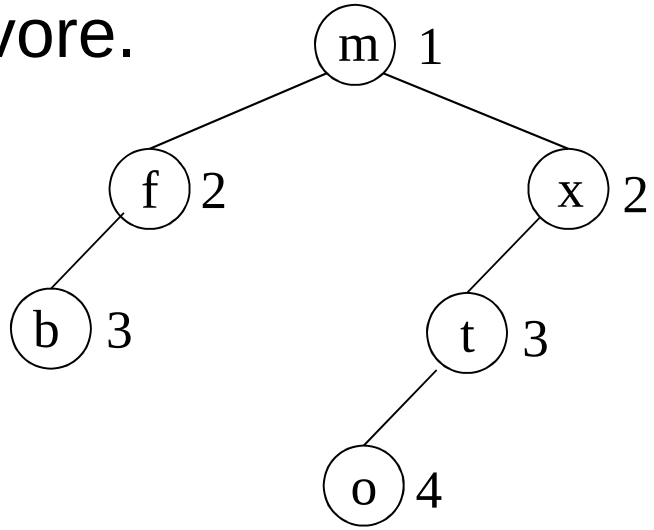
Árvores Binárias de Busca 2

Ivandré Paraboni

Custo de busca bem-sucedida

- *Comprimento do caminho interno* (*CI*): número total de comparações efetuadas para localizar cada chave da árvore.

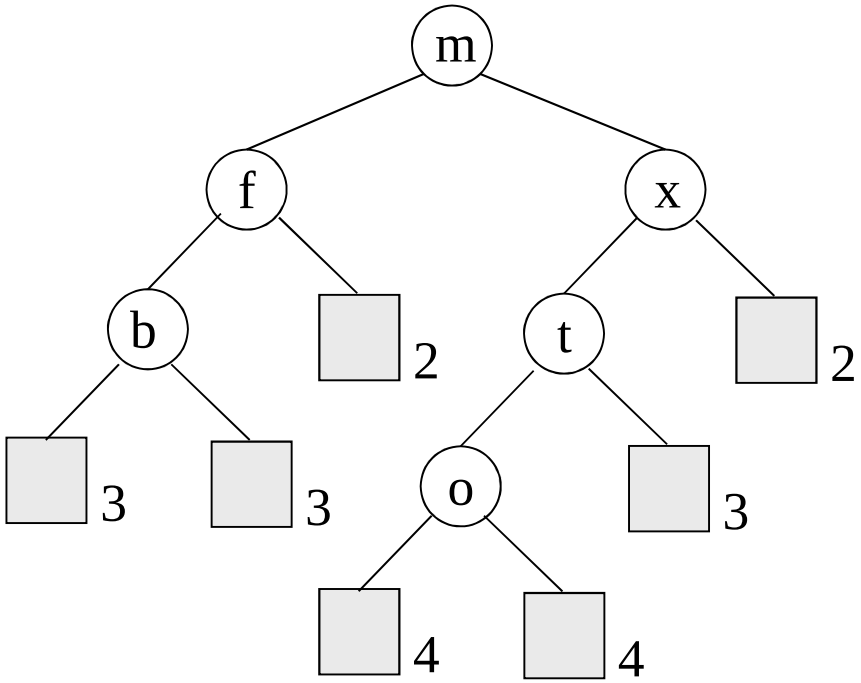


Número de comparações necessárias = nível *k* do nó

$CI = 1+2+2+3+3+4 = 15$

Custo de busca fracassada

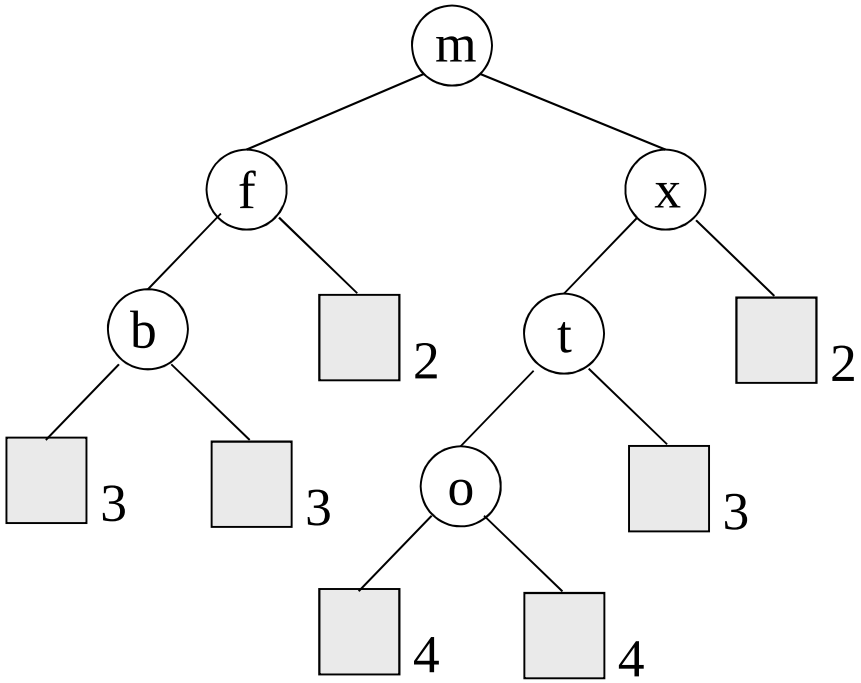
- Igualmente importante é o *comprimento do caminho externo* (*CE*), que é número de comparações necessárias para estabelecer que uma chave não existe



Número de comparações necessárias: nível *k*-1

Custo de busca fracassada

- Exemplo:



Número de comparações necessárias: nível *k*-1

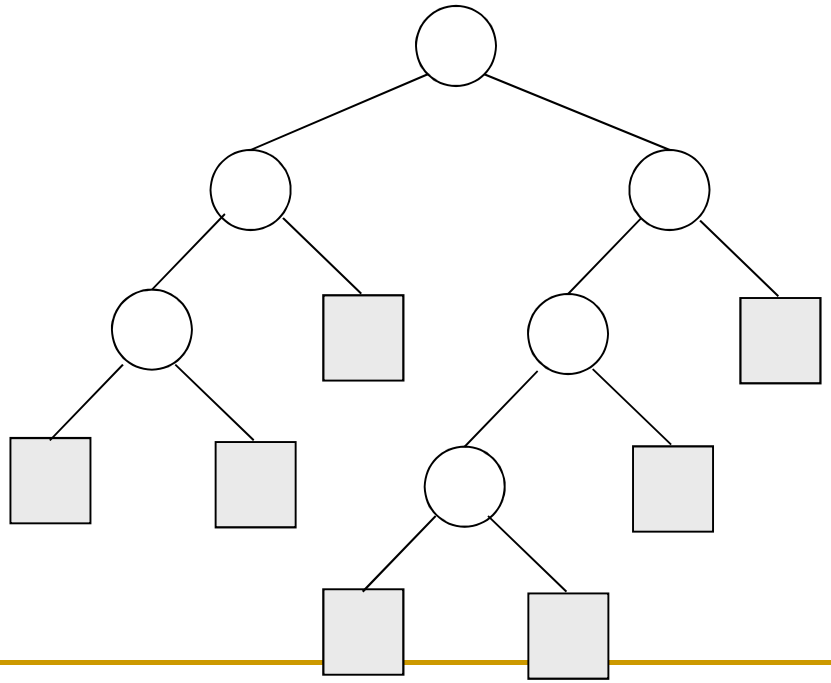
$CE = 2+2+3+3+3+4+4 = 21$

Caminho interno e externo

- Os comprimentos de caminho interno (CI) e externo (CE) são indicativos da qualidade da árvore para o problema de busca
- Número médio de comparações necessárias
  - para sucesso: CI / n
  - para fracasso: CE / n + 1
- A árvore completa minimiza CI e CT
- CI e CE não são independentes: CE = CI + n

Freqüência de acesso diferenciado

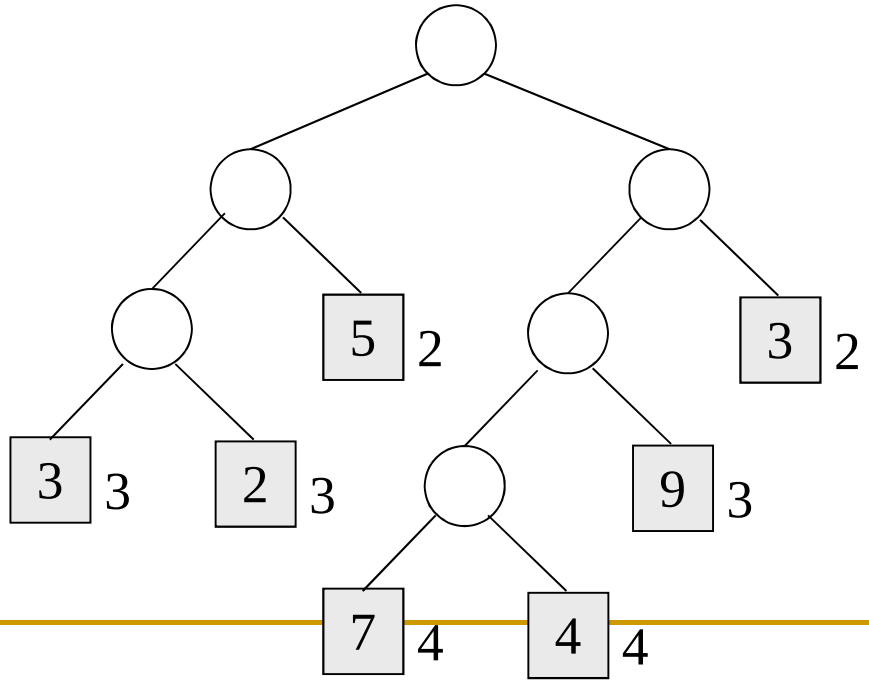
- A árvore completa tem desempenho ótimo se as freqüências de acesso aos nós são idênticas
- Na prática, porém, a freqüência de acesso pode ser desigual



Comprimento de caminho externo ponderado

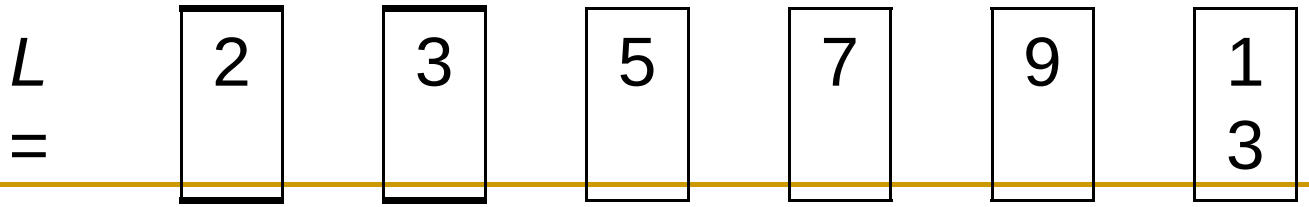
- É a soma dos caminhos externos associados às suas probabilidades

Ex.: (5*2)+(3*2)+(3*3)+(2*3)+(9*3)+(7*4)+(4*4)=102



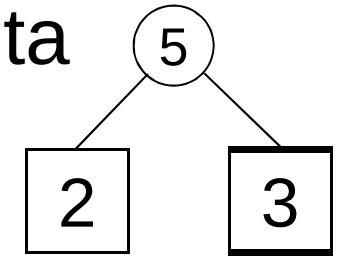
Árvore de caminho externo ponderado mínimo

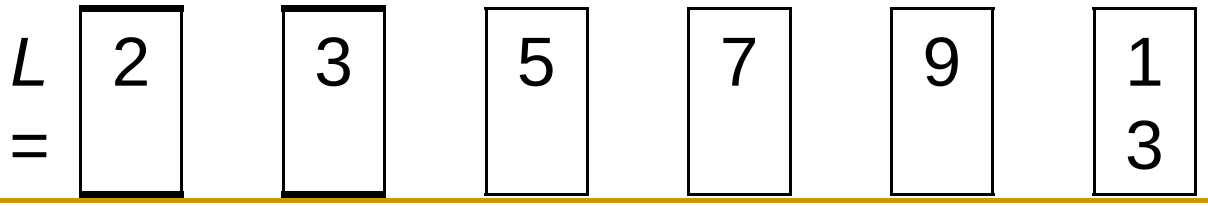
- Algoritmo de Huffman – entrada:
  - Seja uma lista de n pesos $p_1 ... p_n$ fornecidos;
  - Seja uma lista L de árvores binárias expandidas;
  - Cada nó possui três campos: esq, dir e peso.
  - Inicialmente, todas árvores em L possuem apenas um nó, o qual é externo e possui peso pi.
  - Exemplo (n=6): $p_1=2$, $p_2=3$, $p_3=5$, $p_4=7$, $p_5=9$, $p_6=13$



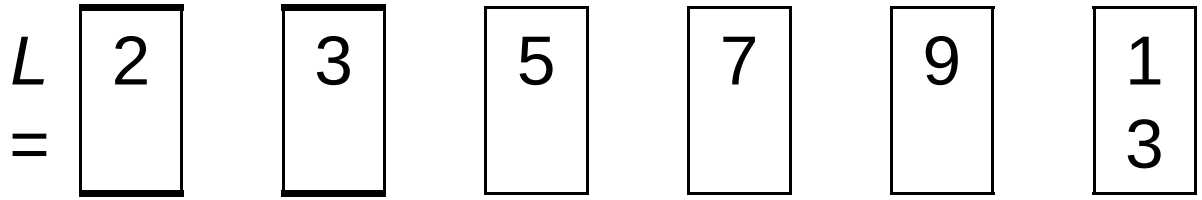
Árvore de caminho externo ponderado mínimo

- A função *menor(L)* retorna a árvore de menor peso em L
- O peso de uma árvore de altura h > 1 é a soma dos pesos de suas sub-árvores esquerda e direita



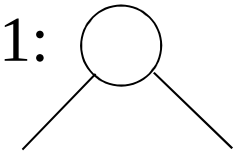


Algoritmo de Huffman



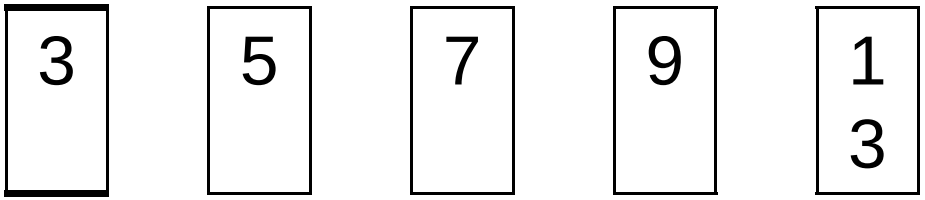

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

Árvore $i = 1$:



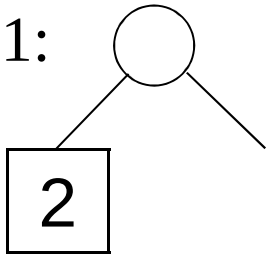
Algoritmo de Huffman

L =



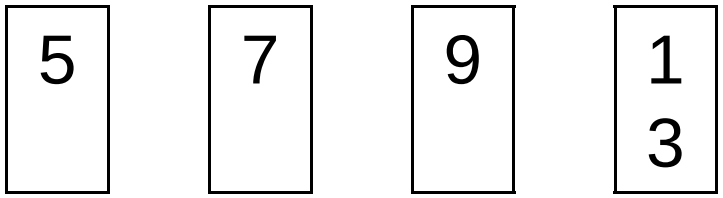

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

Árvore *i*=1:



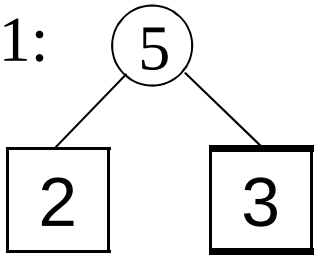
Algoritmo de Huffman

L =



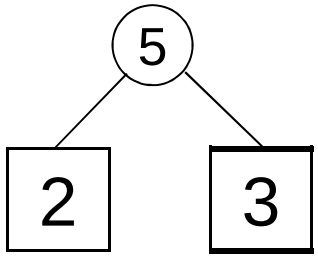

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

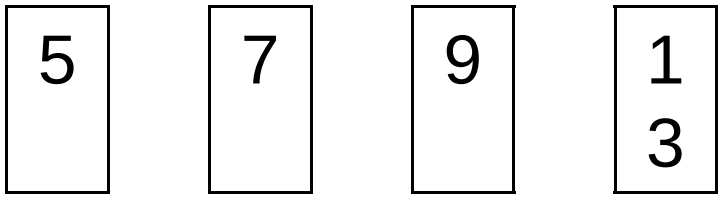
Árvore *i*=1:



Algoritmo de Huffman

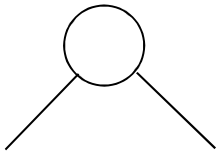
L =





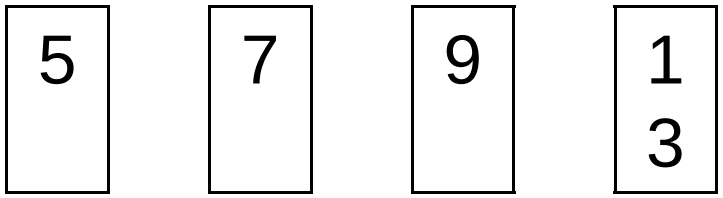

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

Árvore *i*=2:



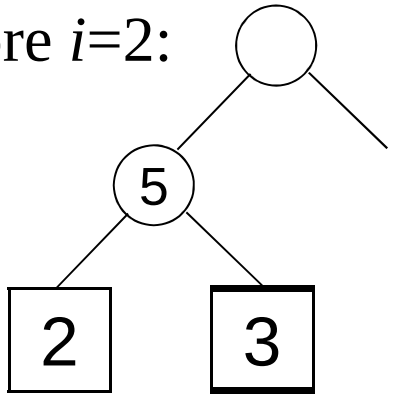
Algoritmo de Huffman

L =



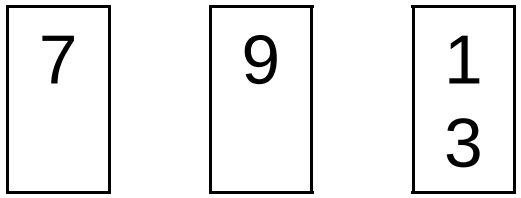

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

Árvore *i*=2:



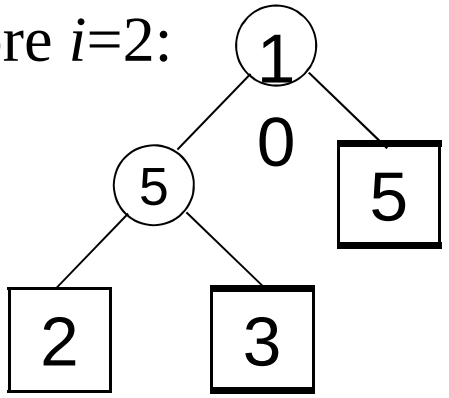
Algoritmo de Huffman

L =



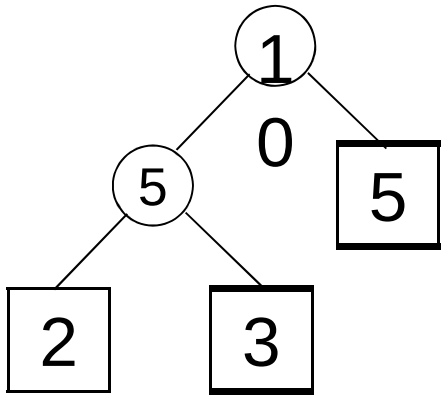

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

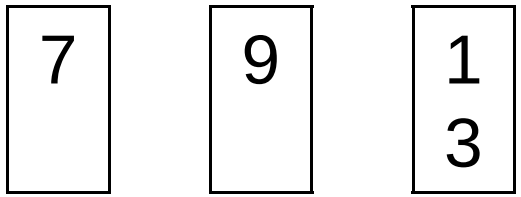
Árvore *i*=2:



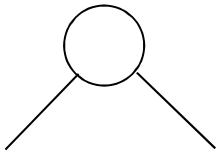
Algoritmo de Huffman

L =





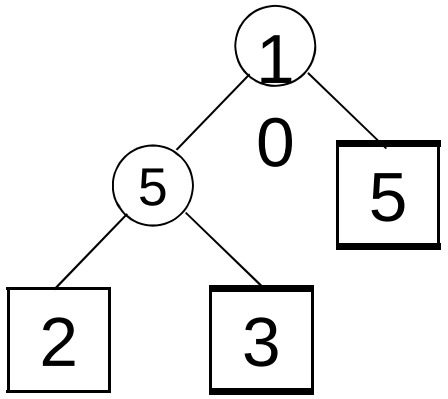
Árvore *i*=3:

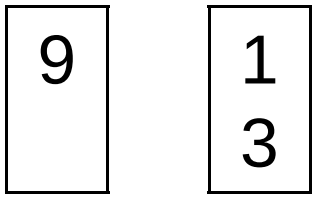


```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

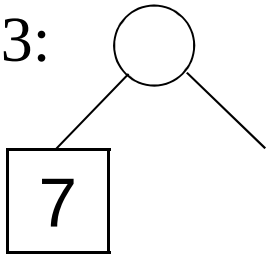
Algoritmo de Huffman

L =





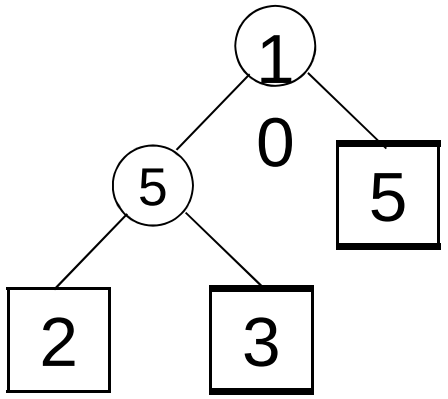
Árvore *i*=3:

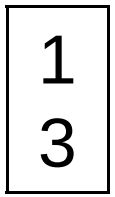



```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

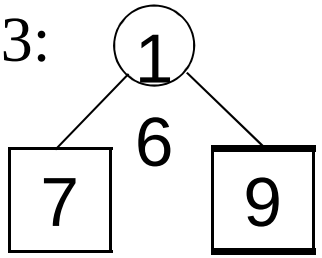
Algoritmo de Huffman

L =





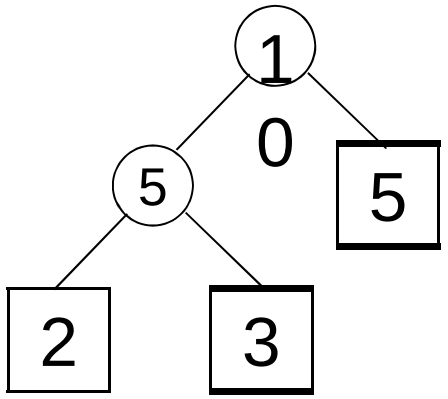
Árvore *i*=3:

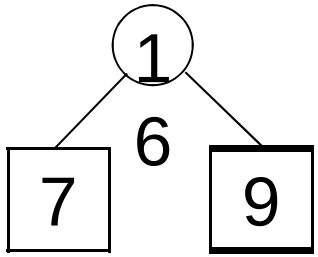


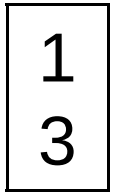

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

Algoritmo de Huffman

L =

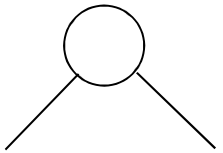






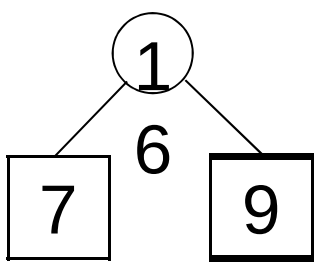

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

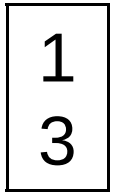
Árvore *i*=4:



Algoritmo de Huffman

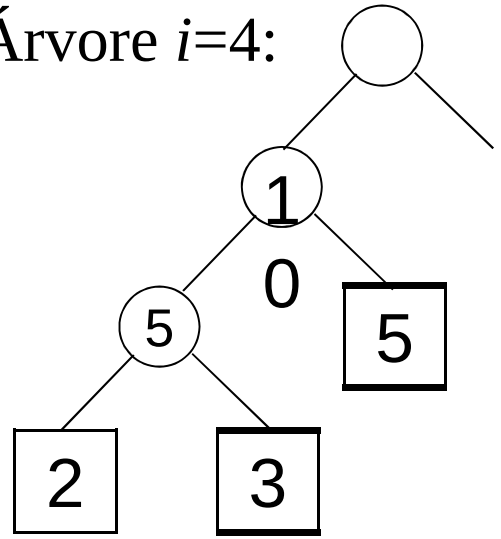
L
=





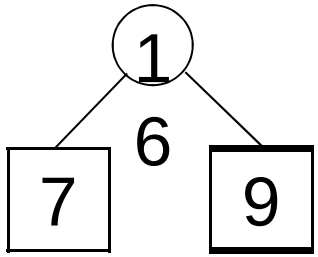

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

Árvore *i*=4:



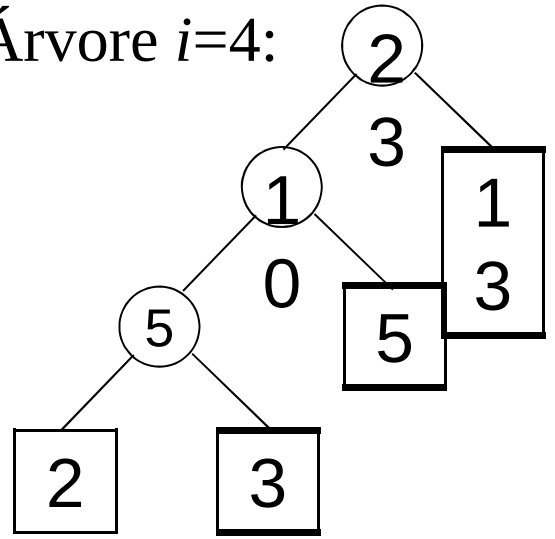
Algoritmo de Huffman

L
=



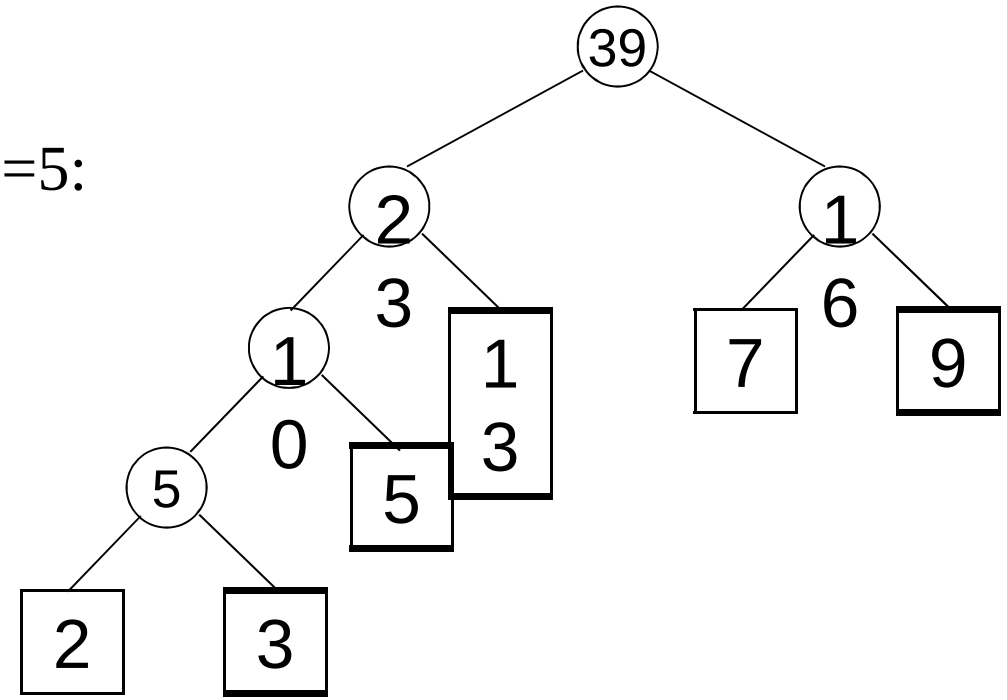

```
Huffman(L, n)

for (i=1;i<n;i++)
{
   nova.esq←menor(L);
   nova.dir←menor(L);
   peso←peso(esq)+peso(dir);
   Insere(L,nova);
}
```

Árvore *i*=4:



Algoritmo de Huffman

Árvore *i*=5:



CE ponderado:(2*4)+(3*4)+(5*3)+(13*2)+(7*2)+(9*2) = 93

Melhor árvore completa = 95

Código de Huffman

- Algoritmo para a compressão de arquivos, principalmente arquivos textos
- Atribui códigos menores para símbolos mais frequentes e códigos maiores para símbolos menos frequentes
- Código é um conjunto de bits

Código de Huffman

- Representação dos dados é feita com códigos de tamanho variável

Código ASCII	Código de Huffman
A=01000001	A=? (0)
B=01000010	B=? (110)
.	.
.	.
.	.
a=01100001	a=? (1111110)
b=01100010	b=? (11111111110)

Exemplo

- Supondo A e C mais frequentes que C e D no conjunto de valores possíveis

Símbolo	Código
A	0
B	110
C	10
D	111

ABACDA= 0 110 0 10 111 0

A | B | A | C | D | A

Requisito

- O código de um símbolo não pode ser prefixo de um outro código
  - Se isso acontece, tem-se ambiguidade na decodificação
- Ex: ACBA = 01010
- Os dois bits em vermelho

é A e C ou B?

- Veja que o código de A é

prefixo do código de B

Símbolo	Huffman
A	0
B	01
C	1

Problema

- Dada uma tabela de frequências como determinar o melhor conjunto de códigos, ou seja, o conjunto que comprimirá mais os símbolos?
- Huffman desenvolveu um algoritmo para isso e mostrou que o conjunto de símbolos obtidos é o melhor para conjuntos de dados que têm a frequência de seus símbolos igual a tabela de frequência usada

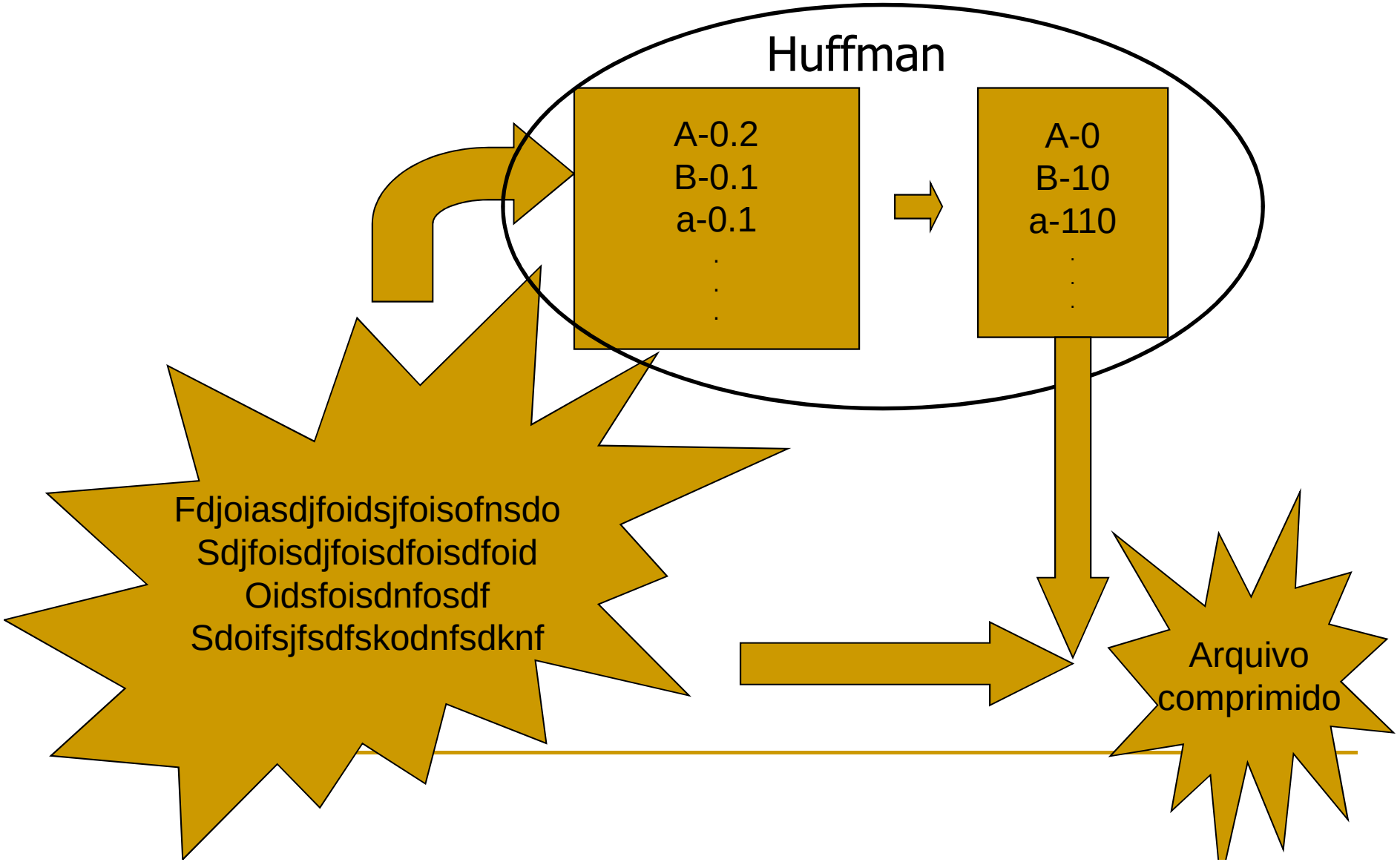
Informações de frequência

- Algoritmo de Huffman produz tabela de códigos baseada em informações de frequência
- Dependência do tipo de dado primário

O algoritmo em si

- Dado: Tabela de freqüências dos N símbolos de um alfabeto
- Objetivo: Atribuir códigos aos símbolos de modo que os mais freqüentes tenham códigos menores (menos bits)

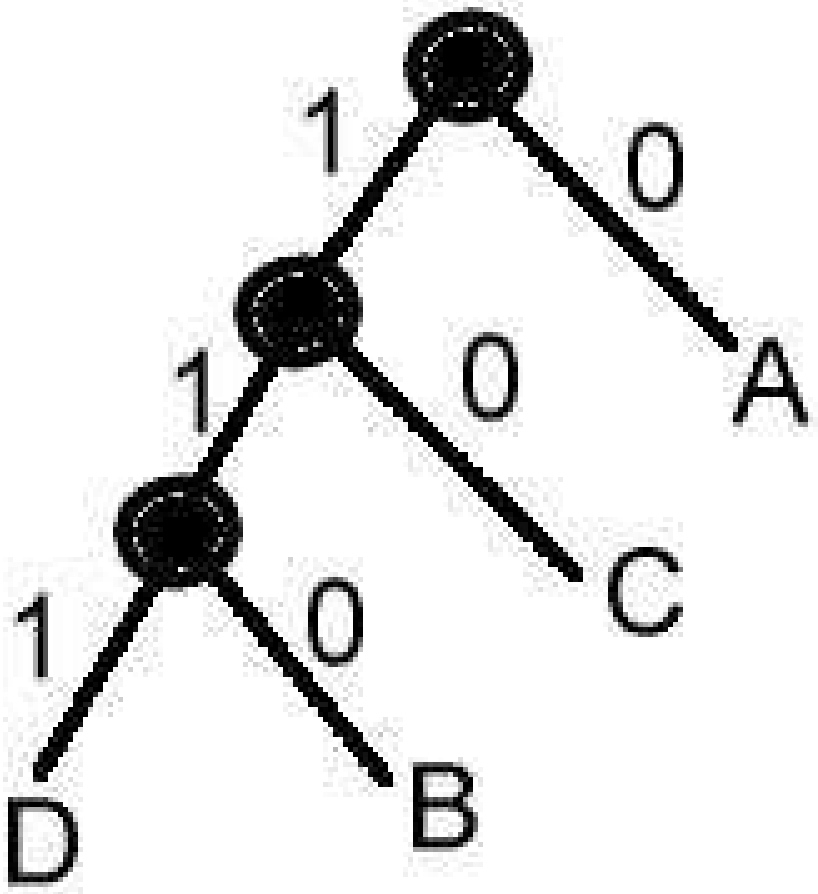
O processo de compressão



Idéia básica

- Construir uma árvore binária tal que
  - A) suas folhas sejam os N símbolos do alfabeto
  - B)cada ramo da árvore seja um valor 1 (esquerda) ou 0 (direita)
    - Isso é uma convenção, o contrário também funciona
  - O código de um símbolo será a seqüência de bits dos ramos da raiz até sua posição na árvore

Exemplo



Símbolo	Código
A	0
B	110
C	10
D	111