

Compacto dos melhores momentos

AULA 16

Compressão de dados

Problema: representar um arquivo **GRANDE**
por outro **menor**.

Exemplo:

arquivo **GRANDE**: ababababababababababababababab

arquivo **menor**: 12ab

Por que comprimir?

Menor espaço de armazenamento.

Menor tempo de transmissão.

Software: gzip, bzip, 7z, etc.

Esquema

representar um dado **fluxo de bits** (*bitstream*) por outro mais curto.

Esquema básico de compressão de dados:

- ▶ um **compressor** transforma um fluxo de bits B em um fluxo $C(B)$ e
- ▶ um **expansor** transforma $C(B)$ de volta em B .



Basic model for data compression

Entradas e saídas binárias

Cadeia de bits (= *bitstring*) é uma sequência de bits:

```
110001000001110101010011111011001110001010000  
1100000000001111110000011101000111
```

fluxo de bits (= *bitstream*) é uma cadeia de bits na **entrada** ou na **saída** de um programa.

A classe **BinaryStdIn** lê um fluxo de bits a partir da **entrada padrão**.

A classe **BinaryStdOut** escreve um fluxo de bits na **saída padrão**.

BinaryDump, HexDump e PictureDump

```
% more abra.txt
```

```
ABRACADARRA!
```

```
% java BinaryDump 16 < abra.txt
```

```
0100000101000010
```

```
0101001001000001
```

```
0100001101000001
```

```
0100010001000001
```

```
0100001001010010
```

```
0100000100100001
```

```
96 bits
```

Genomas

Podemos usar apenas **2 bits** por caractere:

```
% more genomeTiny.txt
```

```
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC
```

```
% java Genome - < genomeTiny.txt | java  
BinaryDump 32
```

```
00000000000000000000000000000000100001
```

```
00110010001110010011001001100100
```

```
11001001110010001110111001110010
```

```
01000000
```

```
104 bits
```

Codificação de comprimento de carreira

Em inglês: *run-length encoding* (=RLE).

Exemplo: cadeia de bits abaixo tem uma carreira de 15 0s, uma carreira de 7 1s, uma de 7 0s, e uma de 11 1s:

0000000000000001111111000000011111111111

Pode ser representada pela sequência 15 7 7 11.

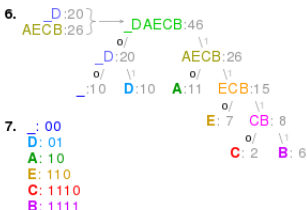
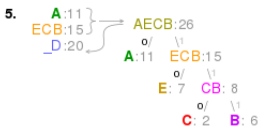
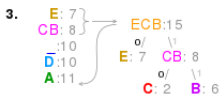
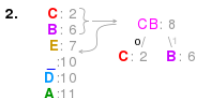
Usando **8 bits** para cada um desses números, teremos uma cadeia de apenas 32 bits (ignore os espaços):

00001111000001110000011100001011

AULA 17

Algoritmo de Huffman

1. "A_DEAD_DAD_CEDD_A_BAD_BABE_A_BEADED_ABACA_BED"



8. "100001110100100011001001110110011100100100011111001001111101111110
00100011111101001110010010101111011101000111111001"

Fonte: Huffman coding

Referências: Algoritmo de Huffman para compressão de dados (PF), Data Compression (SW), slides (SW), video (SW)

Ideias

O algoritmo de Huffman recebe um fluxo de bits e devolve um fluxo de bits.

Fluxo de bits original é lido de 8 em 8 bits

0100000101000010010100100100000101000011
ABRAC

0101001010001000100000100100001001010010
ADABR

0100000100100001
A!

Ideias

Tudo se passa como se o algoritmo transformasse uma **string** em uma **cadeia de bits**.

Exemplo: transforma ABRA**C**ADABRA! em
0111111100**110**01000111111100101.

Cada **caractere** é convertido em uma pequena **cadeia de bits**, que é o seu **código**. Por exemplo, **C** é convertido em **110**.

Suponha que temos um arquivo com **100000 mil caracteres** sobre o alfabeto "**!ABCDR**".

Compressão ou codificação

Uma **tabela de códigos** leva cada **caractere** de 8 bits no seu **código**.

Exemplo de **códigos de comprimento fixo**:

caractere	código
!	001
A	010
B	011
C	100
D	101
R	110

A **tabela de códigos** é uma **ST** em que as **chaves** são os **caracteres** e os **valores** são os **códigos**.

Compressão ou codificação

Usando a **tabela de códigos** anterior gastaríamos

$$100000 \times 3 = 300000 \text{ bits}$$

para codificar o arquivo.

Examinando o arquivo observamos a seguinte **frequência de símbolos**.

caractere	frequência
!	5000
A	45000
B	13000
C	9000
D	16000
R	12000

Compressão ou codificação

Exemplo de **códigos de comprimento variável**:

caractere	código
!	1010
A	0
B	111
C	1011
D	100
R	110

Com essa nova **tabela de códigos** gastaríamos

$$5M \times 4 + 45M \times 1 + 13M \times 3 + 9M \times 4 + 16M \times 3 + 12M \times 3$$

bits = **224000 bits** para representar o arquivo.

Compressão ou codificação

Ideia do algoritmo de Huffman: usar códigos **curtos** para os caracteres que ocorrem com **frequência** e deixar os códigos mais **longos** para os caracteres mais **raros**.

Mesmo princípio que o **código de Morse**:

... the length of each character in Morse is approximately inverse to its frequency of occurrence in English ...

Compressão ou codificação

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

A	• —	U	• • —
B	— • • •	V	• • • —
C	— • — •	W	• — —
D	— • •	X	— • • —
E	•	Y	— • — —
F	• • — •	Z	— — • •
G	— — •		
H	• • • •		
I	• •		
J	• — — —		
K	— • • —	1	• — — — —
L	• — • •	2	• • — — —
M	— —	3	• • • — —
N	— •	4	• • • • —
O	— — —	5	• • • • •
P	• — — •	6	— • • • •
Q	— — • • —	7	— — • • •
R	• — • •	8	— — — • •
S	• • •	9	— — — — •
T	—	0	— — — — —

Fonte: [Morse Code \(Wikipedia\)](#)

Codificação

A **tabela de códigos** será um simples vetor de strings `st[0..255]` indexado pelos **256** caracteres **ASCII**.

É fácil produzir o fluxo de bits codificado a partir da tabela `st[]...`

Codificação

```
public static void compress() {
    String s = BinaryStdIn.readString();
    char[] input = s.toCharArray();
    // aqui vai a construção de st[]...
    BinaryStdOut.write(input.length);
    for (int i=0; i<input.length; i++){
        String code = st[input[i]];
        for(int j=0; j<code.length(); j++)
            if (code.charAt(j) == '1')
                BinaryStdOut.write(true);
            else
                BinaryStdOut.write(false);
    }
    BinaryStdOut.close(); }
```

Expansão (decodificação)

Na **decodificação**, cada **código** deve ser convertido no correspondente **caractere**.

Tabela de códigos deve ser **injetiva**. Mas isso não basta.

Expansão (decodificação)

Exemplo: a tabela de códigos

caractere	código
!	11
A	0
B	1
C	01
D	10
R	00

Transforma **A****B****R****A****C****A****D****A****B****R****A****!** em
0**1****0****0****0****0****1****0****1****0****0****1****0****0****0****1****1**.

Expansão (decodificação)

Exemplo: a tabela de códigos

caractere	código
!	11
A	0
B	1
C	01
D	10
R	00

Transforma **A****B****R****A****C****A****D****A****B****R****A****!** em

0**1****0****0****0****0****1****0****1****0****0****1****0****0****0****1****1**.

0**1****0****0****0****0****1****0****1****0****0****1****0****0****0****1****1** também representa

C**R****R****D****D****C****R****C****B** 8-0

Expansão (decodificação)

A tabela deve ser *livre de prefixos*.

Uma tabela de códigos é **livre de prefixos** (*prefix-free*) se nenhum código é prefixo de outro.

Exemplo: a tabela abaixo é livre de prefixos.

caractere	código
!	101
A	0
B	1111
C	110
D	100
R	1110

Expansão (decodificação)

Exemplo: a tabela abaixo é livre de prefixos.

caractere	código
!	101
A	0
B	1111
C	110
D	100
R	1110

ABRACADABRA! é a única decodificação possível de

011111110011001000111111100101.

Tabelas inversas

A **tabela de códigos inversa** leva cada **código** no correspondente **caractere**. **Exemplo**:

código	caractere
101	!
0	A
1111	B
110	C
100	D
1110	R

A tabela inversa é uma **ST de strings**: as **chaves** são **strings** de 0s e 1s e os **valores** são **caracteres**.

Tabelas inversas

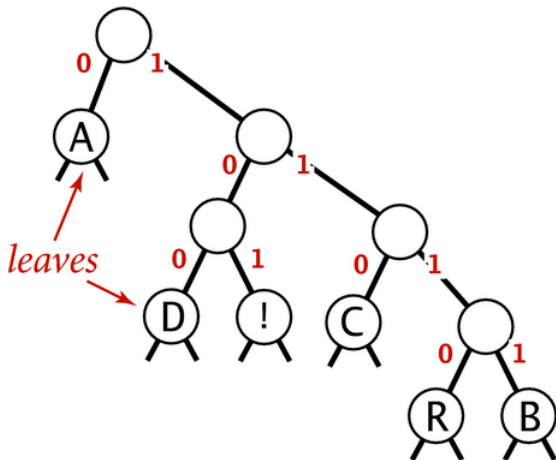
código	caractere
101	!
0	A
1111	B
110	C
100	D
1110	R

A tabela será representada por uma **trie binária**. Portanto, não há nós com apenas um filho. Como a tabela é livre de prefixos, as **chaves estão somente nas folhas**. Diremos que essa é a **trie do código**.

codeword table

key	value
!	101
A	0
B	1111
C	110
D	100
R	1110

trie representation



compressed bitstring

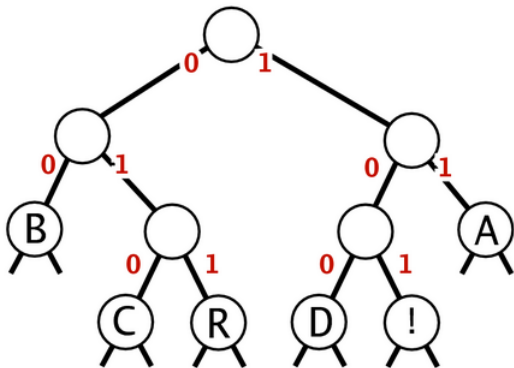
011111110011001000111111100101 ← 30 bits

A B RA CA DA B RA !

codeword table

<i>key</i>	<i>value</i>
!	101
A	11
B	00
C	010
D	100
R	011

trie representation



compressed bitstring

11000111101011100110001111101 ← 29 bits

A B R A C A D A B R A !

Nós de uma trie

```
private static class Node
    implements Comparable<Node> {
    // usado só nas folhas
    private char ch;

    // usado só para construção da trie
    private int freq;

    private final Node left;
    private final Node right;
```

Nós de uma trie

```
Node(char ch, int freq, Node left,  
      Node right){  
    this.ch = ch;  
    this.freq = freq;  
    this.left = left;  
    this.right = right;  
}
```

Nós de uma trie

```
public boolean isLeaf() {  
    return left == null  
        && right == null;  
}  
  
public int compareTo(Node that) {  
    return this.freq - that.freq;  
}  
}
```

Decodificação

```
public static void expand() {
    Node root = readTrie();
    int n = BinaryStdIn.readInt();
    for (int i = 0; i < n; i++) {
        Node x = root;
        while (!x.isLeaf())
            if(BinaryStdIn.readBoolean())
                x = x.right;
            else x = x.left;
        BinaryStdOut.write(x.ch);
    }
    BinaryStdOut.close();
}
```

Construção de um `trie`

O segredo do `algoritmo de Huffman` está na maneira de escolher a `tabela de códigos`.

A `tabela de códigos` de Huffman não é universal.

Ela é construída `sob medida` para a string a ser codificado de tal modo que o comprimento da `cadeia codificada` seja o `menor possível` quando comparado com outros códigos livres de prefixos.

Primeiro, construímos a `trie do código`, depois extraímos dela a `tabela de códigos`.

Construção de um trie

O algoritmo de construção da trie é iterativo.

No início de cada iteração temos um coleção de tries mutuamente disjuntas.

Dada a string a ser codificada, a coleção de tries inicial é construída assim:

1. determine a frequência de cada caractere da string original;
2. para cada caractere, crie um nó e armazene o caractere e sua frequência nos campos `ch` e `freq`
3. cada nó será uma trie da coleção inicial.

Assim, no início da primeira iteração, cada trie tem um único nó.

Construção de um trie

Repita a seguinte iteração até que a coleção de **tries** tenha apenas uma:

1. **escolha** duas **tries** cujas raízes, digamos **x** e **y**, tenham **freq** mínima;
2. **crie** um novo nó **parent** e faça com que **x** e **y** sejam filhos desse nó;
3. **faça** **parent.freq** igual a **x.freq** + **y.freq**;
4. **insira** essa nova **trie** na coleção

Huffman coding demo

- Count frequency for each character in input.

<u>char</u>	<u>freq</u>	<u>encoding</u>
A		
B		
C		
D		
R		
!		

input

A B R A C A D A B R A !

Huffman coding demo

- Count frequency for each character in input.

char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	

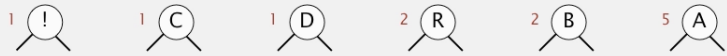
input

A B R A C A D A B R A !

Huffman coding demo

- Start with one node corresponding to each character with weight equal to frequency.

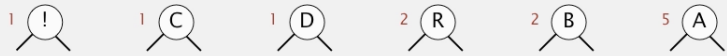
<u>char</u>	<u>freq</u>	<u>encoding</u>
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

<u>char</u>	<u>freq</u>	<u>encoding</u>
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

<u>char</u>	<u>freq</u>	<u>encoding</u>
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

<u>char</u>	<u>freq</u>	<u>encoding</u>
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

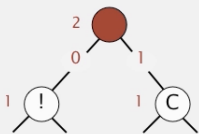
<u>char</u>	<u>freq</u>	<u>encoding</u>
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

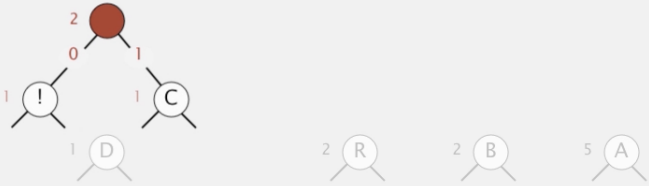
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

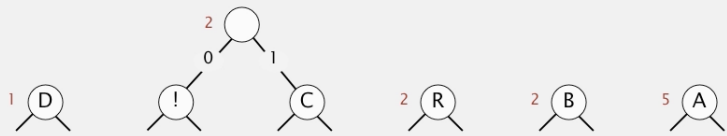
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

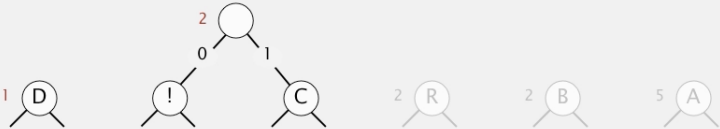
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

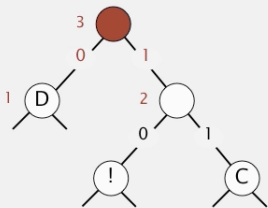
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

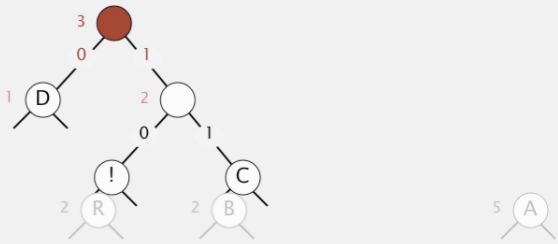
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

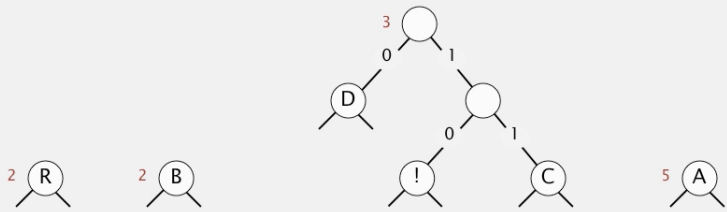
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

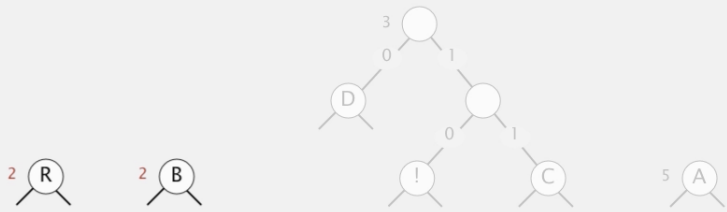
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

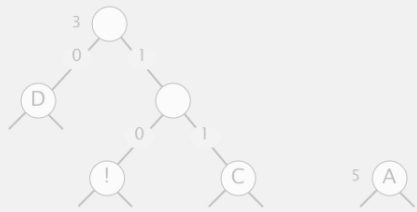
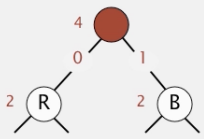
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	
B	2	1
C	1	11
D	1	0
R	2	0
!	1	10



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

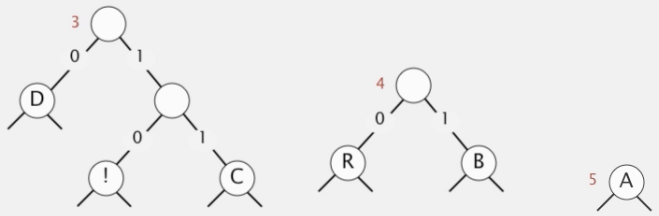
char	freq	encoding
A	5	
B	2	1
C	1	11
D	1	0
R	2	0
!	1	10



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

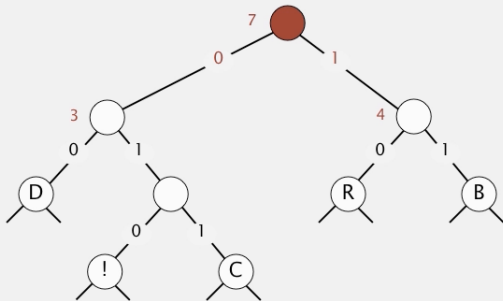
char	freq	encoding
A	5	
B	2	1
C	1	11
D	1	0
R	2	0
!	1	10



Huffman coding demo

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

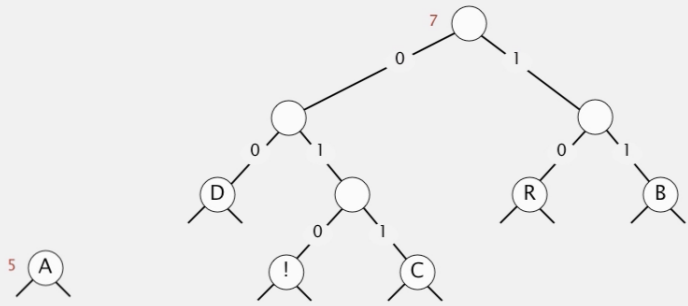
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman coding demo

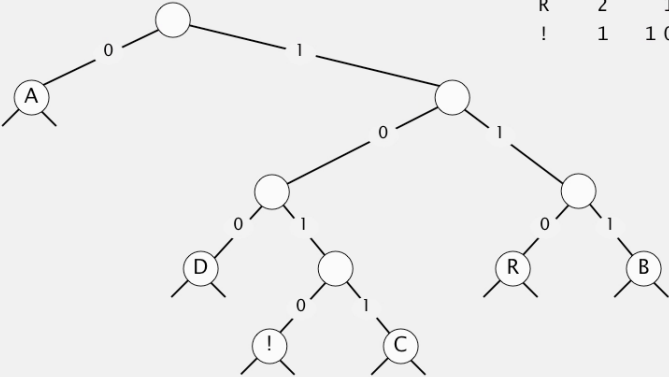
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman coding demo

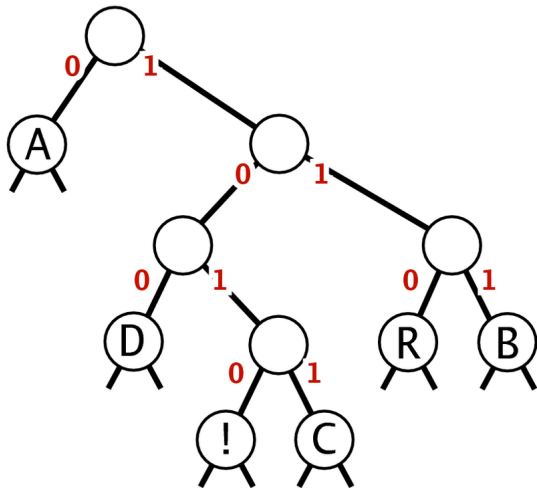
char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Codeword table

<i>key</i>	<i>value</i>
!	1010
A	0
B	111
C	1011
D	100
R	110

Trie representation

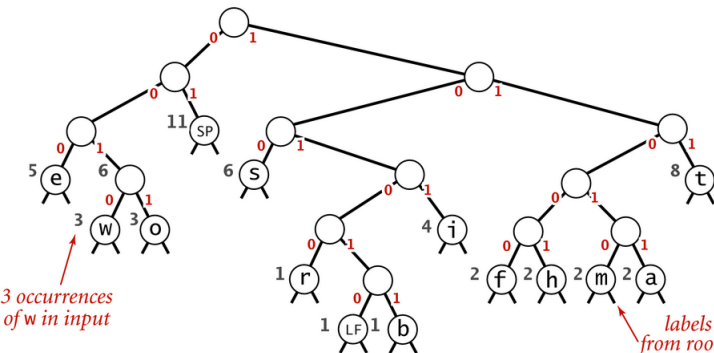


A Huffman code


```
private static Node buildTrie(int[] freq){
    MinPQ<Node> pq= new MinPQ<Node>();
    for (char c = 0; c < R; c++)
        if (freq[c] > 0)
            pq.insert(new Node(c, freq[c],
                                null, null));
    while (pq.size() > 1) {
        Node x = pq.delMin();
        Node y = pq.delMin();
        Node parent = new Node('\0',
                                x.freq + y.freq, x, y);
        pq.insert(parent);
    }
    return pq.delMin();
}
```

Mais um código de Huffman

trie representation



3 occurrences
of w in input

labels on path
from root are 11010
so 11010 is code for m

codeword table

key	value
LF	101010
SP	01
a	11011
b	101011
e	000
f	11000
h	11001
i	1011
m	11010
o	0011
r	10100
s	100
t	111
w	0010

Huffman code for the character stream "it was the best of times it was the worst of times LF"

buildCode()

A tabela de códigos `st[]` usada na codificação é calculada a partir da `trie`:

```
private static String[]
buildCode(Node root) {
    String[] st = new String[R];
    buildCode(st, root, '');
    return st;
}
```

buildCode()

A tabela de códigos `st[]` usada na codificação é calculada a partir da `trie`:

```
private static void
buildCode(String[] st, Node x, String s) {
    if (x.isLeaf()) {
        st[x.ch] = s;
        return;
    }
    buildCode(st, x.left, s + '0');
    buildCode(st, x.right, s + '1');
}
```

A trie de Huffman é ótima

Proposição. A cadeia de bits produzida pelo algoritmo de Huffman é **mínima** no seguinte sentido:

nenhuma outra cadeia produzida por um código livre de prefixos é mais curta que a cadeia produzida pelo algoritmo de Huffman.

Em **MAC0338 Análise de Algoritmos** lembrar disto quando estiverem vendo **Algoritmos Gulosos**.

A **trie** de Huffman é ótima?

```
% more abra.txt | java BinaryDump 0
```

```
96 bits
```

```
% java Huffman - < abra.txt |
```

```
    java BinaryDump 0
```

```
120 bits
```

A trie de Huffman é ótima?

```
% more abra.txt
```

```
ABRACADABRA!
```

```
% java Huffman - < abra.txt |
```

```
    java BinaryDump 30
```

```
010100000100101000100010010000
```

```
110100001101010100101010000100
```

```
0000000000000000000000000000110
```

```
001111100101101000111110010100
```

```
120 bits
```

A trie de Huffman é ótima?

```
% java Huffman - < abra.txt |  
    java BinaryDump 30  
010100000100101000100010010000  
11010000110101010010101000010  
0000000000000000000000000000001100  
01111100101101000111110010100  
120 bits
```


A trie de Huffman é ótima?

01010000010010100010001000100010000
1101000011010101001010101000010

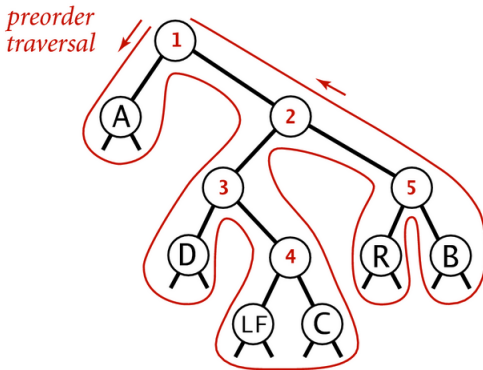
	código	caractere
0 1	01000001	A
0 0 1	01000100	D
0 1	00100001	!
1	01000011	C
0 1	01010010	R
1	01000010	B

Codificação da `trie`

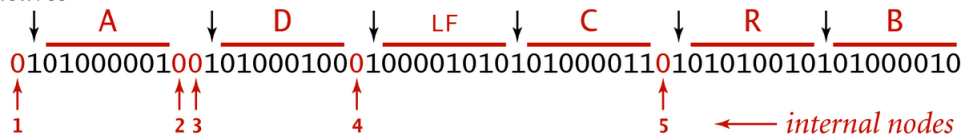
A cadeia de bits produzida pelo `algoritmo de Huffman` não pode ser decodificada sem a correspondente `trie`.

É preciso acrescentar a `trie` à cadeia codificada.

Codificação da trie



leaves



Using preorder traversal to encode a trie as a bitstream

Codificação da trie

```
private static void writeTrie(Node x) {  
    if (x.isLeaf()) {  
        BinaryStdOut.write(true);  
        BinaryStdOut.write(x.ch);  
        return;  
    }  
    BinaryStdOut.write(false);  
    writeTrie(x.left);  
    writeTrie(x.right);  
}
```

Codificação da trie

```
private static Node readTrie() {  
    if (BinaryStdIn.readBoolean()) {  
        char c = BinaryStdIn.readChar();  
        return new Node(c, 0, null, null);  
    }  
    return new Node('\0', 0,  
        readTrie(), readTrie());  
}
```

Compressão completa

Com `buildTrie()`, `buildCode()` e `writeTrie()`, podemos completar o método de compressão

```
public static void compress() {
    String s = BinaryStdIn.readString();
    char[] input = s.toCharArray();
    // aqui vai a construção de st[...]...
    int[] freq = new int[R];
    for (int i = 0; i < input.length; i++)
        freq[input[i]]++;
    Node root = buildTrie(freq);
    String[] st = buildCode(root);
}
```

Compressão completa

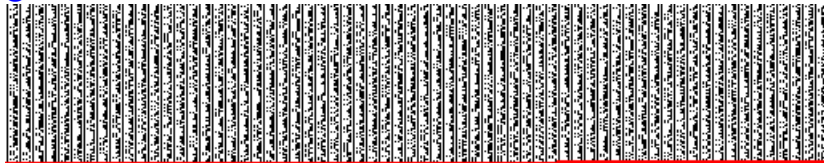
```
writeTrie(root);
BinaryStdOut.write(input.length);
for (int i=0; i<input.length; i++){
    String code = st[input[i]];
    for(int j=0;j<code.length(); j++)
        if (code.charAt(j) == '1')
            BinaryStdOut.write(true);
        else
            BinaryStdOut.write(false);
}
BinaryStdOut.close();
}
```

Esqueleto da classe Huffman

```
public class Huffman {  
    private static int R = 256; // alfabeto  
    private static class Node implements...  
    public static void compress() {...}  
    public static void expand() {...}  
    private static Node buildTrie(int[] freq)  
    private static String[] buildCode(Node...  
    private static void writeTrie(Node x)  
    private static Node readTrie() {...}  
}
```


Genome versus Huffman

```
% java PictureDump 512 100 <  
genomeVirus.txt
```



50008 bits

Genome versus Huffman

```
% java Genome - < genomeVirus.txt | java  
PictureDump 512 25
```



12536 bits

```
% java Huffman - < genomeVirus.txt | java  
PictureDump 512 25
```



12576 bits

RunLength versus Huffman

```
% java PictureDump 32 48 < q32x48.bin
```

```
1536 bits
```



```
% java RunLength - < q32x48.bin | java  
BinaryDump 0
```

```
1144 bits
```

```
% java Huffman - < q32x48.bin | java  
BinaryDump 0
```

```
816 bits
```