CI1057: Algoritmos e Estruturas de Dados III
Compressão de Dados

Profa. Carmem Hara

Departamento de Informática/UFPR

8 de julho de 2024

Compressão de Dados

- Técnica para representar os dados originais em menos espaço
- Idéia: substituir a forma de representação dos símbolos por outra com um menor número de bits ou bytes.

Vantagens:

- Menor espaço de armazenamento
- Menor tempo para leitura do disco e para transmissão

Desvantagens:

- Custo de codificação e decodificação

Exemplo

Considere um texto com 100.000 caracteres no intervalo $[a, f]$.

- Se cada caracter for representado por 1 byte (8 bits)
  Espaço total = 100.000 * 8 = 800.000 bits
- Para 6 caracteres distintos são necessários apenas 3 bits
  a: 000, b: 001, c: 010, d: 011, e: 100, f: 101
  Espaço total = 100.000 * 3 = 300.000 bits

É possível melhorar levando em consideração a frequência de cada letra no texto e utilizando um codificação de tamanho variável.

Codificação de Tamanho Variável

Idéia: Associar códigos menores para os caracteres mais frequentes

	a	b	c	d	e	f
frequência (*1000)	45	13	12	16	9	5
codificação fixa	000	001	010	011	100	101
codificação variável	0	101	100	111	1101	1100

- tamanho total com codificacao fixa:
  100.000 * 3 = 300.000
- tamanho total com codificacao variavel:
  45.000*1 + 13.000*3 + 12.000*3 + 16.000*3 + 9.000*4 + 5.000*4 = 224.000

Razão de Compressão

Razão de Compressão: porcentagem que o arquivo comprimido representa em relação ao tamanho do arquivo não comprimido.

- No exemplo: 224.000 / 300.000 = 0.75 = 75%
- Ganho de 25% no espaço de armazenamento

Codificação de um Texto

	a	b	c	d	e	f
Código	0	101	100	111	1101	1100

- fada: 1100.0.111.0
- cabeca: 100.0.101.1101.100.0

Custo da codificação de ums texto T:

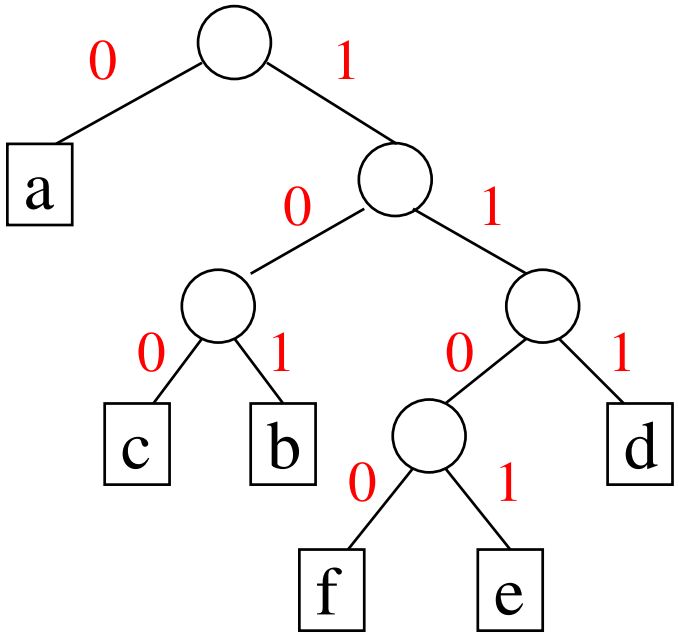
$$C(T) = \sum_{c \in T} freq(c) * tam(c)$$

$freq(c)$ é a frequência do caracter c no texto e
$tam(c)$ é o tamanho da codificação de c (em bits)

Exemplo: $C(cabeca) = 2*1(a) + 1*3(b) + 2*3(c) + 1*4(e) = 15$

Separadores são necessários?

- A codificação pode ser representada por uma trie, na qual os caracteres são as chaves.

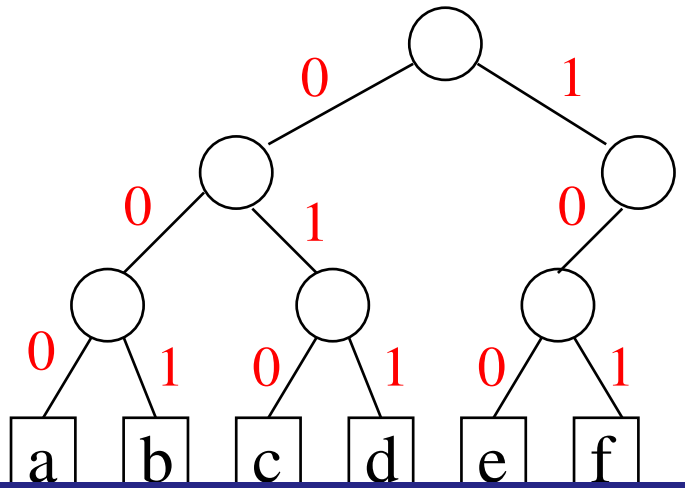


Como *decodificar* a sequência: **100010111011000** ?

Observações

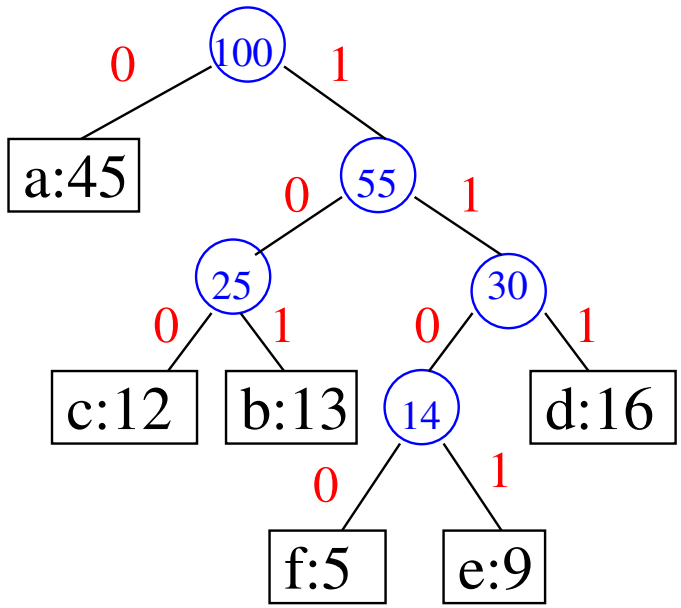
- Tries nas quais todas as chaves são folha formam **códigos de prefixo**
  $\rightarrow$ Nenhum código é prefixo de outro
- Uma codificação ótima é *sempre* representada por uma árvore binária **cheia**
  $\rightarrow$ É uma árvore binária na qual todos os nodos internos tem ou zero ou 2 filhos que são nodos internos.

Exemplo de árvore binária que **NÃO** é cheia:



Codificação de Huffman

- Constrói uma codificação de prefixo ótima baseada nas frequências
- O resultado é uma trie cheia
- Para $|C|$ caracteres:
  - $|C$ nodos folhas
  - $|C| - 1$ nodos internos



Implementação

- Nodos folha tem uma letra e uma frequencia
- Nodos internos não folha tem apenas a frequencia

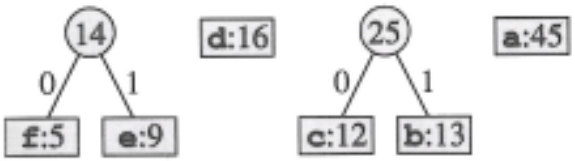
```
1  typedef struct Nodo *ApNodo;
2  typedef struct Nodo {
3    char letra;
4    int freq;
5    ApNodo esq, dir;
6  } Nodo;
7
8  typedef struct freqC {
9    char letra;
10   int freq;
11 } freqC;
12
13 typedef struct codBin{
14   int tam;
15   unsigned cod;
16 } codBin;
```

Geração dos Códigos

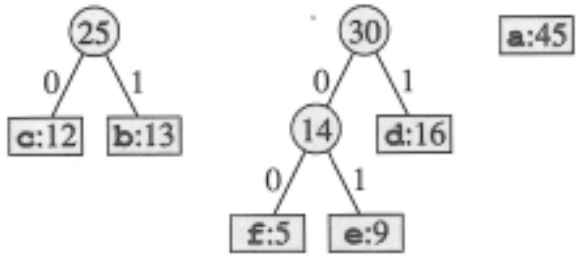
```
1  ApNodo Huffman( freqC c[], int qtdC ){
2    int i;
3    ApNodo p, p1, p2;
4    FilaPrioridade Q;
5
6    iniciaFila( &Q );
7    for( i=0; i<qtdC; i++ ){
8      p = criaNodo( c[i].letra, c[i].freq, NULL, NULL );
9      insereFilaPrioridade( &Q, p );
10   }
11   for( i=0; i<qtdC-1; i++ ){
12     p1 = extraiMinimo( &Q );
13     p2 = extraiMinimo( &Q );
14     p = criaNodo( ' ', p1->freq + p2->freq, p1, p2 );
15     insereFilaPrioridade(&Q, p);
16   }
17   p= extraiMinimo( &Q );
18   freeFila( &Q );
19   return p;
20 }
```

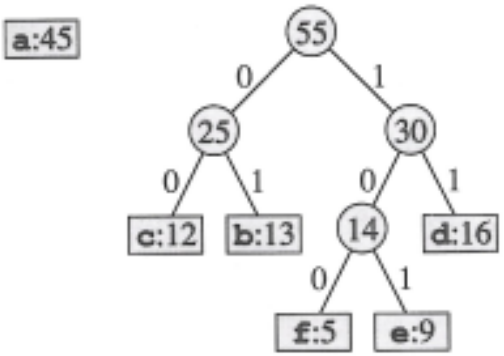
Exemplo de Geração

(b)

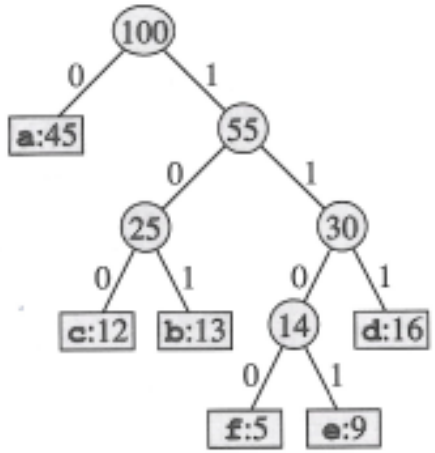


(d)





(f)



Codificação do Texto

```
1  void codifica( FILE *fin, FILE *fout, ApNodo raiz, int qtdChar ){
2    int ch;
3    codBin *cod;
4
5    escreveHeader( fout, raiz, qtdChar );
6    cod= (codBin*) malloc(qtdChar * sizeof(codBin));
7    geraVetorCod( raiz, cod, qtdChar );
8    ch=fgetc(fin);
9    while( ch != EOF ){
10     escreveCodigo( fout, cod, ch );
11     ch=fgetc(fin);
12   }
13   free( cod );
14 }
```

Vetor cod

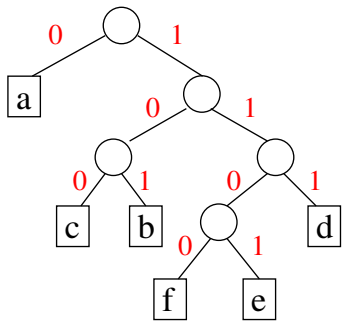
	[a]	[b]	[c]	[d]	[e]	[f]
cod	0	101	100	111	1101	1100
tam	1	3	3	3	4	4

Formato do Arquivo Codificado

qtd. de caracteres	tamanho da trie	tamanho do texto	*trie*	*texto codificado*

Representação da trie:

- Percurso da trie em pós-ordem:
  - Nodo folha: 1 seguido do caracter
  - Nodo não folha: 0



Representação: **1a1c1b01f1e01d000**

Tamanho: 17 bytes = 2* quant.caracteres + (qtd.caracteres - 1)

Geração do Vetor cod

```
1  /* gera o vetor cod indexado pelo caracter */
2  void geraVetorCod( ApNodo raiz, codBin *cod, int qtdChar ){
3    unsigned cc=0; int tamcc=0;
4
5    geraVetorR( raiz, cod, &cc, &tamcc );
6    return;
7  }
8
9  void geraVetorR( ApNodo p, codBin cod[], unsigned *cc, int *tamcc ){
10   if( ehFolha( p )){
11     cod[ind(p->letra)].cod= *cc;
12     cod[ind(p->letra)].tam= *tamcc;
13   } else {
14     *cc= *cc << 1;                  /* coloca zero no final do codigo */
15     (*tamcc)++;
16     geraVetorR( p->esq, cod, cc, tamcc );
17
18     *cc= *cc + 1;                   /* coloca um no final do codigo */
19     geraVetorR( p->dir, cod, cc, tamcc );
20     *tamcc--;
21     *cc = *cc >> 1;
22   }
23 }
```

Decodificação

```
1  void decodifica( FILE *fin, FILE *fout ){
2    ApNodo raiz, p;
3    int bit;
4
5    leHeader( fin, raiz );  /* constroi a trie */
6    p= raiz;
7    bit = leBit( fin );
8    while( bit != EOF ){
9      if( bit == 0 )
10       p= p->esq;
11     else
12       p= p->dir;
13     if( ehFolha( p )){
14       escreveArq( fout, p->letra );
15       p= raiz;
16     }
17     bit= leBit( fin );
18   }
19 }
```

Mais um Exemplo

Codificação para o texto: "ABRACADABRA ALAKAZAM"

Custo da Construção dos Códigos de Huffman

- depende do tempo para obter os caracteres em ordem ascendente de frequencia e inserir novos elementos no conjunto
- Com lista ordenada pela frequência
  - ordenação: $O(nlg(n))$
  - inserção $n * (n-1) : O(n^2)$
- Alternativa: árvore balanceada AVL ou Rubro-Negra
  - Mas os elementos não precisam estar completamente ordenados.
  - Basta obter o valor mínimo de forma eficiente
- Alternativa melhor: **heap** para a implementação da lista de prioridades

  Uma arvore está *em ordem min-heap* se a chave em cada nodo é menor ou igual às chaves armazenadas em todos os seus filhos.

Referências

- Livro Cormen Cap. 17 (Seção 17.3)