

# Métodos de Busca: *Hashing*

Prof. Thiago A. S. Pardo

Baseado no material do Prof. Rudinei Goularte

1

# Introdução

- Acesso sequencial = ?
- Busca binária = ?
- Árvores binárias de busca balanceadas = ?

2

# Introdução

- Acesso sequencial =  $O(n)$ 
  - Quanto mais as estruturas (tabelas, arquivos, etc.) crescem, mais acessos há
- Busca binária =  $O(\log(n))$ 
  - Restrita a arranjos
- Árvores binárias de busca balanceadas =  $O(\log(n))$ 
  - Não importa tamanho da tabela

3

# Introdução

- Acesso sequencial =  $O(n)$ 
  - Quanto mais as estruturas (tabelas, arquivos, etc.) crescem, mais acessos há
- Busca binária =  $O(\log(n))$ 
  - Restrita a arranjos
- Árvores binárias de busca balanceadas =  $O(\log(n))$ 
  - Não importa tamanho da tabela

Questão: é possível ser melhor do que  $O(\log(n))$ ?

4

# Introdução

- Acesso em **tempo constante**
  - Tradicionalmente, endereçamento direto em um arranjo
    - Cada chave  $k$  é mapeada na posição  $k$  do arranjo
      - Função de mapeamento  $f(k)=k$

5

# Introdução

- **Endereçamento direto**
  - Vantagens
    - Acesso direto e, portanto, rápido
      - Via indexação do arranjo
  - **Desvantagens?**

6

## Introdução

- **Endereçamento direto**
  - **Vantagens**
    - Acesso direto e, portanto, rápido
      - Via indexação do arranjo
  - **Desvantagens**
    - **Uso ineficiente do espaço** de armazenamento
      - Declara-se um arranjo do tamanho da maior chave?
      - E se as chaves não forem contínuas? Por exemplo, {1 e 100}
      - Pode sobrar espaço? Pode faltar?

7

## Introdução

- **Hashing**
  - Acesso direto, mas **endereçamento indireto**
    - Função de mapeamento  $h(k) \neq k$ , em geral
    - Resolve uso ineficiente do espaço de armazenamento
  - $O(c)$ , em média, independente do tamanho do arranjo
    - Idealmente,  $c=1$

8

## Introdução

- Hash significa (*Webster's New World Dictionary*):
  1. Fazer picadinho de carne e vegetais para cozinhar
  2. Fazer uma bagunça

9

## Hashing: conceitos e definições

- Também conhecido como **espalhamento** ou **dispersão**
- *Hashing* é uma técnica que utiliza uma **função hash** para mapear uma **chave k** (e seu registro) em um **endereço** em uma **tabela hash**
  - O endereço é usado para **armazenar** e **recuperar** registros

10

## Hashing: conceitos e definições

- **Ideia:** particionar um conjunto de elementos (possivelmente *infinito*) em um número finito de classes
  - B classes, de 0 a B - 1
  - Classes são chamadas de *buckets*

11

## Hashing: conceitos e definições

- **Conceitos relacionados**
  - A função h é chamada de **função hash**
  - $h(k)$  retorna o valor *hash* de k
  - k pertence ao **bucket**  $h(k)$

12

## Hashing: conceitos e definições

- A função hash é utilizada para **inserir**, **remover** ou **buscar** um elemento
  - Deve ser **determinística**, ou seja, resultar sempre no mesmo valor para uma determinada chave
    - Caso contrário, o que acontece?

13

## Exemplo: RGs de quem votou

- Imagine uma tabela hash de **10 posições**
  - Buckets vão de 0 a 9
- A função hash  $h$  retorna o último dígito da soma dos dois primeiros dígitos do RG de quem votou em uma eleição
- Vamos simular o funcionamento para os seguintes RGs fictícios
  - 123
  - 234
  - 567
  - ...

14

## Exemplo: RGs de quem votou

- Imagine uma tabela hash de **10 posições**
  - Buckets vão de 0 a 9
- A função hash  $h$  retorna o último dígito da soma dos dois primeiros dígitos do RG de quem votou em uma eleição
- Vamos simular o funcionamento para os seguintes RGs fictícios
  - 123
  - 234
  - 567
  - ...

Como saber se um RG já votou?

15

## Exemplo: RGs de quem votou

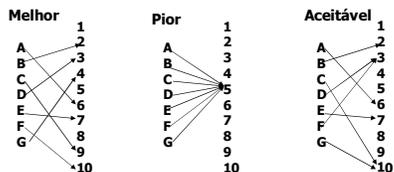
- Imagine uma tabela hash de **10 posições**
  - Buckets vão de 0 a 9
- A função hash  $h$  retorna o último dígito da soma dos dois primeiros dígitos do RG de quem votou em uma eleição
- Vamos simular o funcionamento para os seguintes RGs fictícios
  - 123
  - 234
  - 567
  - ...

E se o novo RG 299 chegar para votar?

16

## Hashing: conceitos e definições

- **Colisão**: ocorre quando a função *hash* produz o mesmo endereço para chaves diferentes
  - As chaves com mesmo endereço são ditas "sinônimos"



17

## Hashing: conceitos e definições

- **Distribuição uniforme é muito difícil**
  - Dependente de cálculos matemáticos e estatísticos complexos
- Função que aparente gerar **endereços aleatórios**
  - Existe chance de alguns endereços serem gerados mais de uma vez e de alguns nunca serem gerados

18

## Hashing: conceitos e definições

- Segredos para um bom *hashing*
  - Escolher uma boa função hash (em função dos dados)
    - Distribui uniformemente os dados, na medida do possível
    - Evita colisões
    - É fácil/rápida de computar
  - Estabelecer uma boa estratégia para *tratamento de colisões*

19

## Exemplo de função hash

- Técnica simples e muito utilizada que produz bons resultados
  - Para chaves inteiras, calcular o resto da divisão  $k/B$  ( $k\%B$ ), sendo que o resto indica a posição de armazenamento
    - $k$  = valor da chave,  $B$  = tamanho do espaço de endereçamento
  - Para chaves tipo *string*, tratar cada caracter como um valor inteiro (ASCII), somá-los e pegar o resto da divisão por  $B$
  - $B$  deve ser primo, preferencialmente

20

## Exemplo de função hash

- Exemplo
  - Seja  $B$  um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24

0	
1	
2	
3	
4	
5	
6	

21

## Exemplo de função hash

- Exemplo
  - Seja  $B$  um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - $1 \% 7 = 1$

0	
1	1
2	
3	
4	
5	
6	

22

## Exemplo de função hash

- Exemplo
  - Seja  $B$  um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24

0	
1	1
2	
3	
4	
5	5
6	

23

## Exemplo de função hash

- Exemplo
  - Seja  $B$  um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - $10 \% 7 = 3$

0	
1	1
2	
3	10
4	
5	5
6	

24

## Exemplo de função *hash*

### Exemplo

- Seja B um arranjo de 7 elementos

- Inserção dos números 1, 5, 10, 20, 25, 24

- $20 \% 7 = 6$

0	
1	1
2	
3	10
4	
5	5
6	20

25

## Exemplo de função *hash*

### Exemplo

- Seja B um arranjo de 7 elementos

- Inserção dos números 1, 5, 10, 20, 25, 24

- $25 \% 7 = 4$

0	
1	1
2	
3	10
4	25
5	5
6	20

26

## Exemplo de função *hash*

### Exemplo

- Seja B um arranjo de 7 elementos

- Inserção dos números 1, 5, 10, 20, 25, 24

- $24 \% 7 = 3$

0	
1	1
2	
3	10, 24
4	25
5	5
6	20

Colisão

27

## Exemplo de função *hash*

### Exemplo com *string*: mesmo raciocínio

- Seja B um arranjo de 13 elementos

- LOWEL = 76 79 87 69 76

- $L+O+W+E+L = 387$

- $h(\text{LOWEL}) = 387 \% 13 = 10$

28

## Exemplo de função *hash*

- Qual a ideia por trás da função hash que usa o resto?

29

## Exemplo de função *hash*

- Qual a ideia por trás da função hash que usa o resto?
  - Os elementos sempre caem no intervalo entre 0 e B-1

30

## Exemplo de função *hash*

- Qual a ideia por trás da função *hash* que usa o resto?
  - Os elementos sempre caem no intervalo entre 0 e B-1
- Como tratar *colisões*?

31

## Exemplo de função *hash*

- Qual a ideia por trás da função *hash* que usa o resto?
  - Os elementos sempre caem no intervalo entre 0 e B-1
- Como tratar *colisões*?
  - Vai depender do tipo de hashing

32

## Hashing

- 2 tipos básicos
  - **Estático**
    - Espaço de endereçamento não muda
  - **Dinâmico**
    - Espaço de endereçamento pode aumentar
      - Fora do escopo desta disciplina (assunto para ALG2)

33

## Hashing estático

- 2 tipos básicos
  - **Fechado**
    - Permite armazenar um conjunto de informações de tamanho limitado, em que os elementos são armazenados na própria tabela
  - **Aberto**
    - Permite armazenar um conjunto de informações de tamanho potencialmente ilimitado

34

## Hashing

- Tipos de *hashing*
  - **Estático**
    - **Fechado**
      - Técnicas de *rehash* para tratamento de colisões
        - *Overflow* progressivo
        - 2ª função *hash*
    - **Aberto**
      - Encadeamento de elementos para tratamento de colisões

35

## Hashing estático

- **Técnicas de *rehash***
  - Se posição  $h(k)$  está ocupada, aplicar técnica de *rehash* sobre  $h(k)$ , que deve retornar o próximo *bucket* livre
    - Características de uma boa técnica de *rehash*
      - Cobrir o máximo de índices entre 0 e B-1
      - Evitar agrupamentos de dados
  - Além de utilizar o índice resultante de  $h(k)$  na técnica de *rehash*, pode-se usar a própria chave  $k$  e outras funções *hash*

36

## Hashing estático

- Overflow progressivo
  - rehash de  $h(k) = (h(k) + i) \% B$ , com  $i$  variando de 1 a  $B-1$  ( $i$  é incrementado a cada tentativa)

37

## Hashing estático

- Overflow progressivo
  - rehash de  $h(k) = (h(k) + i) \% B$ , com  $i$  variando de 1 a  $B-1$  ( $i$  é incrementado a cada tentativa)

Questão: na busca, como saber se o elemento está ou não presente? 38

## Hashing estático

- Exemplo de dificuldade: busca pelo nome "Smith"

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

...
7 ADAMS
8 JONES
9 MORRIS
10 SMITH

Pode ter que percorrer muitos campos

39

## Hashing estático

- Exemplo de dificuldade: busca pelo nome "Smith"

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

...
7 ADAMS
8 JONES
9
10 SMITH

A remoção do elemento no índice 9 pode causar uma falha na busca

40

## Hashing estático

- Exemplo de dificuldade: busca pelo nome "Smith"

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

...
7 ADAMS
8 JONES
9 #####
10 SMITH

Solução para remoção de elementos: não eliminar elemento, mas indicar que a posição foi esvaziada e que a busca deve continuar

41

## Hashing estático

- Overflow progressivo
  - Exemplo anterior
    - rehash de  $h(k) = (h(k) + i) \% B$ , com  $i=1..B-1$ 
      - Chamada **sondagem linear**, pois todas as posições da tabela são checadas, no pior caso
  - Outro exemplo
    - rehash de  $h(k) = (h(k) + c_1 * i + c_2 * i^2) \% B$ , com  $i=1..B-1$  e constantes  $c_1$  e  $c_2$ 
      - Chamada **sondagem quadrática**, considerada melhor do que a linear, pois evita "mais" o agrupamento de elementos

42

## Hashing estático

- **Overflow progressivo**
  - Vantagem
    - Simplicidade
  - Desvantagens
    - Agrupamento de dados (causado por colisões)
    - Com estrutura cheia, a busca fica lenta
    - Dificulta inserções e remoções

Característica do *overflow progressivo*

Características do *hashing fechado*

43

## Hashing estático

- 2ª função *hash*, ou **hash duplo**
  - Uso de 2 funções
    - $h(k)$ : função *hash primária*
    - $h_{aux}(k)$ : função *hash secundária*
  - Exemplo: rehash de  $h(k) = (h(k) + i * h_{aux}(k)) \% B$ , com  $i=1...B-1$
  - Algumas boas funções
    - $h(k) = k \% B$
    - $h_{aux}(k) = 1 + k \% (B-1)$

44

## Hashing estático

- 2ª função *hash*, ou **hash duplo**
  - Vantagem
    - Evita agrupamento de dados, em geral
  - 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

45

## Hashing estático

- Alternativamente, em vez de fazer o *hashing* utilizando uma função *hash* e uma técnica de *rehash*, podemos representar isso em uma única função **dependente do número da tentativa (i)**
  - Por exemplo:  $h(k, i) = (k+i) \% B$ , com  $i=0...B-1$ 
    - A função  $h$  depende agora de dois fatores: a chave  $k$  e a iteração  $i$
    - Note que  $i=0$  na primeira execução, resultando na função *hash* tradicional de divisão que já conhecíamos
    - Quando  $i=1...B-1$ , já estamos aplicando a técnica de *rehash* de sondagem linear

46

## Exercício

- Assumindo que:
  - $B=10$
  - $h(k)=k \% B$
  - rehash de  $h(k) = (h(k)+i) \% B$ , com  $i=1...B-1$
- **Insera** os seguintes elementos em uma tabela hash utilizando *hashing* fechado com *overflow* progressivo
 

41, 10, 8, 7, 13, 52, 1, 89, 15, 21, 31
- **Busque** pelo elemento 41 e depois pelo 31
- **Remova** o elemento 1
  - Lembrete: é importante diferenciar espaços vazios nunca ocupados de espaços vazios já ocupados um dia, pois a busca fica mais eficiente
- **Busque** pelo elemento 21

47

## Hashing estático

- **Exercício em duplas**

48

## Hashing estático

- Implementar uma sub-rotina de inserção utilizando função *hash* anterior

49

## Hashing estático

- Como seria a função de busca?

50

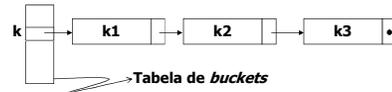
## Hashing estático

- Como seria a função de remoção?

51

## Hashing estático

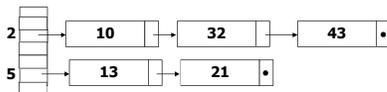
- **Hashing aberto**
  - A tabela de *buckets*, indo de 0 a  $B - 1$ , contém apenas **ponteiros para uma lista de elementos**
  - Quando há colisão, o elemento é inserido no *bucket* como um novo nó da lista
  - Busca deve percorrer a lista



52

## Hashing estático

- Se as listas estiverem **ordenadas**, reduz-se o tempo de busca
  - **Dificuldade deste método?**



53

## Hashing estático

- **Vantagens**
  - A tabela pode receber **mais itens** mesmo quando um *bucket* já foi ocupado
  - Permite percorrer a **tabela por ordem de valor *hash***
- **Desvantagens**
  - **Espaço extra** para as listas
  - **Listas longas** implicam em muito tempo gasto na busca
    - Se as listas estiverem **ordenadas**, reduz-se o tempo de busca
      - **Mas há custo extra** com a ordenação

54

## Hashing

- Pergunta
  - Quais são as principais desvantagens de *hashing*?

## Hashing

- Pergunta
  - Quais são as principais desvantagens de *hashing*?
    - Os elementos da tabela não são armazenados sequencialmente e nem sequer existe um método prático para percorrê-los em sequência

## Métodos de Busca

- Até então...
  - Busca sequencial
  - Busca binária
  - Busca em árvores binárias de busca
    - Balanceadas ou não
  - *Hashing*

57

## Métodos de Busca

- Critérios para se eleger um (ou mais) método(s)?

## Métodos de Busca

- Critérios para se eleger um (ou mais) método(s)
  - Eficiência da busca
  - Eficiência de outras operações
    - Inserção e remoção
    - Listagem e ordenação de elementos
    - Outras?
  - Frequência das operações realizadas
  - Dificuldade de implementação
  - Consumo de memória (interna)
  - Outros?