

# Tabela Hash: tipos de hashing e tratamento de colisões

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados 2 (AE43CP)  
Engenharia de Computação  
Departamento Acadêmico de Informática (Dainf)  
Universidade Tecnológica Federal do Paraná (UTFPR)  
Campus Pato Branco

- Tipos de *Hashing*
- Tratamento de Colisões
  - *Hashing* Estático
  - *Hashing* Dinâmico

## Tipos de *Hashing*

# Tipos de *Hashing*

- Estático
  - Espaço de endereçamento não muda
  - Fechado: Permite armazenar um conjunto de informações de tamanho limitado
    - Tratamento de colisões: sondagem linear (*overflow* progressivo), sondagem quadrática, segunda função hash
  - Aberto: Permite armazenar um conjunto de informações de tamanho potencialmente ilimitado
    - Tratamento de colisões: encadeamento de elementos

# Tipos de *Hashing*

- *Hashing* dinâmico
  - Espaço de endereçamento pode aumentar
  - *Hashing* extensível
  - Pode aumentar se houver colisões

## Tratamento de Colisões

# Tratamento de Colisões

- Uma função *hashing* pode gerar a mesma posição para chaves diferentes
  - Essa situação é chamada de colisão
- Suponha que utilizamos o resto de divisão para definirmos posições em uma tabela de tamanho 50
  - Se inserirmos as chaves 12 e 62, ocorrerá colisão
    - $12\%50 = 12$
    - $62\%50 = 12$
- Qualquer função *hashing* pode acarretar em colisões

- Em uma tabela *hash* deve haver uma forma para tratar colisões
- Desse modo, a estrutura da tabela *hash* é formada em duas partes:
  - Função *hashing*
  - Tratamento de colisões
- As estratégias para tratamento de colisões são aplicadas de acordo com o tipo de *hashing*
  - Estático
    - Endereçamento fechado
    - Endereçamento aberto
  - Dinâmico



# Tratamento de Colisões

## Hashing estático

- Hashing fechado: aplicação de técnicas de *rehash* para lidar com colisões, como nos exemplos abaixo
  - sondagem linear (*overflow* progressivo)
  - Sondagem quadrática
  - Segunda função *hash*
- Técnica *rehash*
  - Se posição  $h(k)$  está ocupada, aplicar função de *rehash* sobre  $h(k)$ , que deve retornar o próximo bucket livre:  $rh(h(k))$
  - Uma boa função *rehash* cobre o máximo de índices entre 0 e o tamanho da tabela - 1 e evita agrupamento de dados
  - Além do índice resultante de  $h(k)$ , na *rehash* também pode ser utilizada a própria chave  $k$  e outras funções *hash*

# Tratamento de Colisões

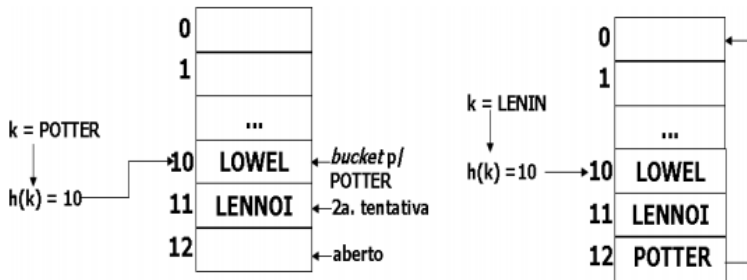
## Hashing estático

- Sondagem linear
  - Também conhecido como *Overflow* progressivo
  - Caso a função *hash* ( $h(k)$ ) resulte em uma posição ocupada, tentar a próxima posição:  $rh(h(k)) = (h(k) + i) \% B$ , sendo  $i$  variando de 1 a  $B - 1$  e  $B$  (*buckets*) é o tamanho da tabela
  - Na primeira tentativa:  $i = 1$
  - A variável  $i$  é incrementada até que seja encontrada uma posição vazia ou todas as opções sejam esgotadas

# Tratamento de Colisões

## Hashing estático

- Sondagem linear



- $h(POTTER) = (80+79+84+84+69+82)\%12 = 478\%12 = 10$
- Como saber se a informação procurada não está armazenada?
- Errata: "LENIN" deve ser substituído por "LeNIN"

# Tratamento de Colisões

## Hashing estático

- Sondagem linear
  - Dificuldade: pode ser necessário percorrer vários campos para encontrar um registro

$h(\text{SMITH}) = 7$

|    |        |
|----|--------|
|    | ...    |
| 7  | ADMS   |
| 8  | JONES  |
| 9  | MORRIS |
| 10 | SMITH  |

- No exemplo acima, a remoção do elemento na posição 9 pode causar uma falha na busca

# Tratamento de Colisões

## Hashing estático

- Sondagem linear
  - Solução para a remoção de elementos: não eliminar o elemento, mas indicar a posição que foi esvaziada, possibilitando a continuação da busca

**$h(\text{SMITH}) = 7$**

|    |       |
|----|-------|
|    | ...   |
| 7  | ADMS  |
| 8  | JONES |
| 9  | ####  |
| 10 | SMITH |

# Tratamento de Colisões

## *Hashing* estático

- Sondagem linear
  - Vantagens
    - Simplicidade
    - Custo computacional baixo
  - Desvantagens
    - Agrupamento de dados (causado por colisões)
    - Com a tabela cheia, a busca fica lenta
    - Dificulta as inserções e remoções

- Sondagem quadrática
  - Também conhecida como espalhamento quadrático, tentativa quadrática ou *rehash* quadrático
  - Utiliza uma equação de segundo grau para calcular *rehash*:
$$rh(h(k), i) = (h(k) + c_1 * i + c_2 * i^2) \% B$$
onde:  $c_1$  e  $c_2$  são constantes diferentes de 0, e  $i$  varia de 1 a  $B - 1$
  - Pode ter um desempenho melhor do que a sondagem linear, mas  $c_1$ ,  $c_2$  e  $B$  são limitados
  - Evita o agrupamento primário (causado pela sondagem linear), mas pode acarretar em agrupamento secundário (quadrático)
    - Exemplo: se duas ou mais chaves possuem o mesmo endereço na aplicação da função *hash*, então as sequências de sondagem serão iguais

- Sondagem quadrática

- Exemplo para  $B = 41$ ,  $c_1 = 0,5$ ,  $c_2 = 0,5$  e  $h(k) = 5$ , então temos:

- Para  $i = 1$ :  $rh(k, i) = (5 + 0,5 * 1 + 0,5 * 1^2) \% 41 = 6$

- Para  $i = 2$ :  $rh(k, i) = (5 + 0,5 * 2 + 0,5 * 2^2) \% 41 = 8$

- Para  $i = 3$ :  $rh(k, i) = (5 + 0,5 * 3 + 0,5 * 3^2) \% 41 = 11$

- Para  $i = 4$ :  $rh(k, i) = (5 + 0,5 * 4 + 0,5 * 4^2) \% 41 = 15$

- Para  $i = 5$ :  $rh(k, i) = (5 + 0,5 * 5 + 0,5 * 5^2) \% 41 = 20$

- Para  $i = 6$ : ?



# Tratamento de Colisões

## *Hashing* estático

- Sondagem quadrática
  - Principal vantagem
    - Evita o agrupamento primário causado pela sondagem linear
  - Principal desvantagem
    - Ainda pode gerar agrupamento (mesmo que seja de forma secundária)

- Segunda função *hash*
    - Também conhecida como *hash* duplo
    - Utiliza duas funções como auxiliares
      - $h1(k)$ : função *hash* primária
      - $h2(k)$ : função *hash* secundária
    - Algumas boas funções auxiliares
      - $h1(k) = k \% B$
      - $h2(k) = 1 + k \% (B - 1)$
    - Função *rehash*
      - $rh(k, i) = (h1(k) + i * h2(k)) \% B$
- ou
- $rh(k, i) = (h1(k) + i * (1 + k \% (B - 1))) \% B$

- Segunda função *hash*

- Exemplo para  $B = 41$ ,  $h1(k) = 5$  e  $h2(k) = 6$ , então temos:

- Para  $i = 1$ :  $rh(k, i) = (5 + 6 * 1) \% 41 = 11$

- Para  $i = 2$ :  $rh(k, i) = (5 + 6 * 2) \% 41 = 17$

- Para  $i = 3$ :  $rh(k, i) = (5 + 6 * 3) \% 41 = 23$

- Para  $i = 4$ :  $rh(k, i) = (5 + 6 * 4) \% 41 = 29$

- Para  $i = 5$ :  $rh(k, i) = (5 + 6 * 5) \% 41 = 35$

- Para  $i = 6$ : ?

# Tratamento de Colisões

## Hashing estático

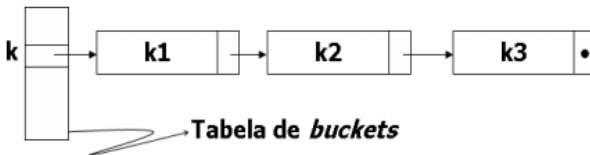
- Segunda função *hash*
  - Vantagem
    - Geralmente evita agrupamento de dados
  - Desvantagens
    - Difícil achar funções *hash* que, ao mesmo tempo, satisfaçam os critérios de cobrir o máximo de índices da tabela e evitem agrupamento de dados
    - Operações de buscas, inserções e remoções são mais difíceis

# Tratamento de Colisões

## Hashing estático

- *Hashing* aberto

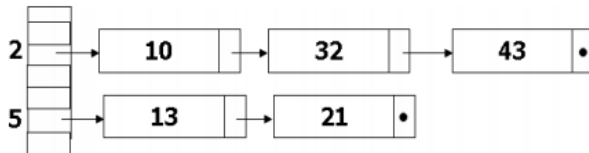
- A tabela de *buckets*, indo de 0 a  $B - 1$ , contém apenas ponteiros para uma lista de elementos
- Quando há colisão, o item é inserido no *bucket* como um novo nó da lista



# Tratamento de Colisões

## Hashing estático

- Hashing aberto
  - Se as listas estiverem ordenadas, reduz-se o tempo de busca



# Tratamento de Colisões

## *Hashing* estático

- *Hashing* aberto
  - Principal vantagem
    - A tabela pode receber mais itens mesmo quando um *bucket* já foi ocupado
  - Desvantagens
    - Quantidade de endereços não pode ser aumentado
    - Espaço extra para as listas
    - Listas longas pode implicar em muito tempo gasto na busca

# Tratamento de Colisões

## *Hashing* estático

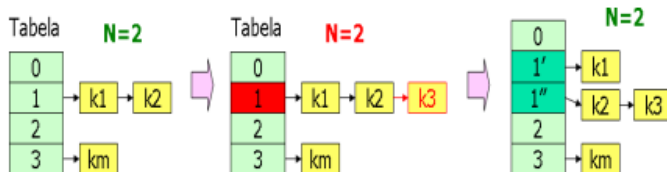
- Eficiência
  - *Hashing* fechado
    - Depende da técnica de *rehash*
    - A tabela pode ficar cheia
  - *Hashing* aberto
    - Depende do tamanho das listas e da função *hash*



# Tratamento de Colisões

## Hashing dinâmico

- O tamanho do espaço de endereçamento (número de *buckets*) pode aumentar
- Exemplo de *hashing* dinâmico:
  - *Hashing* extensível: conforme os elementos são inserido na tabela, o tamanho aumenta se necessário

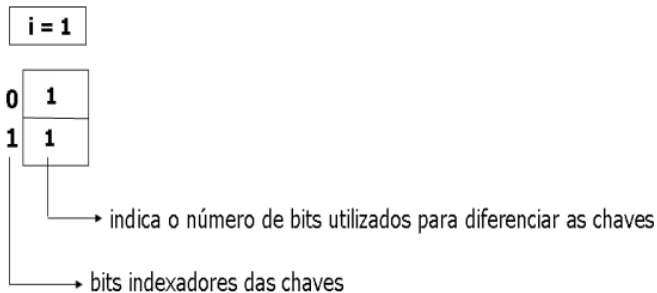


- *Hashing* extensível
  - Em geral, trabalha-se com bits
  - Após  $h(k)$  ser computada, uma segunda função  $f$  transforma o índice  $h(k)$  em uma sequência de bits
  - Alternativamente,  $h$  e  $f$  podem ser unificadas como uma única função *hash* final
  - Função *hash* computa sequência de  $m$  bits para uma chave  $k$ , mas apenas os  $i$  primeiro bits ( $i \leq m$ ) do início da sequência são usados como endereço
    - Se  $i$  é o número de bits usados, a tabela de *buckets* terá  $2^i$  entradas
    - Portanto, tamanho da tabela de *buckets* cresce sempre com potência de 2 (aumenta a quantidade de bits em uma unidade)

# Tratamento de Colisões

## Hashing dinâmico

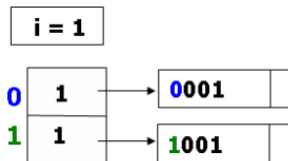
- Hashing extensível
  - $N$  é o número de nós permitidos por bucket
  - Tratamento de colisões: geralmente por listas encadeadas
  - Exemplo: tabela inicialmente vazia,  $m = 4$  (bits) e ( $N = 2$ )



# Tratamento de Colisões

## Hashing dinâmico

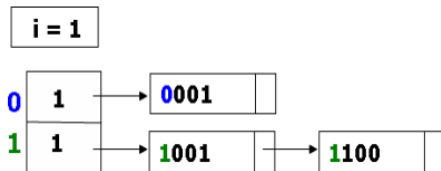
- *Hashing* extensível
  - Exemplo: inserção dos elementos 0001 e 1001, respectivamente



# Tratamento de Colisões

## Hashing dinâmico

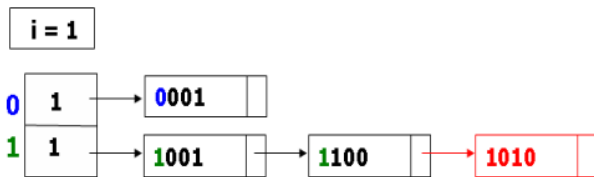
- Hashing extensível
  - Exemplo: inserção do elemento 1100



# Tratamento de Colisões

## Hashing dinâmico

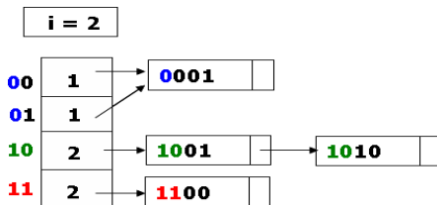
- Hashing extensível
  - Exemplo: inserção do elemento 1010



# Tratamento de Colisões

## Hashing dinâmico

- Hashing extensível
  - Exemplo: inserção do elemento 1010



# Tratamento de Colisões

## Hashing dinâmico

- Hashing dinâmico
  - Vantagens
    - Custo de acesso constante, determinado pelo tamanho de  $N$
    - A tabela pode crescer
  - Desvantagens
    - Complexidade extra para gerenciar o aumento do arranjo e a divisão das listas
    - Podem existir sequências de inserções que façam a tabela crescer rapidamente, tendo, contudo, um número pequeno de registros
- Principal desvantagem de *hashing*: Os elementos da tabela não são armazenados sequencialmente e nem sequer existe um método prático para percorrê-los em sequência





Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.  
*Introduction to Algorithms.*

Third edition, The MIT Press, 2009.



Madalosso, E.

Hashing Universal. AE22CP – Algoritmos e Estrutura de Dados I.

*Notas de Aula.* Engenharia de Computação.

Dainf/UTFPR/Pato Branco, 2019.



Oliva, J. T.

Tratamento de Colisões. AE22CP – Algoritmos e Estrutura de Dados I.

*Notas de Aula.* Engenharia de Computação.

Dainf/UTFPR/Pato Branco, 2020.



Rosa, J. L. G.

Métodos de Busca. SCE-181 – Introdução à Ciência da Computação II.

*Slides.* Ciência de Computação. ICMC/USP, 2018.



Ziviani, N.

*Projeto de Algoritmos - com implementações em Java e C++.*

Thomson, 2007.