MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Fonte: ash.atozviews.com

AULAS 19 e 24

BFS versus DFS

- busca em largura usa **fila**, busca em profundidade usa **pilha**.
- a busca em largura é descrita em **estilo iterativo**, enquanto a busca em profundidade é descrita, usualmente, em **estilo recursivo**.
- busca em largura começa tipicamente num **vértice especificado**, a busca em profundidade, o próprio **algoritmo escolhe o vértice** inicial.
- a busca em largura apenas **visita os vértices que podem ser atingidos** a partir do vértice inicial, a busca em profundidade, tipicamente, **visita todos os vértices** do digrafo.

DFS aplicações

- caminhos x cortes: `DFSpaths`
- ciclos x ordenação topológica (DAGs): `DFStopological`
- componentes conexos: `DFScc`
- grafos bipartidos x ciclos ímpares: `DFScc`
- componentes fortemente conexas: `DFSscc`
- ...

BFS aplicações

- caminhos x cortes: `BFSpaths`
- componentes conexos
- caminhos comprimento mínimo: usa `Queue`
- caminhos de custo/peso mínimo
  - `Dijkstra`: **custo não negativos**, usa `MinPQ`
  - `AcyclicSP`: custos quaisquer, usa `DFStopological`
  - ...
- árvores de custo mínimo/máximo: `PrimMST`

Comentar o 8 Puzzle de COS226 e sua busca A^*.

AULA 25

Compressão de dados

Fonte: `Best VPN for Data Compression`

Referências: Entrada e saída binárias (PF), Compressão de dados (PF), Data Compression (SW), slides (SW), video (SW)

Introdução

Problema: representar um arquivo GRANDE por outro menor.

Exemplo:
arquivo GRANDE: `abababababababababababab`
arquivo menor: `12ab`

Por que comprimir?
Menor espaço de armazenamento.
Menor tempo de transmissão.

Software: `gzip`, `bzip`, `7z`, etc.

Esquema

Problema: 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

Codificador e decodificador

Fluxo `B` é original e o fluxo `C(B)` é codificado.

Fluxo produzido pelo expansor é decodificado.

Fluxo decodificado é idêntico ao original, a compressão não perde informação (*lossless compression*).

Notação: `|B|` é o número de bits de `B`.

Taxa de compressão: `|C(B)|/|B|`.

Desafio:
obter a menor taxa de compressão possível.

Considerações teóricas

Fato. Nenhum algoritmo pode garantir taxa de compressão estritamente menor que 1 para todo e qualquer fluxo de bits.

Fluxos de bits aleatórios são pouco compressíveis.

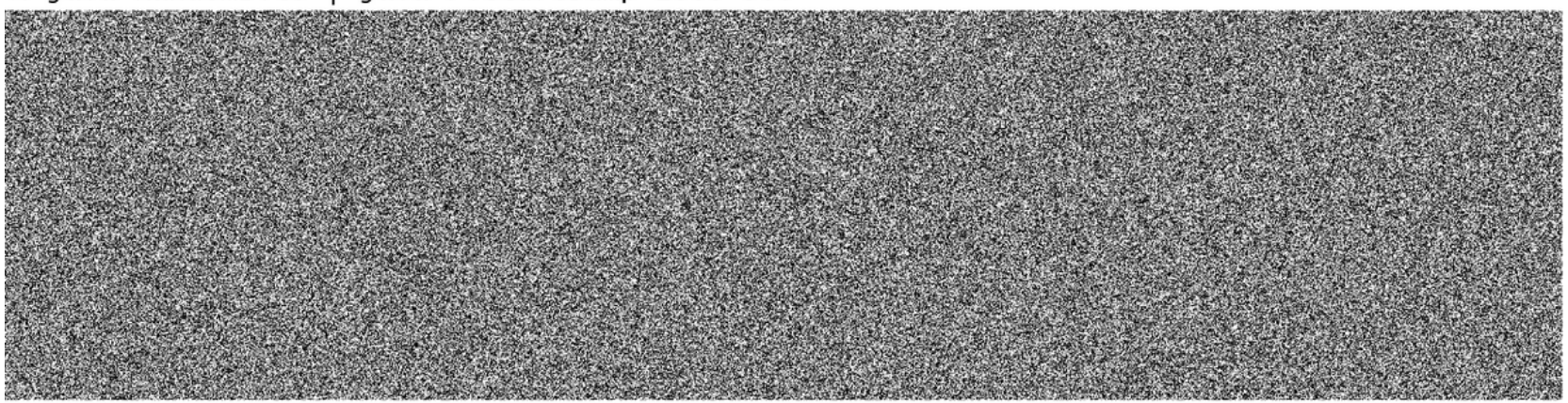
Mesmo fluxos pseudo-aleatórios podem ser difíceis de comprimir.

Considerações teóricas

Muitos fluxos de bits parecem aleatórios
mas não são.

Exemplo: o fluxo de bits parece aleatório. . .

```
% java RandomBits | java PictureDump 2000 500
```



```
1000000 bits
```

A difficult file to compress: 1 million (pseudo-) random bits

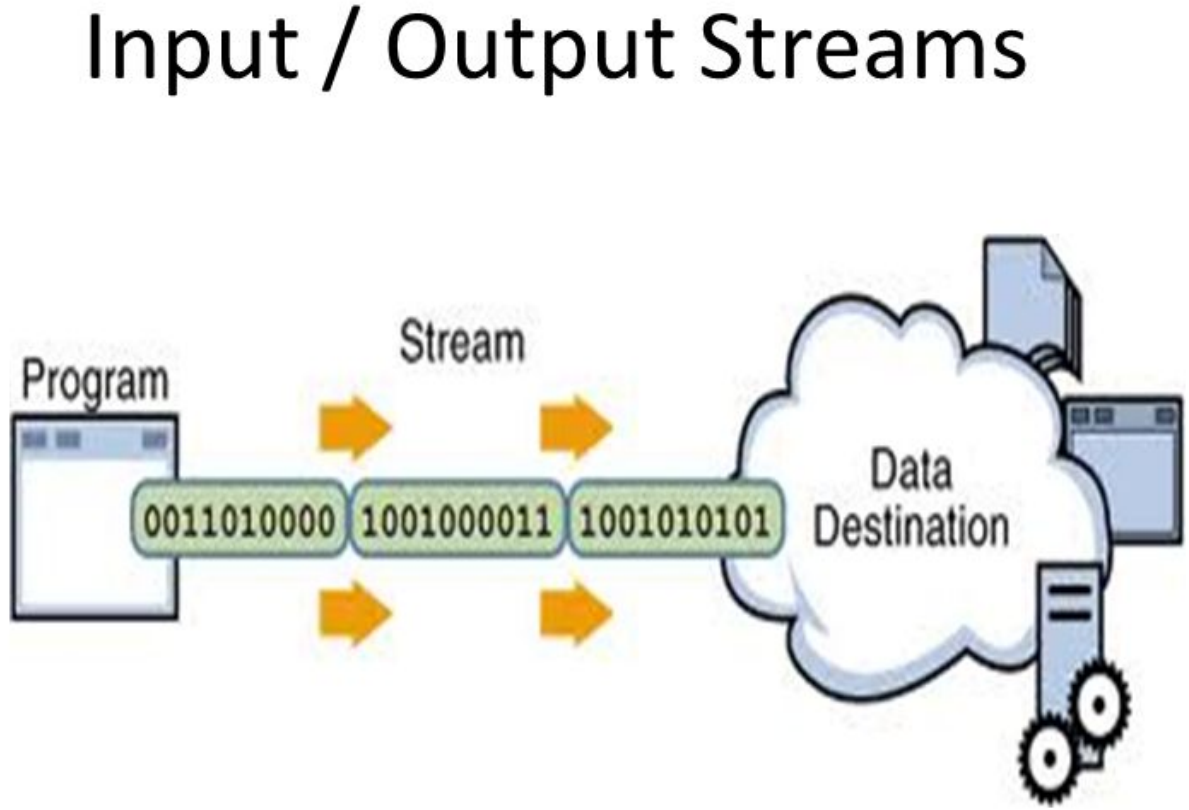
Considerações teóricas

mas foi produzido pelo código em Java

```
public class RandomBits {
    public static void main(String[] args){
    int x = 11111;
    for(int i = 0; i < 1000000; i++) {
        x = x * 314159 + 218281;
        BinaryStdOut.write(x>0);
    }
    BinaryStdOut.close();}
}
```

Taxa de compressão: 0.002

Entradas e saídas binárias



Fonte: `Input / Output Streams Byte streams`

Referências: BinaryStdIn, BinaryStdOut, BinaryIn, BinaryOut

Cadeia de bits e fluxo de bits

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

```
1100010000011101010100111110110011100010100001100000000001111110000011101000111
```

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

A biblioteca `BinaryStdIn` permite a leitura de um fluxo de bits a partir da entrada padrão.

A biblioteca `BinaryStdOut` permite escrever um fluxo de bits na saída padrão.

Biblioteca BinaryStdIn

Para simplificar, demos o mesmo nome para algumas rotinas.

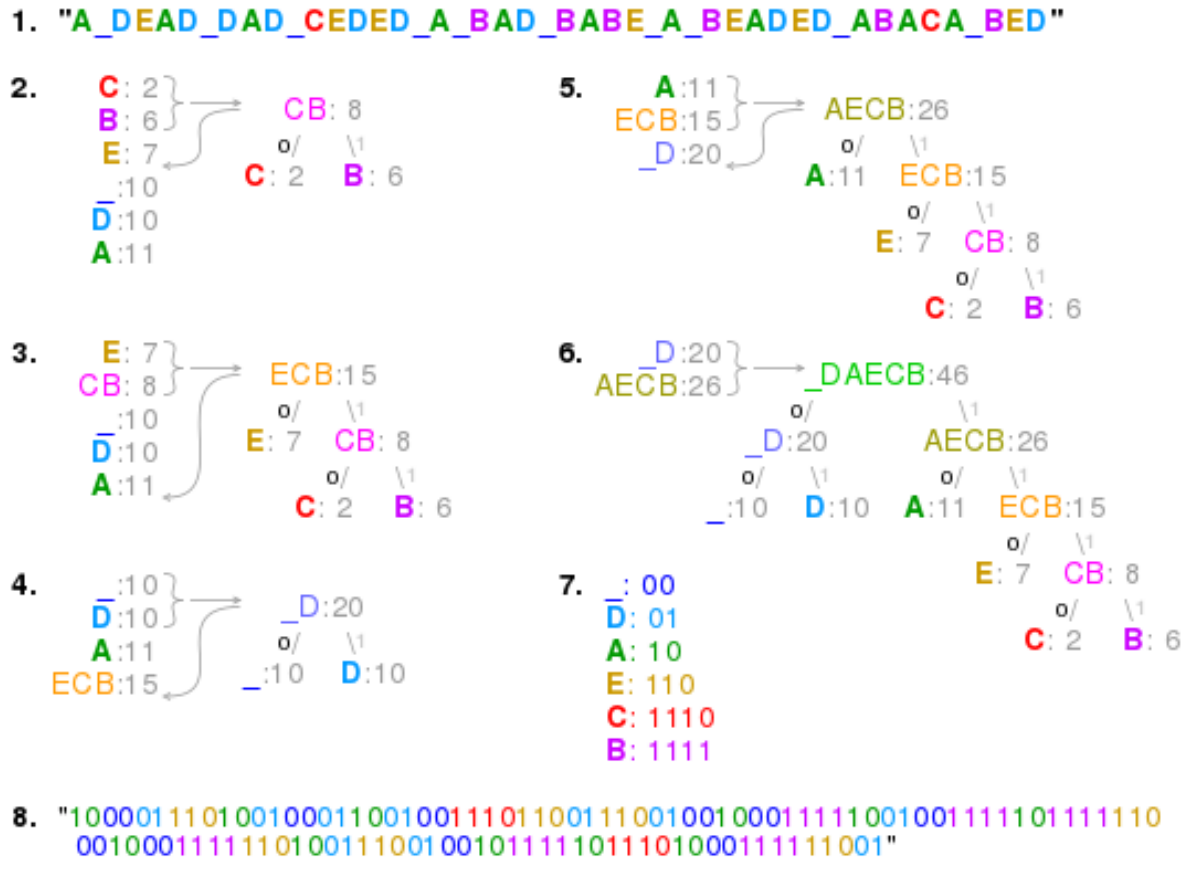
Arquivo	BinaryStdIn.h	
`bool`	`readBoolean();`	lê 1 bit e devolve a boolena correspondente
`char`	`readChar();`	lê 8 bits
`char`	`readChar(int r);`	lê `r` (entre 1 e 8) bits
`char *`	`readString();`	lê fluxo em blocos de 8 bits
`int`	`readInt();`	lê 32 bits
`int`	`readInt(int r);`	lê `r` (entre 1 e 32) bits
`bool`	`isEmpty();`	fluxo está vazio?
`void`	`close();`	feche o fluxo

Biblioteca BinaryStdOut

Para simplificar, demos o mesmo nome para algumas rotinas.

Arquivo	BinaryStdOut.h	
void	write(bool b);	escreve o bit representado por b
void	write(char c);	escreve 8 bits da representação de c
void	write(char c, int r);	escreve os r (1 a 8) bits de c menos significativos
void	close();	feche o fluxo

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

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

Exemplo: transforma `ABRACADABRA!` em `01111111001100100011111111100101`.

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 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

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

para codificar o arquivo.

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

Com essa nova tabela de códigos, gastaríamos

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

bits $=$ 224000 bits para representar o arquivo.

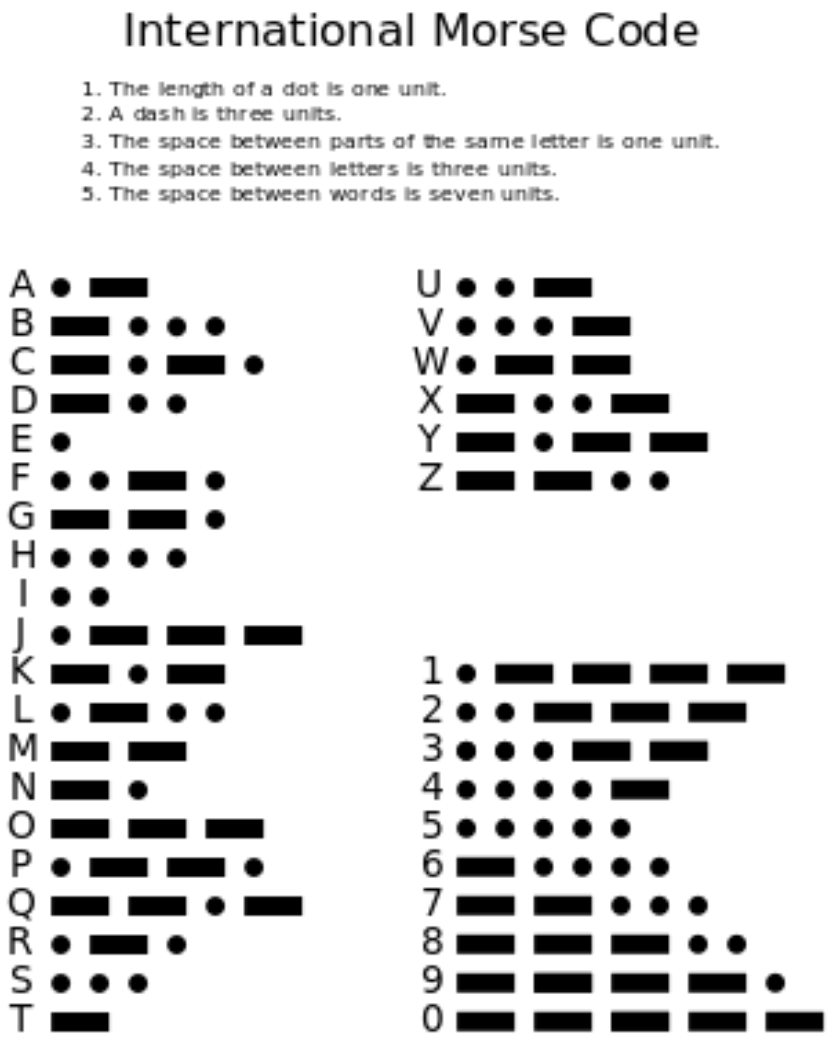
Compressão ou codificação

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



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

```
void compress() {
   char *input = readString();
   /* aqui vai a construção de st[]... */
   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();
}
```

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 ABRACADABRA! em
01000010100100011.

Expansão (decodificação)

Exemplo: a tabela de códigos

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

Transforma ABRACADABRA! em
01000010100100011.
01000010100100011 também representa
CRRDDCRCB 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

01111111001100100011111111100101.

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.

Tabelas inversas

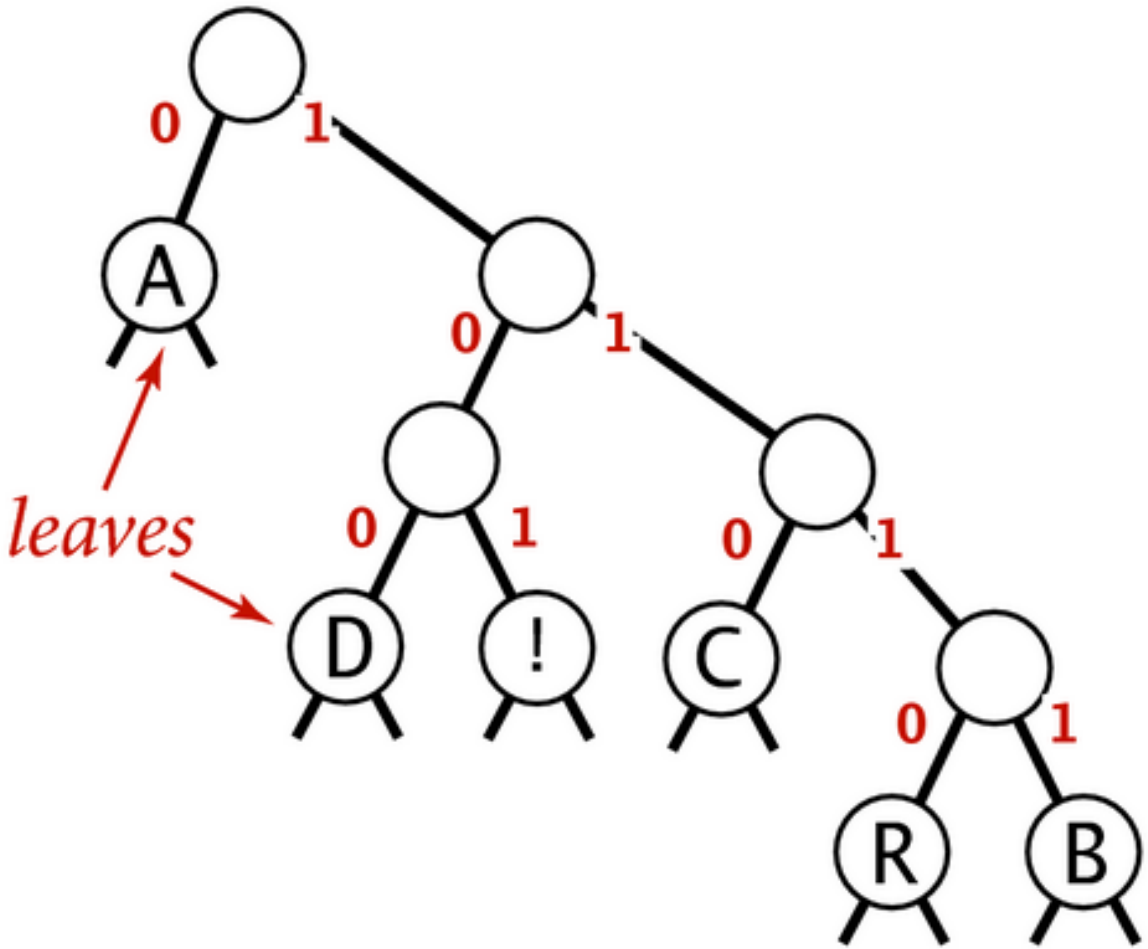
código	caractere
101	!
0	A
1111	B
110	C
100	D
1110	R

A tabela será representada por uma trie binária. 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

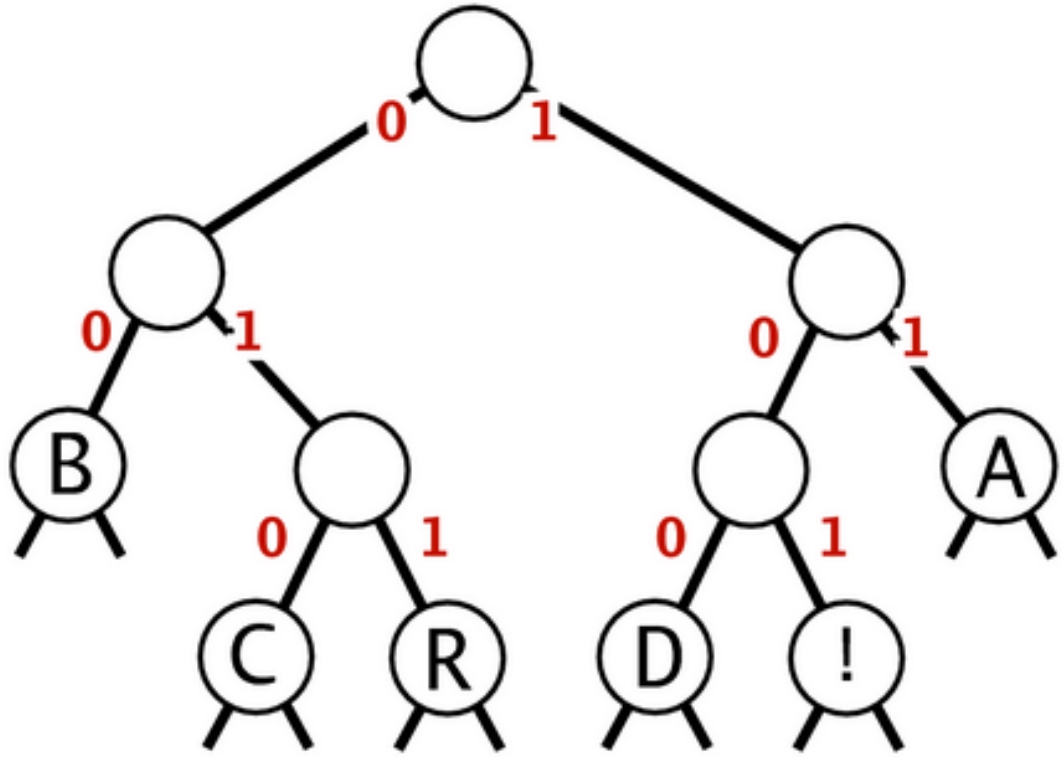
0111111100110010001111111100101 ← *30 bits*

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 !

Nós de uma trie

```
typedef struct node *Node;

struct node {
   /* usado só nas folhas */
   char ch;
   /* usado só para construção da trie */
   int freq;
   Node left;
   Node right;
}
```

Nós de uma trie

```
Node newNode(char ch, int freq,
             Node left, Node right) {
   Node p = mallocSafe(sizeof(*p));
   p->ch = ch;
   p->freq = freq;
   p->left = left;
   p->right = right;
   return p;
}
```

Nós de uma trie

```
bool isLeaf(Node p) {
   return p->left == NULL
        && p->right == NULL;
}

int compareTo(Node this, Node that) {
   return this->freq - that->freq;
}
```

Decodificação

```
void expand() {
    Node root = readTrie();
    int n = readInt();
    for (int i = 0; i < n; i++) {
```

Decodificação

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

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	

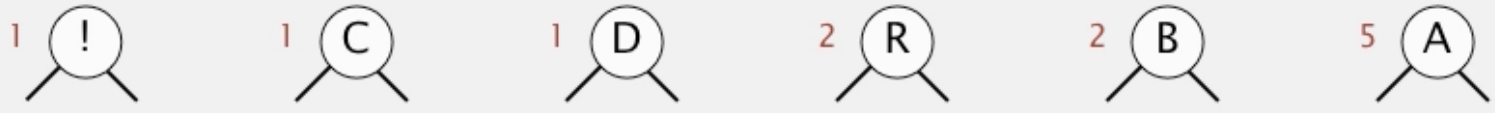
input

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.

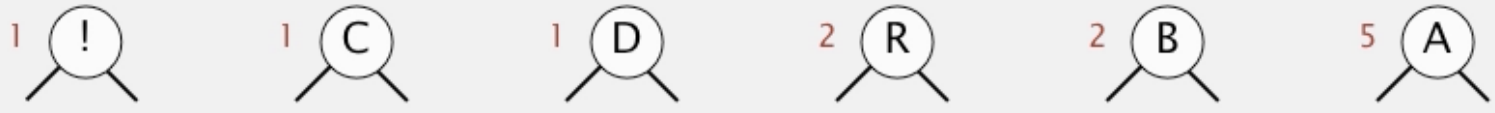
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman coding demo

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

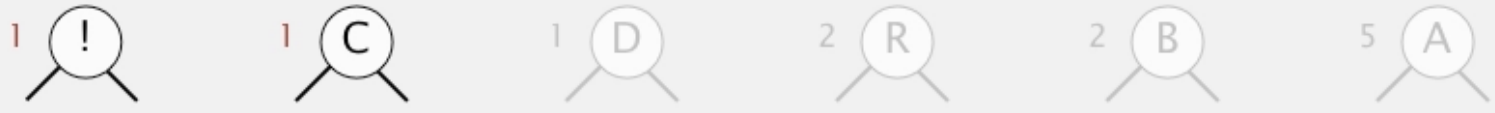
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



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

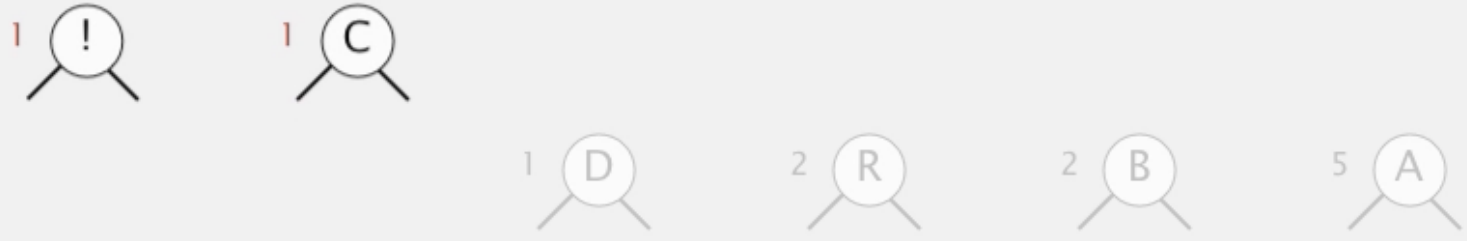
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

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.

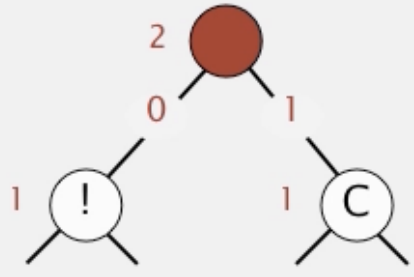
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	

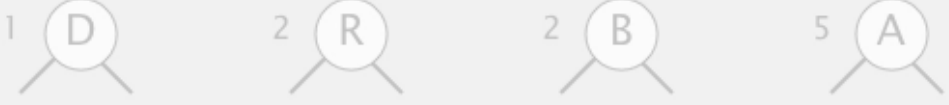


Huffman coding demo

- 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

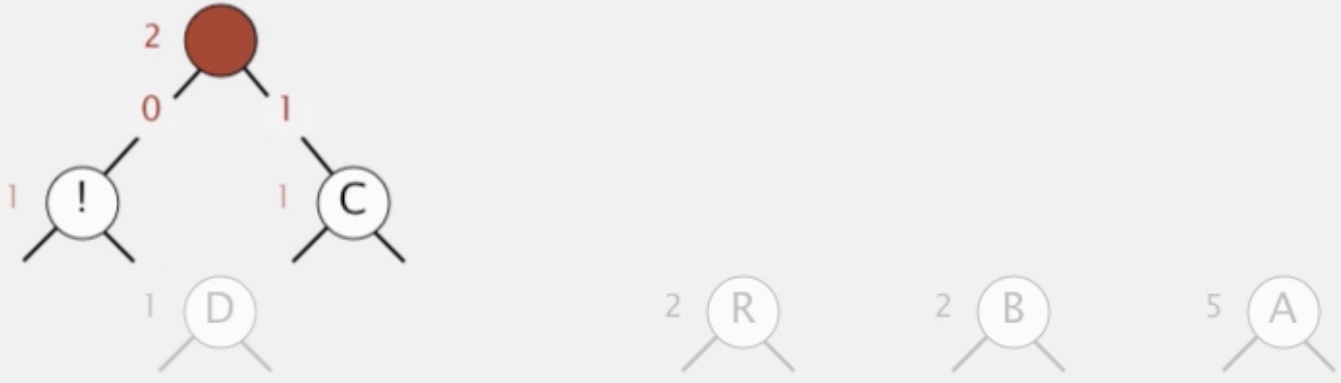




Huffman coding demo

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

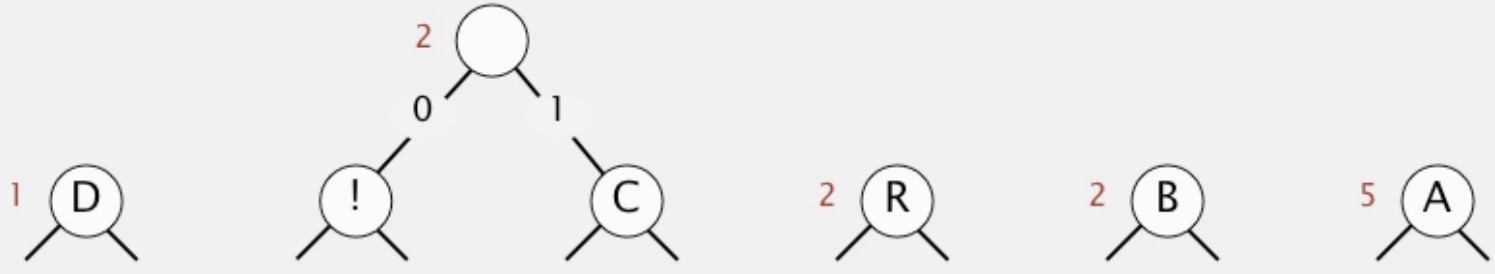
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

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

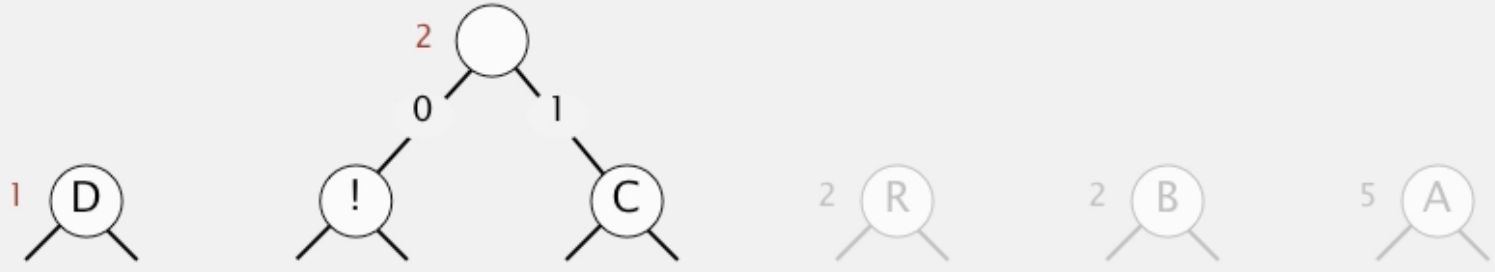
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

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

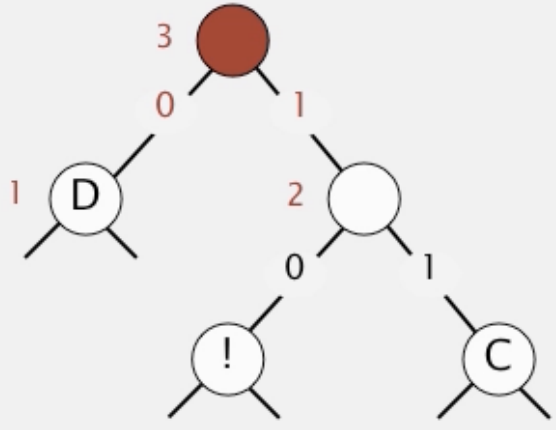
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman coding demo

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

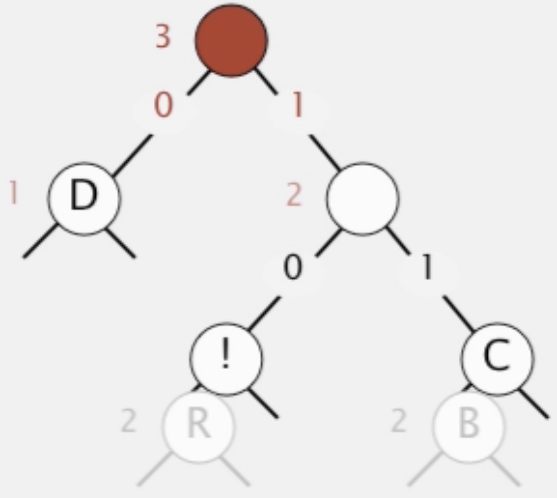
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

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

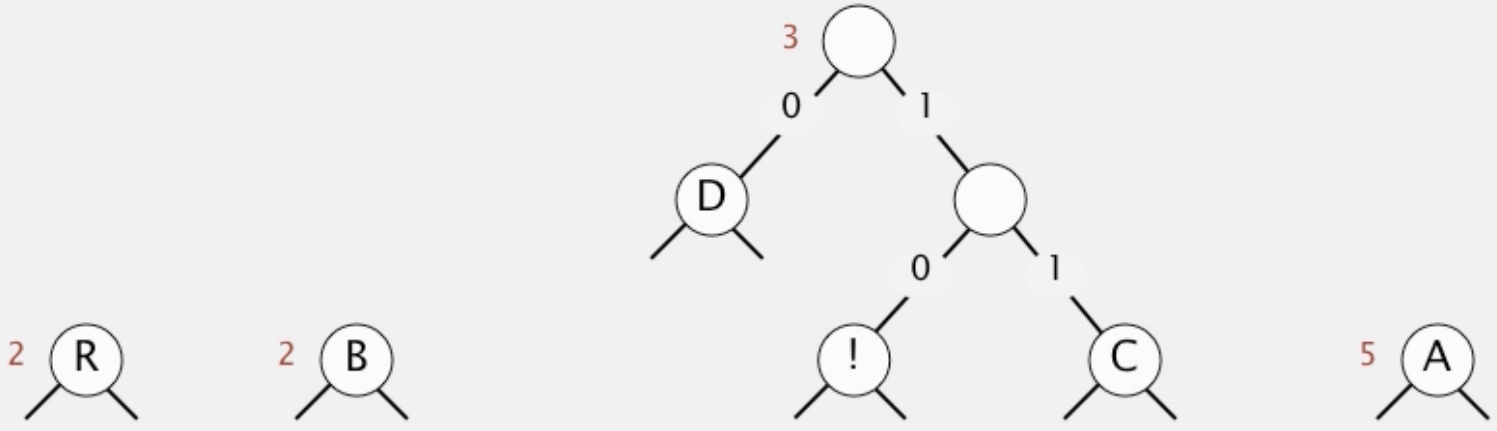
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

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

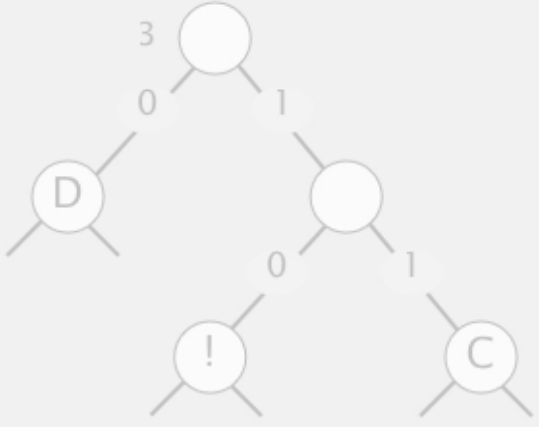
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman coding demo

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

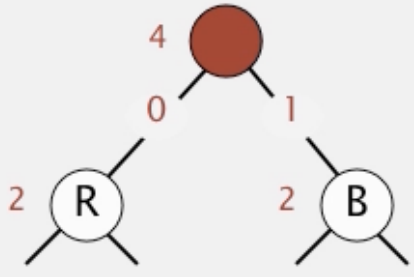
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



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

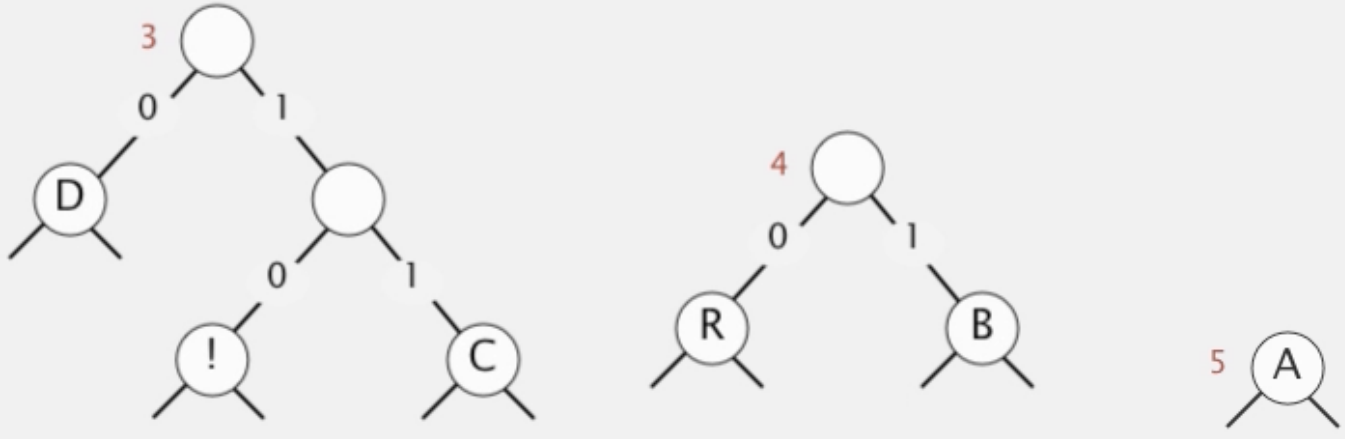
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

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.

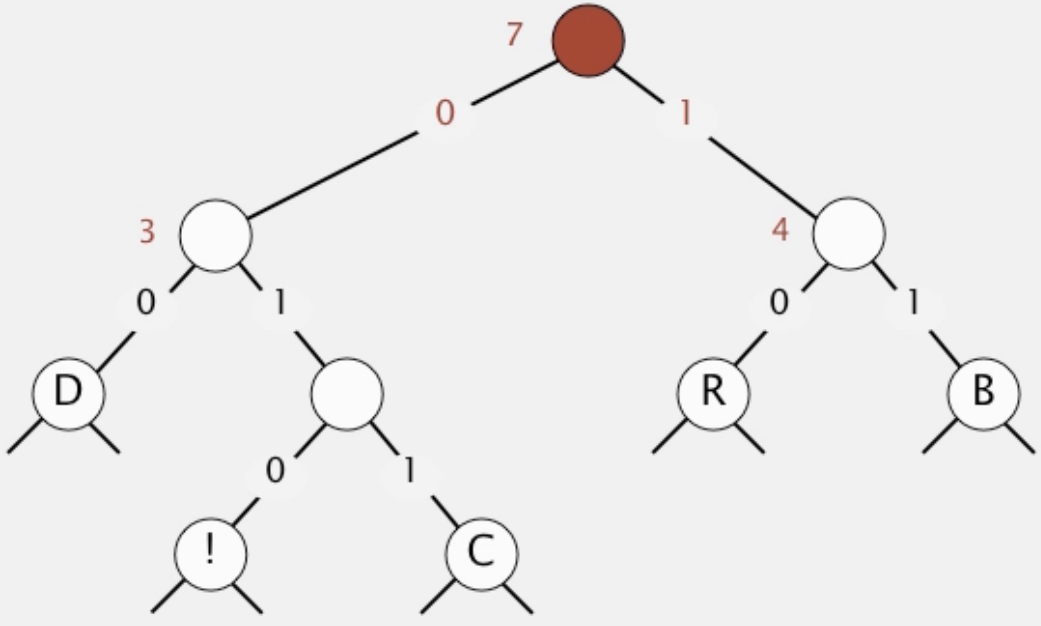
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman coding demo

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

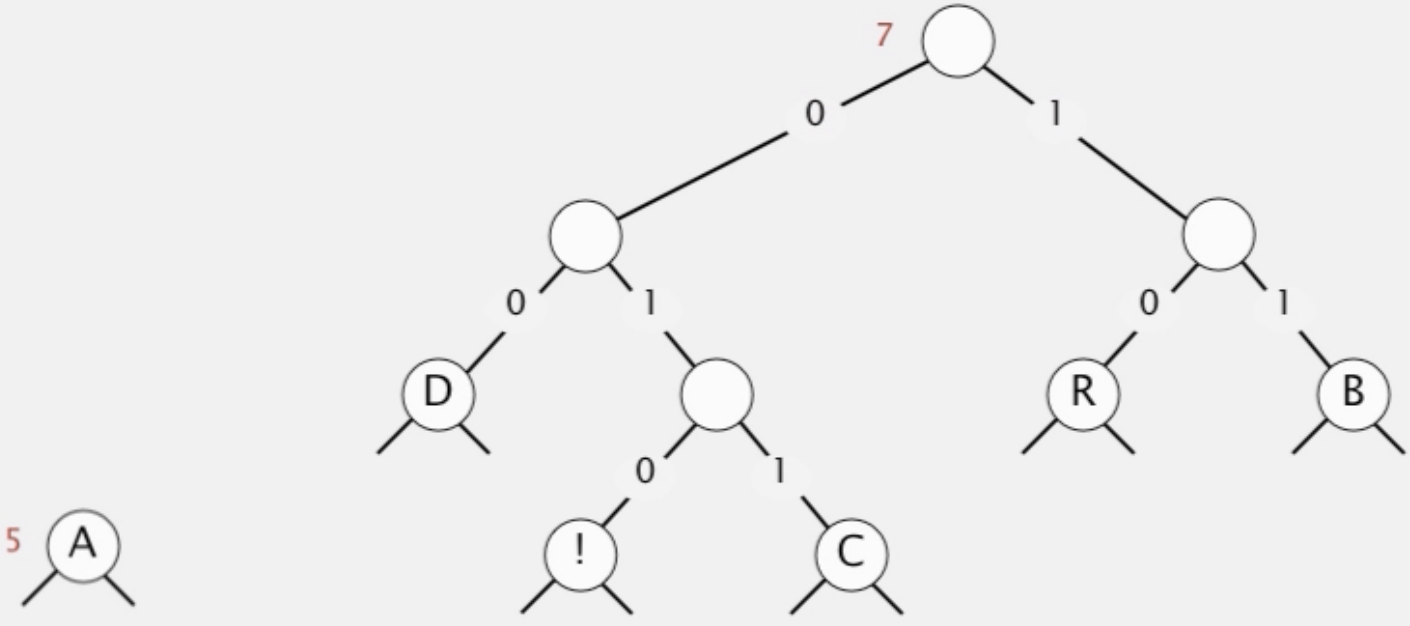
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



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