UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

MATEUS TICIANELI SARTORIO
YURI AIKAU DE CASTRO REIS SANCHEZ

RELATÓRIO DO TRABALHO II DE ESTRUTURAS DE DADOS I

Vitória

2021

MATEUS TICIANELI SARTORIO
YURI AIKAU DE CASTRO REIS SANCHEZ

RELATÓRIO DO TRABALHO II DE ESTRUTURAS DE DADOS I

Relatório de Trabalho apresentado à disciplina de Estrutura de Dados I do curso de graduação em Engenharia de Computação da Universidade Federal do Espírito Santo, como requisito parcial para avaliação. Profª. Patrícia Dockhorn Costa.

Vitória
2021

SUMÁRIO

1. INTRODUÇÃO **4**

2. IMPLEMENTAÇÃO **4**

arvore.h 5

bitmap.h 5

Compacta.h 6

Cmain.c 6

Descompacta.h 7

Dmain.c 7

3. CONCLUSÃO **7**

4. BIBLIOGRAFIA **9**

1. INTRODUÇÃO

O problema proposto consiste basicamente em ler arquivos de texto contendo ASCII simples e os compactando para um arquivo de menor tamanho, além disso realizar a operação inversa, ou seja, descompactar esse arquivo e reescrever o arquivo original.

Para a realização de tal tarefa, utilizou-se a codificação de Huffman, cuja ideia é criar uma árvore binária, onde os caracteres que aparecem com maior frequência no arquivo de entrada possuem menores códigos binários e caracteres que aparecem menos vezes possuem um maior código binário. A figura 1 ilustra uma árvore de compactação de Huffman, onde o caminho até cada caractere é seu código binário.

Os TAD's criados consistem basicamente de dois TAD's principais, que contém as funções de compactação e descompactação.

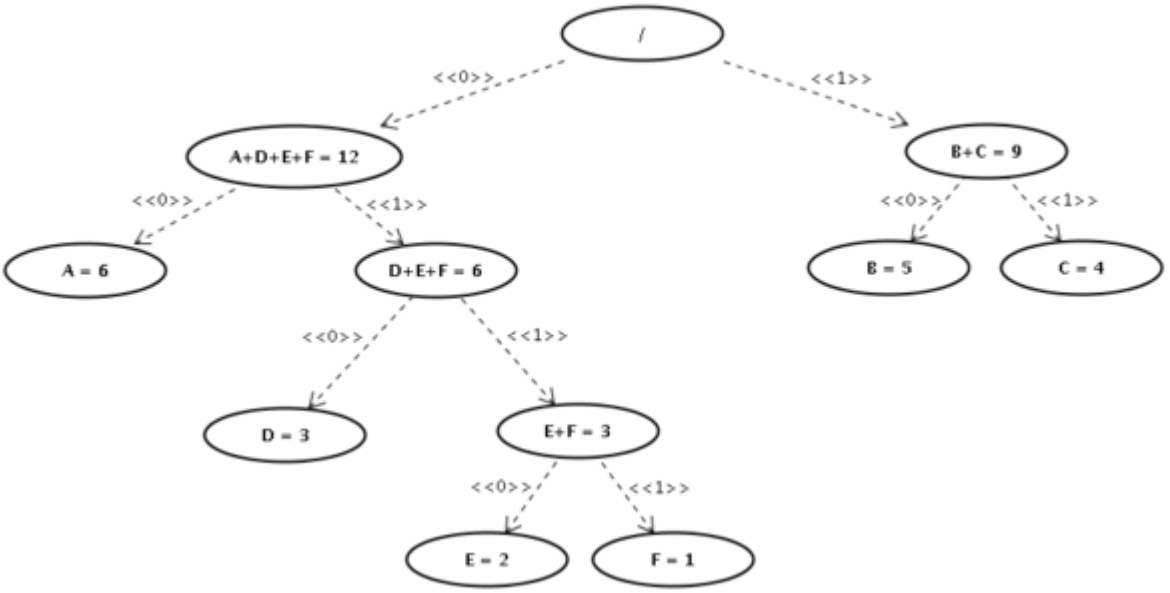

Figura 1 - Exemplo de uma árvore de compactação de Huffman
Fonte: https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman

2. IMPLEMENTAÇÃO

O programa consiste basicamente de TAD's de compactação e descompactação e 2 clientes (Cmain.c e Dmain.c), cada um com suas funções específicas e operações respectivas.

2.1. arvore.h

Primeiramente, dentro desse TAD temos duas estruturas que utilizamos para que possamos realizar as operações, sendo elas: *caractere_st* e *arvore_st*.

A partir disso temos duas funções de inicialização das estruturas caractere e árvore sendo que a última necessita de um *caractere_st* inicializado para que ele possa ser atribuído a *arvore_st* no campo da informação.

Também temos funções de retorno dos campos da estrutura para que outros arquivos também consigam acessá-los fora do arvore.c.

As principais funções que foram de muita importância para a realização do programa foram: caminho(), onde essa estabelece o caminho (em bits) que leva para chegar a um determinado caractere na árvore de huffman, onde registra 0 se foi para a esquerda e 1 para a direita (tudo isso utilizando o TAD de bitmap); criaArvoreCompactada(), essa parte de um arquivo compactado previamente aberto e lê o cabeçalho dele (que anteriormente foi impresso pelo Compactador) e transforma em uma árvore binária idêntica a criada na função huffmann durante a compactação; imprimeArvoreArquivo(), imprime uma árvore passada para essa função no arquivo também passado para essa função, o padrão de impressão são dois caracteres juntos, o primeiro sendo um que indica se o nó representa uma folha (1) ou não folha (0) e o segundo sendo o caractere do nó folha, se for um nó não folha esse valor é de padrão "0".

E por fim, também existe uma função que libera uma árvore passada para ela, nesse caso ela libera todos os campos alocados na *arvore_st* e a árvore em si.

2.2. bitmap.h

Para a manipulação de bits inerente à codificação de Huffman, foi utilizado o TAD bitmap disponibilizado pela professora. Ele permite a manipulação bit a bit de posições de memória, função não suportada nativamente pela linguagem C.

Além disso, criou-se algumas funções a mais no TAD bitmap para auxílio nas funções principais de compactação e descompactação.

A primeira delas foi a bitmapCopia(), que recebe um ponteiro para uma estrutura bitmap, cria e retorna uma cópia dela. Além disso, foi implementada também a função bitmapLimpa(), que recebe um ponteiro para uma estrutura do tipo bitmap, e apaga todas as informações contidas nela, a deixando de forma idêntica a se tivesse acabado de ser inicializada.

Por fim, implementou-se a função bitmapSetByte(), que troca o primeiro byte do vetor de caracteres da estrutura alvo para o caractere passado como argumento.

2.3. Compacta.h

O TAD compacta contém as principais funções de compactação, e é responsável por ler o arquivo de entrada e gerar um arquivo compactado com extensão .comp. No arquivo de impressão, primeiro é impresso um cabeçalho contendo a árvore de compactação (para futura descompressão do arquivo).

O padrão de impressão da árvore consiste de dois bytes “00” para nós não folha e 1c para nós folha, onde c é o caractere armazenado naquele nó.

Um exemplo de cabeçalho é mostrado abaixo:

00000011 1o00001e1s001b1m

Por fim, são impressos os bytes correspondentes aos códigos compactados dos caracteres do arquivo original.

Dentre as principais funções do Compacta.h, está a criaVetorArvores(), que a partir do arquivo de entrada, gera todos os nós folhas da árvore de compactação final e os organiza em um vetor de árvores. Esse vetor é organizado em ordem crescente de acordo com o peso, que é definido como o número de vezes que o caractere aparece no arquivo original.

Outra importante função é a huffman(), que a partir do vetor de árvores unitárias gerada pela criaVetorArvores(), gera a árvore de compactação de Huffman.

Ademais, a função criaTabela() gera um vetor de bitmaps, onde cada posição do vetor corresponde ao código ASCII de um caractere, e então cada posição do vetor é preenchido com um bitmap que corresponde ao seu caminho na árvore de compactação.

Por fim, a função compacta() escreve o arquivo compactado .comp a partir da tabela de compactação. Seu funcionamento consiste basicamente em ler um caractere de entrada, ler seu código compactado e adicionar em um buffer. Quando o buffer atinge capacidade máxima (8 bits), essa sequência é impressa no arquivo compactado.

2.4. Cmain.c

O Cmain.c será o cliente que realizará as funções necessárias para que a compactação seja realizada da forma desejada.

Primeiramente iniciaremos um ponteiro para o arquivo a ser compactado, esse foi passado na linha de comando e para utilizarmos essa informação usaremos argv[1] e passaremos isso de argumento para a fopen() no modo “rb”.

Depois disso iremos criar um vetor de ocorrências de caracteres de um arquivo utilizando a função retornaOcorrencias().

Em seguida criaremos um vetor de Arv* utilizando a função criaVetorArvores() e passaremos o vetor criado anteriormente para essa função. Então utilizaremos esse vetor de Arv* para a função huffman() que realizará todas as operações necessárias e retornará novamente para essa mesma árvore, porém apenas iremos utilizar a primeira posição (que é a árvore de Huffman completa). Em seguida criaremos a nossa tabela de codificação, que será um vetor de bitmap* e armazenará o caminho percorrido na árvore que leva a cada caractere (ou seja o caminho que leva até o caractere 'c' estará armazenado na tabela[c]), isso tudo utilizando a função criatabela().

Com todos os dados acima, finalmente iremos criar o arquivo compactado usando a função compacta().

Por fim iremos liberar todas as variáveis que foram alocadas dinamicamente e fecharemos o ponteiro para o arquivo de entrada.

2.5. Descompacta.h

O TAD Descompacta possui apenas a função de descompactação, chamada descompacta(), que é responsável pela descompactação do arquivo do tipo .comp gerado pelo programa de compactação.

Inicialmente, a função lê o cabeçalho da função e gera a árvore de descompactação, idêntica à gerada na compactação. Em seguida, a função lê um byte do arquivo de entrada por vez, e então procura na árvore qual caractere do arquivo original (o arquivo que deu origem ao arquivo compactado) corresponde à sequência de bits lida, e então imprime o bit lido no arquivo descompactado.

2.6. Dmain.c

O Dmain.c será o cliente do nosso descompactador, onde ele irá chamar todas as funções necessárias e realizar todas as operações para as funções previamente.

Primeiro iremos inicializar um ponteiro para o arquivo compactado passado na linha de comando como argv[1] no modo "rb".

Então iremos chamar a função criaArvoreCompactada() para que ela crie a árvore de compactação de acordo com o cabeçalho do arquivo compactado (o mesmo arquivo que acabamos de inicializar o ponteiro).

Em seguida iremos chamar a função descompacta() que fará as operações para reescrever o arquivo original através do compactado.

E por fim iremos liberar a árvore criada anteriormente e iremos fechar o ponteiro do arquivo compactado que acabamos de ler.

3. CONCLUSÃO

O trabalho foi desafiador, pois envolveu manipulação de bits em C, o que requer um nível de entendimento detalhado do armazenamento de variáveis na heap, e como podem ser feitas operações lógicas bit a bit para acessar bits específicos de blocos de memória. Além disso, foi preciso ter um entendimento sólido de árvores binárias. Ademais, foi necessário usar a criatividade para resolver os problemas propostos usando as estruturas de dados propostas.

A principal dificuldade encontrada no trabalho foi na construção da tabela de compactação a partir da árvore de Huffman. Por fim, a solução foi encontrada ao se utilizar uma função recursiva (caminho()) que percorria cada nó da árvore, passando para cada iteração seguinte o caminho já percorrido, e adicionando um bitmap à tabela de compactação caso um caractere fosse encontrado na árvore.

Outro problema encontrado foi ler o arquivo compactado e reconstruir a árvore de compactação. Ao fim, uma função recursiva (criaTabelaCompactada()) produziu uma solução elegante e simples.

Por fim, observou-se que o programa gera com sucesso arquivos compactados, e em seguida é capaz de descompactá-lo, gerando um arquivo idêntico ao original. No entanto, como é impressa uma árvore binária no início do arquivo compactado, caso o arquivo original seja muito pequeno, a própria impressão da árvore é maior que ele, e portanto o arquivo compactado gerado acaba sendo maior que o arquivo original. Portanto, para o uso do programa, recomenda-se arquivos binários com tamanho de pelo menos 1 kB. No entanto, o arredondamento foi feito grosseiramente para cima, e arquivos muito menores do que isso podem ser compactados com sucesso pelo programa, dependendo muito da quantidade de caracteres diferentes que o arquivo possui.

4. BIBLIOGRAFIA

[1] CELES, W; CERQUEIRA, R; RANGEL NETTO, JM. Introdução a estruturas de dados: com técnicas de programação em C.

[2] Rio de Janeiro: Campus, 2004., 2004. (Série Editora Campus/SBC). ZIVIANI, N. Projeto de algoritmos: com implementações em PASCAL e C. São Paulo, SP: Cengage Learning, 2011., 2011.

[3] SZWARCFITER, JL; MARKENZON, L. Estruturas de dados e seus algoritmos. Rio de Janeiro: Livros Técnicos e Científicos, c1994., 1994.