UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

BRUNO SANTOS FERNANDES
JOÃO PAULO MOURA CLEVELARES

Compactador/Descompactador de Arquivos

Vitória
2022

BRUNO SANTOS FERNANDES
JOÃO PAULO MOURA CLEVELARES

Compactador/Descompactador de Arquivos

Relatório sobre o segundo trabalho do dia 11 de julho de 2022 apresentado à disciplina de Estruturas de dados como requisito parcial para obtenção de nota. Professora Patrícia Dockhorn Costa. Matrículas: Bruno 2021100784 e João 2021100149.

Vitória
2022

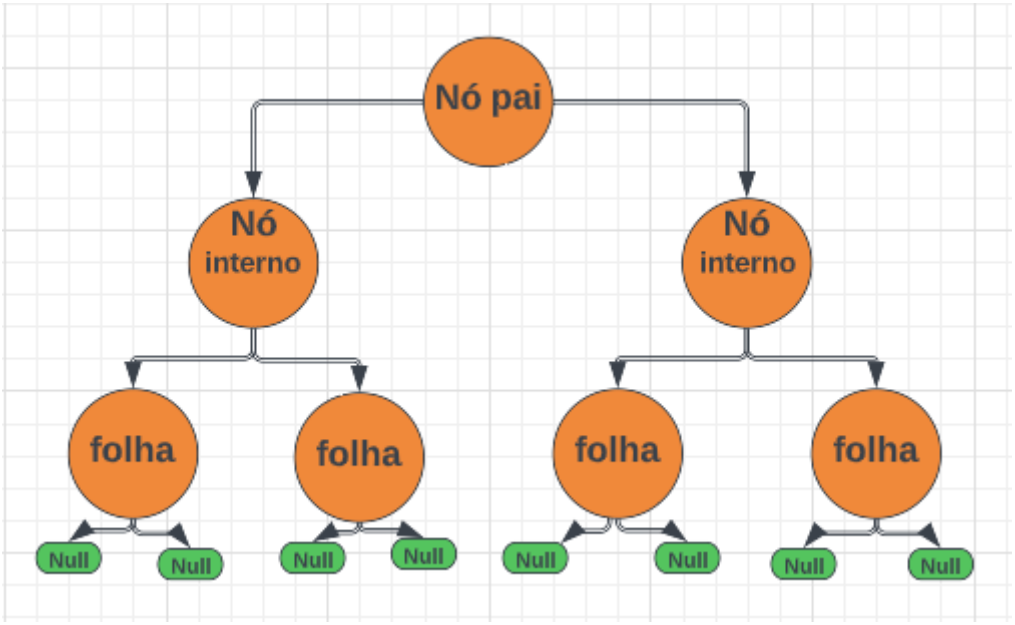
Introdução

O trabalho teve como objetivo a implementação de um codificador e um decodificador usando a linguagem C, aplicando os métodos de codificação de huffman. A codificação de Huffman é um método para a compactação de dados em um arquivo. Usando o número de ocorrências dos símbolos nos dados do arquivo, gera-se uma Árvore Binária em que cada folha armazena um elemento do arquivo, atribuindo 0 para ramos e folhas a esquerda e 1 para ramos e folhas a direita, assim, percorrendo o caminho na árvore até uma folha é gerado a codificação para o elemento armazenado na folha, sendo essa estrutura chamada de Árvore de Huffman. O programa consiste em dois executáveis diferentes que usam os mesmos *TADs* (Tipo Abstrato de Dados) batizados de zip.c (compactador) e unzip.c (descompactador). O primeiro deve ler um arquivo binário de entrada e montar uma tabela, onde cada linha deve conter um código de tamanho variável para cada byte do arquivo. Posteriormente deve gerar um novo arquivo binário(codificado) utilizando os códigos da tabela. O segundo é um descompactador que deve ler o arquivo binário codificado e remontar o arquivo original.

Implementação

ÁRVORE

Primeiramente implementamos um TAD árvore binária *binary_tree.c.*



Árvore binária

Dentro do TAD *binary_tree.c* encontramos funções importantes, como a
`Tree *CreateLeafNode(Tree *left, Tree *right, int weight, unsigned char character)`
responsável por criar um nó folha de uma árvore.

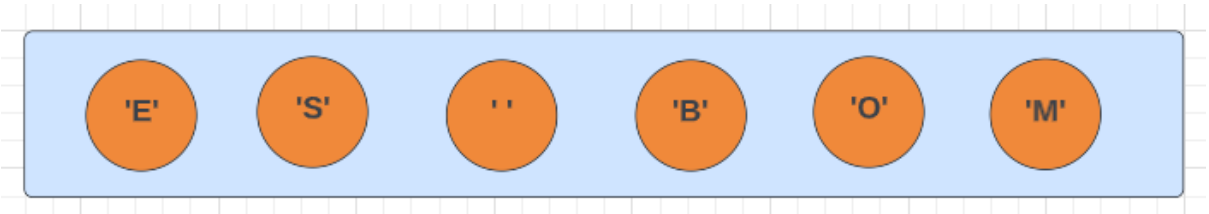
Outra função importante é a de criar um nó interno de uma árvore.
`Tree *CreateInternalNode(Tree *left, Tree *right, int weight)`

*A única coisa que difere as funções acima é o carácter a ser inserido no nó. Apesar de ser algo simples de se resolver com apenas uma função, optamos por criar duas funções para melhor interpretação e legibilidade do código.

Além disso, implementamos algumas funções auxiliares como:
`int TreeHeight(Tree *tree)` — Retorna a altura da árvore.
`int TreeWeight(Tree *tree)` — Retorna o peso de um nó.
`void DestructTree(Tree * tree)` — Libera a memória alocada para a árvore.

LISTA DE ÁRVORE

Após o TAD árvore binária, implementamos um TAD lista de árvores: *ordered_list.c*, onde cada célula da lista é um elemento do tipo *Tree*(árvore). Através dessa lista aplicamos o algoritmo de huffman.



Lista de árvore

Essa lista é feita através do vetor de frequência de caracteres criado pela função:
`CreateFrequencyTable(argv[1]);`, que adiciona o caractere no índice do vetor equivalente ao valor decimal da tabela ASCII de cada caractere e incrementa +1 cada vez que esse caractere for encontrado.

Abaixo temos a função que cria a lista de árvores a partir do vetor de frequência.

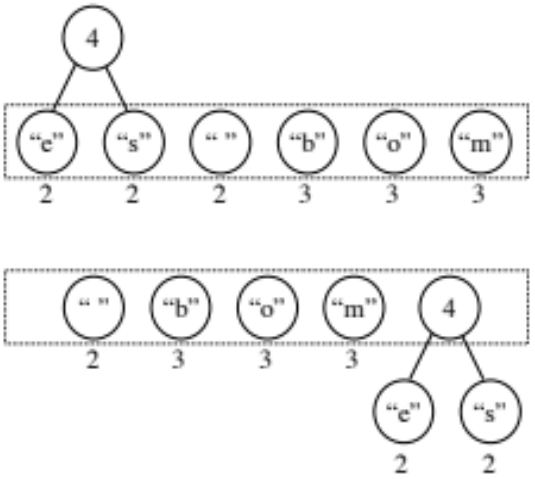
```
*CreateHuffmanList(int *frequencyTable)
```

A função `List *ListPush(List *list, Tree *tree)` é utilizada dentro da função citada acima, visando adicionar as árvores à lista de forma ordenada.

Ainda dentro do TAD *ordered_list.c* se encontra a função:

```
List *Huffman_Execute(List *list)
```

Esta é responsável por executar o algoritmo de Huffman.



Algoritmo de Huffman

TABELA DE FUNÇÕES

No TAD *table.c* temos as principais funções usadas na parte de compactação(zip.c) e na parte de descompactação (unzip.c).

```
int *CreateFrequencyTable(unsigned char *fileWay)
```

-Função responsável pela criação do vetor de frequência.

```
void FillEncodeTable(unsigned char **table, Tree *tree, unsigned char *code, int treeHeight)
```

-Função responsável pelo preenchimento da tabela de codificação.

```
unsigned char **CreateEncodeTable(List *list)
```

-Função responsável pela criação da tabela de codificação.

```
long int GetFileSize(char * fileName)
```

-Função responsável por calcular o tamanho do arquivo em bytes.

```
unsigned char *EncodeText(unsigned char **encodeTable, unsigned char *text,
                          long int fileSize, long int *encodedTextSize) {
```

-Função que codifica uma sequência de bytes com base em uma tabela de codificação.

```
void CreateCompressedFile(unsigned char *text, unsigned char *name, int *frequencyTable, long int encodedTextSize)
```

-Função responsável pela compactação do arquivo.

```
void UnzipFile(FILE *compressedFile, List *list, unsigned char *fileName)
```

-Função responsável pela descompactação do arquivo.

COMPACTAÇÃO

O arquivo zip.c é responsável pela leitura do arquivo de entrada, criação da árvore de Huffman, execução do algoritmo, codificação dos bytes, criação do arquivo de saída *.comp* (com a estrutura da árvore e dados codificados) e liberação da árvore.

ZIP.C

- Primeiramente lemos o tamanho do arquivo em bytes.

```
long int fileSize = GetFileSize(argv[1]);
```

- Criamos o vetor de frequência.

```
int *frequencyTable = CreateFrequencyTable(argv[1]);
```

- Em seguida, criamos a lista de árvores com os bytes e suas frequências, usando a função:

```
List *list = CreateHuffmanList(frequencyTable);
```

- Aplicamos ○ algoritmo de Huffman para obter uma árvore ótima utilizando a função abaixo.

```
Huffman Execute(list);
```

- A partir da árvore ótima criamos uma tabela(dicionário), onde é armazenado (ex: 001100) o caminho da árvore até achar o byte desejado.

```
unsigned char **table = CreateEncodeTable(list);
```

A função acima é responsável por alocar memória dinamicamente para nossa tabela de codificação. Dentro dela, há a função abaixo, responsável por percorrer a árvore e retornar o código de cada byte(que será inserido na tabela)

```
FillEncodeTable(table, tree, "", TreeHeight(tree));
```

-Também temos uma função que salva cada byte do arquivo em um vetor:

```
unsigned char *text = ReadFile(argv[1],fileSize);
```

```
vetor[0] - b
vetor[1] - o
vetor[2] - m
vetor[3] -
vetor[4] - e
vetor[5] - s
vetor[6] - s
vetor[7] - e
vetor[8] -
vetor[9] - b
vetor[10] - o
vetor[11] - m
vetor[12] - b
vetor[13] - o
vetor[14] - m
```

o vetor * text contém cada caractere do arquivo

Após ter a tabela de codificação pronta e a sequência original de bytes do arquivo, é hora de codificar essa sequência de bytes com os códigos da tabela.

```
unsigned char *encodedText = EncodeText(table, text, fileSize, &encodedTextSize);
```

```
encodedText[0] -> 1
encodedText[1] -> 1
encodedText[2] -> 1
encodedText[3] -> 0
encodedText[4] -> 1
encodedText[5] -> 0
encodedText[6] -> 0
encodedText[7] -> 1
encodedText[8] -> 0
encodedText[9] -> 0
```

vetor com a sequência de bytes já codificados

- Com a sequência de bits codificada pronta, podemos criar nosso arquivo compactado. Nele, além de escrevermos nossa sequência de bytes codificada, também adicionaremos um cabeçalho com algumas informações, como o vetor de frequência (a fim de recriar a árvore de Huffman na hora da descompactação), o tamanho da stream codificada e a extensão do arquivo original.

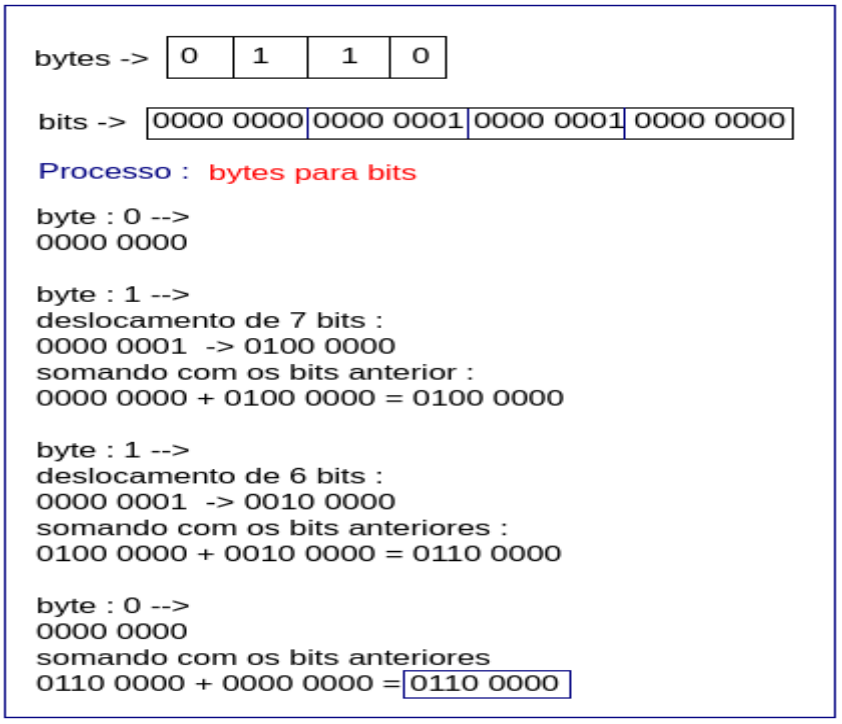
```
CreateCompressedFile(encodedText, argv[1], frequencyTable, encodedTextSize);
```

Para criar nosso arquivo compactado, transformamos 8 bytes em 8 bits, ou seja, apenas 1 byte. O método utilizado foi operação bit a bit. Sendo assim, não foi utilizado o TAD bitmap fornecido pela Professora Patrícia.

Deslocamento de bit

```
if (text[x] == '1') {
  aux = aux << y;
  binary = binary | aux;
}
```

Essa é uma ideia geral de como foi implementada a transformação de 8 bytes para 8 bits



Em seguida liberamos toda a memória alocada com as seguintes funções

```
DestructList(list);
DestructEncodeTable(table);
free(text);
free(encodedText);
```

DESCOMPACTAÇÃO

Agora falaremos um pouco sobre a parte do programa de decodificação que se encontra no arquivo *unzip.c* .

UNZIP.C

Primeiramente recuperamos o vetor de frequências que foi passado para o arquivo compactado com o comando : `fread(frequencyTable,sizeof(int),256,file);`

Recriamos a árvore com as mesmas funções que usamos no arquivo *zip.c*

```
List * list = CreateHuffmanList(frequencyTable);
Huffman_Execute(list);
```

.

A principal função desse arquivo é a `UnzipFile(file, list, argv[1]);` ,responsável por decodificar o arquivo binário (.comp). A decodificação é feita percorrendo a árvore de Huffman com os bits lidos da stream codificada.

DIFICULDADES

A principal dificuldade do trabalho foi otimizar o tempo de compactação para arquivos maiores que 1MB. Na construção da stream codificada foi usada a função ***strcat*** dentro de um loop. Quanto mais iterações, mais lento ficava o algoritmo, pois a função percorria uma string cada vez maior para achar seu final e concatenar a nova string. Para resolver esse problema, optamos por inserir cada byte com acesso direto ao seu respectivo índice no vetor.

```
for (int x = 0; x < fileSize; x++) {
  strcat(code, encodeTable[text[x]]);
}
```

Este trecho foi trocado pelo trecho abaixo

```
long int encodedTextIndex = 0, codeIndex=0;
for(int x=0; x<fileSize;x++){
  while(encodeTable[text[x]][codeIndex] != '\0'){
    code[encodedTextIndex] = encodeTable[text[x]][codeIndex];
    encodedTextIndex++;
    codeIndex++;
  }
  codeIndex=0;
}
```

CONCLUSÃO

O trabalho foi desafiador, pois envolveu manipulação de bits, o que requer um nível de entendimento detalhado sobre leitura e escrita de arquivos binários, além de operações lógicas bit a bit para acessar bits específicos de um bloco de memória. Além disso, foi preciso ter um entendimento sólido de árvores binárias e foi necessário usar a criatividade para resolver os problemas propostos usando as estruturas de dados propostas. Por fim, observou-se que o programa gera com êxito os arquivos compactados, e também é capaz de descompactá-los, gerando um arquivo idêntico ao original. No entanto, como é escrito o vetor de frequência no arquivo compactado, caso o arquivo original seja muito pequeno, o arquivo compactado gerado acaba sendo maior que o arquivo original. Portanto, para o uso do programa, recomenda-se arquivos binários com tamanho acima de 1 kB.

BIBLIOGRAFIA

Playlist no Youtube “Algoritmo de Huffman em C” do canal “Programe seu futuro” <https://www.youtube.com/watch?v=PGIl1gTSPns&list=PLqJK4Oyr5WShtxF1Ch3Vq4b1Dzzb-WxbP&index=2>

Vídeos-aula da professora Patrícia Dockhorn no classroom das aulas Recursão e Árvores.