

## A Codificação de Huffman

A codificação de Huffman é um método de compactação que usa as probabilidades de ocorrência dos símbolos no conjunto de dados a ser compactado para determinar códigos de tamanho variável para cada símbolo.

### ASCII e a codificação

ASCII (*American Standard Code for Information Interchange*) é um padrão de codificação de caracteres usado por muitas languages de programação. Neste padrão, cada caracter é codificado com o mesmo número de bits por caracter (e.x., 8 bits). Desta maneira, há 256 ( $2^8$ ) possíveis combinações para representar os caracteres em ASCII. Os caracteres mais comuns, como os alfanuméricos, pontuação e caracteres de controle usam apenas 7 bits. 128 ( $2^7$ ) caracteres diferentes podem ser codificados com 7 bits. A codificação de Huffman compacta os dados usando um número menor de bits para codificar caracteres que ocorrem mais frequentemente de maneira que nem todos os caracteres precisem ser codificados com 8 bits.

Considere a string “*bom esse bombom*”. Usando a codificação ASCII (8 bits por character), os 16 caracteres dessa string usam 128 bits. A tabela a seguir ilustra como a codificação funciona.

Char	ASCII	Binário
b	98	0110 0010
o	111	0110 1111
m	109	0110 1101
e	101	0110 0101
s	115	0111 0011
Espaço	32	0010 0000

A string “bom esse bombom” seria escrita numericamente assim: 98 111 109 32 101 115 115 101 32 98 111 109 98 111 109. Em binário, seria assim: 0110 0010 0110 1111 0110 1101 0010 0000 0110 0101 0111 0011 0111 0011 0110 0101 0010 0000 0110 0010 0110 1111 0110 1101 0110 0010 0110 1111 0110 1101.

Considere agora que estamos utilizando uma codificação baseada em 3-bits por character:

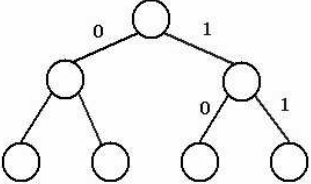
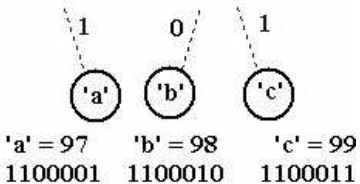
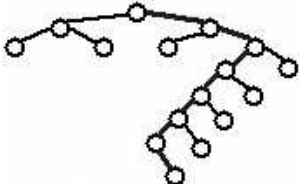
Char		Binário
b	0	000
o	1	001
m	2	010
e	3	011
s	4	100
Espaço	5	101

A string seria numericamente escrita assim: 0 1 2 5 3 4 4 3 5 0 1 2 0 1 2 e em binário: 000 001 010 101 011 100 100 011 101 000 001 010 000 001 010.

Usando 3-bits por character, a string “bom esse bombom” usa um total de 48 bits ao invés de 128. A “economia” de bits poderia ser ainda maior se usarmos menos de 3 bits para codificar caracteres que aparecem mais vezes (b, o, m) e mais bits para caracteres que aparecem menos vezes (e, s). Essa é a idéia básica da codificação de Huffman: usar menor número de bits para representar caracteres com maior frequência. Essa técnica pode ser implementada usando uma árvore binária que armazena caracteres nas folhas, e cujos caminhos da raiz até as folhas provêm a sequência de bits que são usados para codificar os respectivos caracteres.

## Árvore de Codificação

Usando uma árvore binária, podemos representar todos os caracteres ASCII como folhas de uma árvore completa (ver Figuras 1 e 2). Na Figura 3, a árvore tem 8 níveis, o que significa que o caminho da raiz à folhas sempre tem 7 arcos. Arcos da esquerda são numerados com 0 enquanto arcos da direita com 1. O código ASCII para qualquer caracter/folha é obtido seguindo o caminho da raiz à folha e concatenando os 0's e 1's. Por exemplo "a", que tem código ASCII 97 (1100001 em binário) é representando na árvore pelo caminho: direita-direita-esquerda-esquerda-esquerda-direita (Figura 3).

		
Figure 1 - Exemplo de árvore de codificação binária	Figure 2 - Nós folhas de uma árvore binária representando caracteres ASCII	Figure 3 - Caminho de codificação ASCII para o caracter "a"

A estrutura de uma árvore pode ser usada para determinar o código de qualquer folha apenas usando a convenção 0/1 atribuída aos arcos, como descrito. Se usarmos uma árvore diferente, nós teremos uma codificação diferente.

Podemos usar esse tipo de codificação para compactação de arquivos. Para maximizar a compactação, precisamos encontrar uma árvore *ótima*, que apresente um número mínimo de codificação por caracter. O algoritmo para encontrar essa *árvore ótima* é chamado de codificação de Huffman, e foi inventando por D. Huffman em 1952.

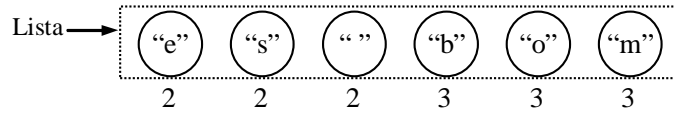
### O algoritmo de Huffman

Assuma que cada caracter em um texto está associado a um peso, que é definido pelo número de vezes que o caracter aparece em um arquivo. Na string "bom esse bombom", os caracteres "b", "o" e "m" têm peso 3, enquanto os caracteres "e", "s" e espaço têm peso 2. Para usar o algoritmo de Huffman, é necessário calcular esses pesos (ver Dica 1).

O algoritmo de Huffman assume que uma árvore será construída a partir de um grupo de árvores. Inicialmente essas árvores têm um único nó com um caracter e o peso deste caracter. À cada iteração do algoritmo, duas árvores são juntadas criando uma nova árvore. Isso faz com que o número de árvores diminua a cada passo. Este é o algoritmo:

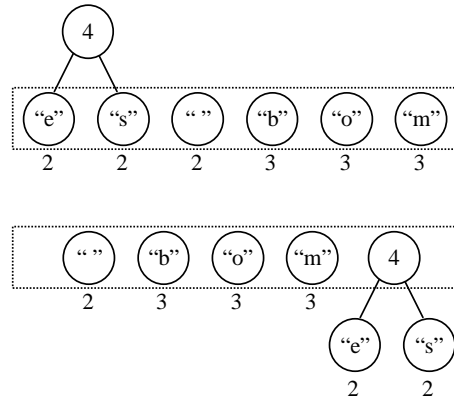
1. Comece com uma lista de árvores. Inicialmente, todas as árvores são compostas de um nó apenas, com o peso da árvore igual ao peso do caracter do nó. Caracteres que ocorrerem mais frequentemente têm o maior peso. Caracteres que ocorrerem menos frequentemente têm o menor peso. Ordene a lista de árvores de forma crescente, fazendo com que o nó raiz da primeira árvore seja o caracter de menor peso e o nó raiz da última árvore seja o caracter de maior peso.
2. Repita os passos a seguir até que sobre apenas uma única árvore:
  - o Pegue as duas primeiras árvores da lista e as chame de T1 e T2. Crie uma nova árvore Tr cuja raiz tenha o peso igual à soma dos pesos de T1 e T2 e cuja subárvore esquerda seja T1 e subárvore direita seja T2.
  - o Exclua T1 e T2 da lista (mantendo T1 e T2 na memória) e inclua Tr na lista, de maneira que a lista seja mantida ordenada.
3. A árvore final será a árvore ótima de codificação.

A Figura 4 mostra a fase inicial do algoritmo para a string "bom esse bombom". Os nós são mostrados com o peso que representa o número de vezes o respectivo caracter aparece na string. Note que a lista está ordenada segundo os pesos dos caracteres.



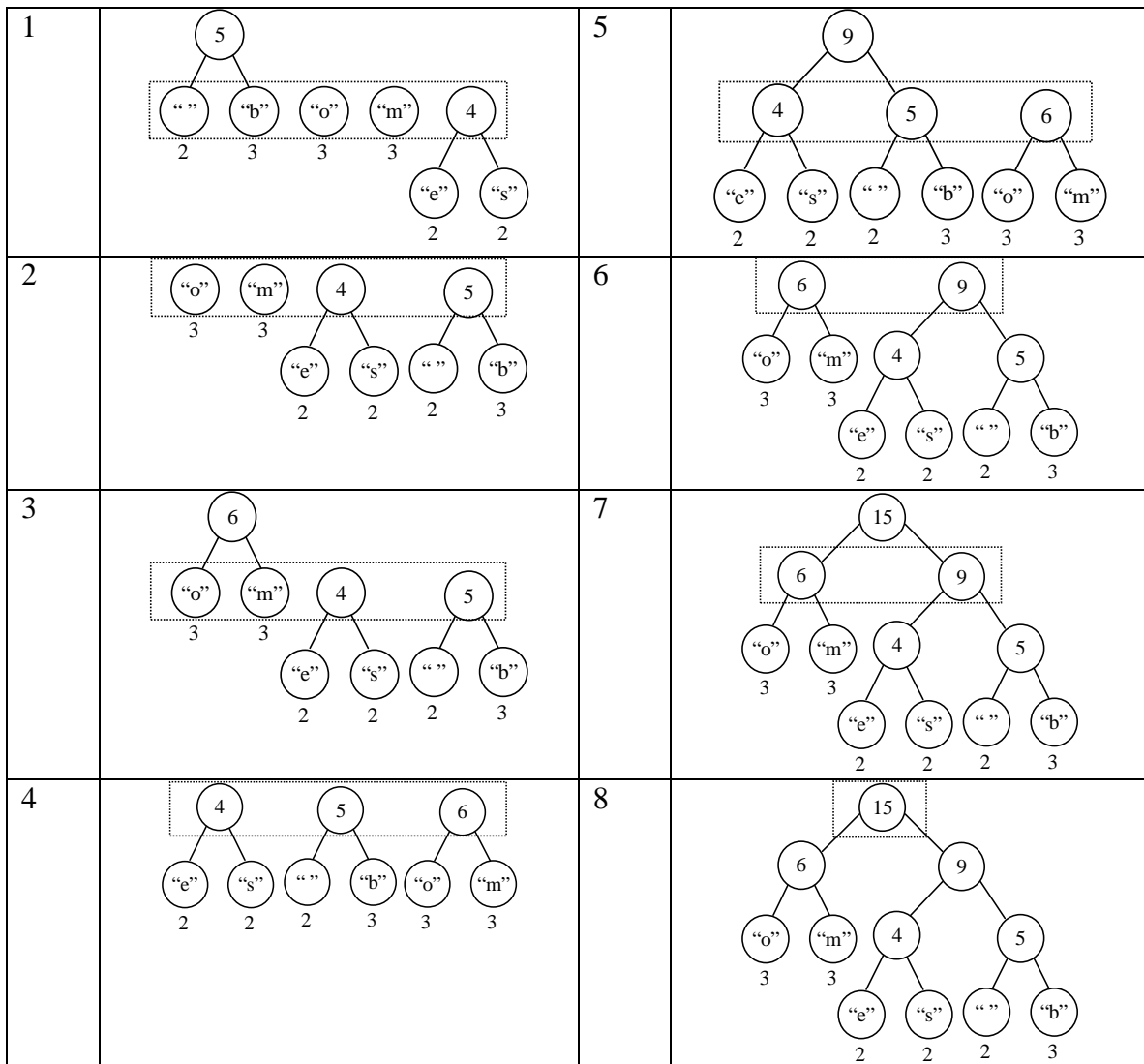
**Figure 4 - Etapa inicial do algoritmo de Huffman (após passo 1)**

Os dois primeiros nós são juntados para formar uma nova árvore, cujo peso do nó raiz é a soma dos nós dos pesos dos nós filhos (Figura 5). Seguindo o algoritmo, os dois primeiros nós são retirados da lista e o nó raiz da árvore criada é inserido ordenadamente na lista de árvores. Figura 5 mostra a nova árvore sendo inserida no final da lista (raiz com o peso 4).



**Figure 5 - Após passo 2 do algoritmo (pela primeira vez)**

Repete-se os passos acima até que sobre apenas uma árvore. A tabela de figuras a seguir demonstra o resto da execução do algoritmo passo-a-passo.



A codificação de caracteres induzida pela última árvore (árvore 8) é a seguinte (considerando 0 para arcos da esquerda e 1 para arcos da direita):

Char	Binário
b	111
o	00
m	01
e	100
s	101
Espaço	110

Usando essa tabela de codificação, a string "bom esse bombom" ficaria assim:

111 00 01 110 100 101 101 100 110 111 00 01 111 00 01

Desta forma usamos 39 bits para codificar a string “bom esse bombom”, ao invés de 48 bits como mostramos usando a codificação de 3-bits.

## **Implementação da Codificação de Huffman**

Há duas partes distintas na implementação: um programa que faz a compactação e um programa que faz a descompactação. Chamaremos o programa que lê de um arquivo e produz um arquivo compactado de *Compactador* e o programa que produz um arquivo normal a partir de um arquivo descompactado de *descompactador*.

Para compactar um arquivo, é necessário a tabela de compactação, como mostramos para o exemplo “bom esse bombom”. Como vimos, esta tabela é construída com uma árvore binária de compactação.

Assumindo que um número fixo de bits é escrito em um arquivo, um arquivo compactado é criado seguindo os seguintes passos:

1. Construa a tabela de codificação;
2. Leia o arquivo a ser compactado e processe um caracter de cada vez (veja dica 1). Para processar o caracter e chegar à sequência de bits que representa o caracter, use a tabela de caracteres que foi criada no passo anterior. Escreva essa sequência de bits no arquivo compactado. Por exemplo, para compactar a string “bom esse bombom”, a sequência 111000111010010110110011011100011110001 seria escrita no arquivo compactado. Veja Dica 2 para manipulação de código binário.

O arquivo compactado deve conter as informações necessárias para se chegar corretamente ao arquivo original a partir do arquivo compactado. As seguintes informações são necessárias: um cabeçalho no arquivo compactado que deve conter a árvore de codificação; e alguma marca de final de arquivo para indicar que a sequência de bits chegou ao fim. O programa descompactador deve criar a árvore de codificação que está no cabeçalho do arquivo, ler os bits e navegar na árvore até encontrar os nós folhas correspondentes àquela sequência de bits lidos. O custo deste algoritmo de busca é proporcional à altura da árvore.

## **Cabeçalho do arquivo compactado**

A árvore de codificação pode ser armazenada no início do arquivo compactado. Existem várias maneiras de armazenar a árvore. Para o trabalho, use a travessia de pré-ordem para escrever cada nó visitado pela travessia. Nós folha devem ser diferenciados de nós não-folhas (nós íternos). Uma maneira de fazer a diferenciação é escrever um único bit para cada nó, por exemplo 1 para nós folhas e 0 para nós não-folha. Para nós-folha, é também necessário escrever o caracter armazenado. Para nós não-folha é necessário apenas o bit indicando de que se trata de um nó não-folha.

## **Caracter de final da sequência de bits**

O sistema operacional não escreve dados bit a bit e sim em “pedaços” maiores (geralmente pedaços de tamanho múltiplo da arquitetura específica do SO). Desta maneira, é muito comum o sistema operacional usar um mecanismo de buffer para acumular os dados escritos e gravá-los de uma vez. Se o seu programa escreve 3 bits e depois 2 bits, todos os bits são em algum momento escritos, mas não é possível ter certeza de quando eles serão escritos. Imagine que o sistema operacional escreva em pedaços múltiplos de 8 bits, e o seu programa tente escrever 61 bits. Nesta situação, o sistema operacional completará os 61 bits com 3 bits para completar os 64 bits (múltiplo de 8). O programa descompactador deve criar algum mecanismo para desconsiderar estes bits “extra”, uma vez que eles não representam nenhuma informação.

Uma solução possível poderia introduzir um pseudo-caracter de final de arquivo, introduzido explicitamente pelo programador ao final da sequencia de bits válidos. Este caracter também deve entrar na árvore/tabela de codificação para que seja corretamente descompactado pelo programa descompactador. Quando um arquivo compactado é escrito, os últimos bits escritos devem ser os bits que representam o pseudo-caracter.

**Dica 1:**

Para calcular a frequencia com a qual os caracteres aparecem no arquivo, use um vetor de inteiros no qual o índice do vetor é o código ASCII do caracter, e o conteúdo do vetor é o número de vezes esse caracter é encontrado no arquivo. Desta forma, é necessário apenas ler o caracter do arquivo e fazer `V[códigoASCII_caracter_lido]++` (em C por exemplo).