13 – Hashing (parte 2)

SCC201/501 - Introdução à Ciência de Computação II

Prof. Moacir Ponti Jr.

www.icmc.usp.br/~moacir

Instituto de Ciências Matemáticas e de Computação – USP

2010/2

Sumário

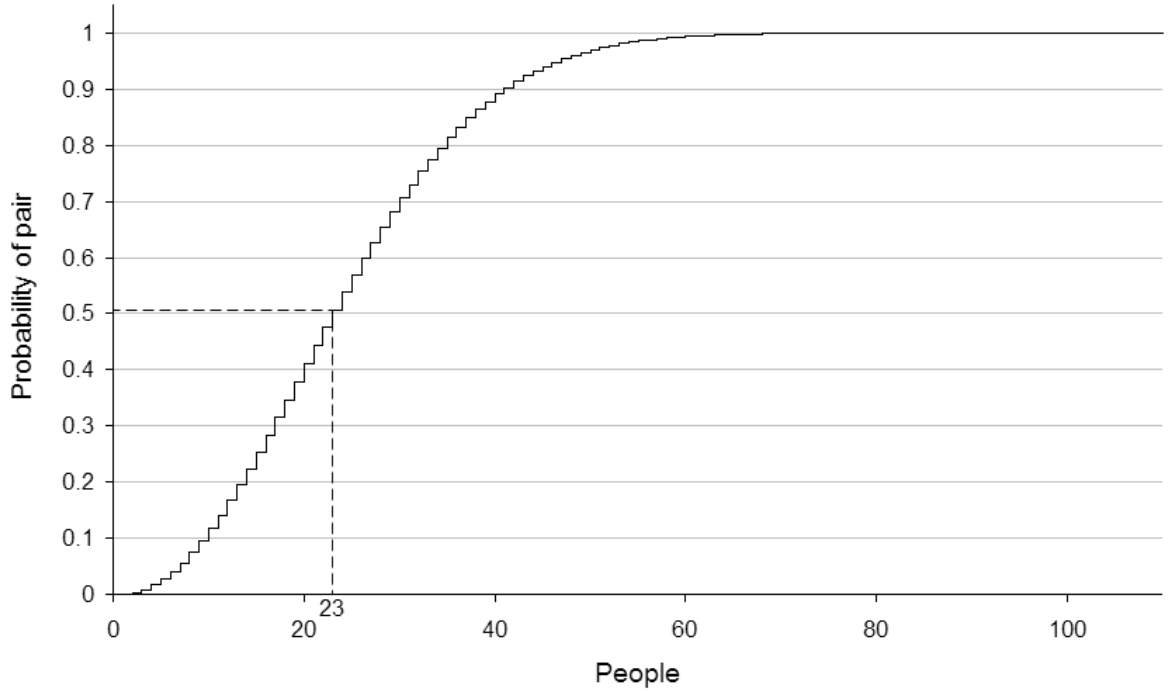
Função Hash — Escolha da Função

- Uma boa função *Hash* $h()$:
  - rápida de computar,
  - distribui chaves de maneira uniforme pela tabela,
  - minimiza a probabilidade de colisões.
- Uma função excelente é muito rara — uma das explicações se refere ao *paradoxo do aniversário*.

- Cautela ao lidar com chaves não-inteiras
  - uma forma é converter cada caractere para um número.
  - se desejamos trabalhar com as letras do alfabeto (são 26 letras), podemos atribuir 0 para 'A' e 26 para 'Z' e trabalhar na base 26.

Função Hash — paradoxo do aniversário

- Em teoria das probabilidades: dado um grupo de 23 (ou mais) pessoas escolhidas aleatoriamente, a chance de que duas pessoas tenham a mesma data de aniversário é $> 50\%$.
  - Para 57 ou mais pessoas, a probabilidade é maior que 99%.
  - A chance é igual a 100% se houver 366 pessoas ou mais.



Função Hash — de chaves a índices

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

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

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

onde CM é o mapeamento de compressão de um inteiro a um intervalo.

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), somar os componentes de 32 bits.
  - porque pode ser ruim para chaves tipo *string* (código ASCII)?

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

$$a_0 + x\,(a_1 + x\,(a_2 + \cdots + x\,(a_{n-2} + x a_{n-1})\cdots)) \tag{3}$$

- A escolha de $x = 33, 37, 39$ ou 41 gera no máximo 6 colisões em um vocabulário de 50.000 palavras em inglês — obtido experimentalmente.

Mapeamentos de código hash: strings

- Cada caracter é um número entre 0 e 255. Uma string pode ser interpretada como a representação em base 256 de um número.
- Se `s` é uma string de tamanho 3. Então o número correspondente seria: `s[0]*256+s[1]`

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

- A base da representação não precisa ser igual ao número de valores possíveis na tabela ASCII (256)
- Pode-se usar como base um número primo e ainda 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;
}
```

Função Hash — Mapeamento de compressão

Método de divisão

- Uso do módulo $h(k) = k \bmod m$, onde k é a chave, m o tamanho da tabela.
- Como escolher m?
- $m = b^e$ (inadequado)
  - se m for potência de 2, função gera os e bits menos significativos de k.
  - todas as chaves com final igual serão mapeadas para o mesmo local.
- m primo (adequado)
  - ajuda a distribuir uniformemente as chaves.

Função Hash — Mapeamento de compressão

- Sedgewick sugere escolher M da seguinte maneira:
  1. Escolha uma potência de 2 que esteja próxima do valor desejado de M (que seja apropriado para a aplicação)
  2. Adote para M o número primo que esteja logo abaixo da potência escolhida.

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 (mod 1).
  3. mapear para $[0, m - 1]$.

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

$$A = \frac{\sqrt{5} - 1}{2} = 0,6180339887 \cdots \tag{4}$$

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 aleatoriamente no início da execução, a partir de um conjunto de constantes $\{0, 1, \cdots, p-1\}$.

características

- diz-se que $h_u()$ 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.

Sumário

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.

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

- $h_1(k) = k \bmod 13$.
- $h_2(k) = 1 + (k \bmod 8)$.
- Inserir as chaves: 18, 41, 22, 44, 59, 32, 31, 73.

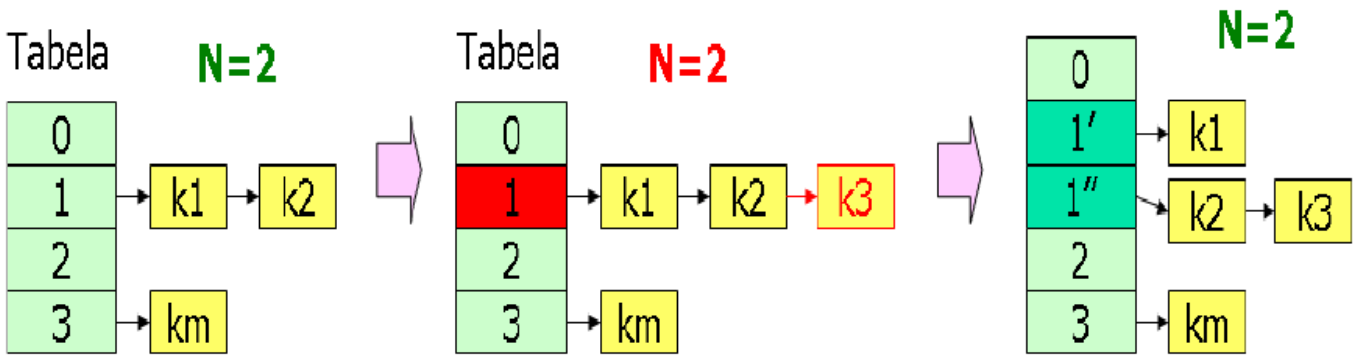
- A função $h_2()$ acima é, em geral, definida como $h_2(k) = 1 + (k \bmod m')$, onde m' é geralmente escolhido como um valor ligeiramente menor do que m.
- Cada par $(h_1(k), h_2(k))$ gera uma sequência de sondagem distinta. Como resultado, o hash duplo pode ter desempenho muito mais próximo do do esquema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

Análise

	Busca	
	sem sucesso	com sucesso
encadeamento	$\mathcal{O}(1+\alpha)$	$\mathcal{O}(1+\alpha)$
sondagem	$\mathcal{O}\left(1+\frac{1}{(1+\alpha)^2}\right)$	$\mathcal{O}\left(1+\frac{1}{1+\alpha}\right)$

Hashing dinâmico

- O tamanho do espaço de endereçamento (número de compartimentos) pode aumentar
  - *Hashing* extensível: auto-ajuste do espaço de endereçamento
  - A idéa é combinar o *hashing* convencional com uma técnica de recuperação de informações denominada *trie*.
- Sendo o N número máximo de elementos por compartimento, sempre que o elemento $N + 1$ for inserido, o compartimento é dividido (*split*)

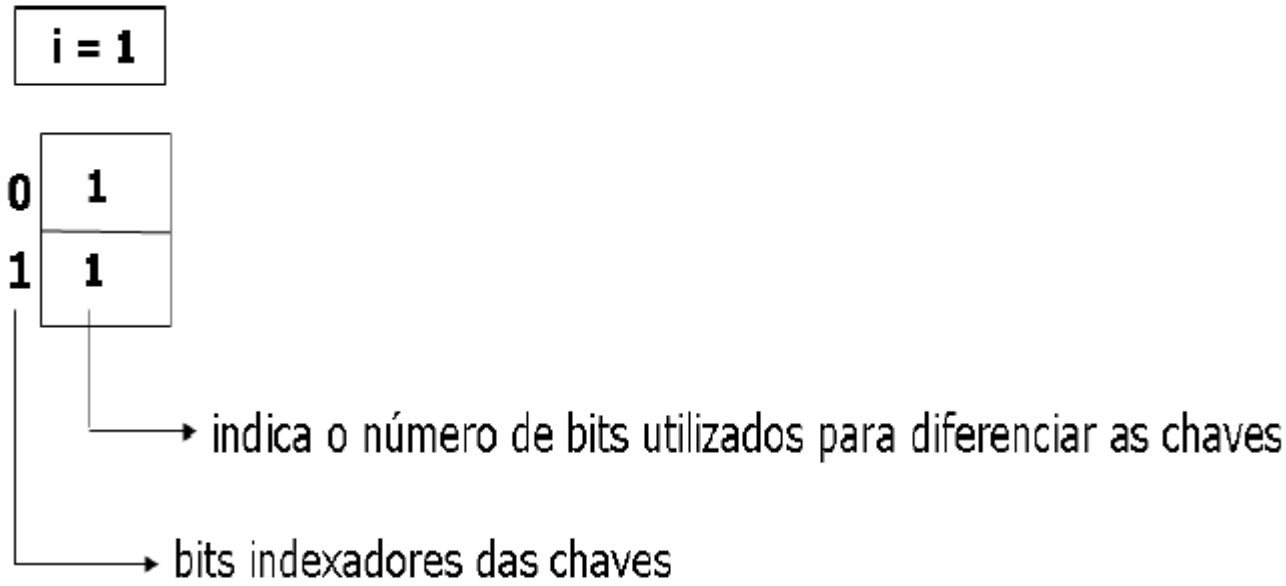


Hashing dinâmico

- Em geral trabalha-se com bits: após $h(k)$ ser computada, uma segunda função transforma o índice em uma sequência de bits
  - Pode-se embutir a transformação em bits na função $h()$
  - Apenas os i primeiros bits ($i \leq m$) da sequência serão usados como endereço: a tabela de terá $2i$ entradas, crescendo como potência de 2.

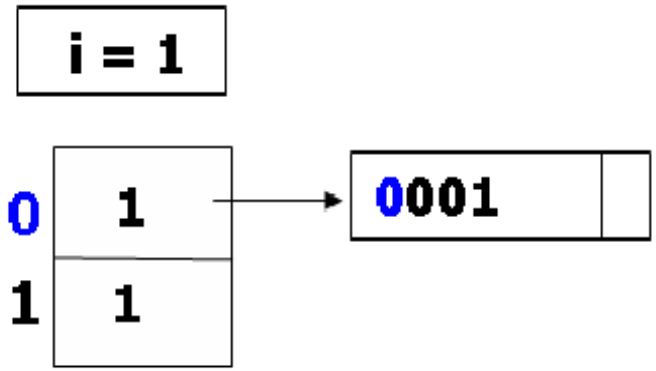
Hashing extensível

- Tabela vazia, $m = 4$ bits, $N = 2$.



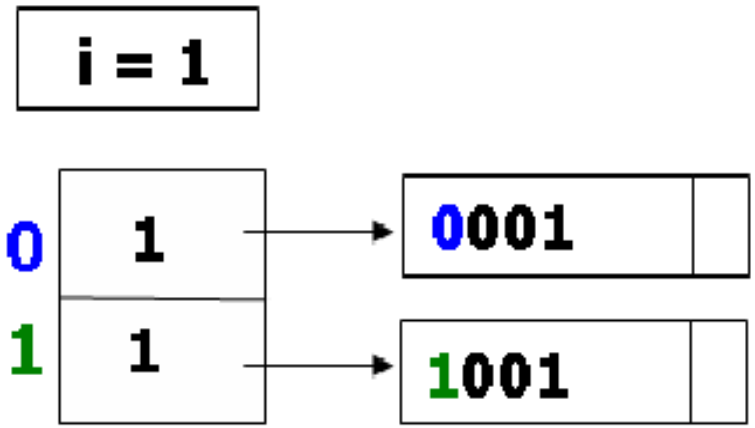
Hashing extensível

- $m = 4$ bits, $N = 2$. Inserção do elemento 0001:



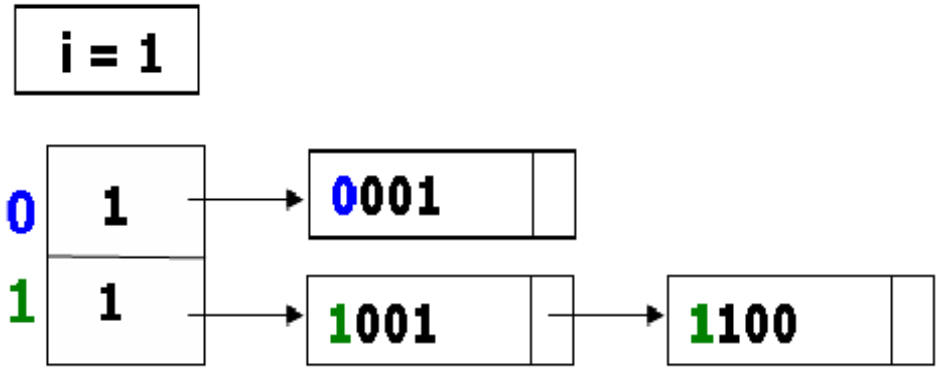
Hashing extensível

- $m = 4$ bits, $N = 2$. Inserção do elemento 1001:



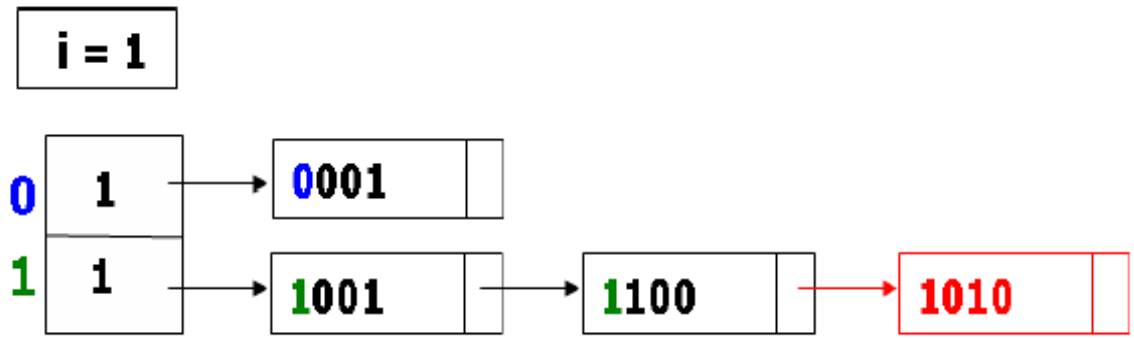
Hashing extensível

- $m = 4$ bits, $N = 2$. Inserção do elemento 1100:



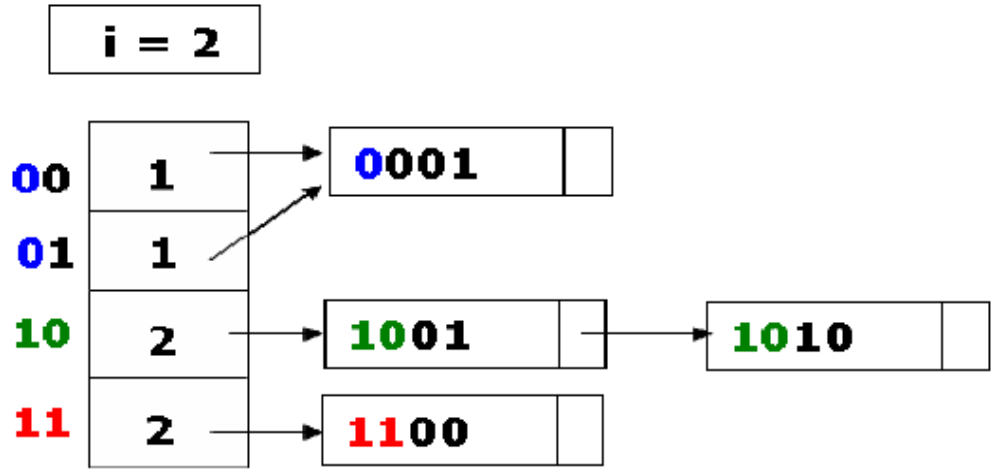
Hashing extensível

- $m = 4$ bits, $N = 2$. Inserção do elemento 1010:
- um único bit não é suficiente para diferenciar os elementos: o compartimento 1 é dividido e tem seu bit incrementado.



Hashing extensível

- $m = 4$ bits, $N = 2$. Rearranjando tabela, i aumenta para observar a restrição de N e chaves são rearranjadas



- Inserção dos elementos 0000 e 0110

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.