Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Algoritmos e Estruturas de Dados II
1º Semestre de 2014
Prof. Raquel Prates e Luiz Chaimowicz

Trabalho Prático 2 – Algoritmo de Huffman para a compactação de arquivos

Valor: 15 pontos
Data de entrega: 03/06/2014

Esse trabalho pretende exercitar diversos conceitos vistos em sala tais como árvores, ordenação e pesquisa. O objetivo é implementar o Algoritmo de Huffman para a compactação e descompactação de arquivos de texto. Basicamente, dado um arquivo de entrada, algoritmos de compactação (ou compressão) devem gerar um arquivo de saída mais compacto (com tamanho menor) que o arquivo de entrada. Normalmente os algoritmos de compactação devem funcionar "sem perdas", ou seja, após a descompactação o arquivo resultante deve ser idêntico ao original. Esse é o caso do Algoritmo de Huffman.

O algoritmo de Huffman trabalha de maneira que a freqüência com que um símbolo aparece em um arquivo determine o tamanho da (nova) representação escolhida para esse símbolo. Ou seja, ao contrário do código ASCII estendido, onde cada caractere é representado por 8 bits, o sistema de codificação de Huffman trabalha com representações de tamanho variado, sendo os caracteres mais frequentes representados por códigos menores.

Considere a seqüência ABRACADABRA que contém 5 símbolos distintos que podem, a princípio, ser representados com 3 bits cada. A seqüência contém 11 letras sendo 5 As, 2 Bs, 2 Rs, 1 C e 1 D. Em uma representação onde um número fixo de bits é utilizada, seriam necessários 33 bits para guardar a seqüência (11 x 3 = 33). Porém, se o código de Huffman for utilizado para representar essa mesma freqüência, poderíamos codificar A, o símbolo mais freqüente, utilizando 1 bit (0) e assim por diante: A: 0 B: 10 R: 110 C: 1110 D: 1111. Com essa codificação, nossa seqüência seria representada da seguinte forma: 01011001110011110101100, ocupando 23 bits (5x1 + 2x2 + 2x3 + 1x4 + 1x4)

Por ser um algoritmo com códigos de tamanho variável, a codificação gerada pelo Algoritmo de Huffman é pré-fixada. Nesse tipo de codificação, um código representando um caractere nunca será o prefixo de um segundo código representando outro caractere.

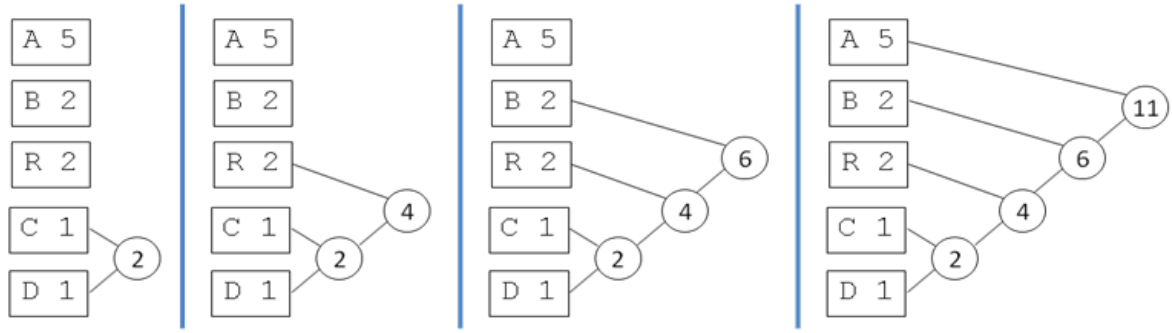
O algoritmo de Huffman cria tal codificação utilizando uma árvore binária, composta de nós internos e nós folha. Nós folha representam um caractere da mensagem, associado a freqüência com que aparecem na mensagem. Nós internos são representados pela soma das frequencias de seus nós filhos. A seqüência de bits associada a um caractere é determinada pelo caminho da raiz da árvore até o caractere. Por convenção, caminhos à esquerda representam 0 e à direita representam 1. A figura abaixo mostra uma possível árvore e codificação para o texto ABRACADABRA (as freqüências são mostradas em vermelho)

Para executar o algoritmo são necessários os seguintes passos:

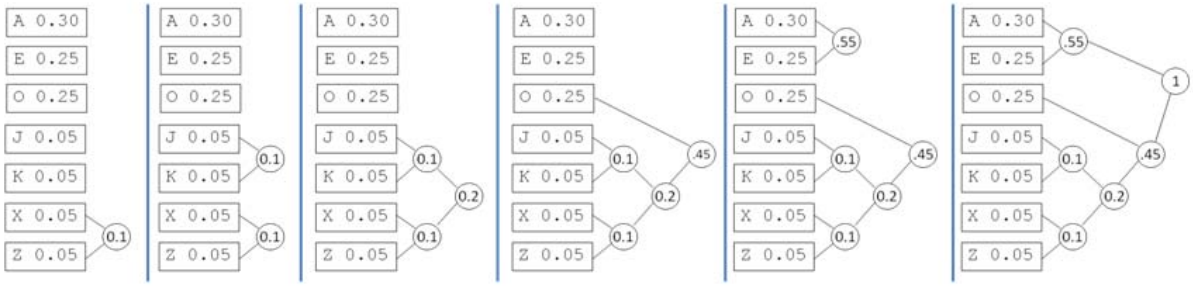
1. **Montar a tabela de freqüências**: ler o arquivo de entrada e montar uma tabela com as freqüências de ocorrência de cada símbolo. Para o passo seguinte, é conveniente que sua tabela esteja ordenada de forma crescente.

2. **Construir a Árvore de Huffman:** a partir da tabela de freqüências você deverá construir a árvore de Huffman. Uma possível maneira de fazer isso é fazer uma abordagem *bottom-up*. Considere que cada símbolo da sua tabela um nodo folha de uma árvore. Selecione os dois símbolos menos freqüentes, e crie uma subárvore cuja raiz e um nó interno com a freqüência igual à soma das freqüências de seus filhos. Repita o processo, selecionando sempre os novos símbolos ou subárvores já existente que tenham a menor freqüência, até que todos os símbolos façam parte da

árvore. Note que os símbolos de maior freqüência estarão mais próximos da raiz. No nosso exemplo, a execução seria:



Outro exemplo, considerando um texto com as seguintes frações (freqüência / total) A (0.3), E, O (0.25), J, K, X, Z (0.05):



Em casos onde letras possuam a mesma frequência, como o exemplo onde R e B aparecem duas vezes, o critério de desempate é o seu código ASCII, ou seja, a letra que tiver o menor código ASCII aparece primeiro na tabela.

3. **Tabela de codificação dos caracteres**: o próximo passo é a construção de uma tabela para a codificação dos caracteres. Para isso, basta fazer um caminhamento na árvore considerando que cada passo a esquerda corresponde ao bit 0 e cada passo a direita corresponde ao bit 1. Nos exemplos acima, as tabelas seriam:

```
A: 0      B: 10     C: 1110   D: 1111   R: 110
```

```
A: 00     E: 01     J: 1100   K: 1101   O: 10     X: 1110   Z: 1111
```

4. **Escrita do Arquivo compactado:** uma vez criada a tabela de símbolos, os caracteres do arquivo de entrada devem ser convertidos e escritos no arquivo de saída. Por exemplo, no caso do texto ABRACADABRA, a codificação seria: 01011001110011110101100. No caso, para facilitar a correção a saída será em arquivo texto. Assim, observe que, na prática, você não compactou o texto. Na verdade, você aumentou o tamanho dele, uma vez que cada dígito armazenado como texto gastará o mesmo espaço para ser armazenado que uma letra, e em alguns casos você transformou uma letra em uma representação de 4 dígitos.!

A implementação do programa que faça uma compactação verdadeira é uma atividade extra. Os detalhes estão descritos abaixo:

Compactação verdadeira (5 pontos extras): Para realmente haver a compactação, os 0s e 1s da codificação devem ser tratados como bits, sendo agrupados e escritos como bytes (*unsigned char*) no arquivo de saída. Faz parte do trabalho extra aprender como manipular bits em C (operadores <<, >>, |, &, etc). Por exemplo, no caso do texto ABRACADABRA, a codificação seria: 010110011110011110101100, que seria escrita no arquivo de saída como **`YÏY`** (01011001 = 89, ASCII(89) = Y; 11001111 = 207, ASCII(207) = Ï; 0101100**1** = 89, ASCII(89) = Y). Na verdade, a aparência do que vai ser escrito depende do editor de texto, codificação do ASCII Estendido, etc. O que importa é que os bytes 89, 207 e 89 sejam escritos. Outro detalhe é que a última parte da codificação, por ter apenas 7 bits foi completada com o número 1 (em vermelho) para inteirar um byte.

A árvore de Huffman a ser implementada deve **obrigatoriamente** fazer o uso de ponteiros e alocação dinâmica. Estamos avaliando o aprendizado da matéria vista em aula, e não sua capacidade de criar um compactador eficiente. O programa deverá ser executado passando-se opções na linha de comando:

./tp2 < *entrada* > *saída*

O arquivo de entrada irá conter uma *string* de tamanho máximo 10000 caracteres. Essa string pode possuir espaços, letras maíusculas e minúsculas. Caracteres especiais e acentos não irão aparecer. Ou seja, "The Quick Brown Fox Jumps Over The Lazy Dog" é uma entrada válida, mas a entrada "Essa entrada é inválida!!!" inválida por causa dos assentos e pontuação.

O arquivo de saída deve conter somente o conjunto de 0s e 1s impresso. No caso de "ABRACADABRA", você deverá imprimir "010110011110011110101100".

Caso deseje implementar o trabalho extra, você deverá criar um outro programa que imprima a codificação usando *unsigned char*. Sua saída deverá conter somente o conjunto dos caracteres gerados. Também usando o exemplo "ABRACADABRA", sua saída deverá ser somente "YÏY".

O que deve ser entregue:

Serão criadas duas páginas de submissão no Prático. Uma para o trabalho normal, onde todos devem submeter o código e a documentação, e uma página para submeter somente o código do trabalho extra.

Para aqueles que fizerem o trabalho extra, a documentação deve conter também a parte de implementação e testes da compactação verdadeira.

- Código fonte do programa em C (todos os arquivos *.c* e *.h*), bem identado e comentado.
- Documentação do trabalho. Entre outras coisas, a documentação deve conter:
    1. Introdução: descrição do problema a ser resolvido e visão geral sobre o funcionamento do programa.
    2. Implementação: descrição sobre a implementação do programa. Deve ser detalhada a estrutura de dados utilizada (de preferência com diagramas ilustrativos), o funcionamento das principais funções e procedimentos utilizados, o formato de entrada e saída de dados, compilador utilizado, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado. Por exemplo, qual método de ordenação usou? Por que?
    3. Estudo de Complexidade: estudo da complexidade do tempo de execução dos procedimentos implementados e do programa como um todo (notação O), considerando conjuntos de tamanho *n*.

4. Testes: descrição dos testes realizados e listagem da saída (não edite os resultados).
5. Conclusão: comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
6. Bibliografia: bibliografia utilizada para o desenvolvimento do trabalho, incluindo sites da Internet se for o caso

Um exemplo de documentação está disponível no Moodle.

Comentários Gerais:

1. Comece a fazer este trabalho logo, enquanto o problema está fresco na memória e o prazo para terminá-lo está tão longe quanto jamais poderá estar.
2. Clareza, indentação e comentários no programa também serão avaliados.
3. O trabalho é individual.
4. A submissão será feita pelo Prático.
5. Trabalhos copiados, comprados, doados, etc serão penalizados conforme anunciado.
6. Logo após a entrega poderá haver um pequeno exercício sobre a implementação do trabalho. A nota do trabalho será ponderada pela nota desse exercício.
7. Penalização por atraso: (2d – 1) pontos, onde d é o número de dias de atraso.