Hashing

SCC201/501/601 - Introdução à Ciência de Computação II

Prof. Moacir Ponti

www.icmc.usp.br/~moacir

Instituto de Ciências Matemáticas e de Computação – USP

com contribuições de Paulo H. Gabriel e Pamela Candida Cortez

2020/2

Sumário

Busca, inserção e remoção em vetores

- Métodos de busca estudados até agora: Comparação de chaves
- Tiramos proveito da ordenação

Custos em vetores

Ordenação: $O(n \log_2 n)$, ou $O(n)$ em casos especiais
Busca: $O(\log_2 n)$ ou $O(\log_2(\log_2 n))$ em casos especiais

Custos após realizar outras operações

Inserção ?
Remoção ?

Tabela *Hash* ou de Espalhamento ou de Dispersão

- Função Hash + Tabela Hash redefine o problema de comparação de valor para um mapa;
- Atinge a posição da chave em tempo constante;

Em outras palavras. . .

Potencial para busca em $O(1)$

Sumário

Conceitos (1/5)

Ideia Central

Utilizar uma função, aplicada a informação (*i.e.* chave), que calcule o índice onde a informação deve (deveria) ser armazenada.

- A função é chamada **função de espalhamento** (ou função *hash*)
- A estrutura de dados é a **tabela de espalhamento** (ou tabela *hash*)

Requer: projetar tabela e função adequada para aproximar $O(1)$

Conceitos (2/5)

Função de Espalhamento

Uma função de espalhamento $h(.)$ transforma uma chave k em um endereço-base $h(k)$ da tabela hash.

- Função hash: não há garantias que seja uma função injetora
- Na prática:
  - pode existir $k \neq p$ tal que $h(k) = h(p)$
  - e se $h(k)$ estiver ocupado?

O maior problema a lidar com Hashing: colisão de resultados.

Conceitos (3/5)

Função de Espalhamento

Formalmente, temos $h : \mathcal{X} \rightarrow [0 \cdots m-1]$, sendo m posições possíveis.

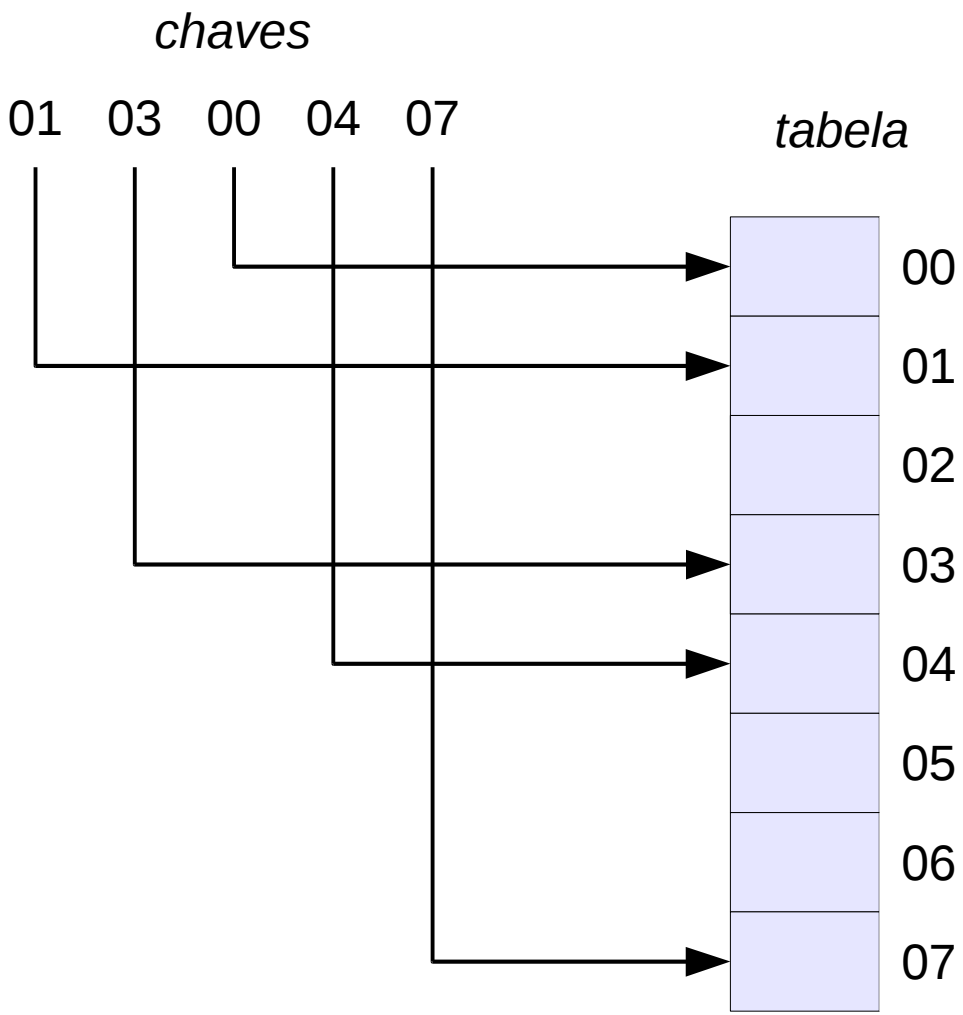
É desejável que $Pr_h(h(k) = i) = \frac{1}{n}$,
i.e., a função distribua de maneira uniforme as posições entre 0 e $m-1$.

Conceitos (5/5)

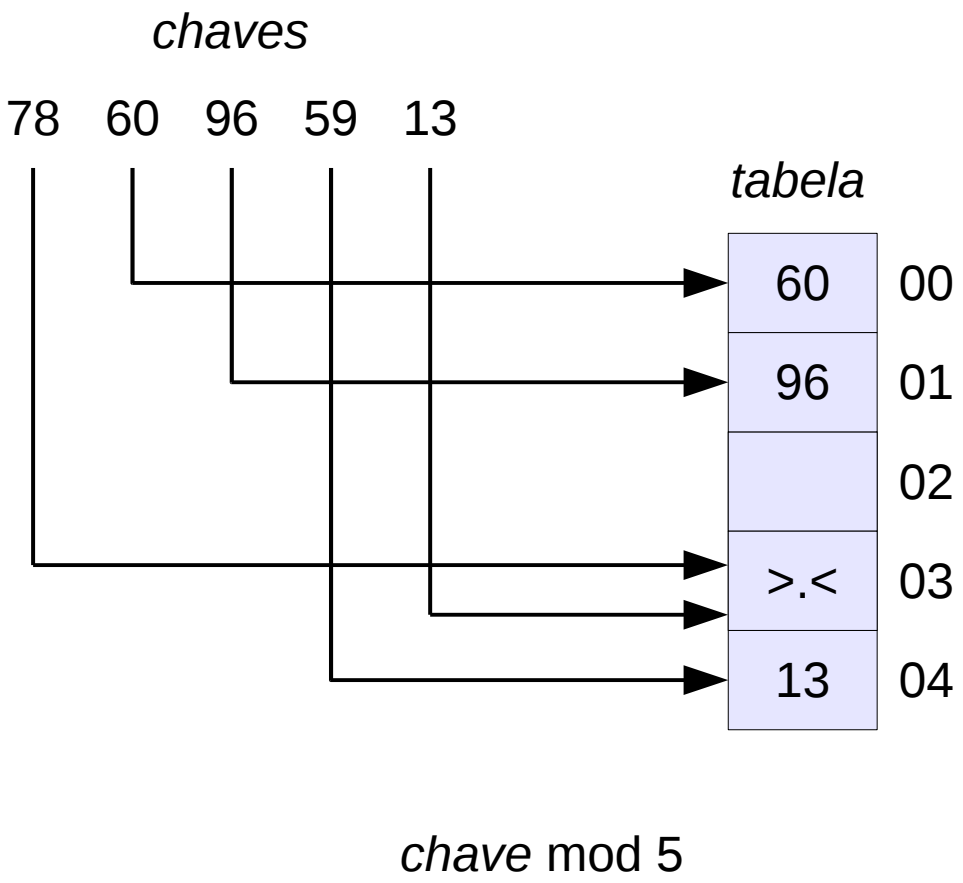
- Desejamos:
  - Número baixo de colisões;
  - Função facilmente computável;
  - Produz distribuição uniforme.

Endereçamento aberto (1/2)

- Indexar elementos em vetores



Colisão



Endereçamento aberto (2/2)

Limitações e desafios

- Chaves não inteiras;
- Quantidade de espaço vazio extra;
- Atualizações: exclusões, inserções, alterações.

Função Hash — Escolha da Função

Uma boa função *Hash* $h()$:

- rápida de computar,
- distribui chaves de maneira uniforme pela tabela,
- minimiza colisões.

Cautela ao lidar com chaves não-inteiras

- pensar em formas de mapear para um interalo discreto

Função Hash — de chaves a índices

$$h : k \rightarrow \text{int}, \tag{1}$$

mapeamento de código, k é uma chave e `int` um número inteiro, caso a chave seja de tipo não-inteiro.

$$c : \text{int} \rightarrow [0, m-1]\,, \tag{2}$$

mapeamento de compressão de um inteiro a um intervalo.

Função Hash — Mapeamento de compressão

Método de divisão

- $h(k) = k \bmod m$, com chave k e m o tamanho da tabela.
- Como escolher m?
- $m = b^e$ (inadequado)
  - todas as chaves com final igual serão mapeadas para o mesmo local.
  - e.g. se potência de 2, gera os e bits menos significativos de k.
- m primo (mais adequado)
  - ajuda a distribuir uniformemente as chaves.
  - o produto de um primo com outro número tem maior chance de ser único, pois o primo é usado para compor esse número.

Função Hash — Mapeamento de compressão

- Sedgewick:
  1. computar potência de 2 próxima do m desejado
  2. definir m como o número primo imediatamente abaixo da potência

k	2^k	m
7	128	127
8	256	251
9	512	509
10	1024	1021
11	2048	2039
12	4096	4093
13	8192	8191
14	16384	16381
15	32768	32749
16	65536	65521
17	131072	131071
18	262144	262139

Função Hash — Mapeamento de compressão

Método de multiplicação

- $h(k) = \lfloor m([k \cdot A] \bmod 1) \rfloor$
- k é a chave, m o tamanho da tabela e $A \in [0, 1]$
  1. mapear $[0, k_{max}] \to A \times [0, k_{max}]$
  2. tomar a parte fracionária $(\bmod 1)$.
  3. mapear para $[0, m - 1]$.

- Escolha de m deixa de ser crítica
- Escolha ótima de A depende dos dados.
- Knuth: o conjugado da razão áurea (*Fibonacci* hashing):

$$A = \frac{\sqrt{5} - 1}{2} = 0,6180339887\cdots \tag{3}$$

Mapeamentos de código hash: chaves não inteiras

- **Integer cast**: para tipos numéricos de 32 bits ou menos, reinterpretar os bits como um inteiro.
- **Soma de componentes**: para numeros de mais de 32 bits (long ou double), soma ponderada dos componentes de 32 bits.
  - a ponderação deve minimizar o overflow
  - porque pode ser ruim para *string* (código ASCII)?

Mapeamentos de código hash: chaves não inteiras

- **Acumulação polinomial**: combinar caracteres (ASCII or Unicode) como coeficientes de um polinômio.
  - computar o polinômio com a regra de Horner, para um valor fixo x:

$$a_0 + x\,(a_1 + x\,(a_2 + \cdots + x\,(a_{n-2} + x a_{n-1}) \cdots)) \tag{4}$$

- Cormen et al: a escolha de $x = 33, 37, 39$ ou 41 gera no máximo 6 colisões em um vocabulário de 50.000 palavras em inglês — obtido empiricamente).

Mapeamentos de código hash: strings

- Considenado caracteres ASCII (entre 0 e 255), uma string é uma representação em base 256 de um número.

Código (1)

```
int hash(char *v, int M) {
    int i, h = v[0];
    for (i = 1; v[i] != '\0'; i++)
        h = h * 256 + v[i];
    return h % M;
}
```

Mapeamentos de código hash: strings

- Base não precisa ser relacionada à tabela ASCII (256)
- Ex: usar número primo (251) e tirar o resto da divisão para evitar *overflow*:

Código (2)

```
int hash(char *v, int M) {
   int i, h = v[0];
   for (i = 1; v[i] != '\0'; i++)
      h = (h * 251 + v[i]) % M;
   return h;
}
```

Mapeamentos de código hash: strings

- Em Java, o hash code de uma String é:
- 31 foi escolhido por ser primo e porque `31 * i == (i << 5) - i`.

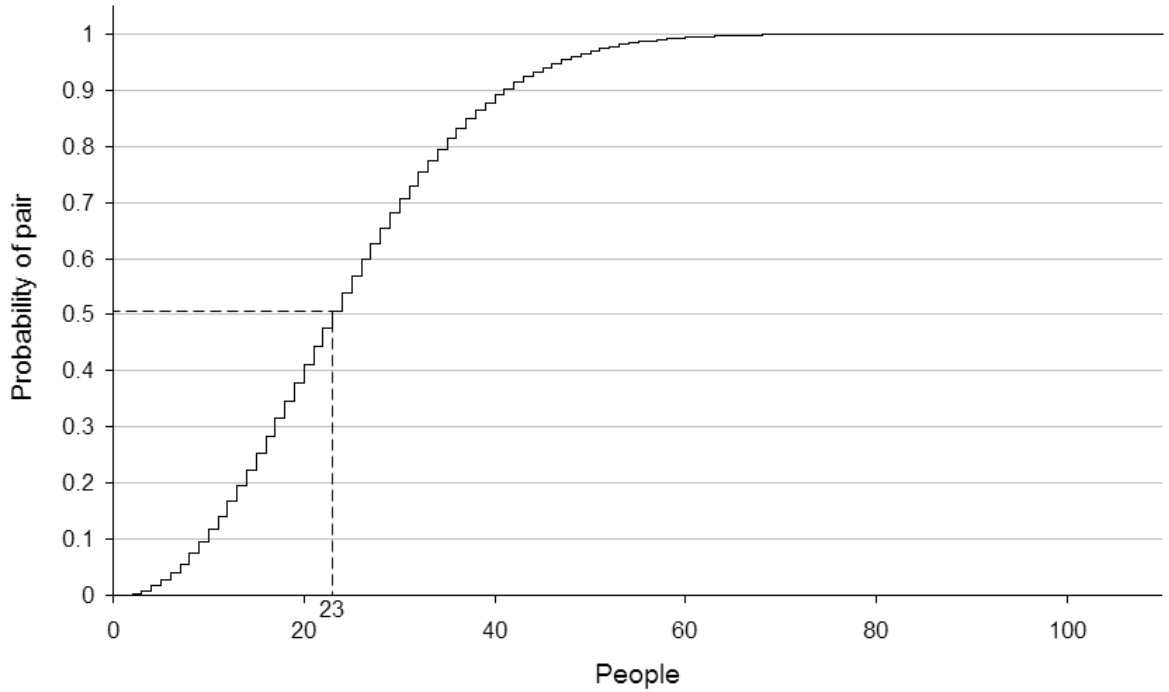
Código (3)

```
int hash(char *v, int M) {
   int i, h = v[0];
   for (i = 1; v[i] != '\0'; i++)
      h += (v[i]*31^(M-i));
   return h;
}
```

Sumário

Função Hash — paradoxo (problema) do aniversário

- De um grupo de 23 ou mais pessoas escolhidas aleatoriamente, a chance de duas terem a mesma data de aniversário é maior que 50%.
- É um paradoxo no sentido de ir contra à intuição/senso comum.
- Para 57 ou mais pessoas, a probabilidade é maior que 99%.
- 100% ocorre só a partir de 366 pessoas.



Função Hash — Conjunto de dados patológico

A melhor função hash!

- Não existe
- para cada $h()$, há um conjunto de dados "patológico";
- há sempre uma posição i para a qual ao menos $\frac{|u|}{n}$ elementos do universo u de chaves para os quais a função h produz resultado i.

Exemplo

- Crosby e Wallach (2003) é possível paralizar sistemas do mundo real explorando funções hash mal projetadas, com conjunto de dados patológico facilmente computável:
  - Projetos open source;
  - Funções hash simples, desenhadas para privilegiar apenas velocidade.

Função Hash — Soluções?

Função hash criptográfica

- e.g. SHA-2;
- é inviável computar um conjunto de dados patológico.

Aleatorização

- uma **família** $\mathcal{H}$ de funções;
- escolher aleatoriamente de $\mathcal{H}$ e tirar vantagem do valor esperado da performance.

Função Hash — Mapeamento de compressão

Hashing Universal

- $h_u(k) = ((ak + b) \bmod p) \bmod m$
  - p é um número primo maior do que m,
  - a e b são constantes escolhidas arbitrariamente no início da execução

definindo a e b

- $a \in \{1, 2, \cdots, p-1\}$
- $b \in \{0, 1, \cdots, p-1\}$

Função Hash — Mapeamento de compressão

Hashing Universal : características

- diz-se que $\mathcal{H}_{p,m} = \left\{ h_{a,b} : a \in Z_p^*, b \in Z_p \right\}$ representa uma coleção de funções universal.
- evita colisões, para o caso em que a não é múltiplo de m.
- a base dessa fórmula é usada em geradores de números *pseudo*-aleatórios — método linear congruencial, onde k é a semente (*seed*) e a e b os limites inferior e superior.

Função Hash — Mapeamento de compressão

Hashing Universal

- $h_u(k) = ((ak + b) \bmod p) \bmod m$

Exemplos:

- $p = 17$ e $m = 6$, usando $a = 3$ e $b = 4$
  - $h_{3,4}(8) = 5$.
  - $h_{3,4}(4)$
- $p = 13$ e $m = 10$, usando $a = 2$ e $b = 5$
  - $h_{2,5}(4)$
  - $h_{2,5}(2)$

Tratamento de Colisões

- Fator de carga (número esperado de espaços ocupados):

$$\alpha = \frac{n}{m}$$

- Menor fator de carga $\rightarrow$ Menor número de colisões
- Sempre pode haver uma colisão
- Previsão de tratamento de colisões
  - Endereçamento aberto
  - Encadeamento

Tratamento de Colisões por Endereçamento Aberto (1/3)

Calcular novo endereço para chaves sinônimas

- Em caso de colisão, calcula-se o novo índice
- Processo iterativo, enquanto a chave não for armazenada

Tratamento de Colisões por Endereçamento Aberto (2/3)

- *Overflow* progressivo
  - Sondagem linear
  - Sondagem quadrática
- *Rehash* – Aplicar segunda função *hash*

Tratamento de Colisões por Endereçamento Aberto (3/3)

- Sondagem linear
  - Todas posições da tabela são checadas

$$(h(x) + k) \mod m, \text{ para } 0 \leq k \leq m - 1$$

  - Longos trechos consecutivos podem ser ocupados
- Sondagem quadrática
  - Diversifica sequência de endereços

$$(h(x) + c_1 \cdot k + c_2 \cdot k^2) \mod m, \text{ para } 0 \leq k \leq m - 1$$

    c_1 e c_2 são constantes, $c_2 \neq 0$
  - Agrupamento ocorre com menos frequência

Hashing duplo (*double hashing / rehashing*)

- Usar duas funções hash: $h_1()$ e $h_2()$.
  - $h_1(k)$ será a primeira posição da tabela a ser verificada.
  - $h_2(k)$ determina o deslocamento usado quando procurado por k.
  - para o caso em que $h_2(k) = 1$, temos o método de sondagem linear (*overflow* progressivo).

- Se m é primo, todas as posições da tabela serão eventualmente examinadas
- Hashing duplo possui vantagens e desvantagens em comum com a sondagem linear. No entanto, ameniza o agrupamento em aplicações onde isso ocorre com mais frequência por outros métodos.

Endereçamento Aberto — Inserção em Hashing Duplo

```
int doublehashing_insert (T, k) {
   if (isFull(T)) {
      return -1;
   }

   sonda = h1(k);
   desloc= h2(k);
   while (T[sonda] != NULL) {
      sonda = (sonda+desloc) % m;
   }

   T[sonda] = k;
}
```

Exemplo de hashing duplo

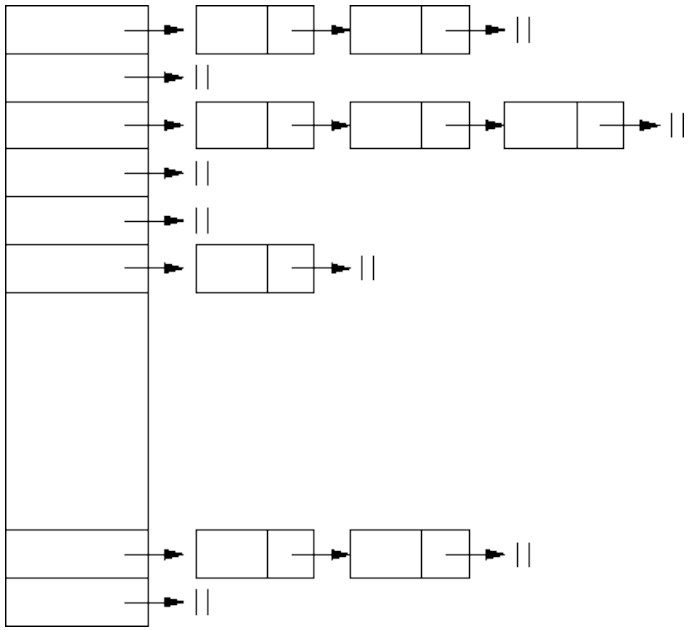
- $h_1(k) = k \bmod 13$.
- $h_2(k) = 1 + (k \bmod 7)$.
- Inserir as chaves: 18, 41, 22, 44, 59, 32, 31, 73.

- A função $h_2()$ acima é, em geral, definida como $h_2(k) = 1 + (k \bmod m')$, onde m' é geralmente escolhido como um valor ligeiramente menor do que m.
- Cada par $(h_1(k), h_2(k))$ gera uma sequência de sondagem distinta. Como resultado, o hash duplo pode ter desempenho muito mais próximo do constante.

Encadeamento Externo (1/2)

Armazena chaves sinônimas em listas encadeadas

- m listas encadeadas
- Endereço base: cabeça da lista
- Busca, inserção e remoção em listas encadeadas



Encadeamento Externo (2/2)

- Melhor caso (inserir): $\mathcal{O}(1)$
- Pior caso (buscar): $\mathcal{O}(n)$
- Pode-se inserir ordenado para reduzir a busca
- **Deve-se questionar:** Qual operação será realizada mais vezes?

Análise de caso médio considerando fator de carga

encadeamento	$\mathcal{O}(1+\alpha)$
endereçamento aberto : inserção	$\mathcal{O}\left(\frac{1}{(1-\alpha)}\right)$
endereçamento aberto : sondagem	$\mathcal{O}\left(\frac{1}{\alpha}\log\frac{1}{(1-\alpha)}\right)$

Hashing e segurança da informação

- Em criptografia, um **hash** é um tipo de assinatura para uma sequência de dados.
- Uma forma de ver isso na prática é usar o programa `md5sum` do linux para ver qual é o hash de arquivos.
  - Faça testes para ver que, pequenas diferenças na entrada, geram saídas bem diferentes, o que é conhecido como **efeito avalanche**.
- Hash não é encriptação
  - pois é preciso haver uma forma de obter o valor original a partir da cifra, e não há uma função *de-hash*.
- Assim, hash é usado para, por exemplo, verificar a integridade de arquivos, armazenar senhas e assinar digitalmente documentos.
  - colisões nesse caso não são toleradas, pois podem significar a quebra da segurança dos códigos.

Bibliografia I


ZIVIANI, N.
Projeto de Algoritmos (Capítulo 5). 3.ed.
Cengage, 2004.


CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C.
Algoritmos: teoria e prática (Capítulo 11).
Campus, 2002.


SEDGEWICK, R.
Algorithms in C
Addison-Wesley, 1998.


ROSA, J.L.G.
Notas de aula: Métodos de Busca
ICMC, 2009.