UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MIKAELLA FERREIRA DA SILVA
LEONARDO DEORCE LIMA DE OLIVEIRA

2º TRABALHO PRÁTICO:

COMPACTADOR/DESCOMPACTADOR DE ARQUIVOS

VITÓRIA
2018

MIKAELLA FERREIRA DA SILVA
LEONARDO DEORCE LIMA DE OLIVEIRA

2º TRABALHO PRÁTICO:

COMPACTADOR/DESCOMPACTADOR DE ARQUIVOS

Trabalho apresentado à disciplina Estrutura de Dados I (INF09292) do Curso de Ciência da Computação da Universidade Federal do Espírito Santo no 2º semestre do ano 2018, como requisito para avaliação.
Orientador: Prof.ª Patrícia Dockhorn Costa

VITÓRIA
2018

RESUMO

Este relatório descorre sobre os programas desenvolvidos de compactação e descompactação de arquivos binários baseados no algoritmo de Huffman. A implementação do algoritmo é feita com o uso de TADs árvore binária e lista encadeada, bem como operadores bitwise em C.

Palavras-chave**:** programas, compactação, descompactação, Huffman, bytes, bitwise.

SUMÁRIO

1 INTRODUÇÃO

Este relatório descreve os processos desenvolvidos de compactação e descompactação de um arquivo binário com a implementação do algoritmo de Huffman.

O algoritmo busca diminuir o tamanho de um arquivo modificando a forma como um símbolo (ou caractere) é representado, atribuindo um número menor de bits para aqueles que ocorrem mais frequentemente. A aplicação do algoritmo de Huffman faz uso de técnicas aprendidas em sala, bem como traz novos desafios.

As funções utilizadas são apresentadas junto as suas características. Algumas funções são compartilhadas entre os códigos de cada programa, portanto o uso de TADs é indispensável para organização do projeto.

2 IMPLEMENTAÇÃO

2.1 Bibliotecas e funções

2.1.1 lista.h

Biblioteca responsável pela manipulação da lista que auxilia na criação da árvore binária de codificação. Inclui a biblioteca arv_binaria.h.

2.1.1.1 struct lista (Lista)

- Função: Estrutura para lista encadeada simples sem sentinela;
- Campos: Um ponteiro para uma estrutura de árvore binária Arv e um ponteiro para o próximo elemento da lista.

2.1.1.2 arv_codif

- Função: Cria árvore de codificação a partir do vetor de frequências;
- Entrada: Vetor cujos índices são os bytes lidos no arquivo original e elementos sao frequências do byte ou índice;

- Saída: Árvore de codificação;
- Pré-condição: Nenhuma;
- Pós-condição: Nós alocados dinamicamente e lista liberada.

2.1.1.3 cria_lista

- Função: Inicializa lista;
- Entrada: Nenhuma;
- Saída: Lista inicializada;
- Pré-condição: Nenhuma;
- Pós-condição: Nenhuma.

2.1.1.4 insere_ordenado

- Função: Insere nó ou árvore de forma crescente com relação à frequência absoluta do byte;
- Entrada: Nó ou árvore a ser ordenado(a);
- Saída: Lista encadeada de árvores binárias ordenada;
- Pré-condição: Lista inicializada;
- Pós-condição: Célula da lista para nó ou árvore alocada e lista ordenada em ordem crescente com relação à frequência.

2.1.2 arv_binaria.h

Biblioteca responsável pela manipulação da árvore binária de codificação.

2.1.2.1 struct arv (Arv)

- Função: Estrutura para armazenar nó da árvore binária de codificação;
- Campos: Tipo de nó (interno ou folha), um byte único lido do arquivo original, a frequência com a qual este aparece e ponteiros para os filhos do nó.

2.1.2.2 cria_arv

- Função: Cria nó de árvore binária Arv;
- Entrada: Campos da estrutura Arv;
- Saída: Nó de árvore binária Arv criado;
- Pré-condição: Nenhuma;
- Pós-condição: Nó alocado dinamicamente.

2.1.2.3 codigos

- Função: Cria tabela cujos índices representam bytes e o conteúdo de cada elemento é o código do byte no algoritmo. Cada elemento diferente de NULL é alocado dinamicamente;
- Entrada: Árvore binária de codificação, string auxiliar, matriz tabela e inteiro auxiliar para recursão;
- Saída: Nenhuma;
- Pré-condição: Árvore, string auxiliar e tabela inicializados;
- Pós-condição: Tabela criada e elementos relevantes da mesma alocados dinamicamente.

2.1.2.4 cria_cabecalho

- Função: Cria maior parte do cabeçalho para arquivo compactado;
- Entrada: Arquivo no qual escrever cabeçalho, árvore binária de codificação;
- Saída: Nenhuma;
- Pré-condição: Arquivo de saída e árvore de codificação inicializados;
- Pós-condição: Parte do cabeçalho escrita no arquivo de saída.

2.1.2.5 libera_arv

- Função: Libera árvore binária Arv;
- Entrada: Árvore binária Arv;
- Saída: Nenhuma;

- Pré-condição: Árvore não nula;
- Pós-condição: Árvore liberada.

2.1.2.6 libera_tab

- Função: Libera tabela de códigos;
- Entrada: Tabela de códigos;
- Saída: Nenhuma;
- Pré-condição: Tabela inicializada;
- Pós-condição: Elementos da tabela liberados.

2.1.2.7 buscaChar

- Função: Auxilia no processo de descompactação escrevendo bytes a partir de seu código;
- Entrada: Nó de árvore binária Arv, direção (0 para esquerda e 1 para direita);
- Saída: Nó na direção fornecida a partir do nó fornecido;
- Pré-condição: Árvore não nula;
- Pós-condição: Nenhuma.

2.1.2.8 retorna_freq

- Função: Retorna frequência absoluta no arquivo original do byte armazenado no nó de uma árvore binária Arv;
- Entrada: Nó de árvore binária Arv;
- Saída: Frequência absoluta do byte referente ao nó no arquivo original;
- Pré-condição: Nó inicializado;
- Pós-condição: Nenhuma.

2.1.2.9 retorna_id

- Função: Retorna tipo de nó;
- Entrada: Nó de árvore binária Arv;
- Saída: Campo identificador do nó;
- Pré-condição: Nó inicializado;
- Pós-condição: Nenhuma.

2.1.2.10 retorna_char

- Função: Retorna caractere representando byte armazenado no nó de uma árvore binária Arv;
- Entrada: Nó de árvore binária Arv;
- Saída: Campo que contém caractere do nó;
- Pré-condição: Nó inicializado;
- Pós-condição: Nenhuma.

2.2 Compactador (compacta.c)

O programa responsável por compactar um arquivo binário recebido como argumento é chamado Compacta. Sua execução em linha de comando é dada por:

```
Compacta nome_arquivo.txt
```

O exemplo mostra nome_arquivo.txt sendo passado como parâmetro para Compacta. Porém, o parâmetro pode ser qualquer arquivo binário.

O arquivo compacta.c contém o código de Compacta, que faz uso da biblioteca lista.h que, por sua vez, inclui a biblioteca arv_binaria.h.

O algoritmo do programa pode ser dividido em cinco passos: verificação do arquivo de entrada; contagem da frequência de cada byte único; criação da árvore binária de codificação; escrita do cabeçalho; e escrita do arquivo codificado.

2.2.1 Verificação do arquivo de entrada

Uma mensagem de erro é exibida caso o usuário esqueça de passar um arquivo como parâmetro ou passe um arquivo que não pode ser aberto pelo programa. Isso é feito com o uso das variáveis argv (int), que guarda o número de parâmetros passados incluindo o nome do programa, e argc (char**), que guarda o nome dos parâmetros.

2.2.2 Contagem da frequência de cada byte único

Um vetor de 256 posições contendo variáveis do tipo int chamado vetChar é declarado, inicializado com todas as posições iguais a zero. Ao ler um byte do arquivo de entrada, este é usado como índice para vetChar, aumentando em uma unidade o valor do elemento naquela posição. Dessa forma, o elemento na posição i será a frequência absoluta do byte i no arquivo de entrada.

2.2.3 Criação da árvore binária de codificação

A árvore binária de codificação é criada e retornada pela função arv_codif que recebe vetChar. Inicialmente, cada elemento de vetChar é verificado e, caso seja maior que zero, armazenado em um nó de árvore binária alocada dinamicamente como frequência absoluta do byte equivalente ao índice de vetChar que representa o elemento verificado. O nó é então inserido como célula alocada dinamicamente em uma lista de forma ordenada, crescente em relação à frequência absoluta do byte que armazena pela função insere_ordenado.

Os primeiros dois elementos da lista tornam-se filhos de um novo nó que contém como frequência a soma das frequências dos filhos. Este novo nó é inserido na lista de forma ordenada por insere_ordenado e o processo é repetido, liberando as alocações dinâmicas de cada célula retirada.

Por fim, uma célula é restante na lista, contendo a árvore binária de codificação. Com a ajuda de uma variável auxiliar, a célula é liberada e a árvore retornada.

2.2.4 Abertura do arquivo de saída e escrita do cabeçalho

O nome do arquivo de entrada é usado para gerar o nome do arquivo de saída, ignorando sua extensão, que será armazenada no cabeçalho. A extensão do arquivo de saída é .comp.

O cabeçalho tem a seguinte estrutura:

- Os dois primeiros bytes são escritos a partir de uma variável do tipo short int que diz o número de bytes distintos existentes no arquivo original;
- O próximo byte será escrito a partir de uma variável do tipo unsigned char que representa um dos bytes lidos;
- Os próximos quatro bytes são escritos a partir de uma variável do tipo int e dizem a frequência absoluta do byte escrito anteriormente;
- Os últimos dois pedaços são escritos para cada byte único do arquivo original, ou seja, pela quantidade de vezes escrita no primeiro byte;
- O próximo byte é escrito a partir de uma variável do tipo char chamada tam que diz o tamanho da extensão do arquivo original. Se tam for zero, o cabeçalho está pronto, caso contrário, a extensão é escrita nos próximos tam bytes.

2.2.5 Escrita do arquivo codificado

São declaradas e inicializadas variáveis auxiliares tais como byte e tamanho. byte é do tipo unsigned int e é usada para formar um byte a ser escrito no arquivo de saída, enquanto tamanho auxilia na formação do byte bem como sua escrita.

Um loop é criado, lendo byte por byte do arquivo original. Em cada iteração, uma variável char* chamada codigo aponta para o elemento da tabela criada anteriormente cujo índice é o byte lido. Portanto codigo aponta para o caminho na árvore binária de codificação que leva ao byte lido.

Um for é usado para percorrer cada digito do codigo, sendo um digito igual a 0 indicador de que devemos seguir para a esquerda na árvore e digito igual a 1 indicador de que devemos seguir para a direita. Se 1 for lido, a variável byte é modificada de forma a atribuir 1 à posição do byte de acordo com o valor de tamanho, que sobe uma

unidade por iteração do for e só é resetado uma vez que byte está cheio e precisa ser escrito.

Ao passar pelos loops, byte pode sair guardando bits de informação que ainda não foram escritos pois tamanho não chegou a 8. Portanto, o valor de tamanho é verificado e o byte é escrito com bits finais que servem apenas para completar o byte.

2.3 Descompactador (descompacta.c)

O programa responsável por descompactar um arquivo compactado recebido como argumento é chamado Descompacta. Sua execução em linha de comando é dada por:

```
Descompacta nome_arquivo.comp
```

O exemplo mostra nome_arquivo.comp sendo passado como parâmetro para Descompacta. O programa funciona somente com arquivos compactados por Compacta e, portanto, de extensão .comp.

O arquivo descompacta.c contém o código de Descompacta, que faz uso da biblioteca lista.h que, por sua vez, inclui a biblioteca arv_binaria.h.

O algoritmo do programa pode ser dividido em quatro passos: verificação do arquivo de entrada; leitura do cabeçalho e criação da árvore binária de codificação; formação do nome do arquivo de saída; e leitura do arquivo codificado e escrita do arquivo original.

2.3.1 Verificação do arquivo de entrada

Uma mensagem de erro é exibida caso o usuário esqueça de passar um arquivo como parâmetro, passe um arquivo de extensão incompatível ou passe um arquivo que não pode ser aberto pelo programa. Isso é feito com o uso das variáveis argv (int), que guarda o número de parâmetros passados incluindo o nome do programa, e argc (char**), que guarda o nome dos parâmetros.

2.3.2 Leitura do cabeçalho e criação da árvore binária de codificação

Um vetor de 256 posições contendo variáveis do tipo int chamado vetChar é declarado, inicializado com todas as posições iguais a zero. Uma variável n do tipo short int é declarada e usada para ler os primeiros dois bytes do arquivo de entrada, que correspondem à quantidade de bytes únicos presentes no arquivo original. Em seguida, uma variável c do tipo unsigned char retém o próximo byte, que corresponde a um dos bytes do arquivo original. Os próximos quatro bytes são lidos e armazenados como int no elemento correspondente ao índice c em vetChar.

A mesma função de criação da árvore binária de codificação é então usada para criar uma árvore idêntica à original a partir do vetChar idêntico ao original.

2.3.3 Formação do nome do arquivo de saída

O próximo byte lido do cabeçalho indica o tamanho em caracteres do nome da extensão do arquivo original, e é lido para a variável tam do tipo unsigned char. Se tam for zero, então o arquivo original não possui extensão e o descompactado passa a ser somente o nome do arquivo compactado sem extensão. Caso contrário, Os próximos tam bytes são lidos e usados para formar uma string que corresponde ao nome da extensão original, que é concatenada com o nome do arquivo compactado. O nome resultante é usado para criar o arquivo descompactado.

2.3.4 Leitura do arquivo codificado e escrita do arquivo original

Variáveis são declaradas a fim de ler o restante do arquivo, correspondente ao arquivo original codificado, e ignorar a possível informação extra gravada no último byte. A variável total do tipo unsigned int recebe o valor no campo frequência do nó raiz da árvore binária de codificação, que é o número total de bytes do arquivo original. Com o uso da variável qtde, é possível parar a leitura assim que qtde atingir o valor de total.

O arquivo é lido byte por byte, armazenando cada um temporariamente na variável byte do tipo unsigned int. Um for é usado para permitir a verificação de cada um dos oito bits de byte. A variável aux do tipo unsigned int é modificada de forma a possuir no bit menos significante o valor do bit lido na iteração, e com isso diz para a função

buscaChar para qual direção da árvore binária de codificação deve seguir, resgatando o nó filho correspondente na variável a do tipo Arv*.

Se o id do nó a for 0, o nó é interno e a condição para escrita de um byte não é atendida. Se o id do nó a for 1, o nó é folha e o byte armazenado em a é escrito no arquivo de saída, somando uma unidade à variável qtde. A condição de parada desse processo de leitura é atendida quando a variável qtde atinge o valor da variável total.

3 CONCLUSÃO

O uso de frequências no cabeçalho aumentou o tamanho mínimo (bem como outras características) de um arquivo necessário para ocorrer redução na compactação, porém permitiu a reutilização de funções em ambos os TADs.

O algoritmo de Huffman alcançou boa compactação dos arquivos testados e apresentou desafios em sua implementação como a manipulação de bits em variáveis em C.

BIBLIOGRAFIA

Stack Overflow. **How to convert decimal value to character in c language**. Disponível em: <https://stackoverflow.com/questions/18330970/how-to-convert-decimal-value-to-character-in-c-language>. Acesso em: 24 nov. 2018.

American Standard Code for Information Interchange. **ASCII table , ascii codes**. Disponível em: <https://theasciicode.com.ar/>. Acesso em: 29 nov. 2018.

Tutorialspoint. **C Library - <string.h>**. Disponível em: <https://www.tutorialspoint.com/c_standard_library/string_h.htm>. Acesso em: 30 nov. 2018.

Tutorialspoint. **Bitwise Operators in C.** Disponível em: <https://www.tutorialspoint.com/cprogramming/c_bitwise_operators.htm>. Acesso em: 30 nov. 2018.