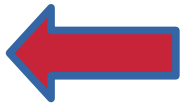
Aula 10: Tabela Hash – Parte 2

DCC405-Estrutura de Dados II

Prof. Me. Acauan C. Ribeiro

Roteiro da Aula

- Contextualização 
- Conceitos principais
- Função Hash
  - Escolha da Função
  - Métodos para mapeamento de compressão
- Evitando e tratando colisões
  - Hashing universal
  - Fator de Carga
  - Endereçamento aberto
  - Hashing duplo
  - Encadeamento
- Análise da Busca usando Hashing

Aula Passada...

- Vimos a **Tabela Hash – Encadeamento Externo**
  - Que utiliza uma **estrutura de dados auxiliar (externa)** para tratar as colisões.

Tabela Hash – Função Hash

- Como escolher a melhor função? 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 – Tipos de Função

- Método da Divisão
- Método da Multiplicação
- Hash Duplo (Double Hashing)

Função Hash – Método da Divisão

- $h(k) = k \bmod m$, com chave k e m o tamanho da tabela.

Função Hash – Método da Divisão

- $h(k) = k \bmod m$, com chave k e m o tamanho da tabela.
- **Como escolher m?**
- $\mathbf{m = b^e}$ (inadequado – base elevado a certo expoente)
  - todas as chaves com final igual serão mapeadas para o mesmo local.
  - Ex.: se potência de 2, gera os e bits menos significativos de k.

Função Hash – Método da Divisão

- $h(k) = k \bmod m$, com chave k e m o tamanho da tabela.
- **Como escolher m?**
- $\mathbf{m = b^e}$ (inadequado – base elevado a certo expoente)
  - todas as chaves com final igual serão mapeadas para o mesmo local.
  - Ex.: 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 – Número Primo

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

- Fazer exemplo para $\mathbf{n = 200}$.. qual seria o melhor **m**?

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 – Método da 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 $[0, k_{max}] \rightarrow A \times [0, k_{max}]$
  2. tomar a parte fracionária (mod 1).
  3. mapear para $[0, m - 1]$.

Função Hash – Método da 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 $[0, k_{max}] \rightarrow A \times [0, k_{max}]$
2. tomar a parte fracionária (mod 1).
3. mapear para $[0, m - 1]$.

Fazer exemplo

- $m = 16$, $k = 13$ e $A = 0{,}5$
  - 1) $[k*A] \rightarrow 13 * 0{,}5 = 6{,}5$
  - 2) $6{,}5 \bmod 1 = 0{,}5$
- 3) $\lfloor 16 * 0{,}5 \rfloor = \mathbf{8}$

Função Hash – Método da 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 $[0, k_{max}] \rightarrow A \times [0, k_{max}]$
  2. tomar a parte fracionária (mod 1).
  3. mapear para $[0, m - 1]$.

Fazer exemplo

- $m = 16$, $k = 13$ e $A = 0{,}5$
  - 1) $[k*A] \rightarrow 13 * 0{,}5 = 6{,}5$
  - 2) $6{,}5 \bmod 1 = 0{,}5$
- 3) $\lfloor 16 * 0{,}5 \rfloor = \mathbf{8}$

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

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

Quando as chaves não forem 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)?

Quando as chaves não forem 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)?

Exemplo: abc

abc =

Quando as chaves não forem 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).

Mapeamento de Hash: Strings (1/3)

- 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;
}
```

Mapeamento de Hash: Strings (2/3)

- 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;
}
```

Mapeamento de Hash: Strings (3/3)

- 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;
}
```

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.

Hashing duplo (inserção)

```
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;
}
```

Hashing duplo - Exemplo

- $h_1(k) = k \bmod 13$.
- $h_2(k) = 1 + (k \bmod 7)$.
- 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.