---> Desenhe vários diagramas (e faça vários exemplos) para entender o que está implementando <----

Compressao de arquivos de texto usando a codificacao de Huffman

Fonte: Winzip.com

A compressão de arquivos é um tema muito importante na computação (e que frequentemente desperta a curiosidade de estudantes curiosos -- espero que você seja um desses).

“Comprimir” um arquivo é algo muito fácil -- difícil é conseguir atingir dois objetivos: ser capaz de restaurar o arquivo original e obter uma boa taxa de compressão.

Há dois tipos importantes de compressão: a com perdas (*lossy*) e a sem perdas (*lossless*). Na compressão sem perdas (usada para arquivos de texto, imagens de alta qualidade, arquivos executáveis, etc) o arquivo descomprimido deve ser idêntico ao original, enquanto na com perdas (utilizado, por exemplo, para imagens, música, filmes, etc) ele precisa ser apenas “similar” (pequenas imperfeições são aceitáveis).

Um método simples (mas não tão eficiente quanto outros) para compressão de arquivos SEM PERDAS (*lossless*) consiste em utilizar a *Codificação de Huffman*. Apesar da taxa de compressão obtida utilizando essa codificação não ser tão alta quanto a obtida por métodos de compressão mais famosos, a codificação de Huffman é muito utilizada como **parte** de algoritmos famosos de compressao (exemplo: JPEG, MP3, Winzip, etc). Ou seja, após utilizar outras técnicas de compressão a codificação de Huffman é aplicada para comprimir ainda mais os dados já comprimidos.

(curiosidade: MP3 utiliza compressao com perdas -- como um algoritmo *lossless* pode ser útil para o MP3?)

---> Desenhe vários diagramas (e faça vários exemplos) para entender o que está implementando <----

Ideia

Para simplificar a explicação, neste roteiro vamos supor que estamos comprimindo um arquivo de texto (na prática não há diferença entre isso e a compressão de outros arquivos -- no fim tudo são bytes).

A ideia básica dessa técnica de compressão consiste em utilizar uma codificação onde os caracteres mais frequentes no arquivo de entrada sejam representados usando menos bytes. Por exemplo, considere o texto “ABBBBBBBBACD”. Nesse texto, temos a seguinte frequência para cada caractere:

Caractere	Frequencia	Codificacao fixa (cod. ASCII)	Codificacao fixa 2	Codificacao variavel
A	2	01000001	00	10
B	8	01000010	01	0
C	1	01000011	10	110
D	1	01000100	11	111

Se o texto acima for representado em disco utilizando a codificação tradicional (usando a tabela ASCII, onde cada caractere ocupa 8 bits), ele seria armazenado como (o espaço em branco entre cada código foi adicionado por legibilidade):
01000001 01000010 01000010 01000010 01000010 01000010 01000010 01000010 01000010 01000001 01000011 01000100

Isso ocuparia 96 bits em disco (12 caracteres ocupando 8 bits cada).

Uma alternativa seria representar o texto utilizando menos bits. Como no exemplo acima há apenas 4 letras poderíamos usar 2 bits para representá-las (veja: codificação fixa 2). Assim, o texto seria armazenado como:
00 01 01 01 01 01 01 01 01 00 10 11

Isso ocuparia 24 bits, 2 por caractere. Obviamente também seria necessário armazenar (junto ao arquivo “compactado”) uma tabela indicando como cada caractere é codificado.

A ideia anterior não é muito boa pois normalmente todos os 256 bytes possíveis aparecem em arquivos e, assim, raramente é possível utilizar uma codificação com menos de 8 bits por caractere. Se o arquivo for de texto e contiver apenas letras do alfabeto inglês, seriam necessários 7 bits para representar todas as 52 possiveis letras minusculas e maiusculas. Com isso, seria possível obter uma taxa de compressão de 7/8 = 88% apenas usando essa codificação mais simples e compacta.

Uma ideia melhor consiste em utilizar uma codificação variável (onde os caracteres mais frequentes usam menos bits). Com a do exemplo acima, o texto “ABBBBBBBBACD” poderia ser representado por:
10 0 0 0 0 0 0 0 0 10 110 111 (usando apenas 18 bits!)

Essa codificação usaria menos espaço e, ao mesmo tempo, permitiria representar qualquer um dos 256 possíveis bytes.

Problema: nos arquivos os bits são armazenados sequencialmente e não há um “espaço em branco” para separar cada conjunto de bits que representa um caractere. Ou seja, ao “olhar um arquivo” em vez de “10 0 0 110” (“ABBC”) veriamos “1000110”. Como saber onde cada caractere termina? (na codificação fixa basta separar os bits em pedaços de tamanho igual ao número de bits da codificação). Como saber que “1000110” é “10 0 0 110” e não “100 0 1 1 0”? Analisando a codificação variável apresentada acima podemos notar que há apenas uma forma de decodificar “1000110” (ou seja, isso só pode representar ABBC). Isso ocorre porque o código de um caractere nunca é prefixo de outro código!

Neste trabalho vamos utilizar essa ideia para comprimir (e descomprimir) arquivos. Ou seja, dado o arquivo a ser comprimido, você deverá contar a frequência de cada byte e gerar uma codificação que usa menos bits para os caracteres mais frequentes (com a propriedade do código de um byte nunca ser prefixo do código de outro). Então, você deverá codificar os dados de entrada e grava-los em um arquivo. Para permitir a descompressão, os códigos associados a cada byte deverão ser armazenados de alguma forma junto com o arquivo comprimido.

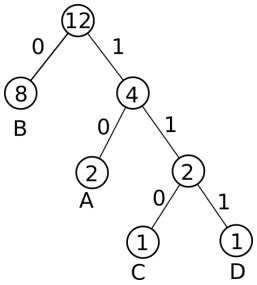
Como gerar uma codificação variável?

David Huffman (um estudante de doutorado) criou em 1951 (durante um trabalho em uma disciplina do MIT) uma técnica para gerar codificações variáveis de forma eficiente e ótima (ou seja, não há como criar uma codificação que use menos bits do que a de Huffman! -- obviamente há outros métodos de compressão que comprimem mais, mas eles usam técnicas diferentes da “codificação de símbolos com número variável de bits”).

Árvore de Huffman

Para gerar uma codificação de Huffman é criada uma árvore binária onde os caracteres (símbolos) ficam nas folhas e a direção utilizada para chegar em cada caractere é utilizada para formar a codificação. Cada nodo contém a soma da frequência dos símbolos em sua sub-árvore (isso é importante apenas para construir a árvore, conforme será visto adiante). Quando é feito um percurso à esquerda temos um símbolo 0 e quando é feito um percurso à direita temos um símbolo 1.

Por exemplo, a codificação acima pode ser representada pela seguinte árvore (árvore de Huffman):



Notem que o caractere A tem codificação 10 (primeiro deve-se ir a direita e depois à esquerda para chegar em A)

Usando uma árvore de Huffman para descomprimir um arquivo

Dado um arquivo comprimido, como descomprimir-lo? Basta percorrer a árvore a partir da raiz (voltando para raiz após chegar em cada folha)

Por exemplo, dado um arquivo contendo os bits “1000110”, começamos na raiz e processamos cada bit do arquivo fazendo o seguinte:
1: vá para a direita na árvore (a partir da raiz)
0: vá para a esquerda. Como chegamos em uma folha (contendo A), grave **A** na saída e volte a raiz.
0: vá para a esquerda a partir da raiz. Como chegamos na folha B, grave **B** e volte a raiz.
0: vá para a esquerda a partir da raiz. Como chegamos na folha B, grave **B** e volte a raiz.
1: vá para a direita a partir da raiz.
1: vá para a direita.
0: vá para a esquerda. Como chegamos na folha 0, grave **C**.

Logo, o arquivo descomprimido será “ABBC” !

Usando uma árvore de Huffman para descomprimir um arquivo

Basta andar na árvore e pré-calcular a codificação de cada símbolo (exemplo acima: o símbolo C teria codificação “110”). A seguir, para cada símbolo, grave sua codificação no arquivo de saída.

Observe que considerar que o lado direito vale 1 e o esquerdo vale 0 não é importante (poderia ser o contrário). O que importa é que a mesma estratégia usada na compressão será utilizada na descompressão.

Criando uma árvore de Huffman

Conforme dito anteriormente, dada a árvore é fácil compactar ou descompactar um arquivo. O desafio é construir a árvore de Huffman. Felizmente o algoritmo que realiza essa tarefa é bem simples.

Algoritmo para criar uma árvore de Huffman (assuma que o peso de uma árvore é a frequência armazenada em sua raiz):

1. Crie uma árvore para cada símbolo. Cada árvore será composta de apenas um nodo e tal nodo terá valor igual à frequência de seu símbolo.
2. Insira essas árvores em uma fila de prioridades PQ.
3. Enquanto o número de árvores não for 1, faça:
    a. Pegue (e as remova da fila) as duas árvores a,b de menor “peso”
    b. Crie uma nova árvore T onde a raiz será um nodo cujo peso é a soma do peso de a e de b. O filho esquerdo de T será a e o direito será b.
    c. Insira T de volta em PQ
4. A árvore de Huffman será a (única) árvore restante na fila de prioridades.

Faça alguns exemplos para entender esse algoritmo (na internet é possível encontrar vídeos com exemplos).

O trabalho

Você deverá inicialmente criar uma classe chamada HuffManTree (coloque sua declaração no arquivo Huffman.h e a implementação no arquivo Huffman.cpp).

Sua classe deverá ter um construtor que recebe como argumento um array automático com a frequência dos 256 possíveis símbolos em um byte (“HuffManTree(int freqs[256])”). Ao ser construída, o construtor deverá criar uma árvore de Huffman (que será armazenada na parte privada de HuffManTree) usando o algoritmo descrito anteriormente (e **obrigatoriamente** usando uma fila de prioridades).

A classe HuffManTree **deverá** ter pelo menos dois métodos públicos (outros métodos podem ser necessários para o correto funcionamento de sua classe):

1. void comprimir(MyVec<bool> &out, const MyVec<char> &in) const;
    a. Essa função deverá ler o vetor de bytes (chars) “in”, comprimi-lo e gravar os bits representando o arquivo comprimido em “out” (cada bool de out representará um bit , sendo 1 representado por true e 0 por false)

2. void descomprimir(MyVec<char> &out, const MyVec<bool> &in) const;
    a. Essa função deverá fazer o inverso do que a função acima faz.

O arquivo teste.cpp (código fonte disponível no final deste documento) contém um exemplo de uso da sua classe. Observe que a codificação gerada pelo seu programa pode ser diferente (a árvore de Huffman para uma determinada frequência de símbolos não é única!). O que importa é que o número de bits usado no total seja o mínimo possível (apesar das árvores de Huffman (para uma mesma frequência) não serem únicas, todas são ótimas).

A seguir, crie um programa main.cpp que usa sua classe para comprimir arquivos. Seu programa deverá receber tres argumentos. O primeiro deverá ser “c” ou “d”. Se for “c”, ele

deverá ler um arquivo (o caminho será o segundo argumento) e gravar a versão comprimida dele em outro arquivo (o caminho será o terceiro argumento). Se o argumento for “d” ele deverá ler um arquivo comprimido (o caminho será o segundo argumento) e gravar a versão descomprimida em outro arquivo (o caminho será o terceiro argumento).

Por exemplo, “./a.out c wikipediaBrasil.html teste.dat” deverá comprimir wikipediaBrasil.html (gravando o arquivo comprimido em teste.dat) e “./a.out d teste.dat wikipediaBrasil.html” deverá restaurar o arquivo (que deverá ficar IDENTICO ao original). Considerando o arquivo wikipediaBrasil.html (disponível como exemplo), espera-se que o arquivo comprimido tenha em torno de 60% do tamanho do arquivo original.

Trabalhando com arquivos, bits, etc.

Note que seu programa main deverá ler o arquivo de entrada, calcular a frequência de cada byte, criar a árvore de Huffman e gravar no arquivo comprimido a codificação da entrada usando um número variável de bits por símbolo.

Assim, se ao comprimir uma determinada string o resultado for “1000100011110000” (representado por uma sequência de booleanos) seu programa deverá gravar dois bytes no arquivo resultante (compactado). I.e, ele deverá gravar os bytes 10001000 e 11110000 (não a “string” formada pelos caracteres 1000100011110000).

O mesmo deverá valer para a descompressão.

A ideia da classe HuffmanTree trabalhar com vetores (MyVec) de booleanos e caracteres (em vez de manipular diretamente os bits dos arquivos de entrada) provavelmente deixará seu código um pouco menos eficiente, mas por outro lado o deixará mais organizado (a manipulação de arquivos e bits ficará fora da classe HuffmanTree).

Armazenando a árvore de Huffman

Notem que para descompactar um arquivo é necessário saber qual árvore de Huffman o gerou. Assim, juntamente com o arquivo compactado (dentro dele) você também deverá armazenar algo que deverá ser utilizado para restaurar a árvore de Huffman.

Por simplicidade, sugiro que no início do arquivo compactado você armazene a frequência de cada um dos 256 possíveis bytes presentes no arquivo original (ou seja, as frequências usadas para gerar a árvore utilizada na compressão). Assim, ao descompactar um arquivo você pode reconstruir a árvore original usando essas frequências. Lembre-se que as árvores de Huffman não são únicas e, assim, essa estratégia só irá funcionar porque o mesmo programa usado para compactar será usado para descompactar (e, dadas as frequências, as árvores serão geradas de forma determinística). Sendo assim, pode ser que um arquivo comprimido usando seu programa não possa ser descomprimido usando o programa de um colega. (isso não é uma boa prática! Está sendo feito apenas para facilitar o trabalho)

---> você está se lembrando de desenhar vários diagramas para entender o que está implementando??? <----

Observacoes

- Nos exemplos apresentados acima consideramos apenas a compressão de arquivos de texto (pois são mais fáceis de entender). Porém, seu programa deverá funcionar com qualquer tipo de arquivo (você não precisa fazer nada de especial para que isso ocorra… não há muitas diferenças entre um arquivo “de texto” e um arquivo de um filme (no fim todos são apenas sequências de bytes)). Obviamente alguns algoritmos de compressão são projetados especialmente para determinados tipos de arquivos (ex: JPEG para imagens).
- O que acontece se você tentar compactar um arquivo já compactado? Um zip? Uma imagem JPEG? O seu próprio “a.out”? O código fonte do seu programa? (faça esses testes).
- Você confia na corretude da sua implementação? Teria coragem de compactar o código fonte do seu trabalho e apagar os arquivos originais (e todo backup)?
- Um arquivo comprimido será sempre menor que o descomprimido?
- (a resposta para estas perguntas não precisa ser entregue)

Arquivo README

Seu trabalho deverá incluir um arquivo README.

Tal arquivo deverá conter:

- Seu nome/matricula
- Informacoes sobre fontes de consulta utilizadas no trabalho

Submissao

Submeta seu trabalho utilizando o sistema Submitty até a data limite. Seu programa será avaliado de forma automática (os resultados precisam estar corretos, o programa não pode ter erros de memória, etc), passará por testes automáticos “escondidos” e a qualidade do seu código será avaliada de forma manual.

Você deverá enviar os arquivos: main.cpp, Huffman.cpp, Huffman.h e README.txt (arquivos adicionais .cpp e .h também podem ser enviados).

Duvidas

Dúvidas sobre este trabalho deverão ser postadas no sistema Piazza. Se esforce para implementá-lo e não hesite em postar suas dúvidas!

Avaliacao manual

A nota obtida nos casos de teste automáticos poderá ser alterada (por exemplo, um programa que apenas copia o arquivo de entrada para o de saída (ou seja, que não compacta) ganha muitos pontos visto que ele boa parte dos pontos são destinados a descompactar corretamente um arquivo comprimido). Assim, não tente obter pontos usando técnicas “duvidosas”.

Principais itens que poderão ser avaliados (além dos avaliados nos testes automáticos):
- Comentarios
- Indentacao
- Nomes adequados para variáveis
- Separacao do código em funções lógicas
- Uso correto de const/referencia
- Uso de variáveis globais apenas quando absolutamente necessário e justificável (uso de variáveis globais, em geral, é uma má prática de programação)
- Etc

Regras sobre plágio e trabalho em equipe

- Este trabalho deverá ser feito de forma individual.
- Porém, os alunos podem ler o roteiro e discutir ideias/algoritmos em alto nível de forma colaborativa.
- As implementações (nem mesmo pequenos trechos de código) não deverão ser compartilhadas entre alunos. Um estudante não deve olhar para o código de outra pessoa.
- Crie um arquivo README (submeta-o com o trabalho) e inclua todas as suas fontes de consulta.
- Não poste seu código (nem parte dele) no Piazza (ou outros sites) de forma pública (cada aluno é responsável por evitar que outros plagiem seu código).
- Trechos de código não devem ser copiados de livros/internet. Se você consultar algum livro ou material na internet essa fonte deverá ser citada no README.
- Se for detectado plágio em algum trecho de código do trabalho a nota de TODOS estudantes envolvidos será 0. Além disso, os estudantes poderão ser denunciados aos órgãos responsáveis por plágio da UFV.

---> Desenhe vários diagramas para entender o que está implementando <<----

Codigo fonte de teste.cpp

```
#include <iostream>
#include <cstdlib>
#include <sstream>
#include <cstdio>
#include "MyVec.h"
#include "Huffman.h"
using namespace std;


int main() {
	int freqs[256] = {0};
	freqs['A'] = 2;
	freqs['B'] = 8;
	freqs['C'] = 1;
	freqs['D'] = 1;
	HuffmanTree arvore(freqs);

	MyVec<char> in;
	in.push_back('A');
	in.push_back('B');
	in.push_back('B');
	in.push_back('C');
	//A frequencia nao precisa ser necessariamente a mesma dos
	//dados de entrada (nessa entrada a frequencia de B e'2, nao 8)
	//O ideal eh que a frequencia seja a mesma! (isso poderia gerar uma compressao)

	MyVec<bool> comprimido;
	arvore.comprimir(comprimido, in);
	for(int i=0;i<comprimido.size();i++)
		cout << comprimido[i];
	cout << endl;
	//deveria imprimir: 1000110 (supondo que a arvore gerada foi igual a apresentada no roteiro)
	MyVec<char> descomprimido;
	arvore.descomprimir(descomprimido, comprimido);
	for(int i=0;i<descomprimido.size();i++)
		cout << descomprimido[i];
	cout << endl;
```

```
        //deveria imprimir: ABBC
}
```