MAC0323 Algoritmos e Estruturas de Dados II

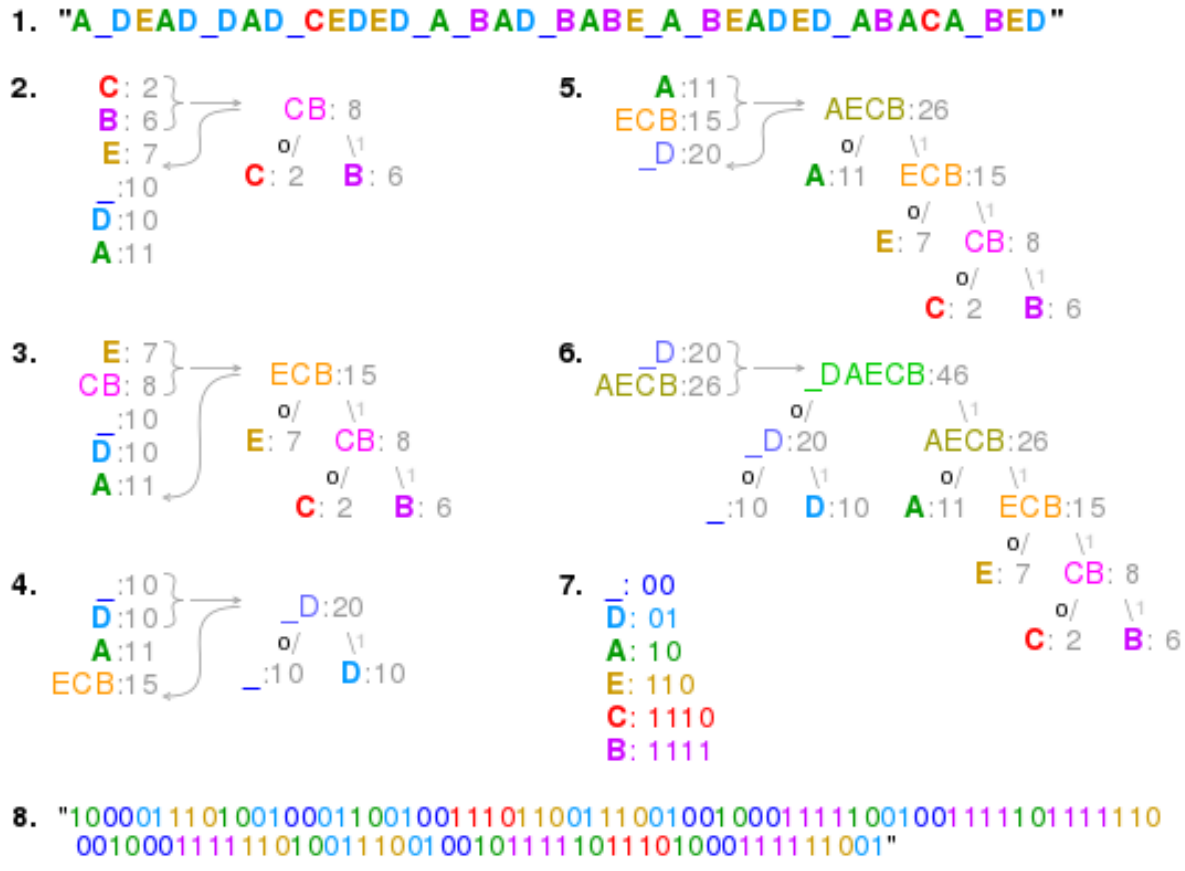
Edição 2020 – 2

Compacto dos melhores momentos

Fonte: ash.atozviews.com

AULA 25

Algoritmo de Huffman



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.

```
01000001010000100101001001000001010000011
ABRAC

01010010100010001000010010000100101010010
ADABR

0100000100100001
A!
```

Ideias

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

caractere	frequência	código
!	5000	1010
A	45000	0
B	13000	111
C	9000	1011
D	16000	100
R	12000	110

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**.

Ideias

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

caractere	frequência	código
!	5000	1010
A	45000	0
B	13000	111
C	9000	1011
D	16000	100
R	12000	110

A tabela deve ser *livre de prefixos*.

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

Trie do código

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

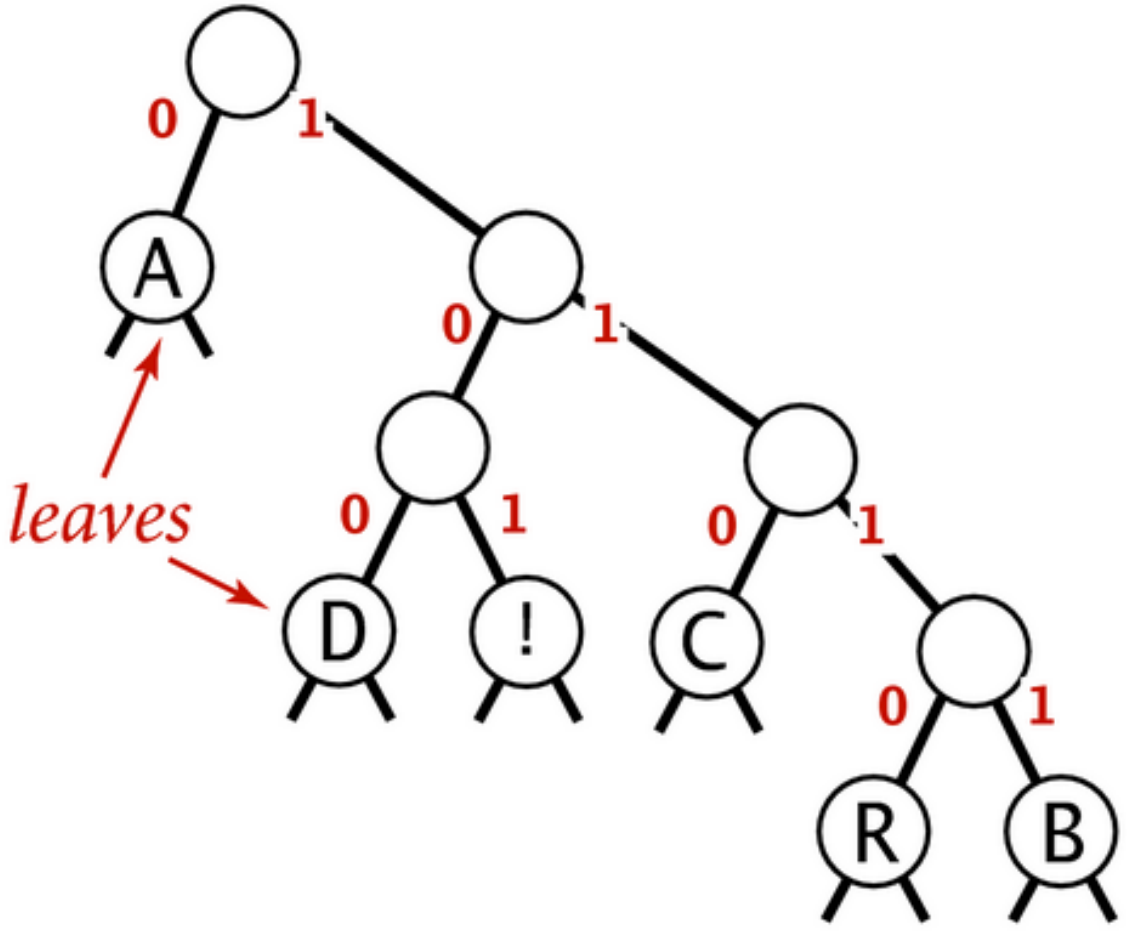
A tabela inversa é uma ST de strings: as chaves são strings de 0s e 1s e os valores são caracteres.

A tabela será representada por uma trie binária.

codeword table

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

trie representation



compressed bitstring

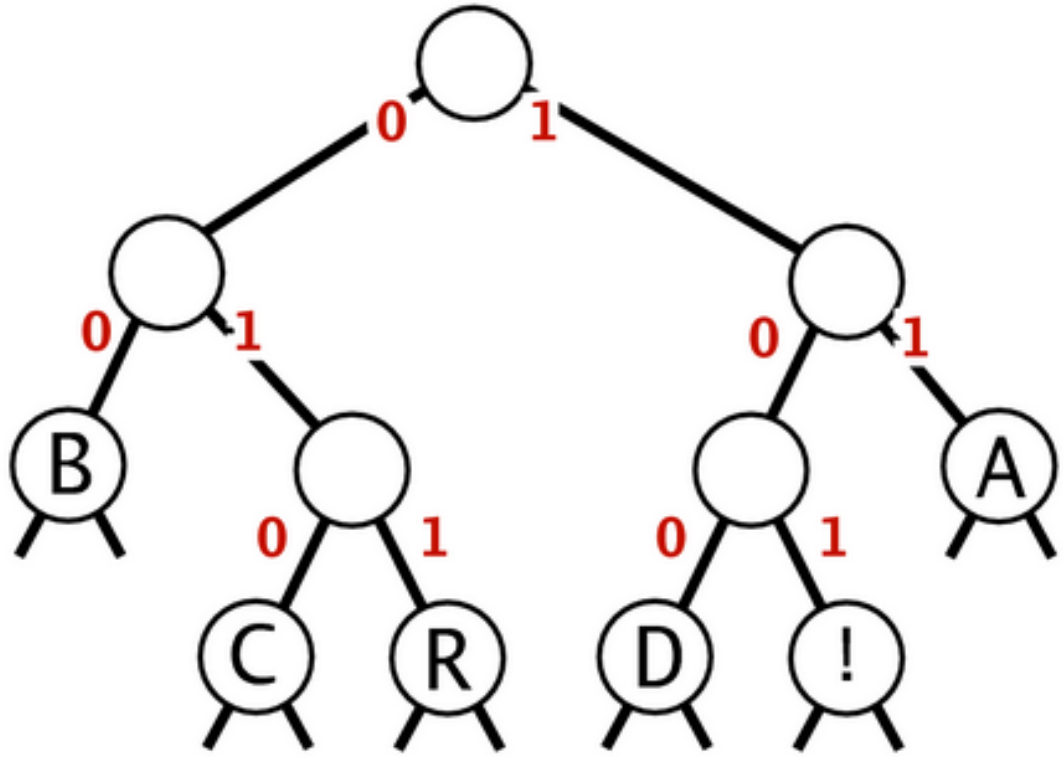
0111111100110010001111111100101 ← *30 bits*

A B RA CA DA B RA !

codeword table

key	*value*
!	101
A	11
B	00
C	010
D	100
R	011

trie representation



compressed bitstring

1100011110101110011000111101 ← *29 bits*

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

AULA 25

Construção de uma `trie`

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

Construção de uma `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 codificada de tal modo que o comprimento da cadeia codificada seja o menor possível quando comparado com outros códigos livres de prefixos.

Construção de uma `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 codificada 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 uma `trie`

O algoritmo de construção da `trie` é iterativo.
No início de cada iteração, temos uma
coleção de `tries` mutuamente disjuntas.

Construção de uma `trie`

O algoritmo de construção da `trie` é iterativo.
No início de cada iteração, temos uma
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.

Construção de uma `trie`

O algoritmo de construção da `trie` é iterativo.
No início de cada iteração, temos uma
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 uma `trie`

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

1. escolha duas `tries` cujas raízes,
   digamos `x` e `y`, tenham `freq` mínima;

Construção de uma `trie`

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

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;

Construção de uma `trie`

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

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.

char	freq	encoding
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	

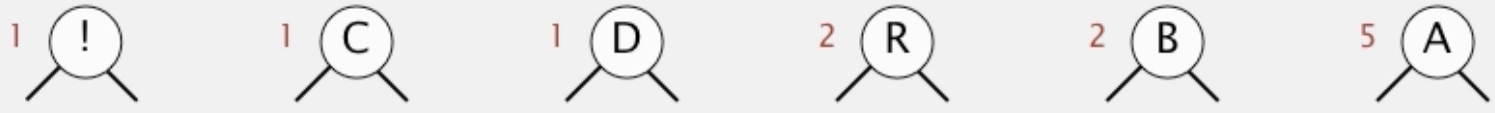
input

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.

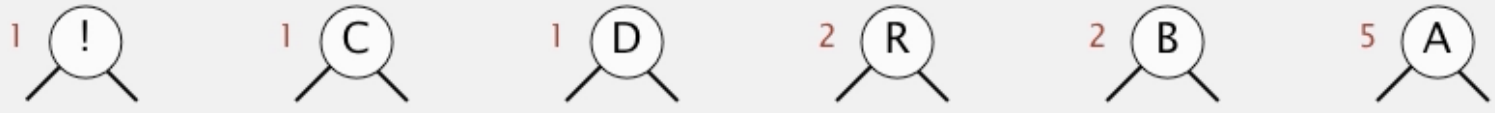
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

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

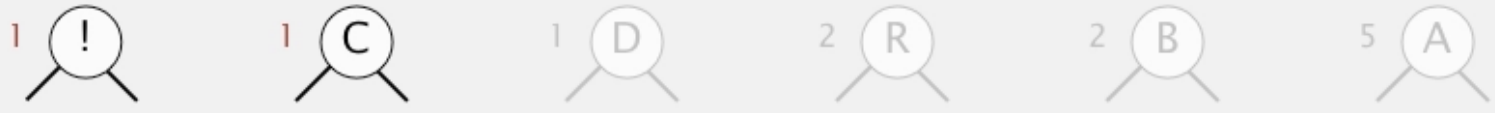
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



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	
D	1	
R	2	
!	1	



Huffman coding demo

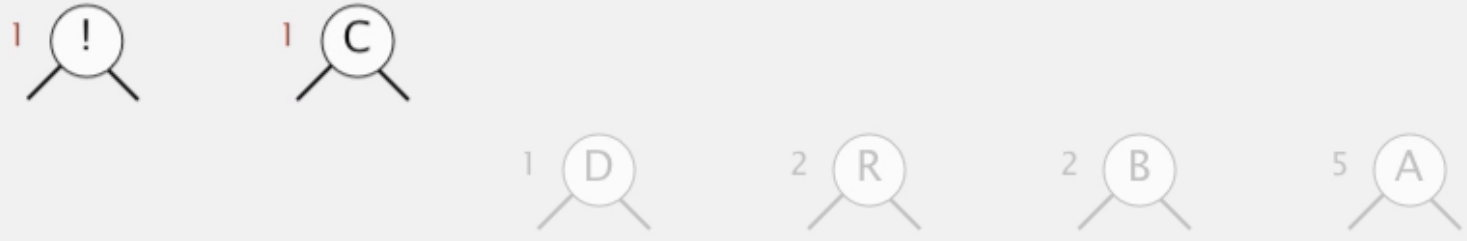
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

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

Huffman coding demo

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

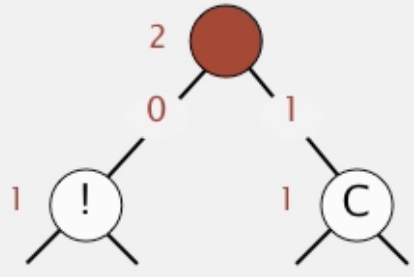
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	

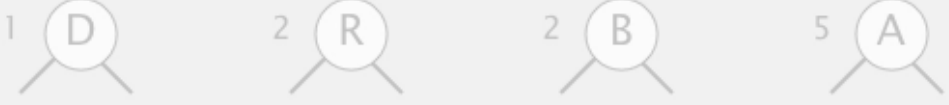


Huffman coding demo

- 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

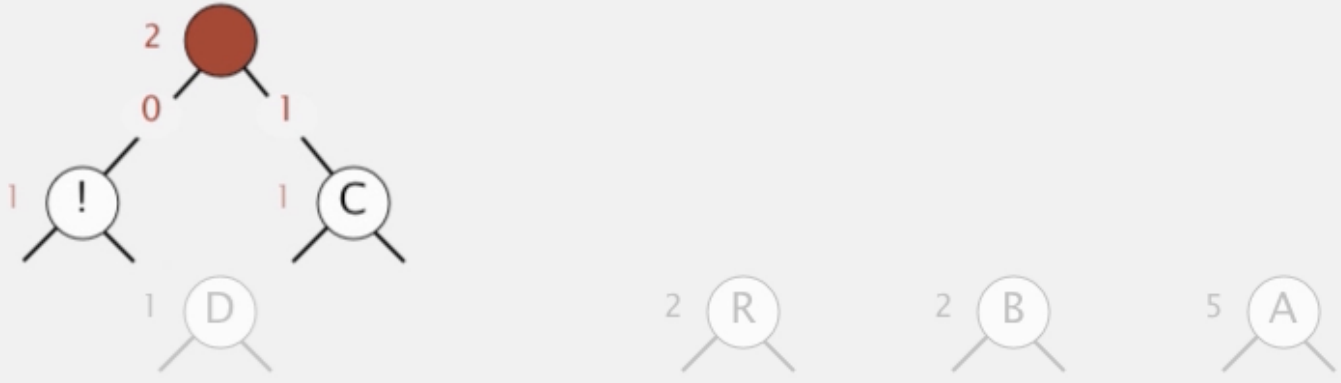




Huffman coding demo

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

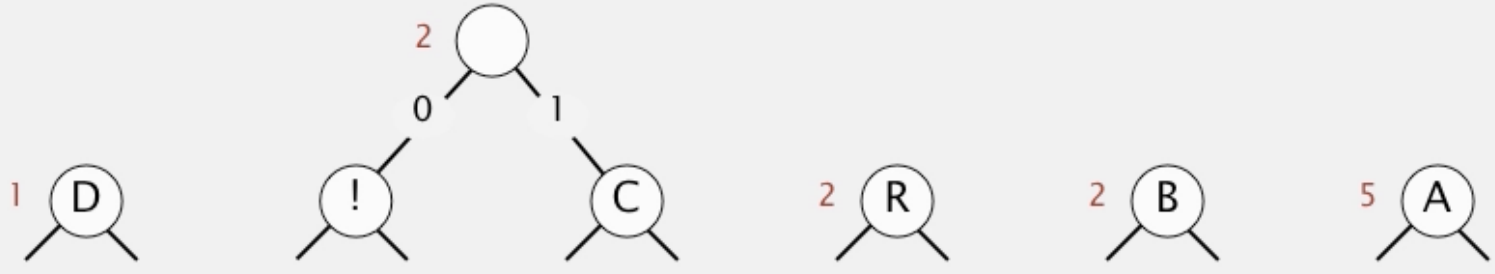
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

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

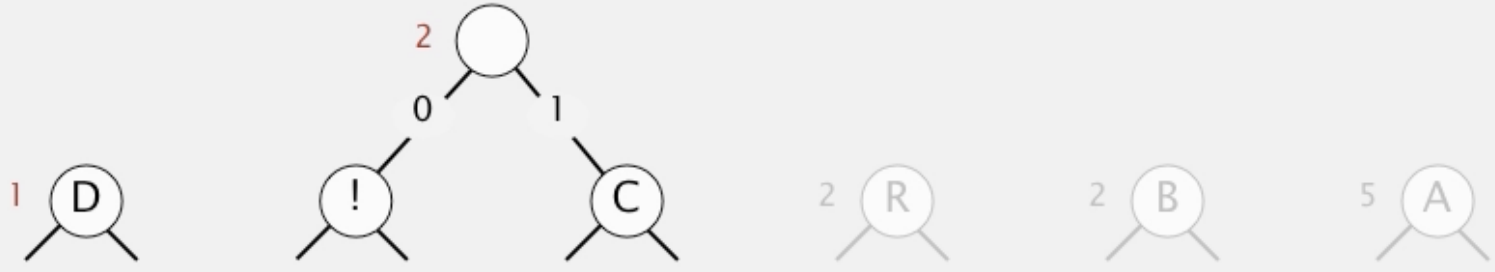
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

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

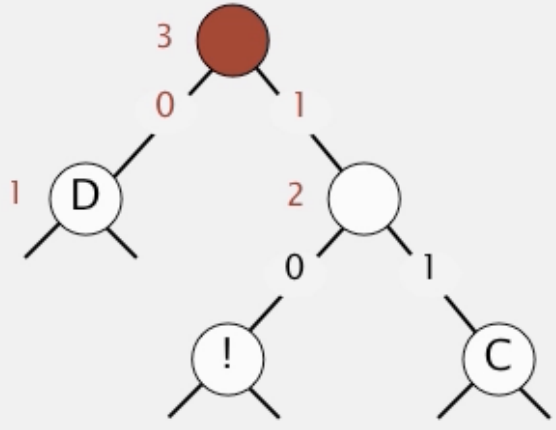
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

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

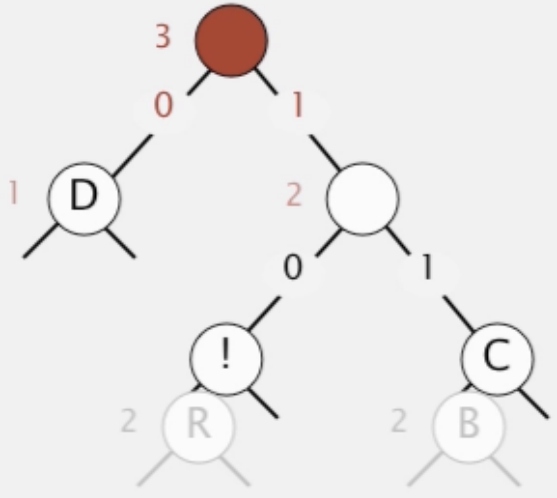
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

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

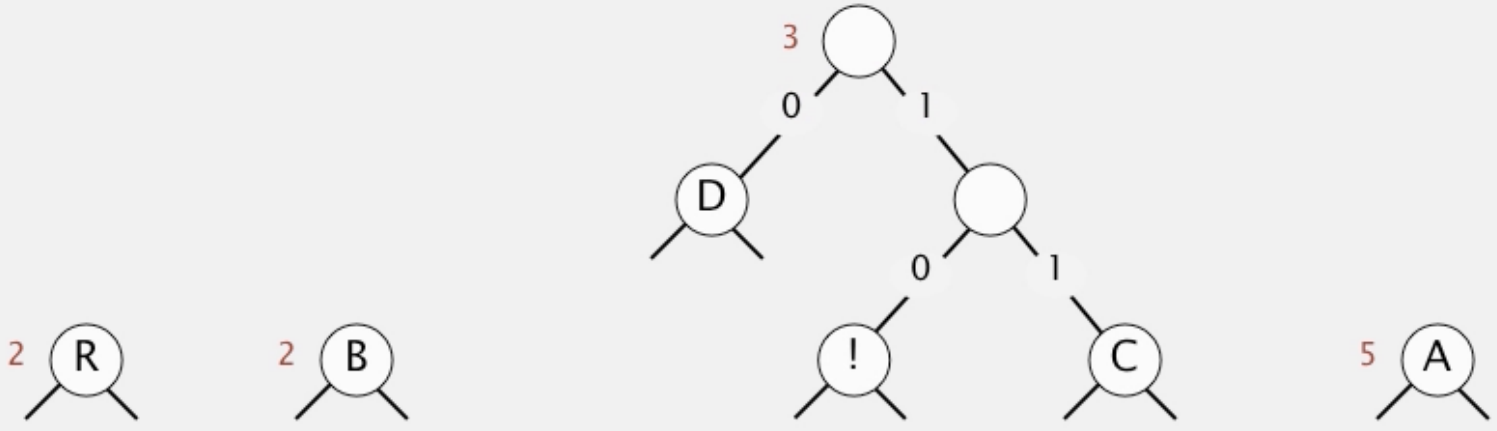
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

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

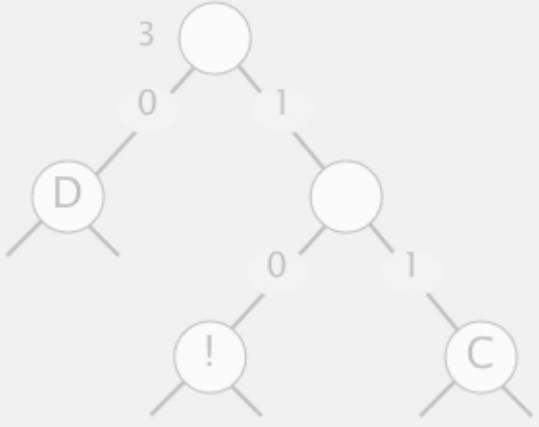
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

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

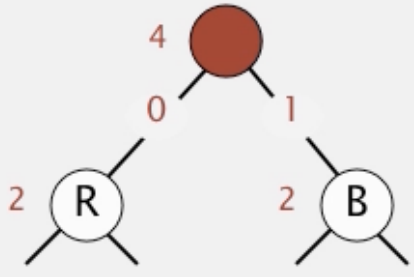
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



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	1 1
D	1	0
R	2	0
!	1	1 0



Huffman coding demo

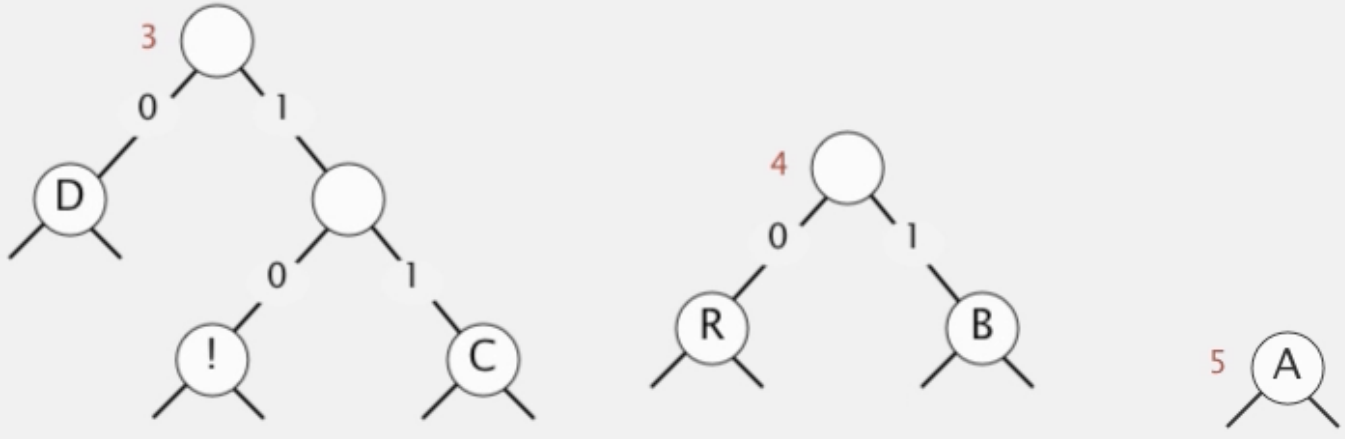
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

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

Huffman coding demo

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

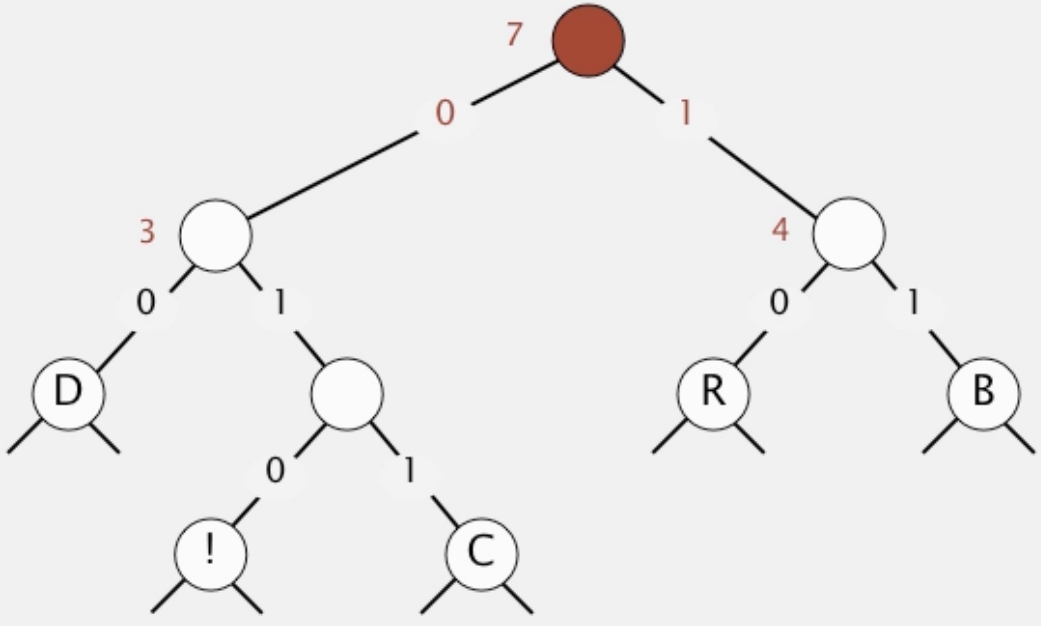
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman coding demo

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

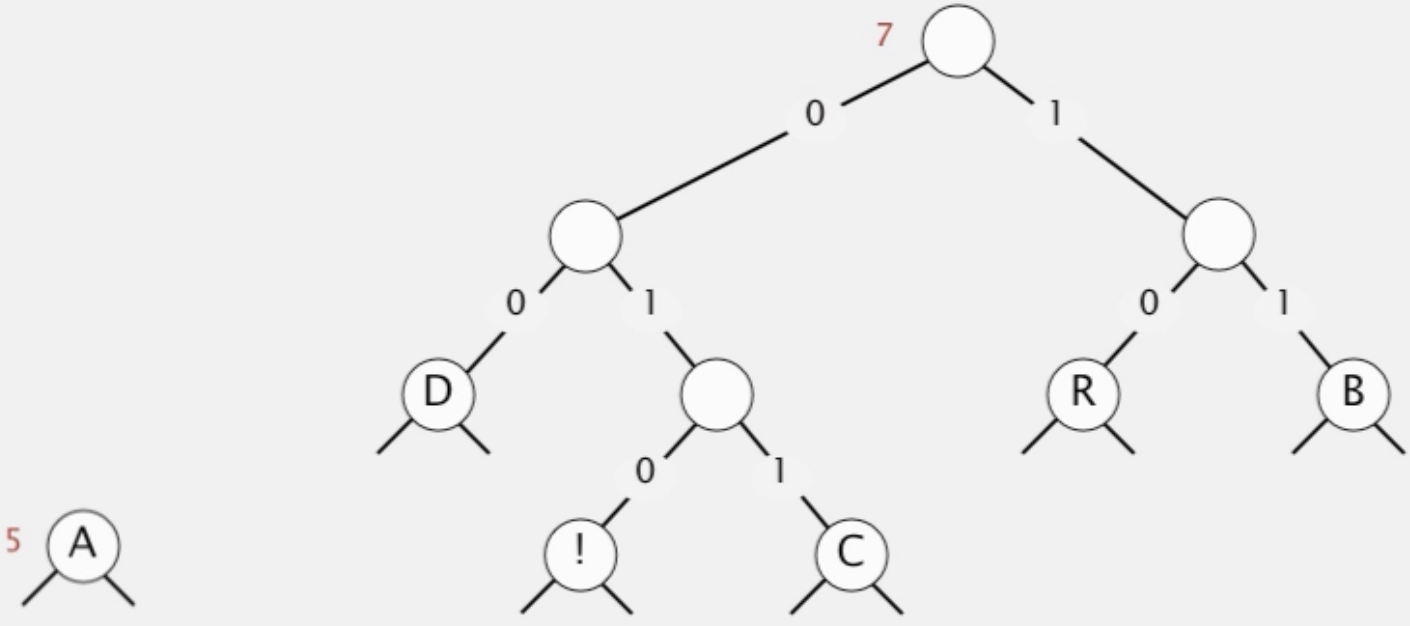
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman coding demo

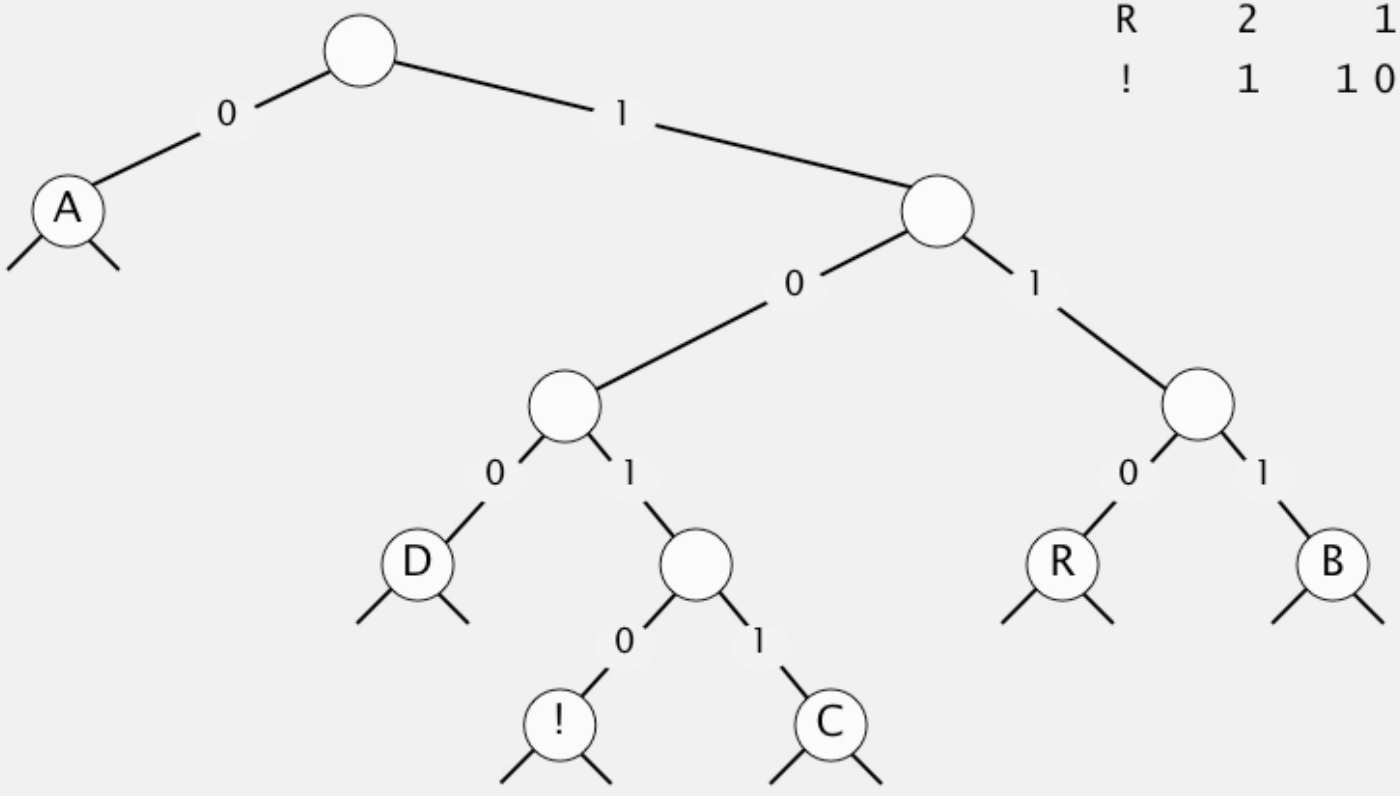
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

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

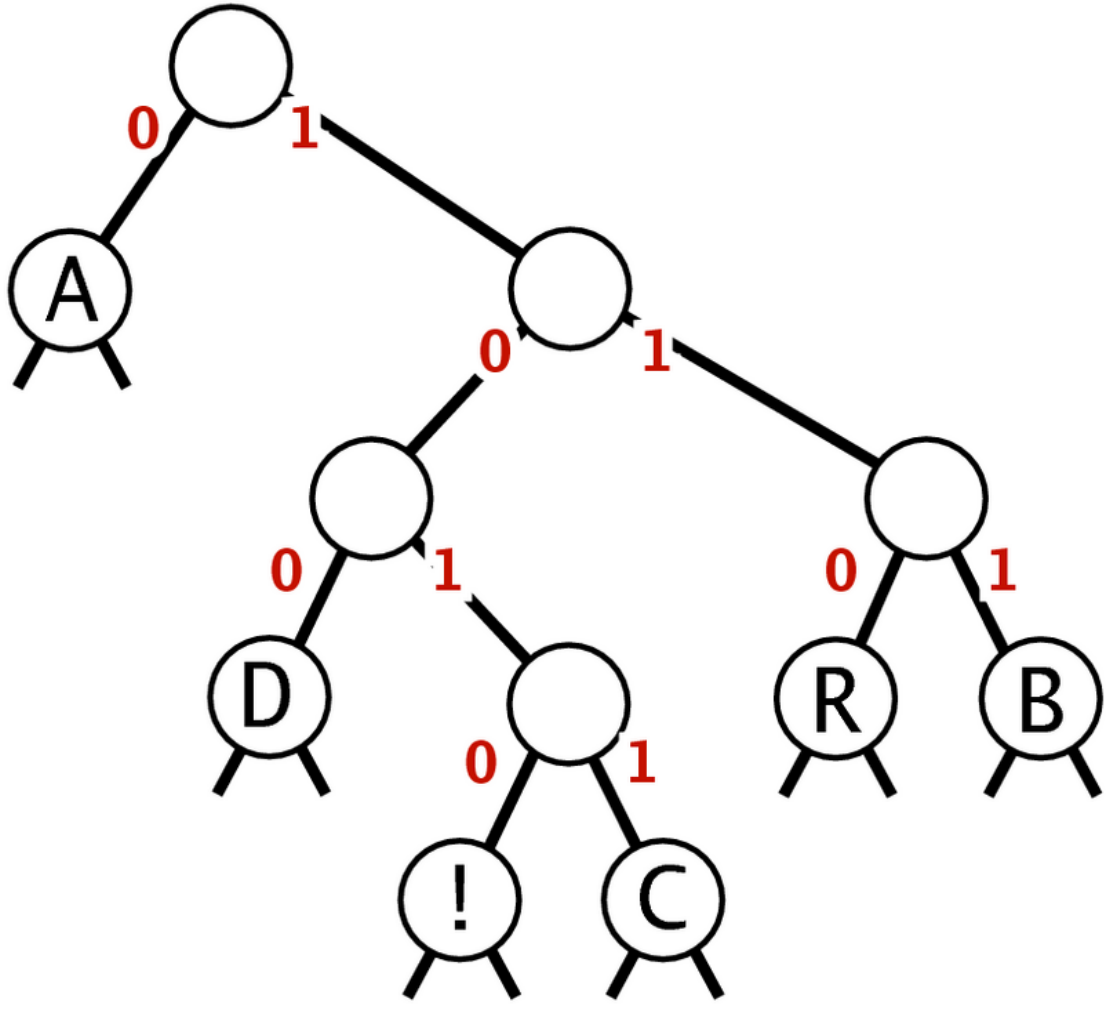
char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Codeword table

key	*value*
!	1010
A	0
B	111
C	1011
D	100
R	110

Trie representation



A Huffman code

Recebe a frequência dos caracteres.

```
static Node buildTrie(int *freq) {
    MinPQ pq = MinPQInit();              /* fila de nós */
    Node x, y, parent;
```

Recebe a frequência dos caracteres.

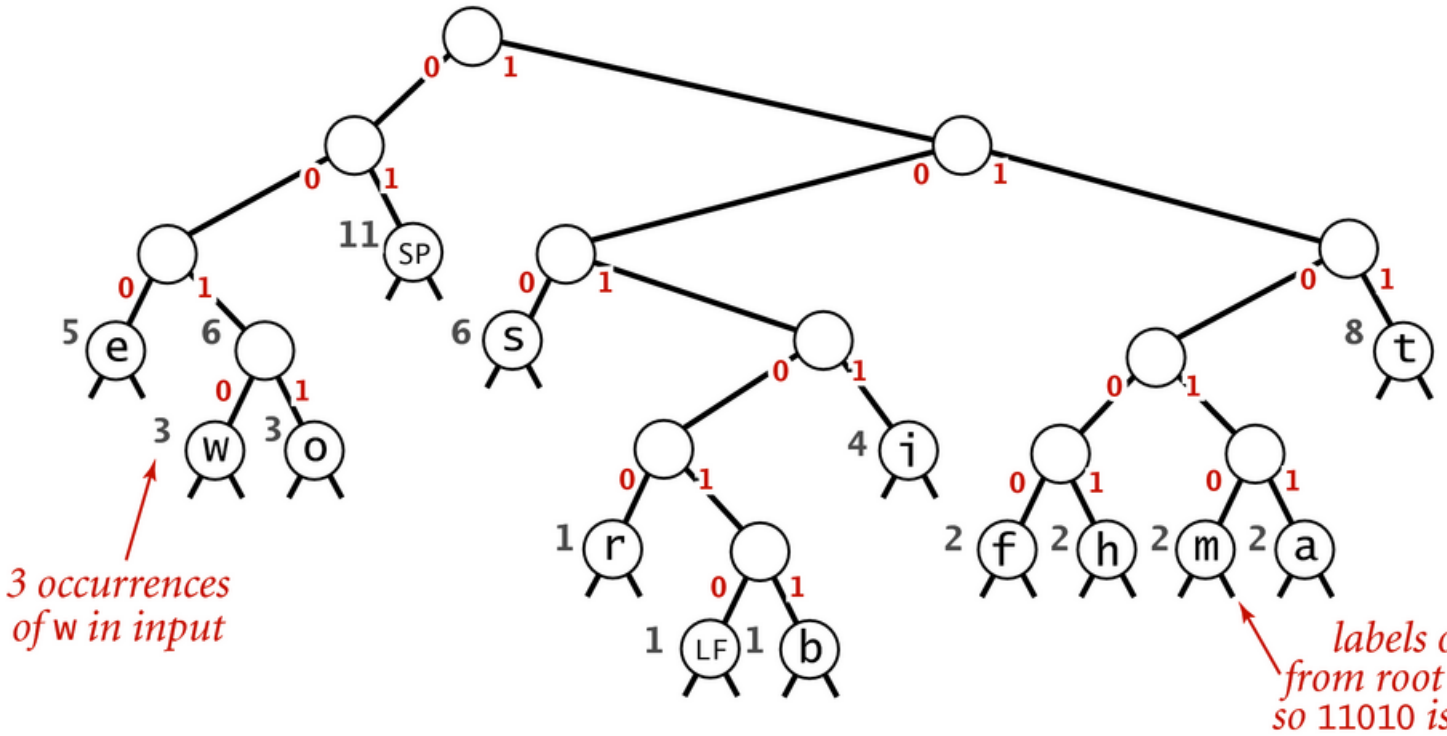
```
static Node buildTrie(int *freq) {
    MinPQ pq = MinPQInit();                 /* fila de nós */
    Node x, y, parent;
    for (char c = 0; c < R; c++)
        if (freq[c] > 0) {
            x = newNode(c, freq[c], NULL, NULL);
            insert(pq, x);
        }
```

Recebe a frequência dos caracteres.

```
static Node buildTrie(int *freq) {
    MinPQ pq = MinPQInit();                 /* fila de nós */
    Node x, y, parent;
    for (char c = 0; c < R; c++)
        if (freq[c] > 0) {
            x = newNode(c, freq[c], NULL, NULL);
            insert(pq, x);
        }
    while (size(pq) > 1) {
        x = delMin(pq);
        y = delMin(pq);
        parent = newNode('\0', x->freq + y->freq, x, y);
        insert(pq, parent);
    }
    return delMin(pq);
}
```

Mais um código de Huffman

trie representation



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`:

```
static char **buildCode(Node root) {
   char **st = mallocSafe(R*sizeof(char *));
   char s[MAX];
   s[0] = '\0';
   buildCode(st, root, s);
   return st;
}
```

buildCode()

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

```
static void buildCode(char **st, Node x, char *s) {
    if (isLeaf(x))
        copy(st[x->ch], s);              /* aloca e copia */
    else {
        int n = strlen(s);
        s[n+1] = '\0';
        s[n] = '0';
        buildCode(st, x->left, s);
        s[n] = '1';
        buildCode(st, x->right, s);
        s[n] = '\0';
    }
}
```

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
000000000000000000000000000110
001111110010110100011111100101 00
120 bits
```

A trie de Huffman é ótima?

```
% java Huffman - < abra.txt |
        java BinaryDump 30
010100000100101000100010010000
110100001101010100101010000010

000000000000000000000000001100

011111001011010001111110010100
120 bits
```

A `trie` de Huffman é ótima?

```
01010000010010100010010010010000
11010000110101010010101000010
```

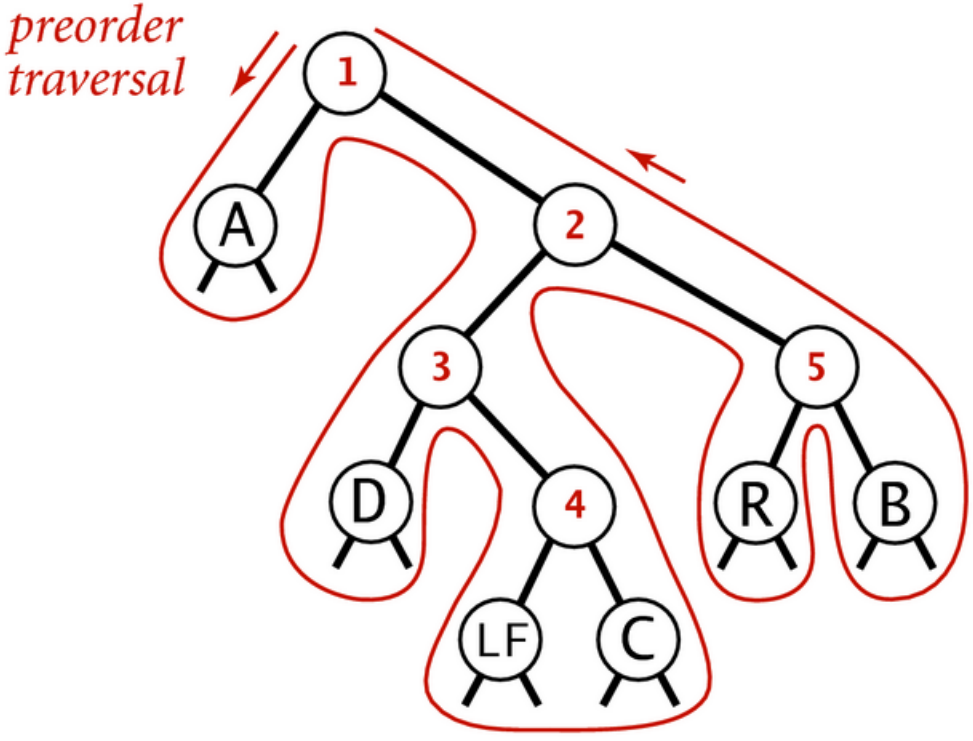
	código	caractere
0 1	01000001	A
0 0 1	01000100	D
0 1	00100001	!
1	01000011	C
0 1	01010010	R
1	01000010	B

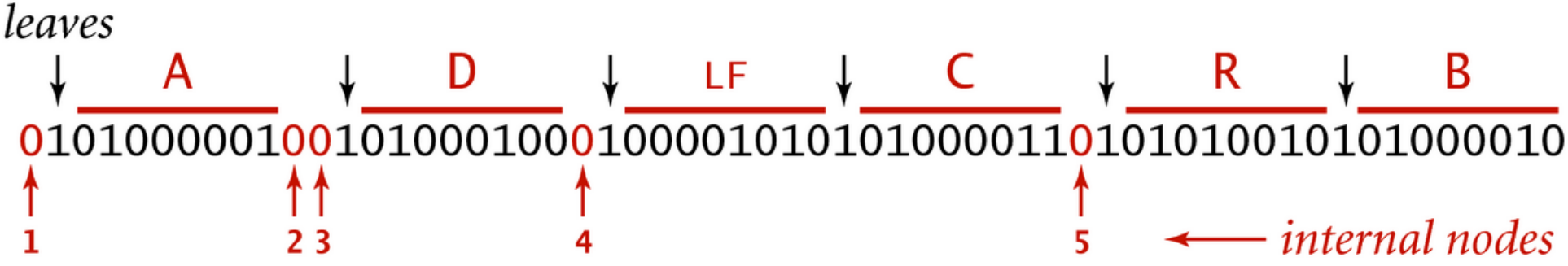
Codificação da `trie`

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`





Using preorder traversal to encode a trie as a bitstream

Codificação da trie

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

Codificação da `trie`

```
static Node readTrie() {
    if (readBoolean()) {
        char c = readChar();
        return newNode(c, 0, NULL, NULL);
    }
    return newNode('\0', 0, readTrie(), readTrie());
}
```

A frequência não é usada mais.

Compressão completa

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

```
void compress() {
   char *input = readString();
   /* aqui vai a construção de st[]...*/
   int *freq = mallocSafe(R*sizeof(int));
   for (int i = 0; i < strlen(input); i++)
      freq[input[i]]++;
   Node root = buildTrie(freq);
   char **st = buildCode(root);
```

Compressão completa

```
    writeTrie(root);
    write(strlen(input));
    for (int i=0; i<strlen(input); i++) {
        char *code = st[input[i]];
        for(int j=0; j<strlen(code); j++)
            if (code[j] == '1')
                write(true);
            else
                write(false);
    }
    close();
}
```

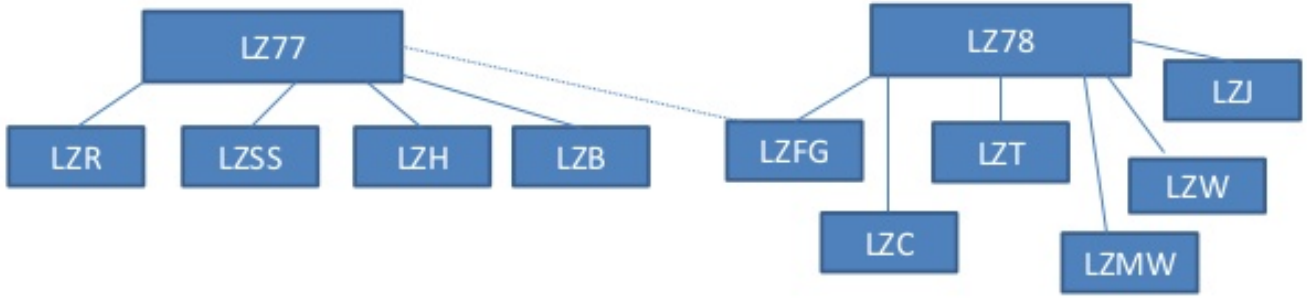
Esqueleto da biblioteca Huffman

Rotinas em Huffman.c:

```
static int R = 256; /* alfabeto */

typedef struct node *Node;
...                 /* rotinas de manipulação de nós */

void compress() {...}
void expand() {...}

static Node buildTrie(int *freq) {...}
static char **buildCode(Node root) {...}
static void writeTrie(Node x) {...}
static Node readTrie() {...}
```

Algoritmo LZW

Limpel Ziv Algorithm Family



APPLICATIONS:
- ZIP
- GZIP
- STACKER

APPLICATIONS:
- GIF
- V.42
- COMPRESS

Fig 1: Limpel Ziv Algorithm Family

Referências: Data Compression (SW), slides (SW), LZW compression, video (SW)

Algoritmo LZW

Desenvolvido por Abraham Lempel e Jakob Ziv em 1977 (LZ77).

Refinamento de LZ78 por Terry Welch em 1978.

Huffman é um método estatístico de compressão de dados.

LZW é um método baseado em dicionários (*dictionary based compression systems*).

Algoritmo LZW

Desenvolvido por Abraham Lempel e Jakob Ziv em 1977 (LZ77).

Refinamento de LZ78 por Terry Welch em 1978.

Huffman é um método estatístico de compressão de dados.

LZW é um método baseado em dicionários (*dictionary based compression systems*).

Diferentemente do algoritmo de Huffman, o algoritmo LZW transforma strings de tamanho variado em códigos de tamanho fixo de W bits (usualmente de 8 a 12).

Métodos baseados em dicionários

Não usam conhecimento estatístico dos dados.

`compress()`: a medida que a entrada é processada, constrói um dicionário e utiliza os índices das strings no dicionário como código.

Métodos baseados em dicionários

Não usam conhecimento estatístico dos dados.

`compress()`: a medida que a entrada é processada, constrói um dicionário e utiliza os índices das strings no dicionário como código.

`expand()`: a medida que a entrada é processada, reconstrói o dicionário para reverter o trabalho de `compress()`.

Exemplos: `LZW`, `LZ77`, Sequitur

Aplicações: Unix compress, `gzip`, GIF

LZW compress()

input = string de entrada

crie dicionário com símbolos do alfabeto

LZW compress()

input = string de entrada

```
crie dicionário com símbolos do alfabeto
repita
   encontre o maior prefixo s de input
             que está no dicionário
   transmita para saída o índice de s
```

LZW compress()

input = string de entrada

```
crie dicionário com símbolos do alfabeto
repita
   encontre o maior prefixo s de input
             que está no dicionário
   transmita para saída o índice de s
   ponha s+c no dicionário onde c é
         o símbolo que segue s em input
         (unmatched symbol).
   apague do input o prefixo s
até que input é vazio
```

LZW exemplo

Decisões de projeto:

- Alfabeto $= \{a, b\} \Rightarrow R = 2$
- $W = 3$ bits $\Rightarrow$ índices entre 0 e 7
- tamanho L do dicionário é $2^W = 8$
- índice R codificará EOF (end of file)

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação											

dicionário

string	código
a	0
b	1
EOF	2

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0										

dicionário

string	código
a	0
b	1
EOF	2
ab	3

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1									

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3								

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3		5						

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5
abab	6

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3		5			4			

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5
abab	6
bab	7

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3		5			4		1	

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5
abab	6
bab	7

LZW compress(): exemplo

input	a	b	a	b	a	b	a	b	a	b	
codificação	0	1	3		5			4		1	2

dicionário

string	código
a	0
b	1
EOF	2
ab	3
ba	4
aba	5
abab	6
bab	7

LZW compress(): exemplo

Resumo

Fomos de $8 \times 10 = 80$ bits para representar

a b a b a b a b a b

para W $\times 7 = 3 \times 7 = 21$ bits:

000 001 011 101 100 001 010

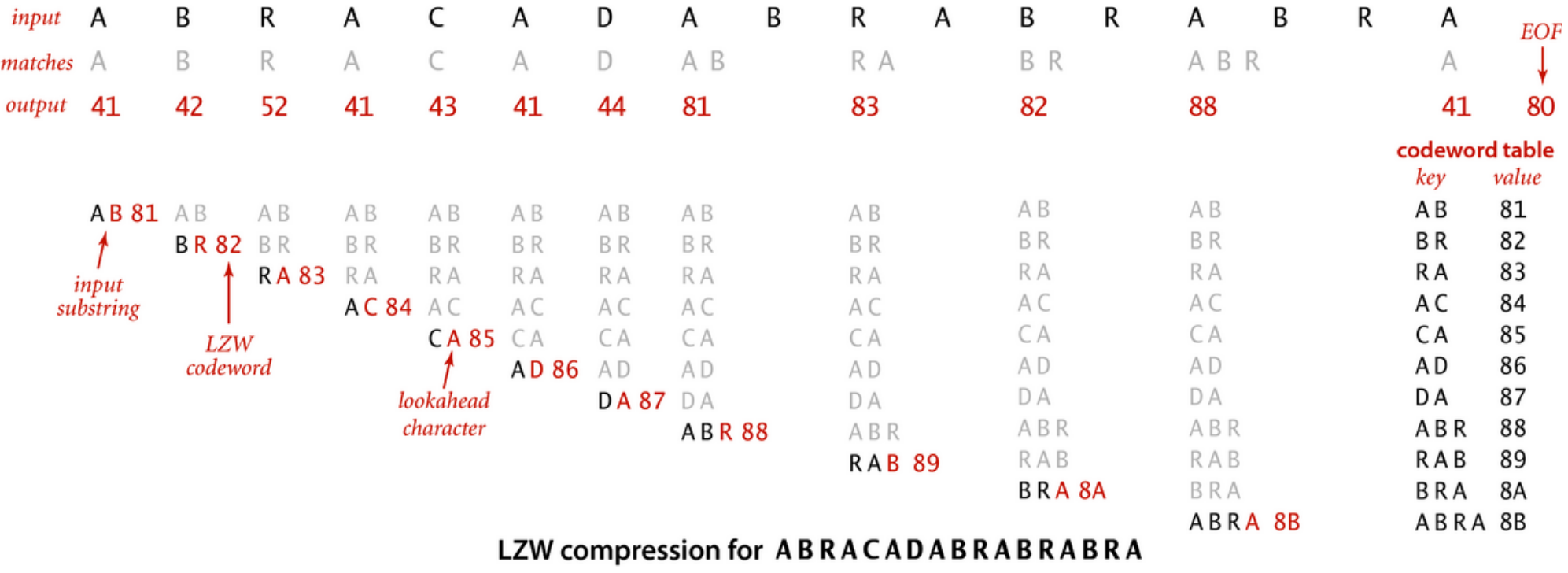
Taxa de compressão $= 21/80 = 0.2625 \sim 26\%$

LZW compress(): mais um exemplo

Decisões de projeto:

- Alfabeto = ASCII $\Rightarrow$ R = 128
- W = 8 bits $\Rightarrow$ índices entre 0 e 255
- tamanho L do dicionário é $2^W = 256$
- índice R = 128 (= 80 hexa) codificará EOF

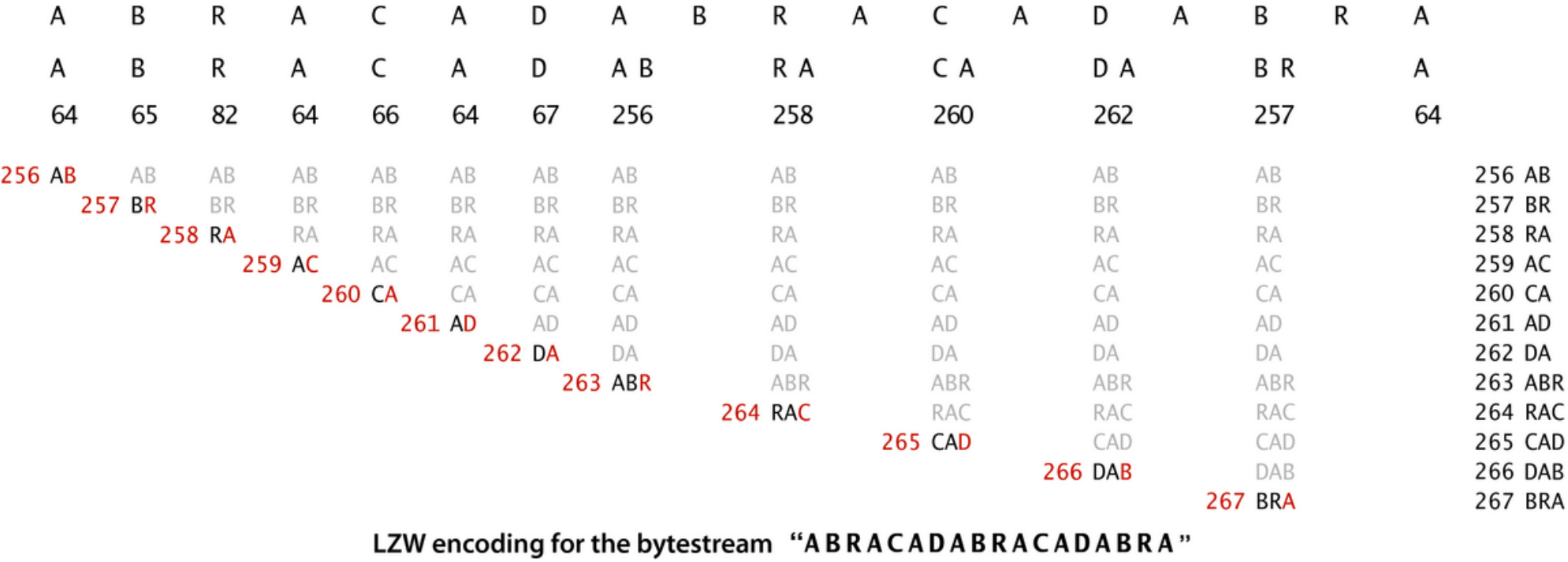
LZW compress(): mais um exemplo



LZW compression for A B R A C A D A B R A B R A B R A

Fonte: algs4

LZW compress(): mais um exemplo



LZW encoding for the bytestream "A B R A C A D A B R A C A D A B R A"

Fonte: algs4

Sobre o dicionário

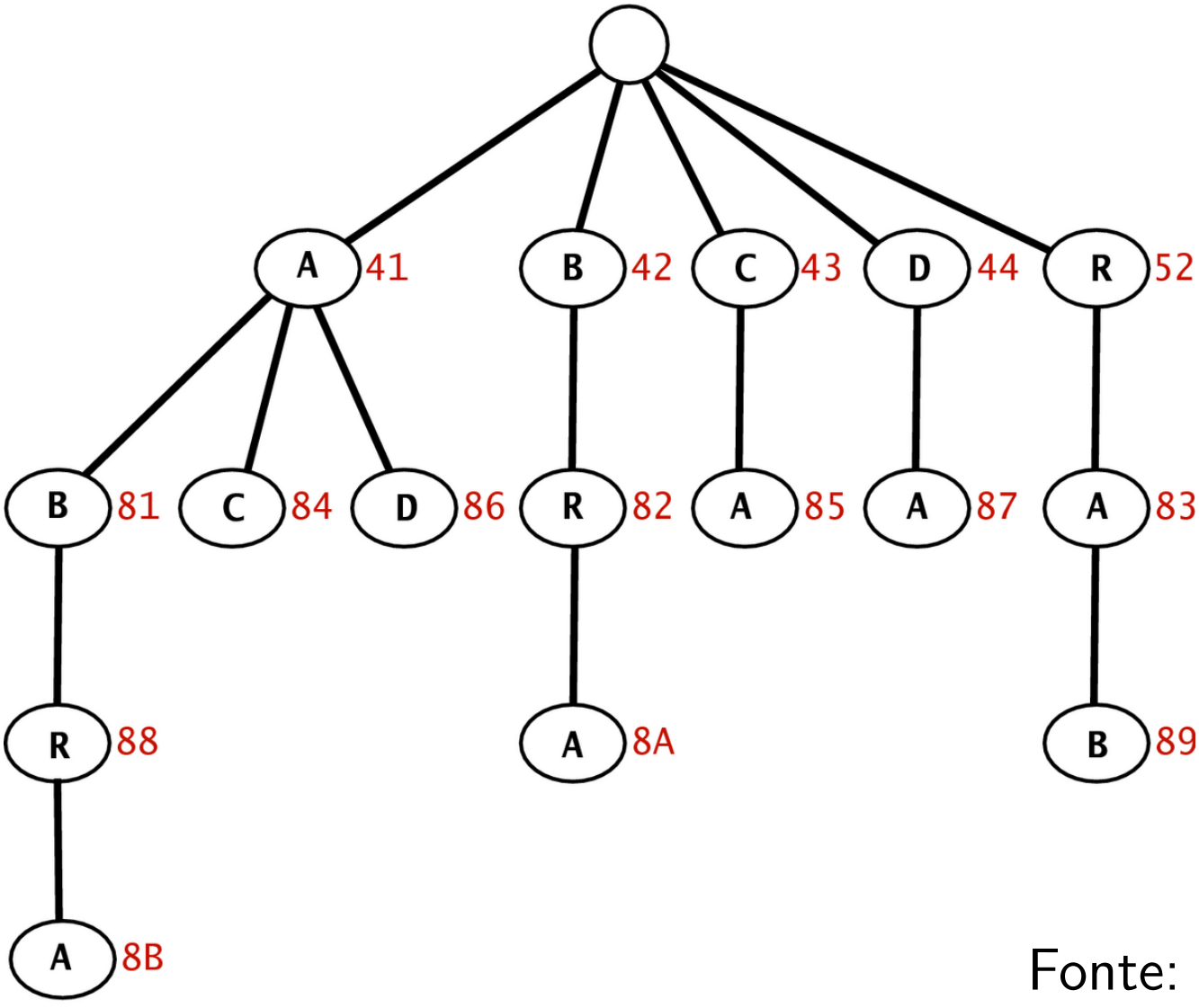
`Dicionário` de tamanho fixo:

- `W` bits de índices permite um dicionário de tamanho $2^{\texttt{W}}$;
- dificilmente todas as entradas no dicionário serão úteis.

Estratégias quando o dicionário atinge seu limite:

- não ponha mais strings,
  somente use o que já está no dicionário;
- jogue tudo fora e comece um novo dicionário;
- dobre o dicionário, acrescentando mais um bit aos índices ($\texttt{W} \leftarrow \texttt{W} + 1$);
- jogue fora as entradas mais recentes
  para fazer espaço para mais entrada;
- ideias?

Dicionário (Trie) de LZW



Fonte: algs4

Trie representation of LZW code table

compress()

Estatística dos dados não precisa ser passada por `compress()` para `expand()`.

string(`i`): devolve string de comprimento 1 com o caracter de código `i`

```
void compress() {
   char *input = readString();
   TST st = TSTInit();     /* trie ternária */
   for (int i = 0; i < R; i++)
      put(st, string(i), i);

   /* R é o código para EOF */
   int code = R+1;
```

compress()

```
  while (strlen(input) > 0) {
     char *s = longestPrefixOf(st, input);
     write(get(st, s), W);
     int t = strlen(s), n = strlen(input);
     if (t < n && code < L) {
        s = substring(input, 0, t+1);
        put(st, s, code++);
     }
     input = substring(input, t, n);
  }
  write(R, W);                          /* EOF */
  close();
}
```

LZW expand(): exemplo

msg codificada

0	1	3	5	4	1	2

output

dicionário

código	string

LZW expand(): exemplo

msg codificada

0	1	3	5	4	1	2

output

dicionário

código	string
0	a
1	b
2	EOF

LZW expand(): exemplo

msg codificada	0	1	3	5	4	1	2
output	a						

dicionário

código	string
0	a
1	b
2	EOF
3	a?

LZW expand(): exemplo

msg codificada	0	1	3	5	4	1	2
output	a	b					

dicionário

código	string
0	a
1	b
2	EOF
3	ab

LZW expand(): exemplo

msg codificada	0	1	3	5	4	1	2
output	a	b					

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	b?

LZW expand(): exemplo

msg codificada	0	1	3		5	4	1	2
output	a	b	a	b				

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba

LZW expand(): exemplo

msg código	0	1	3		5	4	1	2
output	a	b	a	b				

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	ab?

LZW expand(): exemplo

msg código	0	1	3		5	4	1	2
output	a	b	a	b				

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	ab?
	Xiii!

LZW expand(): exemplo

msg código	0	1	3		5			4	1	2
output	a	b	a	b	a	b	?			

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	ab?
	Xiii!

LZW expand(): exemplo

msg código	0	1	3		5			4	1	2
output	a	b	a	b	a	b	a			

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
	:-)

LZW expand(): exemplo

msg código	0	1	3		5			4	1	2
output	a	b	a	b	a	b	a			

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	aba?

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a		

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	aba?

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a		

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	abab

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a		

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	abab
7	ba?

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a	b	

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	abab
7	ba?

LZW expand(): exemplo

msg código	0	1	3		5			4		1	2
output	a	b	a	b	a	b	a	b	a	b	

dicionário

código	string
0	a
1	b
2	EOF
3	ab
4	ba
5	aba
6	abab
7	bab