



Aluno: Vinícius de Oliveira Silva

Curso: Ciência da Computação

Disciplina: Algoritmos e Estruturas de Dados II

Documentação de Trabalho Prático

1. Introdução

A compressão de dados é uma técnica amplamente utilizada em Ciência da Computação. A capacidade de reduzir o tamanho de arquivos tem se mostrado de suma importância na era da informação. Através de técnicas de compactação de dados, é possível, por exemplo, trafegar arquivos com um esforço computacional muito pequeno.

O objetivo desse trabalho é implementar o algoritmo de Huffman para compressão de dados, que baseia-se na frequência de cada caractere no arquivo. Caracteres mais frequentes são representados com menos bits, fazendo assim com que o arquivo gerado pelo algoritmo seja significativamente menor do que o arquivo original.

O algoritmo de Huffman é baseado em uma árvore binária, que é utilizada para determinar-se a nova codificação de cada caractere. Cada letra do texto fica armazenada em uma “folha”, e a codificação binária de cada uma depende da posição que ocupa na árvore.

Deverão ser implementados tipos abstratos de dados (TAD's) para representar árvores binárias de maneira computacional e realizar operações sobre eles, tais como, construir a árvore a partir de um texto, verificar se a árvore se encontra vazia, imprimir a string compactada, etc. É importante frisar que a construção de um TAD auxiliar representando uma “Lista Encadeada” faz-se necessária para que o TAD “Árvore” realize suas funções da maneira correta.

Espera-se com isso, praticar técnicas de programação que permitam a manipulação de diferentes estruturas de dados.

2. Implementação

Estruturas de Dados:

Para a implementação desse trabalho, foram criados dois tipos abstratos de dados:

- Lista
- Arvore

O TAD Lista é usado como auxiliar para o TAD Arvore, sua função é permitir que uma árvore de Huffman possa ser construída a partir de suas folhas.

O TAD Arvore é a estrutura de dados principal do trabalho e é utilizado para representar uma árvore dentro do contexto do programa, possibilitando assim, a execução das operações necessárias.

Representação do TAD Lista:

Atributos:

- Apontador para Celula Primeiro;
- Apontador para Celula Ultimo;

O atributo Primeiro é um apontador usado para indicar qual é a primeira Célula armazenada na lista. É importante observar que as células representam os nós folhas da Árvore no contexto do programa. Uma Célula é composta por:

- Node (estrutura de dados que representa um nó de uma Árvore);
- Apontador para Célula (Apontando para a próxima Célula da lista);

O atributo Ultimo é um apontador usado para indicar qual é a última Célula armazenada na lista.

Representação do TAD Arvore:

Atributos:

- Apontador para Node (estrutura de dados que representa um nó de uma Árvore).

O apontador para Nodes é utilizado para indicar qual é o Nó raiz da árvore, ou seja, o topo da mesma. Um Node é composto por:

- **Código:** inteiro que identifica um nó na árvore. No caso dos nós folha, representa a frequência de uma determinada letra na string original.
- **Caractere:** caractere que indica qual letra da string original está armazenada em um nó da árvore. No caso dos nós internos, o caractere é sempre '~'.
- **Byte:** cadeia de caracteres que indica qual sequência de zeros e uns representará uma determinada letra na saída.
- **Esquerda:** apontador que indica qual é o nó raiz da sub-árvore à esquerda. No caso dos nós-folha, Esquerda aponta para NULL.
- **Direita:** apontador que indica qual é o nó raiz da sub-árvore à direita. No caso dos nós-folha, Direita aponta para NULL.

Funções e Procedimentos:

TAD Lista:

void fazListaVazia (Lista *list): recebe uma lista via ponteiros e a inicializa, alocando memória para o seu apontador "Primeiro".

int vazia(Lista list): recebe uma lista via ponteiros e testa se a mesma encontra-se vazia, retornando 1 em caso afirmativo e 0 em caso negativo.

void insere(Celula cell, Lista *list): recebe uma Celula (por valor), e a insere na lista especificada via ponteiro. É importante observar que a inserção de células é feita de maneira ordenada, tendo dois critérios de ordenação, o código e o caractere do nó.

Celula retiraUltimo(Lista *list): retira o último elemento da lista, libera o espaço de memória utilizado por ele e retorna uma cópia do elemento retirado.

void imprime(Lista list): imprime a lista especificada por parâmetro na saída padrão.

TAD Arvore:

void fazArvoreVazia(Arvore *tree): Inicializa uma nova árvore, alocando memória para o nó raiz e inicializando-o com valores nulos.

Arvore construir (char* texto, Node tabelaFreq):** Recebe um texto e a tabela de frequência dos caracteres no mesmo. A função cria a árvore de Huffman representando a situação correspondente, e a retorna. A geração da árvore é feita utilizando a abordagem bottom-up, que cria uma árvore a partir de suas folhas, e não a partir de sua raiz. Esse procedimento requer o uso do TAD auxiliar Lista, e por isso, a função "construir" faz diversas chamadas a funções desse TAD.

Essa é a principal função do programa, visto que é a responsável por montar toda a estrutura da árvore utilizada para a codificação dos caracteres.

char* getBytes(char letra, Arvore tree): Função que recebe uma letra como parâmetro e pesquisa na árvore especificada por sua codificação binária. Caso a letra exista na árvore, a codificação binária correspondente é retornada. Essa função poderia ser somente uma função auxiliar do TAD, mas é disponibilizada no header pois contribui para uma melhor abstração de uma árvore de Huffman.

void imprimeBytes(Node* tabelaFreq, Arvore tree, char* texto): Função que imprime toda a sequência binária correspondente ao texto digitado pelo usuário. Para cada letra distinta da string, uma única busca na árvore é feita, e o código encontrado é armazenado na tabela de frequência para que, caso a letra se repita na string, a busca seja feita de maneira mais rápida, uma vez que não será necessário consultar a árvore novamente.

void imprimeCompactado(Node* tabelaFreq, Arvore tree, char* texto): Possui basicamente a mesma finalidade que a função *imprimeBytes*, porém essa função imprime a string da entrada de forma verdadeiramente compactada. A cada 8 caracteres da sequência de 0's e 1's que representa a string codificada, um número inteiro é gerado (através de conversão de base binária para decimal) e o caractere Ascii corresponde é impresso. É interessante observar que essa função só é utilizada na implementação da parte extra do trabalho.

Node criaNoInt(Node* filhoEsq, Node* filhoDir): Cria um nó interno para a árvore, tendo como filho da esquerda o nó recebido por parâmetro "filhoEsq" e como filho da direita, o nó recebido por parâmetro "filhoDir". O código do nó gerado é a soma dos códigos de seus filhos. É interessante salientar que essa é uma função interna do TAD, e portanto, não está disponível para chamadores externos.

void preencheBytes(Arvore tree): Função recursiva que percorre a árvore especificada por parâmetro atribuindo códigos binários para cada um de seus nós. Sua lógica de funcionamento parte do princípio de que o filho da direita de um nó sempre terá seu código binário igual ao de seu pai concatenado de "1" e o filho da esquerda sempre terá seu código binário igual ao de seu pai concatenado de "0". Essa função também é interna ao TAD, e por isso, não foi disponibilizada em seu header. É importante ressaltar que a função só deve ser chamada depois que a árvore já está totalmente construída, e por isso, essa função só é chamada pela função *construir*.

int posicao(Node* array, char letra): Função interna do TAD que permite a busca da posição de uma determinada letra na tabela de frequência. Essa é a função que possibilita a eficiência da função *imprimeBytes*, pois permite recuperar a posição onde a cadeia binária de uma determinada letra está armazenada na tabela de frequência.

int byteToInt(char* byte): Função interna do TAD que recebe uma cadeia de 8 caracteres 0's e 1's e retorna o número inteiro correspondente àquela codificação binária. Essa função só é utilizada no desenvolvimento da parte extra do trabalho.

Programa Principal:

O programa principal recebe a entrada de dados do usuário através da função *fgets*, que permite a leitura de strings com espaços, e em seguida, declara uma variável do tipo *Arvore*, que é posteriormente preenchida através da função *construir* do TAD.

Depois que a árvore já está montada, a função *main* faz uma chamada ao procedimento *imprimeBytes* (na implementação principal) que exibe a string digitada pelo usuário de forma codificada de acordo com a árvore de Huffman. Na implementação extra, o procedimento *imprimeCompactado* é chamado para que a saída gerada corresponda à string de entrada efetivamente compactada.

Toda a entrada de dados é lida de *stdin* e impressa em *stdout*.

É importante também observarmos que em momento algum, os atributos do TAD *Arvore* são acessados no programa principal. Apenas as funções enunciadas na interface do TAD são acessadas.

Organização do Código, Decisões de Implementação e Detalhes Técnicos:

O código está dividido em 5 arquivos principais. Os arquivos *Lista.c* e *Lista.h* implementam o TAD *lista*, já os arquivos *Arvore.c* e *Arvore.h* implementam o TAD *Arvore*, enquanto o arquivo *TP2.c* implementa o programa principal.

Uma importante decisão de implementação que foi tomada durante a implementação desse trabalho, foi a criação da função *getBytes*, dedicada à tarefa de pesquisar na árvore qual o código binário de uma determinada letra da string digitada pelo usuário. Essa função foi criada com o intuito de preservar a abstração, permitindo assim maior flexibilidade em caso de reutilização de código.

O compilador utilizado foi o MacOSX GCC através da IDE Eclipse Kepler, no sistema operacional Mac OSX 10.9.2 (Mavericks).

3. Análise de Complexidade

A análise de complexidade será feita em função da variável n que representa o número de caracteres distintos da string de entrada.

TAD Lista:

Função fazListaVazia: como a função apenas executa comandos $O(1)$, e não há loops, podemos dizer que sua Ordem de complexidade é $O(1)$.

Função vazia: como a função apenas executa comandos $O(1)$, e não há loops, podemos dizer que sua Ordem de complexidade é $O(1)$.

Função insere: a complexidade dessa função é $O(n)$ no pior caso (quando se quer inserir uma letra cuja frequência é menor do que a de todas as outras já inseridas) e $O(1)$ no melhor caso (quando se quer inserir uma letra cuja frequência é maior do que o de todas as outras letras já inseridos).

Função retiraUltimo: como a lista não é duplamente encadeada, e o objetivo da função é retirar o último elemento da mesma, é necessário que se percorra toda a lista a fim de obter um apontador que aponte para o penúltimo elemento da lista (que por sua vez possui um apontador que indica qual é o elemento a ser removido). Assim sendo, podemos dizer que a complexidade da função é $O(n)$.

Função imprime: como cada elemento da lista deve ser impresso, concluímos que a Ordem de complexidade dessa função é $O(n)$.

TAD Arvore:

Função fazArvoreVazia: A função apenas executa comandos $O(1)$, e não há loops, por isso, podemos dizer que sua ordem de complexidade é $O(1)$.

Função construir: A primeira vista, podemos pensar que a complexidade dessa função é no pior caso, $O(2n^2)$, uma vez que ela faz duas chamadas à função *retiraUltimo* (que por sua vez é $O(n)$), dentro de um loop que executa n vezes. Porém, é importante observar que essa função faz uma chamada ao procedimento recursivo *preencheBytes*, e por isso, devemos somar sua complexidade (que será posteriormente calculada) ao valor já determinado.

Função getBytes: Como a função *getBytes* é recursiva, sua análise de complexidade passa pela solução de uma equação de recorrência. Neste caso, a complexidade da função é dada pela seguinte função:

$F(n) = 2F(n/2)+3$, onde $F(1)=1$. Resolvendo a equação, chegamos em $F(n) = 4n-3$. Assim, podemos dizer que a função é $O(4n)$.

É importante notar que a análise é feita considerando o pior caso, ou seja, quando a árvore de Huffman está balanceada (isto é, o número de elementos a esquerda da raiz é igual ao número de elementos a direita).

Função *imprimeBytes*: Para se fazer a análise de complexidade dessa função, é necessário definir-se uma variável que represente a quantidade de caracteres da string de entrada. Podemos chamá-la de “m”. Inicialmente, a função percorre toda a tabela de frequência fazendo 53 iterações, e em cada uma delas, a função *getBytes* é chamada. Em seguida, a função *posição* (cuja complexidade é $\Theta(53)$) é executada m vezes. Assim, a complexidade da função *imprimeBytes* é: $53*(4n-3) + m*53$.

Cabe destacar que é essa a função mais demorada do programa (sem contar as funções exclusivas da parte extra), visto que ela tem de trabalhar com o total de caracteres da string, não somente os distintos.

Função *imprimeCompactado*: Para se fazer a análise de complexidade dessa função, também é necessário que se defina uma variável que represente a quantidade de caracteres na string de entrada. Aqui, também utilizaremos o nome “m” para nos referir a essa variável. Também é necessário que se defina uma variável para representar o número de caracteres da string codificada. Utilizaremos a letra “L” para esse fim.

Inicialmente, essa função executa um algoritmo bastante parecido com o algoritmo implementado pelo procedimento *imprimeBytes*, e por isso podemos partir de sua função de complexidade para determinar a complexidade da função *imprimeCompactado*.

Além de repetir o algoritmo da *imprimeBytes*, a função também tem de percorrer a string de saída (de tamanho L) e converter cada grupo de 8 zeros e/ou uns em um número inteiro (a função *byteToInt* é utilizada para esse fim). A complexidade para se fazer isso é de 8L. Assim, a complexidade total dessa função é:

$$53*(4n-3) + m*53 + 8L.$$

Função *criaNoInt*: A função apenas executa comandos $O(1)$, e não há loops, por isso, podemos dizer que sua ordem de complexidade é $O(1)$.

Função *preencheBytes*: Assim como a função *getBytes*, essa função também é recursiva e sua análise de complexidade exige a solução de uma equação de recorrência. Neste caso, a equação é: $F(n) = 2F(n/2) + 1$, onde $F(1)=1$.

Resolvendo a equação, encontramos: $F(n) = 2n-1$, ou seja, a ordem de complexidade dessa função é $O(2n)$.

Função *Posição*: Essa função faz sempre 53 iterações na tabela de frequência, por isso, sua ordem de complexidade é $\Theta(53)$.

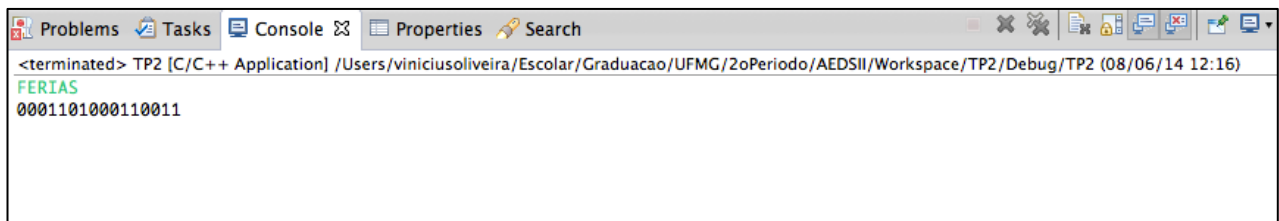
Função byteToInt: Como essa função sempre executa um loop durante 8 iterações, podemos dizer que sua complexidade é $\Theta(8)$.

4. Testes

Inúmeros testes foram realizados com o programa no intuito de avaliar seu comportamento em diversas situações específicas, tais como strings que geram empate de código entre nós internos, strings onde cada caractere tem frequência única, strings com muitos caracteres, etc. Todos os testes foram bem sucedidos e a saída gerada corresponde à saída esperada.

Os testes foram executados em um computador equipado com processador Intel Core 2 Duo @2.4GHz e 4GB de memória. A saída abaixo mostra uma execução típica do programa:

Console do Eclipse executando o trabalho prático:



```
<terminated> TP2 [C/C++ Application] /Users/viniciusoliveira/Escolar/Graduacao/UFMG/2oPeriodo/AEDSII/Workspace/TP2/Debug/TP2 (08/06/14 12:16)
FERIAS
0001101000110011
```

Obs.: Os caracteres em verde representam a entrada do Trabalho e os em preto representam a saída.

5. Conclusão

A implementação desse trabalho foi mais longa e custosa do que o esperado devido ao fato de que problemas relacionados ao tempo de execução do algoritmo estavam impedindo que o programa fosse avaliado no sistema “Prático”, causando portanto, a atribuição de nota insatisfatória. Como o problema que causava esses erros ainda não era conhecido, muito tempo foi gasto procurando por falhas que sequer existiam.

Após a identificação de que o problema era a complexidade do algoritmo, bastou uma otimização do código para que o programa apresentasse a precisão e eficiência esperadas.

6. Referências

[1] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning.