Aula 6

Estrutura de dados

Prof. Vinicius Pozzobon Borin

Conversa Inicial

- O objetivo desta aula é apresentar os conceitos que envolvem a estrutura de dados do tipo *hash*
- Funções *hash*
  - Método da divisão
  - Método da multiplicação
  - *Hashing* universal

- Implementação e tratamento de colisões
  - Tentativa linear
  - Tentativa quadrática
  - Listas encadeadas
- Análise de complexidade
  - Inserção
  - Busca
  - Remoção

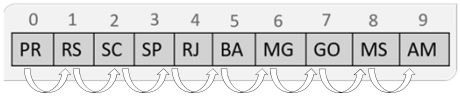
Hashs: definições

Cenário

- Você precisa armazenar um grande volume de dados de diferentes estados brasileiros, como capital, população total, lista de cidades, PIB, índice de criminalidade, nome do governador, dentre inúmeros outros dados
- Você resolve adotar a sigla de cada estado como palavra-chave para as buscas
- Cada posição do vetor conterá a sigla do estado

Solução 1: endereçamento direto

- Usar um vetor de dimensão *m* e armazenar sequencialmente as informações
- Inserir os dados na sequência em que são cadastrados



- Vetor
  - Tempo de acesso ao dado constante: O(1)
  - O tempo de busca do dado no endereçamento direto é dependente do tamanho do vetor
  - Usamos algoritmos de buscas estudados na aula 2, como busca sequencial O(n) e busca binária O(logn)
- Lista encadeada
  - O acesso aos dados depende do tamanho da lista: O(n)

Solução 2: tabela *hash*

- Possibilidade de tornar o tempo de acesso e de busca em um vetor independentemente do tamanho do conjunto de dados

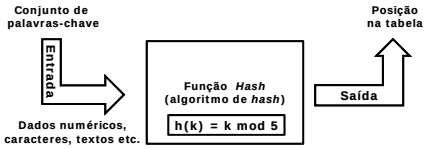
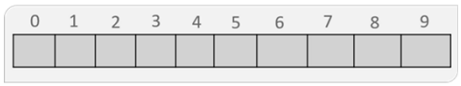


Tabela *hash*: exemplo

Palavra-chave *k*	Função *hash* h(k)
2 caracteres	$h(k) = (CHAR1_{ASCII} + CHAR2_{ASCII}) MOD\ m$
	$h(k) = (CHAR1_{ASCII} + CHAR2_{ASCII}) MOD\ 10$



$m = 10$

Tabela *hash*: exemplo

Palavra-chave *k*	Função *hash* h(k)	Resultado (posição)
PR	$h(PR) = (P_{ASCII} + R_{ASCII}) MOD\ 10$	$h(PR) = 2$
	$h(PR) = (80 + 82) MOD\ 10$	
	$h(PR) = 162\ MOD\ 10$	

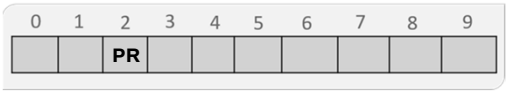


Tabela *hash*: exemplo

Palavra-chave *k*	Função *hash* h(k)	Resultado (posição)
SC	$h(SC) = (S_{ASCII} + C_{ASCII}) MOD\ 10$	$h(SC) = 0$
	$h(SC) = (83 + 67) MOD\ 10$	
	$h(SC) = 150\ MOD\ 10$	

Tabela *hash*: exemplo

Palavra-chave *k*	Função *hash* h(k)	Resultado (posição)
RS	$\mathbf{h}(RS) = (R_{ASCII} + S_{ASCII}) MOD\ 10$	$\mathbf{h}(RS) = 5$
	$\mathbf{h}(RS) = (82 + 83) MOD\ 10$	
	$\mathbf{h}(RS) = 165\ MOD\ 10$	

Tabela *hash*

- Se aplicarmos um algoritmo de busca no vetor a seguir, estaremos dependendo do tamanho do vetor: O(n)
- Se usarmos a função *hash* para reaver o dado de uma posição, poderemos fazer isso sem dependência do tamanho do vetor

Busca sequencial: 6 iterações
Busca binária: 3 iterações
Hash: acesso imediato

Aplicações

- Rastreamento de jogadas efetuadas no xadrez
- Compiladores necessitam manter uma tabela com variáveis mapeadas na memória do programa
- Aplicações voltadas para a segurança, como autenticação de mensagens e assinatura digital, empregam *hashs*
- A estrutura de dados-base que permite às populares criptomoedas (como *bitcoins*) operarem são *hashs*

Funções *hash*

O que uma boa função *hash* deve ter?

- Facilidade de ser calculada
- Capacidade de distribuir palavras-chave o mais uniformemente possível
- Capacidade de minimizar colisões – os dados devem ser inseridos de uma forma que as colisões sejam as mínimas possíveis, reduzindo o tempo gasto na resolução de colisões e também reavendo os dados
- Capacidade de resolver qualquer colisão que ocorrer

Método da divisão

$$h(k) = k\ MOD\ m$$

$k = 100$, $m = 12$ ⇨ $h(100) = 100\ MOD\ 12 = 4$

$k = 100$, $m = 15$ ⇨ $h(100) = 100\ MOD\ 15 = 10$

- Alterar o tamanho do vetor pode alterar todas as posições das chaves!

Método da divisão

$$h(k) = k \; MOD \; m$$

- **8 dígitos telefônicos**

$k = 99882233$

$k = 99 + 88 + 22 + 33$ ⇨ $h(242) = 242 \; MOD \; 12 = 2$

$k = 242$

Caracteres alfanuméricos $h(k) = \left(\sum k_{ASCII\_Dec}\right) MOD \; m$

Método da multiplicação

$$h(k) = \lfloor m(k\,A \; MOD \; 1) \rfloor$$

$$h(k) = \lfloor m(k\,A - \lfloor k\,A \rfloor) \rfloor$$

$k = 123456$

$m = 16384$ ⇨ $h(123456) = \lfloor 16384(123456.0,618 - \lfloor 123456.0,618 \rfloor) \rfloor$

$A = 0,618$ $\qquad h(123456) = 67$

Valor recomendado (sempre 0 < A < 1)

Comparativo dos métodos

- **Método da divisão**
  - **Computacional rápido de executar**
  - **O valor de *m* deve ser escolhido minuciosamente**
  - **Múltiplos de 2 devem ser evitados. É recomendável adotar números primos**
- **Método da multiplicação**
  - **Computacional mais lento que o método da divisão**
  - **O valor de *m* não tem impacto**

Hashing universal

- **Observe a situação a seguir para** $h(k) = k \; MOD \; 6$

$k = 6$ ⇨ $h(6) = 6 \; MOD \; 6 = 0$

$k = 12$ ⇨ $h(12) = 12 \; MOD \; 6 = 0$

$k = 18$ ⇨ $h(18) = 18 \; MOD \; 6 = 0$

$k = 24$ ⇨ $h(24) = 24 \; MOD \; 6 = 0$

$k = 30$ ⇨ $h(30) = 30 \; MOD \; 6 = 0$

Excesso de colisões!

Solução para minimizar colisões

Hashing universal

Hashing universal

- **O uso de uma única função *hash* pode resultar em uma situação em que todas as chaves precisam ser inseridas na mesma posição, gerando colisão e, consequentemente, piorando o desempenho do algoritmo**
- **Para minimizar esse problema, adotamos um conjunto *H* de funções *hash*. Sorteamos uma função dentro da classe de funções disponíveis para fazer a inserção do dado**

Hashing universal: exemplo

Classe de funções *hash*

$h_{a,b}(k) = ((ak + b) \; MOD \; p) \; MOD \; m$

p é um número primo

$b = \{0, 1, 2 \ldots p - 1\}$

$a = b - \{0\}$

- **Podemos escolher um valor para p e m e variar a e b aleatoriamente para gerar funções diferentes com resultados diferentes**

Tabela *hashing* de endereçamento aberto e tentativa linear

- **Endereçamento aberto**
  - **Quando todas as posições são conhecidas e se tem em cada uma, no máximo, uma chave**

```
1  registro Hash
2      chave: inteiro
3      status: caractere
4      //L = Livre, O = Ocupado, R = Removido
5  fimregistro
6
7  algoritmo "HashMenu"
8  var
9      Tabela: Hash[TAMANHO_VETOR]
```

Colisões

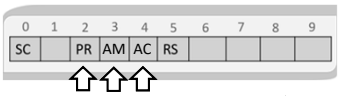
- **Ocorrem quando uma chave precisa ser posicionada em uma posição em que já existe outra chave**

Inserindo a sigla *AM*:

$h(AM) = 142\ MOD\ 10 = 2$ **Colisão!**

Resolvendo colisões: tentativa linear

- **Quando uma colisão ocorre, busca-se a próxima posição livre**



$h(AM) = 2$ ⇨ **Colisão**

$h(AM) = 3$ ⇨ **Livre**

$h(AC) = 2$ ⇨ **Colisão**

$h(AC) = 3$ ⇨ **Colisão**

$h(AC) = 4$ ⇨ **Livre**

Tentativa linear: inserção

```
1   função InserirNaHash(Tabela: Hash, pos: inteiro, n: inteiro)
2   var
3       i: inteiro
4   inicio
5       enquanto ((i < TAMANHO_VETOR)
6       E (Tabela[(pos+i) MOD TAMANHO_VETOR].status <> 'L')
7       E (Tabela[(pos+i) MOD TAMANHO_VETOR].status <> 'R'))
8           i = i + 1
9       fimenquanto
10
11      se (i < TAMANHO_VETOR) então
12          Tabela[(pos+i) MOD TAMANHO_VETOR].chave = n
13          Tabela[(pos+i) MOD TAMANHO_VETOR].status = 'O'
14      senão
15          escreva("Tabela Cheia!")
16      fimse
17  fimfunção
```

Complexidade: O(n)

Tentativa linear: remoção

```
19  função RemoverDaHash(Tabela: Hash, n: inteiro)
20  var
21      posicao: inteiro
22  inicio
23      posicao = BuscarNaHash(Tabela, n)
24
25      se (posicao < TAMANHO_VETOR) então
26          Tabela[posicao].status = 'R'
27      senão
28          escreva("Elemento não existente na tabela!")
29      fimse
30  fimfunção
```

Complexidade: O(n)

Tentativa linear: busca

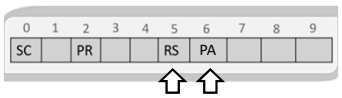
```
32  função BuscarNaHash(Tabela: Hash,  n: inteiro)
33  var
34     i, pos: inteiro
35  início
36     i ← 0
37     pos ← FuncaoHashing(n)
38
39     enquanto ((i < TAMANHO_VETOR)
40     E (Tabela[(pos+i) MOD TAMANHO_VETOR].status <> 'L')
41     E (Tabela[(pos+i) MOD TAMANHO_VETOR].chave <> n))
42        i ← i +1
43     fimenquanto
44
45     se ((Tabela[(pos+i) MOD TAMANHO_VETOR].chave <> n)
46     E (Tabela[(pos+i) MOD TAMANHO_VETOR].status <> 'R')) então
47        retorne (pos+i) MOD TAMANHO_VETOR
48     senão
49        retorne TAMANHO_VETOR //Não encontrado
50     fimse
51  fimfunção
```

Complexidade: O(n)

Tabela *hashing* de endereçamento aberto e tentativa quadrática

Resolvendo colisões: tentativa quadrática

- **Quando uma colisão ocorre, a próxima posição é calculada somando-se o valor da variável incremental *i***



$d = h(PA) = 5$ ⇨ **Colisão**

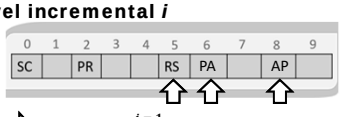
$i = 1$

$d = h\ (PA) = (5+1) MOD\ 10 = 6$ ⇨ **Livre**

$d = h(PA) = (d+i) MOD\ 10$

Resolvendo colisões: tentativa quadrática

- **Quando uma colisão ocorre, a próxima posição é calculada somando-se o valor da variável incremental *i***



$d = h(AP) = 5$ ⇨ **Colisão**

$i = 1$

$d = h\ (AP) = (5+1) MOD\ 10 = 6$ ⇨ **Colisão**

$i = 2$

$d = h\ (AP) = (6+2) MOD\ 10 = 8$ ⇨ **Livre**

$d = h(AP) = (d+i) MOD\ 10$

Tentativa quadrática: inserção

```
1   função InserirNaHash(Tabela: Hash, n: inteiro)
2   var
3      d, i: inteiro
4   início
5      d ← FuncaoHashing(n)
6      i ← 1
7
8      enquanto ((i<= TAMANHO_VETOR)
9      E (Tabela[d].status <> 'L')
10     E (Tabela[d].status <> 'R'))
11        d ← (d + i) MOD TAMANHO_VETOR
12        i ← i +1
13     fimenquanto
14
15     se (i <= TAMANHO_VETOR) então
16        Tabela[d].chave ← n
17        Tabela[d].status ← 'O'
18     senão
19        escreva("Tabela cheia!")
20     fimse
21  fimfunção
```

Complexidade: O(n)

Tentativa quadrática: remoção

```
19  função RemoverDaHash(Tabela: Hash, n: inteiro)
20  var
21     posicao: inteiro
22  início
23     posicao ← BuscarNaHash(Tabela, n)
24
25     se (posicao < TAMANHO_VETOR) então
26        Tabela[posicao].status ← 'R'
27     senão
28        escreva("Elemento não existente na tabela!")
29     fimse
30  fimfunção
```

Complexidade: O(n)

Tentativa quadrática: busca

```
35  função BuscarNaHash(Tabela: Hash,  n: inteiro)
36  var
37      d, i: inteiro
38  inicio
39      d = FuncaoHashing(n)
40      i= 1
41
42      enquanto ((i <= TAMANHO_VETOR)
43      E (Tabela[d].status <> 'L')
44      E (Tabela[d].chave <> n))
45          d = (d + i) MOD TAMANHO_VETOR
46          i = i +1
47      fimenquanto
48
49      se ((Tabela[d].chave == n)
50      E ((Tabela[d].status <> 'R')) então
51          retorne d
52      senão
53          retorne TAMANHO_VETOR
54      fimse
55  fimfunção
```

Complexidade: O(n)

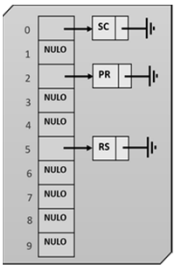
Tabela *hashing* com endereçamento em cadeia

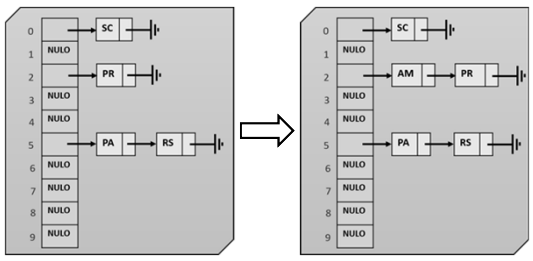
- **Endereçamento em cadeia**
  - **Cada posição poderá conter diversas chaves encadeadas**
  - **A forma de implementar é utilizando uma lista encadeada simples**

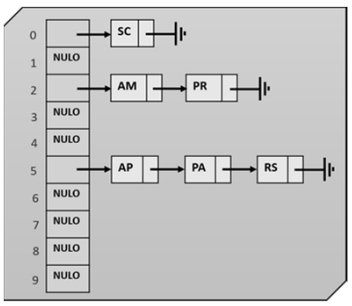
```
1  registro HashLista
2      chave: inteiro
3      prox: HashLista[->)
4  fimregistro
5
6  algoritmo "HashMenuLista"
7  var
8      Tabela: HashLista[TAMANHO_VETOR][->)
```

Resolvendo colisões com listas

- **Uma lista encadeada de chaves é criada para cada posição do vetor**
- **Quando uma colisão ocorre, a nova chave é inserida no início dessa lista**







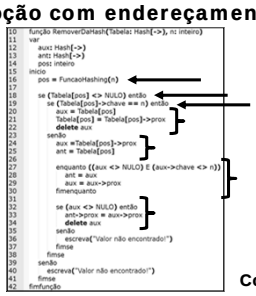
Inserção com endereçamento em cadeia

- Funciona como uma inserção no início da lista

```
1  função InserirNaHash(Tabela: Hash[->], pos: inteiro, n: inteiro)
2  var
3     Novo: Hash[->]
4  inicio
5     Novo->chave = n
6     Novo->prox = Tabela[pos]
7     Tabela[pos] = Novo
8  fimfunção
```

Complexidade: O(1)

Remoção com endereçamento em cadeia



Complexidade: O(n)

Referências

- ASCENCIO, A. F. G.; ARAÚJO, G. S. Estrutura de dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson, 2011.
- ASCENCIO, A. F. G.; CAMPOS, E. A. V. Fundamentos da programação de computadores: algoritmos, Pascal, C/C++ (padrão ANSI) e JAVA. 3. ed. São Paulo: Pearson, 2012.
- CORMEN, T. H. et al. Algoritmos: teoria e prática. 3. ed. São Paulo: Elsevier Brasil, 2012.
- KNUTH, D. E. The art of computer programming. 2. ed. Addison-Wesley, 1998. v. 3. Sorting and searching.
- LAUREANO, M. Estrutura de dados com algoritmos e C. Rio de Janeiro: Brasport, 2008.
- MIZRAHI, V. V. Treinamento em linguagem C. 2. ed. São Paulo: Pearson, 2008.
- PUGA, S.; RISSETTI, G. Lógica de programação e estrutura de dados com aplicações em Java. 3. ed. São Paulo: Pearson, 2016.