UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CURSO CIÊNCIA DA COMPUTAÇÃO

ELAINE DIAS PIRES
FILIPE GOMES ARANTE DE SOUZA

RELATÓRIO 2° TRABALHO PRÁTICO DE ESTRUTURAS DE DADOS: COMPACTADOR DE HUFFMAN

Vitória
2021

ELAINE DIAS PIRES
FILIPE GOMES ARANTE DE SOUZA

RELATÓRIO 2° TRABALHO PRÁTICO DE ESTRUTURAS DE DADOS: COMPACTADOR DE HUFFMAN

Trabalho apresentado à disciplina de Estruturas de Dados da Universidade Federal do Espírito Santo - UFES, como requisito parcial para avaliação semestral.

Professor: Patrícia Dockhorn Costa
Turma: 2020/1

VITÓRIA
2021

SUMÁRIO

1 INTRODUÇÃO

O objetivo deste trabalho é implementar uma compactação / descompactação de arquivos utilizando o algoritmo de huffman na codificação.

Este algoritmo funciona da seguinte forma: A partir de uma lista de árvores, o algoritmo captura as duas primeiras árvores da lista e cria uma nova árvore inserindo-a novamente na lista de forma ordenada por ordem crescente de acordo com o peso do nó da árvore. Esse passo a passo é repetido até que se tenha apenas uma árvore na lista, sendo esta a árvore de Huffman.

Essa forma de montar a árvore faz com que os caracteres de maior peso tenham um caminho menor (logo, menos bits para codificá-los) em relação à raiz da árvore, enquanto os caracteres de menor peso tenham um caminho maior (portanto, mais bits para codificá-los). Por isso, trata-se de um algoritmo baseado na probabilidade de ocorrência dos caracteres no arquivo que será compactado.

Neste relatório explicaremos a lógica utilizada, bem como as TADs e principais funções criadas.

2 IMPLEMENTAÇÃO

2.1 TADS

Para modularizar o trabalho, criamos as seguintes TADs:

- TAD tree.c
- TAD list.c

No TAD **tree.c** foram implementadas todas as operações que podem ser feitas sobre uma árvore, e sua função é dar suporte para a criação da árvore de codificação de Huffman.

No TAD **list.c** implementamos uma lista de árvores, cujo comportamento é de uma fila com prioridade, ou seja, a inserção de elementos é ordenada de acordo com o peso do nó da árvore e a retirada é do primeiro elemento da lista. Sua função também é dar suporte na criação da árvore de codificação de Huffman.

Além destes dois TADS criados, utilizamos o TAD **bitmap.c** cedido pelo professor João Paulo Andrade Almeida para auxiliar na tarefa de codificar e decodificar em bits e permitir o intermédio entre o arquivo compactado e os arquivos de saída gerados.

Segue abaixo definição das structs para entendimento da organização dos TADS:

```
struct tree{
    unsigned char elem;
    int peso;
    Tree* left;
    Tree* right;
};
```

```
struct list{
    Celula* first;
    Celula* last;
    int tam;
};
```

```
struct map {
    unsigned int max_size;
    unsigned int length;
    unsigned char* contents;
};
```

Figura 1: Definição das structs.

2.2 MÓDULOS

Tendo como base os TADS mencionados acima, criamos as seguintes bibliotecas para criação do compactador e descompactador de Huffman:

- huffman.c
- compactador.c
- descompactador.c

No módulo **huffman.c** foram implementadas todas as funções para criação da árvore de huffman e também para a tabela de codificação, tendo como pilares os TADS tree.c e list.c.

Na biblioteca **compactador.c** , foram desenvolvidas todas as funções para criar um bitmap com o cabeçalho e o texto/imagem/áudio codificados e gerar o arquivo compactado .comp.

Por fim, no módulo **descompactador.c**, foram desenvolvidas todas as rotinas para ler o arquivo compactado .comp, inserir seu conteúdo em um bitmap, para enfim percorrê-lo bit a bit decodificando o arquivo, gerando o arquivo original.

2.3 FLUXOGRAMAS

Segue abaixo fluxogramas em alto nível de cada etapa do trabalho:

2.3.1 FLUXOGRAMA DE COMPACTAÇÃO

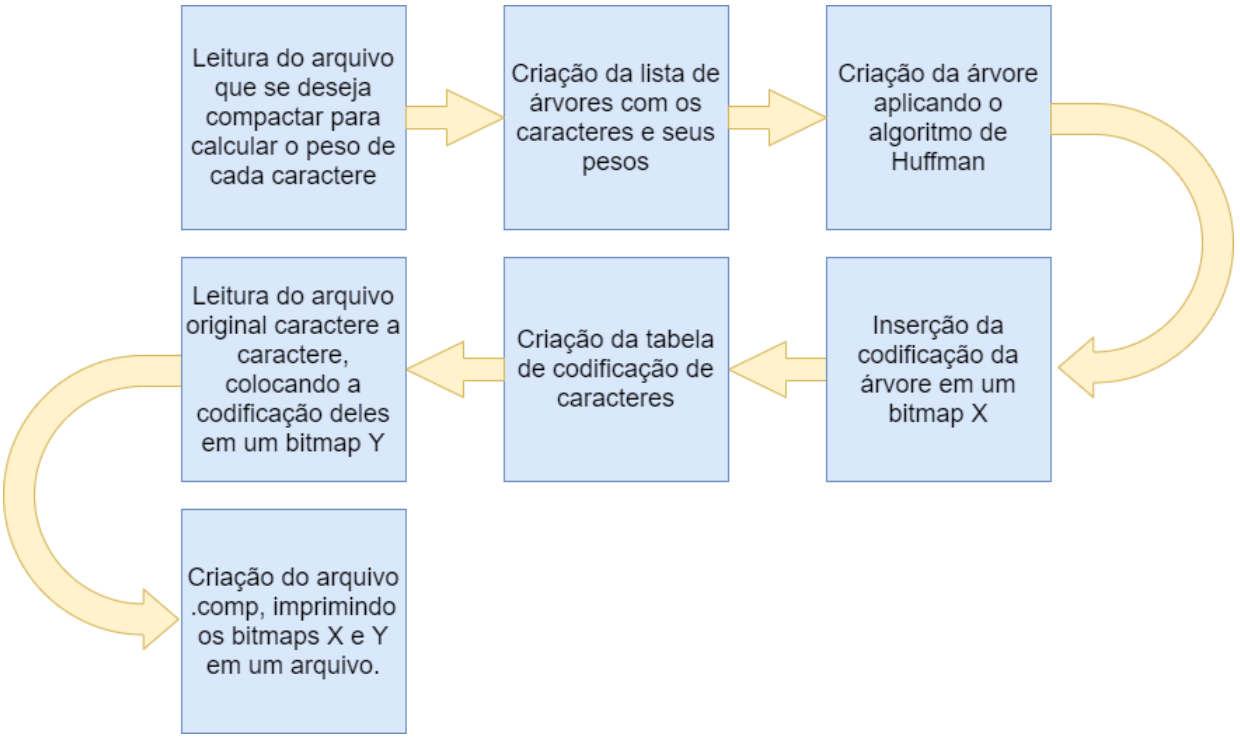

Figura 2: Fluxograma de compactação do arquivo original.

2.3.2 FLUXOGRAMA DE DESCOMPACTAÇÃO DO ARQUIVO

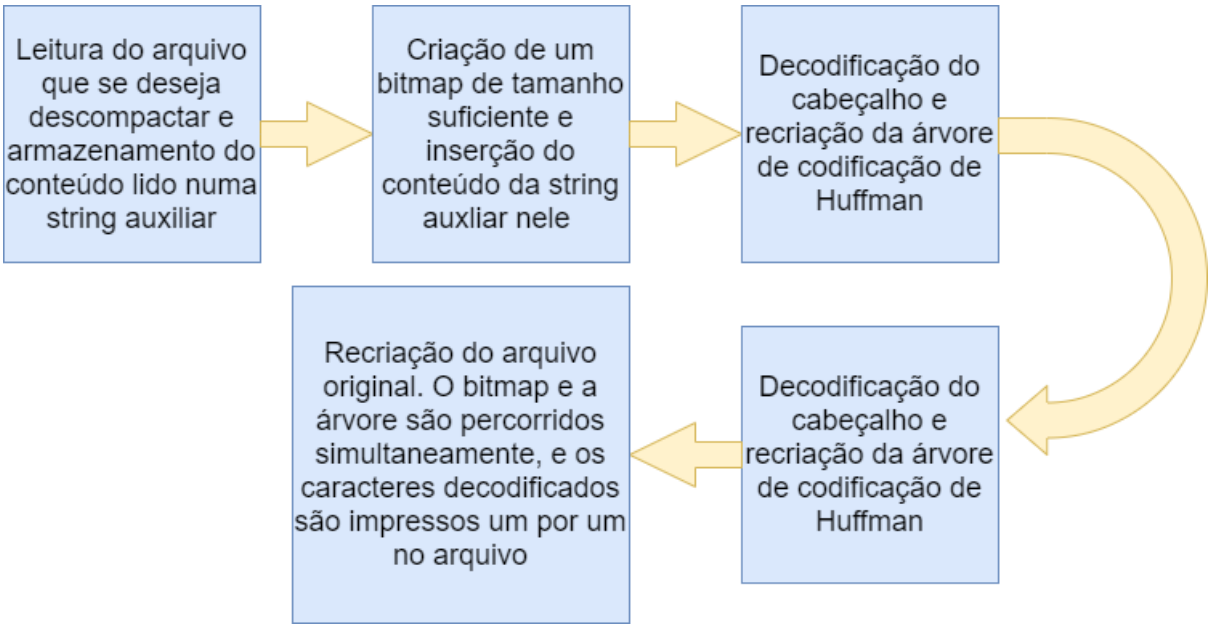

Figura 3: Fluxograma de descompactação do arquivo compactado.

2.4 CABEÇALHO

O cabeçalho do arquivo **.comp** foi feito da seguinte maneira:

INFORMAÇÃO	QUANTIDADE DE BITS
Qrd. de folhas da árvore de Huffman	8
Lixo em bits do arquivo codificado	3
Nó folha	1
Nó não-folha	1
Caractere de determinado nó folha	8

O cabeçalho ficou organizado na seguinte ordem:

[Qtd. folhas] + [Tamanho do lixo do arquivo] + [Caminhamento na árvore em travessia pré-ordem]

No momento de recriar a árvore de Huffman a leitura do cabeçalho era a seguinte:

- **Leitura dos 8 primeiros bits:**
  Quantidade de folhas, condição de parada da leitura do cabeçalho. São 8 bits, pois existem somente 256 caracteres na tabela ascii, assim uma árvore de Huffman possui no máximo essa quantidade de folhas.

- **Leitura dos próximos 3 bits:**
  Tamanho do lixo do arquivo compactado, condição de parada do caminhamento do bitmap na descompactação. São 3 bits, pois este tamanho é sempre um número entre 0 e 7.

- **Leitura do próximo bit:**
  Se o bit for 0, não é folha. Logo, ocorre a leitura do próximo bit.
  Se o bit for 1, é folha. Então são lidos os próximos 8 bits, que representam o caractere ascii dessa mesma folha. Esse processo de ler folhas e não folhas se repete até todos os bits 1 de folhas da árvore serem lidos.

- **Ignorar bits de lixo no cabeçalho:**
  À medida que o cabeçalho é percorrido no bitmap, inserimos um contador que verifica quantos bits já foram lidos. Quando a última folha e o último caractere da árvore é lido, calculamos o quanto falta para esse contador ser um múltiplo de 8. Assim, encontramos a quantidade de bits que devem ser ignoradas no bitmap para poder começar a decodificar os caracteres no lugar correto.

2.5 TEXTO

O “texto” no arquivo .comp estava da seguinte forma:

[codHuff_caractere_0] + … + [codHuff_caractere_n] + [lixoTexto]

Para ignorar o lixo do texto fizemos da seguinte forma:
A priori já sabíamos qual era a quantidade de lixo, pois a colocamos no cabeçalho. Dessa forma, na decodificação do texto, utilizamos a função **bitmapGetLength** e subtraímos o valor do lixo do texto, assim tínhamos a quantidade exata dos bits que continham a informação codificada no bitmap.

2.6 PRINCIPAIS FUNÇÕES

```
void compacta(unsigned char* nomeArquivo)
```

Função principal da compactação. Abrimos o arquivo a ser compactado, chamamos **calculaPesos** que cria o vetor de pesos dos caracteres. Chamamos outras funções como a **geraArvoreCodificacao** e a **imprimeBitmapNoArquivo .** A função de imprimir bitmap no arquivo foi utilizada na função compacta tanto para imprimir o cabeçalho quanto o texto codificado.

```
Tree* huffman(List* list)
```

Nessa função, implementamos o algoritmo de huffman a partir de uma lista de árvores. Cada elemento da lista é uma árvore que contém um caractere do arquivo com seu respectivo peso. O peso é a quantidade de vezes que o caractere aparece no arquivo.

```
unsigned char** inicializaTabelaCodificacao(Tree* tree, unsigned char** tabela, unsigned char* caminho)
```

Nessa função, inserimos na tabela a codificação de cada caractere do arquivo. O caminho percorrido na árvore de huffman até chegar ao caractere é a codificação do mesmo. Por exemplo, seja o caractere “e”. Se na árvore de huffman percorremos esquerda, direita, direita até chegar ao nó folha que contém o caractere “e”, a codificação do “e” será 011. Dessa forma, inserimos na tabela tal codificação para o caractere “e”. A recursão foi bastante importante na implementação dessa função.

```
void decodifica(bitmap* bm, unsigned char* nomeArquivoCompactado){
```

Essa função é extremamente importante, pois é ela que faz a decodificação, chamando a função **recriaTree** , a qual recria a árvore de huffman e chama a função **decodificaTexto** . A função que decodifica o texto faz a decodificação a partir do bitmap gerado pela função **leArquivoCompactado,** a qual insere toda a informação do arquivo .comp em um bitmap. É também na função **decodificaTexto** que nós geramos o arquivo descompactado.

```
void descompacta(unsigned char* nomeArquivoCompactado){
```

Função principal na descompactação, chama outras importantes funções como a **leArquivoCompactado** e a **decodifica**

3 DIRETÓRIOS

Dividimos o trabalho nas seguintes pastas:

- **src:** pasta que contém os arquivos .c do trabalho.
- **include:** pasta que contém os arquivos .h do trabalho.
- **client:** pasta que contém os arquivos clientes do trabalho (programas Compacta.c e Descompacta.c).

- **data:** pasta que deve conter o arquivo de entrada do trabalho.
  Para executar o programa compactador, é necessário que o arquivo que se deseja compactar esteja na pasta “data”. Após executar o “Compacta”, um arquivo **.comp** é gerado nessa pasta e o arquivo original é apagado.
  Já ao executar o programa descompactador, a ideia é a mesma. O arquivo **.comp** deve estar na pasta “data” e, após executar, o **.comp** é deletado e o arquivo descompactado é gerado na pasta “data”.

4 MAKEFILE

Criamos um makefile para facilitar a compilação do trabalho.

- Para compilar o programa basta utilizar o comando: **`$ make`**

  Serão criados dois arquivos executáveis na pasta raiz, intitulados como “Compacta” e "Descompacta".

- Para executar o programa “Compacta”, digite: **`$ make runComp`**
- Para executar o programa “Descompacta”, digite: **`$ make runDescomp`**
- Para apagar os arquivos objeto e os executáveis, digite: **`$ make clean`**

É importante salientar que o trabalho está configurado para uma entrada “string.txt”. Dessa forma, caso o arquivo a ser compactado seja outro, é necessário uma pequena alteração no makefile, na seguinte parte:

```
4    NOMECOMPACTA := string.txt
5    NOMEDESCOMPACTA := $(NOMECOMPACTA).comp
```

Assim, caso o arquivo a ser compactado seja, por exemplo,“redacao.txt”, é necessário a substituição de **string.txt** por **redacao.txt**.
Porém, se a intenção for executar algum dos programas sem o Makefile, pode-se utilizar os seguintes comandos:

Para compactar um arquivo:

`$ valgrind ./Compacta nome_arquivo.(formato)`

Para descompactar um arquivo:

`$ valgrind ./Descompacta nome_arquivo.(formato).comp`

5 CONCLUSÃO

Conseguimos realizar a compactação e descompactação.Nossa maior dificuldade foi a manipulação de bits, a conversão para bytes e ter que pensar de uma forma mais baixo nível. Tivemos dificuldade em determinar uma condição de parada para o cabeçalho e para o texto na codificação, mas conseguimos ultrapassá-las colocando a quantidade folhas e o lixo do texto no cabeçalho.

Aferir se a codificação estava correta também foi um processo complicado, conseguimos aferir com o auxílio do comando do terminal WSL

`$ xxd -b nome_do_arquivo` .

Além disso, de início, não sabíamos que poderiam ter diferentes tipos de formato de entrada no trabalho, então ficamos preocupados com isso, mas no final conseguimos codificar e decodificar os arquivos binários assim mesmo.

Achamos o trabalho bem interessante em termos de aprendizado e também, bastante desafiador.

Em relação à taxa de compactação dos arquivos, chegamos ao seguinte resultado:

Para arquivos pequenos, a compactação acabava expandindo o arquivo, pois o cabeçalho ocupava bastante espaço, de modo que ele, aliado ao texto codificado, ficava maior que o arquivo original. Em arquivos grandes de texto, a taxa de compactação foi próxima de 50%, enquanto arquivos de imagem/música tiveram taxas de compactação menores que 10%. Essa diferença ocorre pois em arquivos de imagem/música, a árvore de huffman tende a ser mais balanceada devido aos caracteres terem frequência muito próximas uns dos outros, enquanto em arquivos de texto, a árvore de codificação tende a ser desbalanceada. Esse desbalanceamento ocorre pois é comum em texto, alguns caracteres, como as vogais, aparecem mais que outros, como algumas consoantes , por exemplo “z”,”y” e “w”.

6 BIBLIOGRAFIA

Slides e vídeo aulas disponibilizados no classroom da turma.