# MC202 — Estruturas de Dados

# Laboratório 2 - Pílulas de Nanicolina

Primeiro semestre de 2017 - Turmas B e C
Professor: Emilio Francesquini
francesquini@ic.unicamp.br

## 1 O problema

As famosas pílulas de Nanicolina[1] eram capazes de encolher o tamanho de um determinado herói colorado mexicano e mais tarde devolvê-lo ao seu tamanho original. Neste laboratório vamos tentar fazer o mesmo, mas desta vez com arquivos de computadores. Para isto vamos utilizar um algoritmo clássico de compactação de dados criado pelo David A. Huffman durante o seu doutorado em 1952[2]. No final, teremos um programa que será capaz de codificar (=*compactar*) e decodificar (=*descompactar*) arquivos de quaisquer tipos.

## 2 Codificação de Huffman

Tipicamente quando queremos que o envio de um arquivo via uma conexão de rede seja feito mais rapidamente ou quando queremos diminuir o espaço ocupado no disco nós utilizamos uma aplicação de compressão de arquivos. O código de Huffman é uma maneira clássica de compactar arquivos que é baseada em uma árvore binária.

Existem algumas maneiras de fazer a compressão de arquivos. A mais simples é utilizar uma tabela de tamanho fixo. Por exemplo, em um texto em português nós provavelmente não vamos utilizar mais do que 128 caracteres diferentes (52 letras, 10 números, sinais de pontuação, ...). Suponhamos que este seja o caso, ou seja, queremos compactar um arquivo de texto que usa no máximo 128 caracteres diferentes. Então poderíamos usar um código de tamanho fixo de 7 bits ($2^7 = 128$ possibilidades) para codificar todas as letras do texto. Economizaríamos 1 bit a cada 8, o que nos daria uma taxa

[1]Pílulas de Nanicolina, Polegarina ou apenas Pastilhas Encolhedoras (em espanhol *Pastillas de Chiquitolina*)

[2]D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes,"in Proceedings of the IRE, vol. 40, no. 9, pp. 1098-1101, Sept. 1952.

de compressão de $\sim 7/8 * tamanhoOriginal$, ou aproximadamente 87% para arquivos grandes[3].

13% de redução de tamanho é muito pouco, podemos fazer melhor. A Codificação de Huffman nada mais é do que uma maneira mais inteligente de se criar uma tabela para codificação. Para isto, em lugar de usar códigos de tamanho fixo, ela utiliza códigos de tamanho variável usando a frequência que aquele caractere aparece no texto. Caracteres que tem uma maior frequência recebem códigos mais curtos do que caracteres que aparecem raramente. Usando este princípio é possível diminuir consideravelmente o tamanho de arquivos codificados, principalmente de texto, pois é esperado que algumas letras sejam muito mais frequentes do que outras. A ideia é a de que o número de bits economizados para codificar os caracteres mais usados sejam mais do que o suficiente para cobrir o deficit que ocorre ao codificarmos os caracteres menos comuns.

O algoritmo de Huffman que deverá ser implementado será composto de três fases:

1. Na primeira fase a frequência de cada um dos caracteres que ocorre no texto deverá ser calculada[4].

2. A segunda fase do algoritmo consiste em construir a árvore de Huffman (veja a Seção 3). A árvore de Huffman é uma árvore binária que é construída baseando-se na frequência (calculada na fase anterior) do uso de cada um dos caracteres no arquivo sendo analisado. Em cada passo desta fase teremos uma coleção de árvores (ou seja, uma floresta de árvores binárias). As folhas de cada uma destas árvores correspondem a um conjunto de caracteres que ocorrem no texto. À raiz de cada uma dessas árvores será associado um número que corresponde à frequência com que os caracteres associados às folhas desta árvore ocorrem no texto. Escolheremos duas árvores desta floresta com a a menor frequência e as transformaremos em uma única árvore, ligando-as a uma nova raiz criada para este fim e cujo valor será dado pela soma dos valores das frequências armazenadas nas raízes de cada uma das duas subárvores.

---

[3]Na verdade como precisamos guardar a tabela de símbolos também, a taxa vai ser um pouco pior. Contudo como a tabela é de tamanho fixo independentemente do tamanho do arquivo, a taxa de compressão tende a ~87% conforme o arquivo cresce.

[4]Note que falamos em *texto* e em *caracteres*, mas efetivamente estamos nos referindo ao conteúdo de um *arquivo* e os seus *bytes* que não necessariamente são um texto legível já que podem ser uma foto, um vídeo, uma música...

3. Finalmente, na terceira fase, a árvore de Huffman será usada para codificar o texto, ou seja, compactar o arquivo. Veja as Seções 4, 5 e 6 para mais informações de como isto deverá ser feito. Note que a mesma árvore usada para codificar um arquivo deverá ser usada para decodificá-lo, mas como depois de compactado não temos mais acesso ao arquivo original para recalcular a árvore, é preciso armazenar a árvore no próprio arquivo codificado.

Para entender em mais detalhes como uma árvore de Huffman funciona **leia** as excelentes páginas escritas pelo Prof. Paulo Feofiloff do IME/USP:

**Compressão de dados**:
`https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/compress.html`
**Codificação de Huffman**
`https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html`

# 3 Árvore de Huffman - Exemplo

Suponha que o arquivo a ser compactado seja composto pelos caracteres A, B, C, D, E, R e que a frequência de cada um deles foi calculada e resultou na tabela abaixo:

| A | B | C | D | E | R |
|---|---|---|---|---|---|
| 22 | 9 | 10 | 12 | 16 | 8 |

O algoritmo para a construção da árvore de Huffman funciona utilizando a tabela de frequências para fazer com que as letras com menor frequência B, R estejam mais distantes da raiz enquanto as letras mais comuns A, E estejam mais próximas.

A construção da árvore é feita de baixo para cima, ou seja, começamos construindo as folhas e vamos trabalhando até chegar à raiz. Nesta árvore, as letras estarão **exclusivamente** nas folhas. Os nós intermediários conterão apenas a soma das frequências das letras contidas nas suas subárvores esquerda e direita.

Vejamos como a árvore é construída para o exemplo acima. Primeiramente criamos um nó para cada uma das letras e suas frequências. Começamos portanto com n árvores onde n é o número de letras distintas no texto (veja Figura 1).

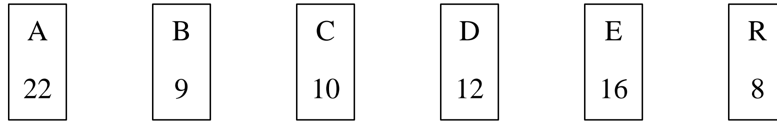



Figura 1: Floresta no início do algoritmo.

Em seguida, escolhemos as duas árvores com as menores frequências (árvores R e B com frequências de 8 e 9 respectivamente) e as juntamos, usando um novo nó criado justamente com esta finalidade. O valor associado a este nó é a soma das frequências das duas subárvores e, como ele não é uma folha, ele não possui um caractere associado a ele (Figura 2).

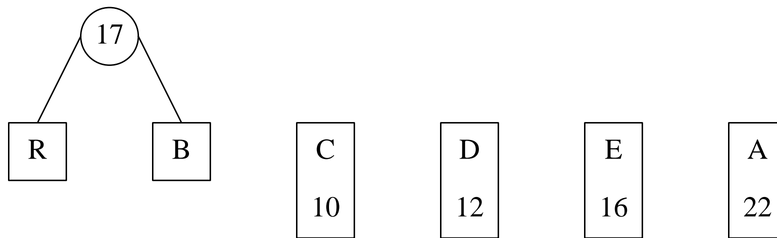



Figura 2: Floresta após termos juntado as árvores com frequências 8 e 9

Repita o processo. Desta vez escolha as árvores com raízes 10 e 12, e junte-as. A coleção de árvores tem a forma mostrada na Figura 3:

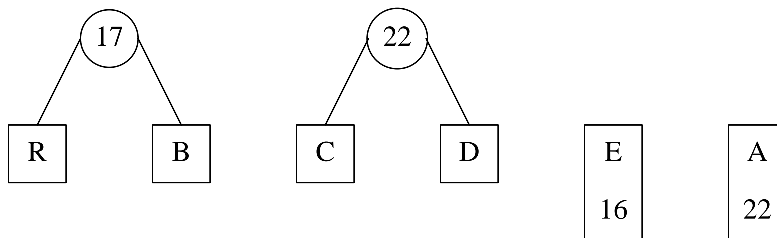



Figura 3: Floresta após termos juntado as árvores com frequências 10 e 12, formando a árvore com frequência associada 22.

Continuando o mesmo processo, escolha as árvores frequências 16 e 17, formando a floresta que pode ser vista na Figura 4.

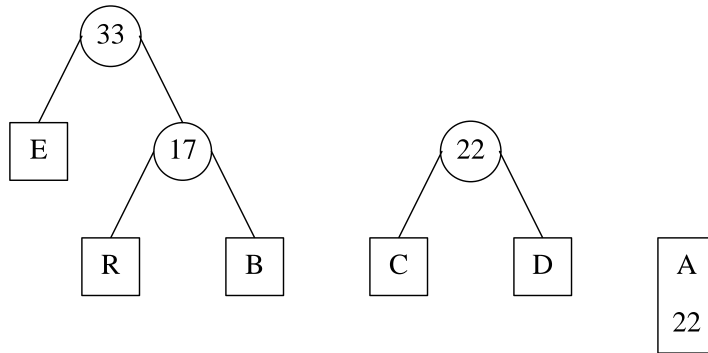



Figura 4: Floresta após termos juntado as árvores com frequências 16 e 17.

Depois, juntamos as árvores com frequência 22 e 22 formando a floresta exibida na Figura 5.

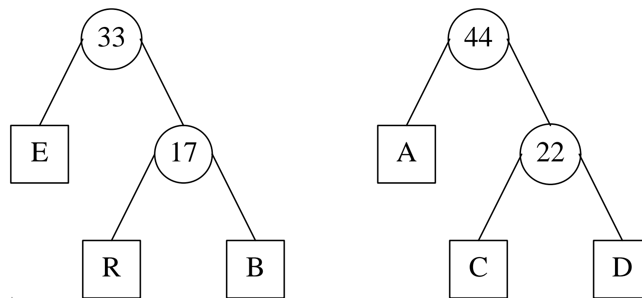



Figura 5: Floresta após termos juntado as árvores com frequências 22 e 22.

E, finalmente juntamos as duas árvores restantes para obtermos a árvore mostrada na Figura 6.

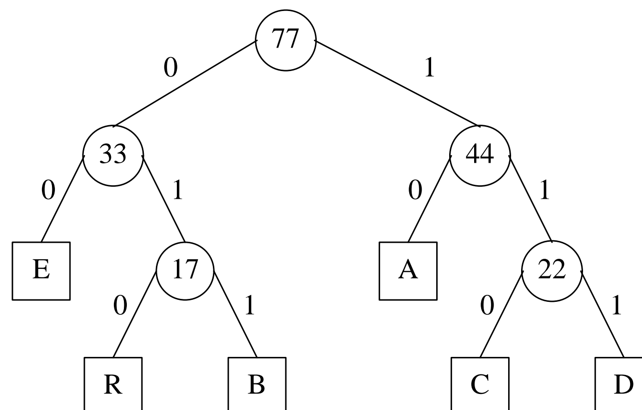



Figura 6: Árvore de Huffman

Assim, a árvore de Huffman está completamente construída.

# 4 Codificação

Associe 0 às arestas da árvore de Huffman que ligam um nó com o seu filho esquerdo e 1, às arestas que ligam um nó com seu filho direito. O código correspondente a cada letra será formado pelo número binário associado ao caminho da raiz até a folha correspondente. Com isso, a tabela de códigos resultante da árvore de Huffman da Figura 6 é:

| A | B | C | D | E | R |
|---|---|---|---|---|---|
| 10 | 011 | 110 | 111 | 00 | 010 |

Repare que em casos muito específicos é possível construir uma árvore onde a codificação de caracteres diminui (menos de 8 bits) enquanto outros podem chegar até 256 bits.

# 5 Decodificação

Para decodificar, com a tabela acima, é bem fácil. Por exemplo, a seguinte sequência:

1 0 0 1 1 0 1 0 1 0 1 1 0 1 0 1 1 1 1 0 0 1 1 0 1 0 1 0

Corresponde ao texto original: ABRACADABRA. Como isso é feito?

Para decodificar basta ir utilizando cada bit e percorrendo a árvore de Huffman desde a raiz até chegar em uma folha. A folha vai, portanto, conter um caractere que deve ser utilizado para recompor o texto original. Repita o processo desde a raiz até que todos os bits da mensagem codificada tenham sido lidos. Observe que a mensagem original ”ABRACADABRA”tem 11 caracteres (88 bits) e foi compactada para apenas 28 bits (taxa de compressão de 31%!).

# 6 Sobre a implementação

O seu programa deverá saber lidar tanto com arquivos texto quanto com arquivos binários. Ele deverá receber 3 parâmetros:

- *c* ou *d* – Indica se a operação desejada é para codificar (compactar) ou decodificar (descompactar) um arquivo

- *nomeDoArquivoDeOrigem* – Nome do arquivo a ser compactado ou descompactado
- *nomeDoArquivoDeDestino* – Nome do arquivo criado como resultado da opção escolhida aplicada ao *nomeDoArquivoDeOrigem*

Para a criação da árvore de Huffman **deverá ser utilizado obrigatoriamente** um Heap baseado em um vetor.

## 6.1 Formato do arquivo codificado

O formato do arquivo codificado será o seguinte:

- 4 bytes para guardar o tamanho do arquivo original (o que significa que o tamanho máximo dos arquivos que podemos compactar é de 4GB).
- Árvore de Huffman codificada (veja subseção a seguir).
- Arquivo codificado seguindo a árvore acima.
- Filler – como o arquivo codificado não terá necessariamente um número múltiplo de 8 de bits, o último byte poderá ter que ser completado. Para completá-lo podem ser necessários de 1 a 7 bits. O filler deve ser obrigatoriamente composto de APENAS bits com valor 0.

| Tamanho do original | Árvore de Huffman | Arquivo Codificado | Filler |
|---|---|---|---|
| Fixo: 4 Bytes | Variável | Variável | De 1 a 7 bits |

## 6.2 Codificando a árvore de Huffman

Para codificar a árvore de Huffman no arquivo percorra a árvore em pré-ordem (visite a raiz, depois a subárvore esquerda, depois a subárvore direita). Quando visitar um nó interno, escreva um bit 0. Quando visitar uma folha, escreva um bit 1 seguido pelos 8 bits do caractere associado a esta folha. Esse representação da árvore é fácil produzir e fácil decodificar. Veja: `https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html`.

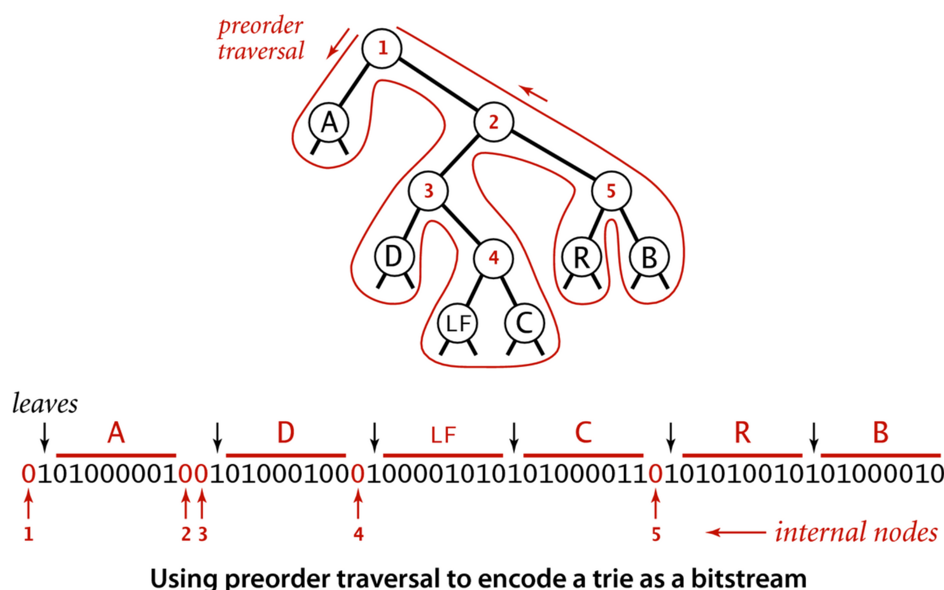


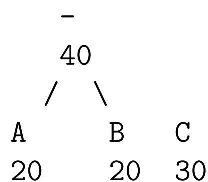

Figura 7: Figura retirada do livro do Sedgewick (veja bibliografia na página do curso).

## 6.3 Dicas

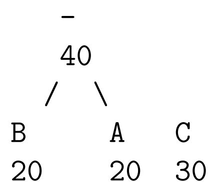
- Tente começar com arquivos pequenos, com apenas algumas letras ou uma palavra antes de tentar compactar arquivos grandes.
- Crie 3 fases bem distintas no seu código. Primeiro faça o código de cálculo das frequências. Escreva o tamanho do arquivo original no arquivo de saída. Só depois crie a árvore de Huffman e escreva-a codificada no arquivo. E finalmente comece a fazer a codificação do arquivo compactado.
- Durante a criação da árvore de Huffman, juntamos dois nós com as frequências mais baixas criando uma nova raiz e colocando um dos nós como filho direito e o outro como filho esquerdo. Não há uma regra que define qual dos nós deve ser colocado como direito e qual como esquerdo. Então, por exemplo, se temos a seguinte situação:

  ```
  A   B   C
  20  20  30
  ```

  Após a junção podemos ter:



ou



Ambas as árvores estão CORRETAS. O fato de existirem diversas árvores possíveis faz com que o arquivo compactado não seja exatamente igual para cada uma dessas árvores. Isto não faz diferença nenhuma para o algoritmo. A árvore gerada vai ser realmente diferente, mas continuará tendo as mesmas características. Utilize o compactador/descompactador fornecido como base para construir o seu. Faça o comportamento do seu código ficar **compatível** com código fornecido. Em outras palavras:

- Diversas árvores CORRETAS existem para a mesma tabela de frequências.
- Independentemente de como você montar a sua árvore, o tamanho do arquivo compactado gerado será EXATAMENTE IGUAL.
- Arquivos CORRETOS, independentemente da árvore, devem ser compatíveis entre o seu código e o código fornecido:
  - Arquivos compactados com o seu código DEVEM ser descompactáveis com o código fornecido
  - Arquivos compactados com o código fornecido DEVEM ser descompactáveis com o seu código

- Para ver o conteúdo binário de um arquivo utilize o comando `xxd` no Linux. Exemplo:

```
$ xxd -b arquivo
00000000: 00000001 00000000 00000000 00000000 00001101 00000001
......
```

- Para comparar dois arquivos binários sugiro fazer o seguinte:

```
$ xxd -b arquivo1 >arquivo1.bin
$ xxd -b arquivo2 >arquivo2.bin
$ diff arquivo1.bin arquivo2.bin
```

- Para escrever e ler um inteiro de 4 bytes com o tamanho no início do arquivo codificado utilize as funções `fread` e `fwrite` e o tipo `uint32_t`:

```
#include <inttypes.h>
…
//Escrevendo
uint32_t val = 12345;
//note o modo de abertura binário "b"
FILE *file1 = fopen("nomeArquivoEscrita", "wb");
fwrite(&val, sizeof(uint32_t), 1, file1);
//Lendo
uint32_t val2;
FILE *file2 = fopen("nomeArquivoLeitura", "rb");
fread(&val2, sizeof(uint32_t), 1, file2);
```

- Utilize o arquivo `bitstream.h` fornecido na sua implementação para facilitar a manipulação de bits. Para usá-lo, existem dois tipos principais, o `bitstreamoutput` usado para a escrita e o `bitstreaminput` usado para leitura. Para criá-los utilize as funções `criaEncodedOutput` e `criaEncodedInput`. A função `destroiEncodedOutput` já cria o *filler* do seu arquivo automaticamente. Exemplo de uso (tamanho do arquivo e os primeiros bits da árvore da Figura 7):

```
FILE *saida = fopen("nomeArquivo", "wb");
uint32_t tamanho = 12345;
fwrite(&tamanho, sizeof(uint32_t), 1, saida);
bitstreamoutput *bso = criaEncodedOutput(saida);
bitstream bs;

bsClean(&bs);
//existe também uma versão da função
//bsPushInPlace chamada bsPush que devolve
//uma cópia do bs, em vez de modificá-lo
bsPushInPlace(0, &bs);
bsPushInPlace(1, &bs);
```

```
encodedOutputWrite(bso, &bs);
encodedOutputRawWrite(bso, 'A');
bsClean(&bs);
bsPushInPlace(0, &bs);
bsPushInPlace(0, &bs);
bsPushInPlace(1, &bs);
encodedOutputWrite(bso, &bs);
encodedOutputRawWrite(bso, 'D');

destroiEncodedOutput(bso);
fclose(saida);

FILE *entrada = fopen("nomeArquivo", "rb");
uint32_t tamanho2;
int itensLidos = fread(&tamanho2, sizeof(uint32_t), 1, entrada);
assert(itensLidos == 1);
bitstreaminput *bsi = criaEncodedInput(entrada);
//sendo o mesmo arquivo acima, vale 0
int bit = bsiPop(bsi);
bit = bsiPop(bsi); //1
byte val = bsiPopByte(bsi);//A
bit = bsiPop(bsi); //0
bit = bsiPop(bsi); //0
bit = bsiPop(bsi); //1
val = bsiPopByte(bsi);//D

destroiEncodedInput(bsi);
fclose(entrada);
```

# 7 Entrega e avaliação

Este laboratório poderá ser feito em duplas. A entrega deverá ser feita pelo Susy **apenas** por um dos integrantes da dupla. O arquivo enviado deve conter um cabeçalho contendo o nome completo e RA de cada um. Verifique a página da disciplina (`http://www.ic.unicamp.br/~francesquini/mc202/`) para informações sobre as datas de entrega e acesso ao Susy.

Todos os trabalhos entregues passarão por uma correção automática no Susy. Não passar nos testes do Susy indica nota 0 (zero), no entanto passar pelo sistema de correção automática não indica que a nota será maior que

zero.

**ATENÇÃO:** Plágios serão severamente punidos com a reprovação na disciplina.

Bom trabalho!