```

Calcula o tamanho do lixo.

Parâmetros

frequency	A tabela de frequência dos bytes
table	O dicionário

Retorna

O tamanho do lixo

```
00099                                                                   {
00100
00101     unsigned long long int totalBits = 0;
00102     for(int i = 0;i < 256;i++){
00103         if(frequency[i] > 0) {
00104             totalBits += frequency[i] * table[i].size;
00105         }
00106     }
00107     unsigned short int trash = (8 - (totalBits % 8));
00108     return trash;
00109 }
```

4.3.2.4 getTreeSize()

```
void getTreeSize (
            NODE * tree_root,
            unsigned short int * treeSize )
```

Calcula o tamanho da árvore contabilizando também o scape.

Parâmetros

tree_root	A raiz da árvore de Huffman
treeSize	Ponteiro para o tamanho do lixo

```
00111                                                                   {
00112     if(tree_root != NULL) {
00113         if((is_leaf(tree_root)) && (get_data(tree_root) == '*' || get_data(tree_root) == '\\'))
00114             (*treeSize) += 2;
00115         else
00116             (*treeSize)++;
00117
00118         getTreeSize(tree_root->left, treeSize);
00119         getTreeSize(tree_root->right, treeSize);
00120     }
00121 }
```

4.3.2.5 init_frequency_table()

```
unsigned int * init_frequency_table ( )
```

Inicializa a tabela de frequência de bytes.

Todos os campos alocados são inicializados com zero

Retorna

A tabela de frequência de bytes

```
00007                                   {
00008
00009   unsigned int* frequency_table = calloc(TAM, sizeof(unsigned int));
00010   malloc_error_reporter(frequency_table);
00011
00012   return frequency_table;
00013
00014 }
```

4.3.2.6 isSetedLong()

```
bool isSetedLong (
            unsigned long long int code,
            int i )
```

Verifica se o bit no indice i da codificação de Huffman está ativado.

Parâmetros

code	Codificação de Huffman de um byte x
i	Indice que será verificado

Retorna

Se o bit na posição i está setado ou não

```
00122                                                   {
00123     unsigned long long int mask = 1;
00124     mask «= i;
00125     // 1000101 tam = 7
00126     // mask = 000000000000000001
00127     // 1000101
00128     // 1000000
00129     return code & mask;
00130 }
```

4.3.2.7 set_bit()

```
void set_bit (
             int j,
             uint8_t * byte )
```

Ativa o bit na posição j.

Parâmetros

j	Indice do bit a ser ativado
byte	Byte que terá seu bit ativado

```
00028                                   {
00029     uint8_t mask = 1; // 00000001
00030     mask = mask « j;
00031     // 10000000
00032     // 00000000
00033     // 10000000
00034     *byte = *byte | mask;
00035 }
```

4.3.2.8 setBytes()

```
void setBytes (
             FILE * fileIn,
             FILE * fileOut,
             BitHuff table[] )
```

Escreve a codificação de Huffman completa no arquivo compactado.

Lê o arquivo original byte por byte, busca a codificação do byte no dicionário, Cria um novo byte compactado, e escreve no compactado

Parâmetros

fileIn	Arquivo de entrada (Que será compactado)
fileOut	Arquivo compactado
table	Dicionário com a codificação de Huffman de cada byte

```
00132                                                   {
00133     unsigned char buffer = 0; // Buffer que sera escrito no arquivo quando cheio
00134     unsigned char original_byte;
00135     int buffer_index = 7;
00136 
00137     while(fread(&original_byte, sizeof(unsigned char), 1, fileIn) == 1) {
00138         unsigned long long int huffCode = table[original_byte].bitH;
00139         // 01011001
00140         // char exemplo[255]
00141         // 0 254
00142 
00143         for(int i = table[original_byte].size; i > 0; i--){
00144             if(isSetedLong(huffCode, i - 1)){
00145                 set_bit(buffer_index, &buffer);
00146             }
00147             buffer_index--;
00148             if(buffer_index < 0){
00149                 fprintf(fileOut, "%c", buffer);
00150                 buffer = 0;
00151                 buffer_index = 7;
00152             }
00153         }
00154     }
00155     if(buffer_index != 7){
00156         fprintf(fileOut, "%c", buffer);
00157     }
00158 
00159     fclose(fileIn);
00160     fclose(fileOut);
00161 }
```

4.3.2.9 setFirstTwoBytes()

```
unsigned short int setFirstTwoBytes (
            unsigned short int trash_size,
            unsigned short int tree_size )
```

Inicializa os dois primeiros bytes.

Parâmetros

trash_size	O tamanho do lixo
tree_size	O tamanho da árvore

Tamanho do lixo: Primeiros 3 bits. Tamanho da árvore: próximos 13 bits

Retorna

Um unsigned short setado com o tamanho do lixo e tamanho da árvore

```
00073                                                                                   {
00074     unsigned short int tree_and_trash_size = 0;
00075     trash_size «= 13;
00076     tree_and_trash_size = tree_size | trash_size;
00077     return tree_and_trash_size;
00078 }
```

4.3.2.10 write_header()

```
void write_header (
            FILE * cmp_file,
            unsigned short int tree_and_trash_size,
            NODE * root )
```

Escreve o header no arquivo compactado.

Parâmetros

cmp_file	O arquivo compactado
tree_and_trash_size	Os dois primeiros bytes do arquivo compactado (lixo e tamanho da árvore)
root	Raiz da árvore de Huffman

```
00037                                                                                                                {
00038
00039
00040     // 10100000 00100100
00041     // 00000000 10100000 first_byte
00042     // 00100100 00000000 «= 8
00043     // »= 8
00044     // 00000000 00100100 second_byte
00045
00046
00047     unsigned char first_byte = tree_and_trash_size » 8;
00048     tree_and_trash_size «= 8;
00049     tree_and_trash_size »= 8;
00050     unsigned char second_byte = tree_and_trash_size;
00051
00052     fwrite(&first_byte, sizeof(unsigned char), 1, cmp_file);
00053     fwrite(&second_byte, sizeof(unsigned char), 1, cmp_file);
00054     write_tree(root, cmp_file);
00055 }
```

4.3.2.11 write_tree()

```
void write_tree (
             NODE * root,
             FILE * cmp_file )
```

Escreve a árvore de Huffman em pré-ordem no arquivo compactado.

Parâmetros

root	A raiz da árvore de Huffman
cmp_file	O arquivo compactado

```
00057                                                {
00058
00059     if (root != NULL) {
00060         uint8_t data = get_data(root); // Pega o byte do no
00061         if(((data == '*') || (data == '\\')) && is_leaf(root)){
00062             uint8_t temp = '\\';
00063             fwrite(&temp, sizeof(uint8_t), 1, cmp_file); // Escreve scape
00064         }
00065
00066         fwrite(&data, sizeof(uint8_t), 1, cmp_file); // Escreve o byte no arquivo
00067         write_tree(root->left, cmp_file); // Percorre os filhos esquerdos
00068         write_tree(root->right, cmp_file); // Percorre os filhos direitos
00069     }
00070 }
```

4.4 encode.h

Ir para a documentação desse arquivo.

```
00001 #ifndef ENCODE_H
00002 #define ENCODE_H
00003 #include "list.h"
00004
00009 typedef struct BitHuff {
00010     unsigned long long int bitH;
00011     int size;
00012 } BitHuff;
00013
00022 unsigned int* init_frequency_table();
```

```
00023
00031 void count_frequency(unsigned int* frequency_table, char* file_name);
00032
00040 void set_bit(int j, uint8_t* byte);
00041
00050 void write_header(FILE* cmp_file, unsigned short int tree_and_trash_size, NODE* root);
00051
00059 void write_tree(NODE* root, FILE* cmp_file);
00060
00072 unsigned short int setFirstTwoBytes(unsigned short int trash_size, unsigned short int tree_size);
00073
00082 void buildTable(NODE* tree_root, BitHuff table[], BitHuff code);
00083
00092  unsigned short int getTrashSize(unsigned int frequency[], BitHuff table[]);
00093
00100 void getTreeSize(NODE* tree_root, unsigned short int* treeSize);
00101
00110 bool isSetedLong(unsigned long long int code, int i);
00111
00112
00123 void setBytes(FILE *fileIn,FILE *fileOut,BitHuff table[]);
00124
00125
00126 #endif
```

4.5 Referência do Arquivo inc/list.h

Estruturas de Dados

- struct NODE

  Estrutura de um nó
- struct HEAD

  Estrutura da lista encadaeada e árvore.

Definições de Tipos

- typedef struct NODE NODE

  Estrutura de um nó
- typedef struct HEAD HEAD

  Estrutura da lista encadaeada e árvore.

Funções

- void malloc_error_reporter (void ∗ptr)

  Verifica se a alocação dinâmica foi bem sucedida.
- void file_error_reporter (FILE ∗file)

  Funcao para reportar possível erro ao abrir arquivos.
- void init_struct (HEAD ∗head)

  Prepara a estrutura para uso.
- NODE ∗ create_node ()

  Cria um nó genérico.
- void insert_in_linked_list (HEAD ∗Mystruct, unsigned int ∗frequency_table)

  Prepara o nó que será inserido na lista encadeada.
- void insert_sorted (HEAD ∗Mystruct, NODE ∗new_node)

  Insere o nó de maneira crescente.
- uint8_t removeFirst (NODE ∗∗list, int ∗currentSize)

  Remove logicamente o primeiro nó da lista.

4.5.1 Definições dos tipos

4.5.1.1 HEAD

```
typedef struct HEAD HEAD
```

Estrutura da lista encadaeada e árvore.

Esta estrutura representa tanto a cabeça da lista encadeada quanto a raiz da árvoreas

4.5.1.2 NODE

```
typedef struct NODE NODE
```

Estrutura de um nó

Esta estrutura representa o nó da lista encadeada e árvore

4.5.2 Funções

4.5.2.1 create_node()

```
NODE * create_node ( )
```

Cria um nó genérico.

: Inicializa data com '*'. Inicializa frequency com zero

Retorna

Retorna o novo nó

```
00026                   {
00027     NODE* new_node = malloc(sizeof(NODE));
00028     malloc_error_reporter(new_node);
00029
00030     new_node->data = malloc(sizeof(uint8_t));
00031     malloc_error_reporter(new_node->data);
00032     new_node->frequency = malloc(sizeof(int));
00033     malloc_error_reporter(new_node->frequency);
00034
00035     *(uint8_t*)new_node->data = '*';
00036     *(int*)new_node->frequency = 0;
00037     new_node->left = NULL;
00038     new_node->right = NULL;
00039     new_node->next = NULL;
00040
00041     return new_node;
00042 }
```

4.5.2.2 file_error_reporter()

```
void file_error_reporter (
             FILE * file )
```

Funcao para reportar possível erro ao abrir arquivos.

Parâmetros

file	Arquivo que será verificado

```
00014                                   {
00015     if(ptr == NULL){
00016         perror("Erro ao abrir arquivo");
00017         exit(EXIT_FAILURE);
00018     }
00019 }
```

4.5.2.3 init_struct()

```
void init_struct (
            HEAD * head )
```

Prepara a estrutura para uso.

Parâmetros

head	Estrutura que será preparada

```
00021                                   {
00022   Mystruct->head = NULL;
00023   Mystruct->size = 0;
00024 }
```

4.5.2.4 insert_in_linked_list()

```
void insert_in_linked_list (
            HEAD * Mystruct,
            unsigned int * frequency_table )
```

Prepara o nó que será inserido na lista encadeada.

Funcao auxiliar de insert sorted. Esta funcao já prepara o novo no a ser inserido de maneira ordenada

Parâmetros

Mystruct	A estrutura da lista/árvore
frequency_table	A tabela de freqência de bytes

```
00044                                                                                   {
00045     for(unsigned int i = 0; i<TAM; i++){
00046         if(frequency_table[i] > 0){
00047
00048         NODE* new_node = create_node(); // '*'
00049         *((uint8_t*)(new_node->data)) = i; // Desrreferencia o ponteiro e converte um int para byte
00050         *((int*)(new_node->frequency)) = frequency_table[i];
00051
00052         insert_sorted(Mystruct, new_node);
00053     }
00054   }
00055 }
```

4.5.2.5 insert_sorted()

```
void insert_sorted (
            HEAD * Mystruct,
            NODE * new_node )
```

Insere o nó de maneira crescente.

Parâmetros

Mystruct	Estrutura da lista/árvore
new_node	Nó que será inserido

```
00057                                                   {
00058     NODE* current = Mystruct->head;
00059     NODE* prev = NULL;
00060
00061     // Encontre o local adequado para inserir o novo nó com base na frequência
00062     while (current != NULL && get_frequency(current) < get_frequency(new_node)) {
00063         prev = current;
00064         current = current->next;
00065     }
00066
00067     // Insira o novo nó na posição correta
00068     if (prev == NULL) {
00069         new_node->next = Mystruct->head;
00070         Mystruct->head = new_node;
00071     } else {
00072         prev->next = new_node;
00073         new_node->next = current;
00074     }
00075     Mystruct->size++;
00076 }
```

4.5.2.6 malloc_error_reporter()

```
void malloc_error_reporter (
             void * ptr )
```

Verifica se a alocação dinâmica foi bem sucedida.

Parâmetros

ptr	Ponteiro genérico

```
00007                                       {
00008     if(ptr == NULL){
00009         perror("Erro na alocação de memória");
00010         exit(EXIT_FAILURE);
00011     }
00012 }
```

4.5.2.7 removeFirst()

```
uint8_t removeFirst (
             NODE ** list,
             int * currentSize )
```

Remove logicamente o primeiro nó da lista.

Parâmetros

list	Ponteiro para a cabeça da lista
currentSize	Ponteiro para o tamanho da estrutura

Retorna

byte do nó que foi removido

```
00078                                                   {
00079     uint8_t item = get_data(*list_head);
00080     (*list_head) = (*list_head)->next;
00081     (*currentSize)--;
00082     return item;
00083 }
```

4.6 list.h

Ir para a documentação desse arquivo.

```
00001 #ifndef LIST_H
00002 #define LIST_H
00003
00010 typedef struct NODE {
00011   void *data;
00012   void* frequency;
00013   struct NODE* left;
00014   struct NODE* right;
00015   struct NODE* next;
00016 } NODE;
00017
00023 typedef struct HEAD{
00024   struct NODE* head;
00025   int size;
00026 } HEAD;
00027
00034 void malloc_error_reporter(void* ptr);
00035
00041 void file_error_reporter(FILE* file);
00042
00048 void init_struct(HEAD* head);
00049
00057 NODE* create_node();
00058
00068 void insert_in_linked_list(HEAD* Mystruct, unsigned int* frequency_table);
00069
00077 void insert_sorted(HEAD* Mystruct, NODE* new_node);
00078
00087 uint8_t removeFirst(NODE **list, int *currentSize);
00088
00089 #endif
00090
```

4.7 Referência do Arquivo inc/std.h

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <string.h>
#include <stdint.h>
```

Definições e Macros

- #define INPUT_DIR "input/"
- #define UNZIPED_OUT_DIR "unziped/"
- #define ZIPED_OUT_DIR "ziped/"
- #define TAM 256

4.7.1 Definições e macros

4.7.1.1 INPUT_DIR

```
#define INPUT_DIR "input/"
```

4.7.1.2 TAM

```
#define TAM 256
```

4.7.1.3 UNZIPED_OUT_DIR

```
#define UNZIPED_OUT_DIR "unziped/"
```

4.7.1.4 ZIPED_OUT_DIR

```
#define ZIPED_OUT_DIR "ziped/"
```

4.8 std.h

Ir para a documentação desse arquivo.

```
00001 #ifndef STD_H
00002 #define STD_H
00003
00004 #include <stdbool.h>
00005 #include <stdio.h>
00006 #include <stdlib.h>
00007 #include <locale.h>
00008 #include <string.h>
00009 #include <stdint.h>
00010 // Definindo diretorios de entrada/saida
00011 #define INPUT_DIR "input/"
00012 #define UNZIPED_OUT_DIR "unziped/"
00013 #define ZIPED_OUT_DIR "ziped/"
00014 #define TAM 256
00015
00016 #endif
```

4.9 Referência do Arquivo inc/tree.h

```
#include "std.h"
#include "list.h"
#include "decode.h"
#include "encode.h"
```

Funções

- bool is_leaf (NODE *tree_node)

  Verifica se o nó é uma folha.
- uint8_t get_data (NODE *node)

  Pega o byte do nó x.
- void pre_order_trasversal (NODE *tree_root)

  Exibe a árvore em pré-ordem.
- void huffmanTree (HEAD *myStruct, int *currentSize)

  Constrói a árvore de Huffman.
- int get_frequency (NODE *node)

  Pega a frquência do nó x.

4.9.1 Funções

4.9.1.1 get_data()

```
uint8_t get_data (
            NODE * node )
```

Pega o byte do nó x.

Parâmetros

node	Nó da estrutura

Retorna

byte do nó

```
00011                               {
00012     if(node == NULL){
00013         puts("NO NULO");
00014     }
00015     uint8_t *data = (uint8_t*)node->data;
00016     return *data;
00017 }
```

4.9.1.2 get_frequency()

```
int get_frequency (
            NODE * node )
```

Pega a frquência do nó x.

Parâmetros

node	Nó lista/árvore

Retorna

A frequência do nó x

```
00018                                {
00019     if(node == NULL){
00020         puts("NO NULO");
00021         return -1;
00022     }
00023     int* frequency = (int*)node->frequency;
00024     return *frequency;
00025 }
```

4.9.1.3 huffmanTree()

```
void huffmanTree (
              HEAD * myStruct,
              int * currentSize )
```

Constrói a árvore de Huffman.

Constrói a árvore a partir da lista encadeada

Parâmetros

myStruct	Estrutura da lista/árvore
currentSize	Ponteiro para o tamanho da estrutura

```
00038                                                {
00039     while(*currentSize > 1){
00040         int frequency = get_frequency(myStruct->head) + get_frequency(myStruct->head->next);// soma
      das frequências dos primeiros nós
00041         NODE* newNode = create_node();
00042         *((int*)(newNode->frequency)) = frequency;
00043 
00044         newNode->left = myStruct->head;// esquerda do novo nó vai receber o primeiro elemento
00045         removeFirst(&myStruct->head,currentSize);
00046 
00047         newNode->right = (myStruct->head);//direita do novo nó vai receber o segundo elemento
00048         removeFirst(&myStruct->head,currentSize);
00049 
00050         insert_sorted(myStruct, newNode);// adiciona o novo nó de maneira ordenada
00051     }
00052     return;
00053 }
```

4.9.1.4 is_leaf()

```
bool is_leaf (
              NODE * tree_node )
```

Verifica se o nó é uma folha.

Parâmetros

tree_node	Nó da árvore de Huffman

Retorna

Se é ou não uma folha

```
00007                          {
00008     return node->left == NULL && node->right == NULL;
00009 }
```

4.9.1.5 pre_order_trasversal()

```
void pre_order_trasversal (
            NODE * tree_root )
```

Exibe a árvore em pré-ordem.

```
00027                                       {
00028     if (root == NULL) {
00029         return;
00030     }
00031
00032     printf("%c", get_data(root));
00033     pre_order_trasversal(root->left);
00034     pre_order_trasversal(root->right);
00035
00036 }
```

4.10 tree.h

Ir para a documentação desse arquivo.

```
00001 #ifndef TREE_H
00002 #define TREE_H
00003 #include "std.h"
00004 #include "list.h"
00005 #include "decode.h"
00006 #include "encode.h"
00007
00015 bool is_leaf(NODE* tree_node);
00016
00024 uint8_t get_data(NODE* node);
00025
00029 void pre_order_trasversal(NODE* tree_root);
00030
00039 void huffmanTree(HEAD* myStruct,int *currentSize);
00040
00048 int get_frequency(NODE* node);
00049
00050 #endif
```

4.11 Referência do Arquivo src/decode.c

```
#include "std.h"
#include "list.h"
#include "tree.h"
#include "decode.h"
#include "encode.h"
```

Funções

- unsigned long long int find_file_size (FILE *file)

  retorna o tamanho do arquivo compactado
- void get_tree_and_trash_size (FILE *compressed_file, int *trashSize, int *treeSize)

  Pega o tamanho da árvore e o tamanho do lixo do cabeçalho.
- NODE * getTree (FILE *archive, int *treeSize)

  Refaz a árvore de Huffman a partir do cabeçalho.
- bool is_seted (uint8_t byte, int i)

  Verifica se o byte está setado na posição i.
- void unzip (FILE *compressed_file, FILE *fileOut, unsigned long long coded_size, NODE *treeRoot, int trashSize)

  Descomprimi o arquivo compactado.

4.11.1 Funções

4.11.1.1 find_file_size()

```
unsigned long long int find_file_size (
            FILE * file )
```

retorna o tamanho do arquivo compactado

Parâmetros

file	O arquivo compactado

Retorna

O tamanho do arquivo compactado

```
00007                                                 {
00008
00009     if (file == NULL) {
00010         perror("Erro ao abrir o arquivo");
00011         return 1;
00012     }
00013
00014     unsigned long long int current_postition = ftell(file);
00015
00016     fseek(file, 0, SEEK_END); // Move o indicador de posição para o final do arquivo
00017     unsigned long long int file_size = ftell(file);   // Obtém a posição atual, que é o tamanho do
      arquivo em bytes
00018     fseek(file, current_postition, SEEK_SET); // Move o indicador de posição de volta para o início do
      arquivo
00019
00020     return file_size;
00021 }
```

4.11.1.2 get_tree_and_trash_size()

```
void get_tree_and_trash_size (
            FILE * compressed_file,
            int * trashSize,
            int * treeSize )
```

Pega o tamanho da árvore e o tamanho do lixo do cabeçalho.

Parâmetros

compressed_file	O arquivo compactado
trashSize	Ponteiro para o tamanho do lixo
treeSize	Ponteiro para o tamanho da árvore

```
00023                                                                                 {
00024     uint16_t first_byte = fgetc(compressed_file);
00025     uint16_t second_byte = fgetc(compressed_file);
00026
00027     uint16_t trash_and_tree_size = 0;
00028     trash_and_tree_size |= first_byte;
00029     trash_and_tree_size «= 8;
00030     trash_and_tree_size |= second_byte;
00031
00032     *trashSize = trash_and_tree_size » 13;
00033     trash_and_tree_size «= 3;
00034     *treeSize = trash_and_tree_size » 3;
00035 }
```

4.11.1.3 getTree()

```
NODE * getTree (
            FILE * archive,
            int * treeSize )
```

Refaz a árvore de Huffman a partir do cabeçalho.

Parâmetros

archive	Arquivo compactado
treeSize	Ponteiro para o arquivo compactado

Retorna

A raiz da árvore de Huffman

```
00037                                           {
00038     uint8_t byte;
00039     NODE *huffTree = NULL;
00040 
00041     //tratando caractere especial (sempre folha)
00042     if(*treeSize > 0){
00043         fread(&byte,sizeof(uint8_t),1,archive);
00044         if(byte == '\\'){
00045             (*treeSize)--;
00046             fread(&byte,sizeof(uint8_t),1,archive);
00047             huffTree = create_node();
00048             *(uint8_t*)huffTree->data = byte;
00049             (*treeSize)--;
00050             return huffTree;
00051         }
00052         huffTree = create_node();
00053         *(uint8_t*)huffTree->data = byte;
00054         (*treeSize)--;
00055 
00056         //se for um nó intermediario
00057         if(byte == '*'){
00058             //busca primeiro na esquerda até acha uma folha
00059             huffTree->left = getTree(archive,treeSize);
00060             huffTree->right = getTree(archive,treeSize);
00061         }
00062         return huffTree;
00063     }
00064     return huffTree;
00065 }
```

4.11.1.4 is_seted()

```
bool is_seted (
            uint8_t byte,
            int i )
```

Verifica se o byte está setado na posição i.

Parâmetros

byte	O byte que será avaliado
i	O indice do byte que iremos verificar

Retorna

Se está ou não setado

```
00067                                   {
00068     uint8_t mask = 1; // coloca o bit 1 na posicao i
00069     mask «= i;
00070     return byte & mask; // Se o bit nao estiver setado retorna 0
00071 }
```

4.11.1.5 unzip()

```
void unzip (
            FILE * compressed_file,
            FILE * fileOut,
            unsigned long long coded_size,
            NODE * treeRoot,
            int trashSize )
```

Descomprimi o arquivo compactado.

Parâmetros

compressed_file	O arquivo compactado
FileOut	O arquivo descompactado
coded_size	O tamanho total da codificação
treeRoot	A raiz da árvore de Huffman
trashSize	O tamanho do lixo

```
00073
     {
00074     unsigned long long int index;
00075     uint8_t cmpByte;
00076     NODE* aux = treeRoot;
00077     int j = 7;
00078
00079     for(index = 0; index < coded_size; index++) {
00080         j = 7;
00081         cmpByte = fgetc(compressed_file);
00082
00083         while(j >= 0){
00084             if(is_seted(cmpByte, j))
00085                 treeRoot = treeRoot->right;
00086             else
00087                 treeRoot = treeRoot->left;
00088
00089             if(is_leaf(treeRoot)){
00090                 uint8_t byte = get_data(treeRoot);
00091                 fprintf(fileOut, "%c", byte);
00092                 treeRoot = aux;
00093             }
00094             if ((index == coded_size - 1) && (j == trashSize))
00095                 break;
00096             j--;
00097         }
00098     }
00099 }
```

4.12 Referência do Arquivo src/encode.c

```
#include "std.h"
#include "list.h"
#include "tree.h"
#include "decode.h"
#include "encode.h"
```

Funções

- unsigned int * init_frequency_table ()

  Inicializa a tabela de frequência de bytes.
- void count_frequency (unsigned int *frequency_table, char *file_name)

  Conta a frequencia de cada byte.
- void set_bit (int j, uint8_t *byte)

  Ativa o bit na posição j.
- void write_header (FILE *cmp_file, unsigned short int tree_and_trash_size, NODE *root)

  Escreve o header no arquivo compactado.
- void write_tree (NODE *root, FILE *cmp_file)

  Escreve a árvore de Huffman em pré-ordem no arquivo compactado.
- unsigned short int setFirstTwoBytes (unsigned short int trash_size, unsigned short int tree_size)

  Inicializa os dois primeiros bytes.
- void buildTable (NODE *tree_node, BitHuff table[ ], BitHuff code)

  Constroi o dicionário que mapeia cada byte a sua codificação.
- unsigned short int getTrashSize (unsigned int frequency[ ], BitHuff table[ ])

  Calcula o tamanho do lixo.
- void getTreeSize (NODE *tree_root, unsigned short int *treeSize)

  Calcula o tamanho da árvore contabilizando também o scape.
- bool isSetedLong (unsigned long long int code, int i)

  Verifica se o bit no indice i da codificação de Huffman está ativado.
- void setBytes (FILE *fileIn, FILE *fileOut, BitHuff table[ ])

  Escreve a codificação de Huffman completa no arquivo compactado.

4.12.1 Funções

4.12.1.1 buildTable()

```
void buildTable (
            NODE * tree_root,
            BitHuff table[],
            BitHuff code )
```

Constroi o dicionário que mapeia cada byte a sua codificação.

Parâmetros

tree_root	A raiz da árvore de Huffman
table	O dicionário
code	A codificação de Huffman

```
00080                                                                {
00081     if(is_leaf(tree_node))
00082     {
00083         table[get_data(tree_node)] = code;
00084         return;
00085     }
00086     else
00087     {
00088         code.size++;
00089         code.bitH «= 1;
00090         if(tree_node->left != NULL)
00091         buildTable(tree_node->left, table, code);
00092         code.bitH++;
00093         if(tree_node->right != NULL)
```

```
00094         buildTable(tree_node->right, table, code);
00095     }
00096 }
```

4.12.1.2 count_frequency()

```
void count_frequency (
             unsigned int * frequency_table,
             char * file_name )
```

Conta a frequencia de cada byte.

Parâmetros

frequency_table	A tabela de frequência dos bytes
file_name	Nome do camninho do arquivo de entrada (arquivo que será compactado)

```
00016                                                                    {
00017     FILE* file = fopen(file_name, "rb");
00018     file_error_reporter(file);
00019     uint8_t byte;
00020
00021     while(fread(&byte, sizeof(uint8_t), 1, file) == 1){ // Mudar vericacao?
00022         frequency_table[byte]++;
00023     }
00024
00025     fclose(file);
00026 }
```

4.12.1.3 getTrashSize()

```
unsigned short int getTrashSize (
             unsigned int frequency[],
             BitHuff table[] )
```

Calcula o tamanho do lixo.

Parâmetros

frequency	A tabela de frequência dos bytes
table	O dicionário

Retorna

O tamanho do lixo

```
00099                                                                        {
00100
00101     unsigned long long int totalBits = 0;
00102     for(int i = 0;i < 256;i++){
00103         if(frequency[i] > 0) {
00104             totalBits += frequency[i] * table[i].size;
00105         }
00106     }
00107     unsigned short int trash = (8 - (totalBits % 8));
00108     return trash;
00109 }
```

4.12.1.4 getTreeSize()

```
void getTreeSize (
```

```
            NODE * tree_root,
            unsigned short int * treeSize )
```

Calcula o tamanho da árvore contabilizando também o scape.

Parâmetros

tree_root	A raiz da árvore de Huffman
treeSize	Ponteiro para o tamanho do lixo

```
00111                                                                   {
00112     if(tree_root != NULL) {
00113         if((is_leaf(tree_root)) && (get_data(tree_root) == '*' || get_data(tree_root) == '\\'))
00114             (*treeSize) += 2;
00115         else
00116             (*treeSize)++;
00117
00118         getTreeSize(tree_root->left, treeSize);
00119         getTreeSize(tree_root->right, treeSize);
00120     }
00121 }
```

4.12.1.5 init_frequency_table()

```
unsigned int * init_frequency_table ( )
```

Inicializa a tabela de frequência de bytes.

Todos os campos alocados são inicializados com zero

Retorna

A tabela de frequência de bytes

```
00007                                   {
00008
00009   unsigned int* frequency_table = calloc(TAM, sizeof(unsigned int));
00010   malloc_error_reporter(frequency_table);
00011
00012   return frequency_table;
00013
00014 }
```

4.12.1.6 isSetedLong()

```
bool isSetedLong (
            unsigned long long int code,
            int i )
```

Verifica se o bit no indice i da codificação de Huffman está ativado.

Parâmetros

code	Codificação de Huffman de um byte x
i	Indice que será verificado

Retorna

Se o bit na posição i está setado ou não

```
00122                                                   {
00123     unsigned long long int mask = 1;
00124     mask «= i;
00125     // 1000101 tam = 7
00126     // mask = 000000000000000001
00127     // 1000101
00128     // 1000000
00129     return code & mask;
00130 }
```

4.12.1.7 set_bit()

```
void set_bit (
              int j,
              uint8_t * byte )
```

Ativa o bit na posição j.

Parâmetros

j	Indice do bit a ser ativado
byte	Byte que terá seu bit ativado

```
00028                                   {
00029     uint8_t mask = 1; // 00000001
00030     mask = mask « j;
00031     // 10000000
00032     // 00000000
00033     // 10000000
00034     *byte = *byte | mask;
00035 }
```

4.12.1.8 setBytes()

```
void setBytes (
              FILE * fileIn,
              FILE * fileOut,
              BitHuff table[] )
```

Escreve a codificação de Huffman completa no arquivo compactado.

Lê o arquivo original byte por byte, busca a codificação do byte no dicionário, Cria um novo byte compactado, e escreve no compactado

Parâmetros

fileIn	Arquivo de entrada (Que será compactado)
fileOut	Arquivo compactado
table	Dicionário com a codificação de Huffman de cada byte

```
00132                                                           {
00133     unsigned char buffer = 0; // Buffer que sera escrito no arquivo quando cheio
00134     unsigned char original_byte;
00135     int buffer_index = 7;
00136
00137     while(fread(&original_byte, sizeof(unsigned char), 1, fileIn) == 1) {
00138         unsigned long long int huffCode = table[original_byte].bitH;
00139         // 01011001
```

```
        // char exemplo[255]
        // 0 254

        for(int i = table[original_byte].size; i > 0; i--){
            if(isSetedLong(huffCode, i - 1)){
                set_bit(buffer_index, &buffer);
            }
            buffer_index--;
            if(buffer_index < 0){
                fprintf(fileOut, "%c", buffer);
                buffer = 0;
                buffer_index = 7;
            }
        }
    }
    if(buffer_index != 7){
        fprintf(fileOut, "%c", buffer);
    }

    fclose(fileIn);
    fclose(fileOut);
}
```

4.12.1.9 setFirstTwoBytes()

```
unsigned short int setFirstTwoBytes (
            unsigned short int trash_size,
            unsigned short int tree_size )
```

Inicializa os dois primeiros bytes.

Parâmetros

trash_size	O tamanho do lixo
tree_size	O tamanho da árvore

Tamanho do lixo: Primeiros 3 bits. Tamanho da árvore: próximos 13 bits

Retorna

Um unsigned short setado com o tamanho do lixo e tamanho da árvore

```
{
    unsigned short int tree_and_trash_size = 0;
    trash_size «= 13;
    tree_and_trash_size = tree_size | trash_size;
    return tree_and_trash_size;
}
```

4.12.1.10 write_header()

```
void write_header (
            FILE * cmp_file,
            unsigned short int tree_and_trash_size,
            NODE * root )
```

Escreve o header no arquivo compactado.

Parâmetros

cmp_file	O arquivo compactado
tree_and_trash_size	Os dois primeiros bytes do arquivo compactado (lixo e tamanho da árvore)
root	Raiz da árvore de Huffman

```
00037                                                                                {
00038
00039
00040     // 10100000 00100100
00041     // 00000000 10100000 first_byte
00042     // 00100100 00000000 «= 8
00043     // »= 8
00044     // 00000000 00100100 second_byte
00045
00046
00047     unsigned char first_byte = tree_and_trash_size » 8;
00048     tree_and_trash_size «= 8;
00049     tree_and_trash_size »= 8;
00050     unsigned char second_byte = tree_and_trash_size;
00051
00052     fwrite(&first_byte, sizeof(unsigned char), 1, cmp_file);
00053     fwrite(&second_byte, sizeof(unsigned char), 1, cmp_file);
00054     write_tree(root, cmp_file);
00055 }
```

4.12.1.11 write_tree()

```
void write_tree (
             NODE * root,
             FILE * cmp_file )
```

Escreve a árvore de Huffman em pré-ordem no arquivo compactado.

Parâmetros

root	A raiz da árvore de Huffman
cmp_file	O arquivo compactado

```
00057                                          {
00058
00059     if (root != NULL) {
00060         uint8_t data = get_data(root); // Pega o byte do no
00061         if(((data == '*') || (data == '\\')) && is_leaf(root)){
00062             uint8_t temp = '\\';
00063             fwrite(&temp, sizeof(uint8_t), 1, cmp_file); // Escreve scape
00064         }
00065
00066         fwrite(&data, sizeof(uint8_t), 1, cmp_file); // Escreve o byte no arquivo
00067         write_tree(root->left, cmp_file); // Percorre os filhos esquerdos
00068         write_tree(root->right, cmp_file); // Percorre os filhos direitos
00069     }
00070 }
```

4.13 Referência do Arquivo src/list.c

```
#include "std.h"
#include "list.h"
#include "tree.h"
#include "decode.h"
#include "encode.h"
```

Funções

- void malloc_error_reporter (void *ptr)

  Verifica se a alocação dinâmica foi bem sucedida.
- void file_error_reporter (FILE *ptr)

Funcao para reportar possível erro ao abrir arquivos.

- void init_struct (HEAD *Mystruct)

  Prepara a estrutura para uso.

- NODE * create_node ()

  Cria um nó genérico.

- void insert_in_linked_list (HEAD *Mystruct, unsigned int *frequency_table)

  Prepara o nó que será inserido na lista encadeada.

- void insert_sorted (HEAD *Mystruct, NODE *new_node)

  Insere o nó de maneira crescente.

- uint8_t removeFirst (NODE **list_head, int *currentSize)

  Remove logicamente o primeiro nó da lista.

4.13.1 Funções

4.13.1.1 create_node()

```
NODE * create_node ( )
```

Cria um nó genérico.

: Inicializa data com '*'. Inicializa frequency com zero

Retorna

Retorna o novo nó

```
00026                   {
00027     NODE* new_node = malloc(sizeof(NODE));
00028     malloc_error_reporter(new_node);
00029
00030     new_node->data = malloc(sizeof(uint8_t));
00031     malloc_error_reporter(new_node->data);
00032     new_node->frequency = malloc(sizeof(int));
00033     malloc_error_reporter(new_node->frequency);
00034
00035     *(uint8_t*)new_node->data = '*';
00036     *(int*)new_node->frequency = 0;
00037     new_node->left = NULL;
00038     new_node->right = NULL;
00039     new_node->next = NULL;
00040
00041     return new_node;
00042 }
```

4.13.1.2 file_error_reporter()

```
void file_error_reporter (
             FILE * file )
```

Funcao para reportar possível erro ao abrir arquivos.

Parâmetros

file	Arquivo que será verificado

```
00014                                   {
00015     if(ptr == NULL){
00016         perror("Erro ao abrir arquivo");
```

```
00017         exit(EXIT_FAILURE);
00018     }
00019 }
```

4.13.1.3 init_struct()

```
void init_struct (
              HEAD * head )
```

Prepara a estrutura para uso.

Parâmetros

head	Estrutura que será preparada

```
00021                                 {
00022   Mystruct->head = NULL;
00023   Mystruct->size = 0;
00024 }
```

4.13.1.4 insert_in_linked_list()

```
void insert_in_linked_list (
              HEAD * Mystruct,
              unsigned int * frequency_table )
```

Prepara o nó que será inserido na lista encadeada.

Funcao auxiliar de insert sorted. Esta funcao já prepara o novo no a ser inserido de maneira ordenada

Parâmetros

Mystruct	A estrutura da lista/árvore
frequency_table	A tabela de freqência de bytes

```
00044                                                                 {
00045     for(unsigned int i = 0; i<TAM; i++){
00046         if(frequency_table[i] > 0){
00047 
00048         NODE* new_node = create_node(); // '*'
00049         *((uint8_t*)(new_node->data)) = i; // Desrreferencia o ponteiro e converte um int para byte
00050         *((int*)(new_node->frequency)) = frequency_table[i];
00051 
00052         insert_sorted(Mystruct, new_node);
00053     }
00054   }
00055 }
```

4.13.1.5 insert_sorted()

```
void insert_sorted (
              HEAD * Mystruct,
              NODE * new_node )
```

Insere o nó de maneira crescente.

Parâmetros

Mystruct	Estrutura da lista/árvore
new_node	Nó que será inserido

```
00057                                                   {
00058     NODE* current = Mystruct->head;
00059     NODE* prev = NULL;
00060 
00061     // Encontre o local adequado para inserir o novo nó com base na frequência
00062     while (current != NULL && get_frequency(current) < get_frequency(new_node)) {
00063         prev = current;
00064         current = current->next;
00065     }
00066 
00067     // Insira o novo nó na posição correta
00068     if (prev == NULL) {
00069         new_node->next = Mystruct->head;
00070         Mystruct->head = new_node;
00071     } else {
00072         prev->next = new_node;
00073         new_node->next = current;
00074     }
00075     Mystruct->size++;
00076 }
```

4.13.1.6 malloc_error_reporter()

```
void malloc_error_reporter (
             void * ptr )
```

Verifica se a alocação dinâmica foi bem sucedida.

Parâmetros

ptr	Ponteiro genérico

```
00007                                       {
00008     if(ptr == NULL){
00009         perror("Erro na alocação de memória");
00010         exit(EXIT_FAILURE);
00011     }
00012 }
```

4.13.1.7 removeFirst()

```
uint8_t removeFirst (
             NODE ** list,
             int * currentSize )
```

Remove logicamente o primeiro nó da lista.

Parâmetros

list	Ponteiro para a cabeça da lista
currentSize	Ponteiro para o tamanho da estrutura

Retorna

byte do nó que foi removido

```
00078                                                   {
00079     uint8_t item = get_data(*list_head);
00080     (*list_head) = (*list_head)->next;
00081     (*currentSize)--;
00082     return item;
00083 }
```

4.14 Referência do Arquivo src/main.c

```
#include "std.h"
#include "list.h"
#include "tree.h"
#include "decode.h"
#include "encode.h"
```

Funções

- int main ()

4.14.1 Funções

4.14.1.1 main()

```
int main ( )
00007           {
00008     int option;
00009
00010     puts("DIGITE A OPCAO: ");
00011     puts("COMPACTAR[1]");
00012     puts("DESCOMPACTAR[2]");
00013
00014     scanf("%d", &option);
00015
00016     if(option == 1){
00017
00018         HEAD Mystruct;
00019         init_struct(&Mystruct);
00020         unsigned int* frequency_table = init_frequency_table();
00021         char file_name[50];
00022         unsigned short int trash_size, tree_size = 0;
00023
00024         printf("Informe o nome do arquivo a ser compactado\n");
00025         scanf("%s", file_name);
00026
00027         char inPath[TAM]; // Caminho do input
00028         strcpy(inPath, INPUT_DIR); // Copia o diretorio padrao de inputs para inPath
00029         strcat(inPath, file_name); // Adciona o nome do arquivo
00030
00031         // Parte 1: Mapeia a frequência de cada caractere
00032         printf("\nPARTE 1\n");
00033         count_frequency(frequency_table, inPath);
00034
00035         // Parte 2: Cria a lista encadeada ordenada
00036         printf("\nPARTE 2\n");
00037         insert_in_linked_list(&Mystruct, frequency_table);
00038
00039         // Parte 3: Cria a ávore de Huffman
00040         printf("\nPARTE 3\n");
00041         huffmanTree(&Mystruct, &Mystruct.size);
00042         puts("Saiu tree");
00043
00044         // Parte 4: Cria um dicionário que mapeia cada caractere
00045         BitHuff code;
00046         BitHuff *table = malloc(sizeof(BitHuff) * 256);
00047         code.bitH = 0;
00048         code.size = 0;
00049         memset(table,0,sizeof(BitHuff) * 256); //
```

```
00050         printf("\nPARTE 4\n");
00051         buildTable(Mystruct.head, table, code); //
00052 
00053         // Parte 5: Compactar
00054         printf("\nPARTE 5\n");
00055         puts("Comprimindo arquivo");
00056         trash_size = getTrashSize(frequency_table, table);
00057         getTreeSize(Mystruct.head, &tree_size);
00058         unsigned short int tree_and_trash_size = setFirstTwoBytes(trash_size, tree_size);
00059 
00060         char *remove = strrchr(file_name,'.'); // remove o '.'
00061         if(remove != NULL) *remove = '\0'; // Adciona a char terminal onde era o '.'
00062         char out_path[TAM];
00063         sprintf(out_path, "%s%s%s",ZIPED_OUT_DIR, file_name, ".huff"); // Caminho completo
00064         FILE* compressed_file = fopen(out_path, "wb");
00065         file_error_reporter(compressed_file);
00066 
00067 
00068         FILE* fileIn = fopen(inPath, "rb");
00069         file_error_reporter(fileIn);
00070 
00071         write_header(compressed_file, tree_and_trash_size, Mystruct.head);
00072 
00073         setBytes(fileIn, compressed_file, table);
00074 
00075         puts("FIM");
00076 
00077     }
00078     else if(option == 2){
00079         char file_name[50];
00080         char file_format[10];
00081         char out_file_path[TAM];
00082 
00083         puts("Digite o nome do arquivo: ");
00084         scanf("%s", file_name);
00085 
00086         puts("Digite o formato do arquivo: ");
00087         scanf("%s", file_format);
00088         //strcat(out_file_name, file_format);
00089 
00090         char inPath[TAM];
00091         // Adciona o diretorio do nosso arquivo de entrada que esta no diretorio padrao de arquivos
    comprimidos
00092         sprintf(inPath, "%s%s", ZIPED_OUT_DIR, file_name);
00093         FILE* compressed_file = fopen(inPath, "rb");
00094         file_error_reporter(compressed_file);
00095 
00096         int trashSize, treeSize;
00097         get_tree_and_trash_size(compressed_file, &trashSize, &treeSize);
00098 
00099         printf("Tamanho da arvore: %d\n", treeSize);
00100         printf("Tamanho do lixo: %d\n", trashSize);
00101 
00102         unsigned long long int codedSize = find_file_size(compressed_file) - 2 - treeSize;
00103 
00104         NODE* treeRoot = getTree(compressed_file, &treeSize);
00105 
00106         char *remove = strrchr(file_name,'.');
00107         if(remove != NULL) *remove = '\0';
00108         sprintf(out_file_path, "%s%s%s", UNZIPED_OUT_DIR, file_name, file_format);
00109         FILE* fileOut = fopen(out_file_path, "wb");
00110         file_error_reporter(fileOut);
00111 
00112         unzip(compressed_file, fileOut, codedSize, treeRoot, trashSize);
00113 
00114         fclose(compressed_file);
00115         fclose(fileOut);
00116     }
00117     else {
00118         puts("OPCACO INVALIDA");
00119         puts("TERMINANDO PROGRAMA");
00120         return 0;
00121     }
00122 
00123     return 0;
00124 }
```

4.15 Referência do Arquivo src/tree.c

```
#include "std.h"
#include "list.h"
```

```
#include "tree.h"
#include "decode.h"
#include "encode.h"
```

Funções

- bool is_leaf (NODE *node)

  Verifica se o nó é uma folha.
- uint8_t get_data (NODE *node)

  Pega o byte do nó x.
- int get_frequency (NODE *node)

  Pega a frquência do nó x.
- void pre_order_trasversal (NODE *root)

  Exibe a árvore em pré-ordem.
- void huffmanTree (HEAD *myStruct, int *currentSize)

  Constrói a árvore de Huffman.

4.15.1 Funções

4.15.1.1 get_data()

```
uint8_t get_data (
            NODE * node )
```

Pega o byte do nó x.

Parâmetros

node	Nó da estrutura

Retorna

byte do nó

```
00011                               {
00012     if(node == NULL){
00013         puts("NO NULO");
00014     }
00015     uint8_t *data = (uint8_t*)node->data;
00016     return *data;
00017 }
```

4.15.1.2 get_frequency()

```
int get_frequency (
            NODE * node )
```

Pega a frquência do nó x.

Parâmetros

node	Nó lista/árvore

Retorna

A frequência do nó x

```
00018                                {
00019     if(node == NULL){
00020         puts("NO NULO");
00021         return -1;
00022     }
00023     int* frequency = (int*)node->frequency;
00024     return *frequency;
00025 }
```

4.15.1.3 huffmanTree()

```
void huffmanTree (
              HEAD * myStruct,
              int * currentSize )
```

Constrói a árvore de Huffman.

Constrói a árvore a partir da lista encadeada

Parâmetros

myStruct	Estrutura da lista/árvore
currentSize	Ponteiro para o tamanho da estrutura

```
00038                                                {
00039     while(*currentSize > 1){
00040         int frequency = get_frequency(myStruct->head) + get_frequency(myStruct->head->next);// soma
      das frequências dos primeiros nós
00041         NODE* newNode = create_node();
00042         *((int*)(newNode->frequency)) = frequency;
00043 
00044         newNode->left = myStruct->head;// esquerda do novo nó vai receber o primeiro elemento
00045         removeFirst(&myStruct->head,currentSize);
00046 
00047         newNode->right = (myStruct->head);//direita do novo nó vai receber o segundo elemento
00048         removeFirst(&myStruct->head,currentSize);
00049 
00050         insert_sorted(myStruct, newNode);// adiciona o novo nó de maneira ordenada
00051     }
00052     return;
00053 }
```

4.15.1.4 is_leaf()

```
bool is_leaf (
              NODE * tree_node )
```

Verifica se o nó é uma folha.

Parâmetros

tree_node	Nó da árvore de Huffman

Retorna

Se é ou não uma folha

```
00007                           {
00008     return node->left == NULL && node->right == NULL;
00009 }
```

4.15.1.5 pre_order_trasversal()

```
void pre_order_trasversal (
              NODE * root )
```

Exibe a árvore em pré-ordem.

```
00027                                       {
00028     if (root == NULL) {
00029         return;
00030     }
00031
00032     printf("%c", get_data(root));
00033     pre_order_trasversal(root->left);
00034     pre_order_trasversal(root->right);
00035
00036 }
```

Índice Remissivo

bitH
  BitHuff, 5
BitHuff, 5
  bitH, 5
  encode.h, 13
  size, 5
buildTable
  encode.c, 32
  encode.h, 13

count_frequency
  encode.c, 33
  encode.h, 14
create_node
  list.c, 38
  list.h, 21

data
  NODE, 7
decode.c
  find_file_size, 29
  get_tree_and_trash_size, 29
  getTree, 29
  is_seted, 30
  unzip, 31
decode.h
  find_file_size, 9
  get_tree_and_trash_size, 10
  getTree, 10
  is_seted, 11
  unzip, 11

encode.c
  buildTable, 32
  count_frequency, 33
  getTrashSize, 33
  getTreeSize, 33
  init_frequency_table, 34
  isSetedLong, 34
  set_bit, 35
  setBytes, 35
  setFirstTwoBytes, 36
  write_header, 36
  write_tree, 37
encode.h
  BitHuff, 13
  buildTable, 13
  count_frequency, 14
  getTrashSize, 14
  getTreeSize, 15
  init_frequency_table, 15
  isSetedLong, 15
  set_bit, 17
  setBytes, 17
  setFirstTwoBytes, 18
  write_header, 18
  write_tree, 19

file_error_reporter
  list.c, 38
  list.h, 21
find_file_size
  decode.c, 29
  decode.h, 9
frequency
  NODE, 7

get_data
  tree.c, 43
  tree.h, 26
get_frequency
  tree.c, 43
  tree.h, 26
get_tree_and_trash_size
  decode.c, 29
  decode.h, 10
getTrashSize
  encode.c, 33
  encode.h, 14
getTree
  decode.c, 29
  decode.h, 10
getTreeSize
  encode.c, 33
  encode.h, 15

HEAD, 6
  head, 6
  list.h, 21
  size, 6
head
  HEAD, 6
huffmanTree
  tree.c, 44
  tree.h, 27

inc/decode.h, 9, 12
inc/encode.h, 12, 19
inc/list.h, 20, 24
inc/std.h, 24, 25

inc/tree.h, 25, 28
init_frequency_table
encode.c, 34
encode.h, 15
init_struct
list.c, 39
list.h, 22
INPUT_DIR
std.h, 25
insert_in_linked_list
list.c, 39
list.h, 22
insert_sorted
list.c, 39
list.h, 22
is_leaf
tree.c, 44
tree.h, 27
is_seted
decode.c, 30
decode.h, 11
isSetedLong
encode.c, 34
encode.h, 15

left
NODE, 7
list.c
create_node, 38
file_error_reporter, 38
init_struct, 39
insert_in_linked_list, 39
insert_sorted, 39
malloc_error_reporter, 40
removeFirst, 40
list.h
create_node, 21
file_error_reporter, 21
HEAD, 21
init_struct, 22
insert_in_linked_list, 22
insert_sorted, 22
malloc_error_reporter, 23
NODE, 21
removeFirst, 23

main
main.c, 41
main.c
main, 41
malloc_error_reporter
list.c, 40
list.h, 23

next
NODE, 7
NODE, 6
data, 7
frequency, 7
left, 7
list.h, 21
next, 7
right, 7

pre_order_trasversal
tree.c, 45
tree.h, 27

removeFirst
list.c, 40
list.h, 23
right
NODE, 7

set_bit
encode.c, 35
encode.h, 17
setBytes
encode.c, 35
encode.h, 17
setFirstTwoBytes
encode.c, 36
encode.h, 18
size
BitHuff, 5
HEAD, 6
src/decode.c, 28
src/encode.c, 31
src/list.c, 37
src/main.c, 41
src/tree.c, 42
std.h
INPUT_DIR, 25
TAM, 25
UNZIPED_OUT_DIR, 25
ZIPED_OUT_DIR, 25

TAM
std.h, 25
tree.c
get_data, 43
get_frequency, 43
huffmanTree, 44
is_leaf, 44
pre_order_trasversal, 45
tree.h
get_data, 26
get_frequency, 26
huffmanTree, 27
is_leaf, 27
pre_order_trasversal, 27

unzip
decode.c, 31
decode.h, 11
UNZIPED_OUT_DIR
std.h, 25

write_header

encode.c, 36
encode.h, 18
write_tree
encode.c, 37
encode.h, 19

ZIPED_OUT_DIR
std.h, 25