

Hashing

Organização e Recuperação de Dados

Profa. Valéria

UEM – CTC – DIN

Slides preparados com base no Cap. 10 do livro FOLK, M.J. & ZOELLICK, B. *File Structures*. 2nd Edition, Addison-Wesley Publishing Company, 1992 e nos slides disponibilizados pelo Prof. Pedro de Azevedo Berger (DCC/UnB)

Tipos de acesso

- Até este ponto, vimos como organizar arquivos de registros para três tipos de acesso: **sequencial**, **indexado** e **sequencial-indexado**
 - Arquivo sequencial
 - Busca sequencial, ordenação + busca binária
 - Arquivo indexado
 - Índices lineares, árvore-B, árvore-B*
 - Arquivo sequencial-indexado
 - Arquivo sequencial em bloco + índice de blocos, árvore-B⁺
- E o acesso **direto**? Queremos $O(1)$ → *hashing*

Introdução

- O que é *hashing*?
 - A ideia é descobrir a localização de uma chave simplesmente examinando o seu conteúdo
 - Para isso, precisamos de uma função que transforme a chave em um endereço
 - Função *hash* $\rightarrow h(k) = e$, sendo *k* uma chave e *e* um endereço
 - O endereço *e* é chamado de **endereço base** da chave
- O espaço de endereços da tabela *hash* tem que ser escolhido antecipadamente
 - P.e., podemos escolher que o *hashing* terá 1.000 endereços
 - Por isso esse tipo de *hashing* por vezes é chamado de **estático, pois seu espaço de endereços é estático**
 - A tabela *hash* é pode ser armazenada em arquivo como um conjunto de registros de tamanho fixo

Hashing

- Exemplo
 - Desejamos armazenar 75 registros em um arquivo *hash*
 - A chave de cada registro é um sobrenome de pessoa
 - Determinamos que o arquivo terá 1.000 endereços disponíveis
 - Seja U o conjunto de todas as chaves possíveis e h a função *hash*

$$h: U \rightarrow \{0,1, \dots, 999\}$$

Exemplo de h : pegue os valores ASCII correspondentes as duas primeiras letras da chave, multiplique um valor pelo outro e use os 3 dígitos mais à direita do resultado como endereço base



k	Código ASCII das 2 primeiras letras	Produto	Endereço base ($h(k) = \text{produto mod } 1.000$)
BALL	66, 65	$66 * 65 = 4.290$	290
LOWELL	76, 79	$76 * 79 = 6.004$	004
TREE	84, 82	$84 * 82 = 6.888$	888

Hashing

- Não existe uma relação óbvia entre a chave e o endereço
- Quem define essa relação é a função *hash*

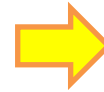
k	Código ASCII das 2 primeiras letras	Produto	Endereço base ($h(k) = \text{produto} \bmod 1.000$)
BALL	66, 65	$66 * 65 = 4.290$	290
LOWELL	76, 79	$76 * 79 = 6.004$	004
TREE	84, 82	$84 * 82 = 6.888$	888



RRN	Arquivo
000	
001	
⋮	⋮
004	LOWELL
⋮	⋮
290	BALL
⋮	⋮
888	TREE
⋮	⋮
999	

Hashing

k	Código ASCII das 2 primeiras letras	Produto	Endereço base ($h(k) = \text{produto} \bmod 1.000$)
BALL	66, 65	$66 * 65 = 4.290$	290
LOWELL	76, 79	$76 * 79 = 6.004$	004
TREE	84, 82	$84 * 82 = 6.888$	888



RRN	Arquivo
000	
001	
⋮	⋮
004	LOWELL
⋮	⋮
290	BALL
⋮	⋮
888	TREE
⋮	⋮
999	

A função *hash* do exemplo pode mapear nomes diferentes para o mesmo endereço.
Um exemplo seriam as chaves LOWELL e OLIVER.

Hashing

- Chaves mapeadas para um mesmo endereço são chamadas de **sinônimas**
- Os sinônimos geram **colisões** → tentativa de inserção em um endereço ocupado
- Evitar completamente as colisões é difícil, por isso utiliza-se **técnicas específicas para lidar com colisões**
 - P.e., endereçamento direto, encadeamento em área separada, duplo *hashing*, etc.

O hashing só será eficiente se tiver poucas colisões

Redução de colisões

- Como reduzir as colisões?
 - **Melhorar o espalhamento dos registros**
 - Dado o espaço de endereços, encontrar uma função *hash* que mapeie as chaves do modo mais uniforme possível, evitando funções que gerem “agrupamentos” em certas regiões do espaço de endereços
 - **Aumentar o espaço de endereços → Fator de carga**
 - É “mais fácil” espalhar 75 registros em 1.000 endereços do que em 100
 - *Trade-off* entre espaço e desempenho
 - **Reservar espaço para mais de um registro por endereço → Buckets**
 - Cada endereço armazenará um bloco de registros de tamanho fixo
 - Esse bloco de registros é chamado de *bucket*

Redução de colisões

- Como reduzir as colisões?

- **Melhorar o espalhamento dos registros**

- Dado o espaço de endereços, encontrar uma função *hash* que mapeie as chaves do modo mais uniforme possível, evitando funções que gerem “agrupamentos” em certas regiões do espaço de endereços

- **Aumentar o espaço de endereços → Fator de carga**

- É “mais fácil” espalhar 75 registros em 1.000 endereços do que em 100
 - *Trade-off* entre espaço e desempenho

- **Reservar espaço para mais de um registro por endereço → Buckets**

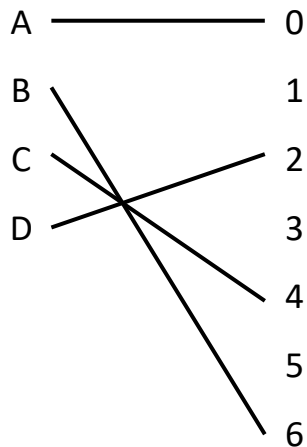
- Cada endereço armazenará um bloco de registros de tamanho fixo
 - Esse bloco de registros é chamado de *bucket*

Espalhamento dos registros

- Exemplos de possíveis distribuições

Uniforme
(sem sinônimos)

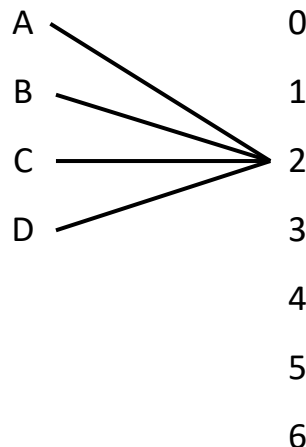
Chave	Endereço
-------	----------



Ótima

Todas as chaves são
sinônimas

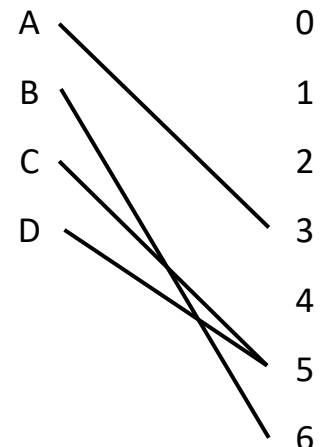
Chave	Endereço
-------	----------



Péssima

Aleatória
(alguns sinônimos)

Chave	Endereço
-------	----------



Aceitável

Distribuições uniformes são raras

Distribuições aleatórias são aceitáveis e mais fáceis de serem obtidas

Espalhamento dos registros

- Não existe uma função que gere uma distribuição melhor que a aleatória para todos os casos
 - A distribuição gerada por uma função *hash* depende do conjunto de chaves que serão espalhadas
- Portanto, a escolha da função adequada envolve
 - Consideração das chaves
 - Alguma experimentação


Exemplo de função *hash*

- Algoritmo *fold-and-add*:
 1. Se a chave não for numérica, transforme-a de modo a gerar um número
 - Por ex., uso dos valores ASCII dos caracteres da chave
 2. Divida a chave em partes, some as partes duas a duas e divida o resultado por um número primo
 3. Divida o valor resultante pelo tamanho do espaço de endereços e use o resto da divisão como endereço base para a chave em questão
 - O espaço de endereços pode ser projetado de forma que o seu tamanho também seja um número primo

Exemplo de função *hash*

1. Represente a chave como um número
 - Se a chave já é um número, pule esse passo
 - Se é a chave é uma *string* (sequência de caracteres), considere os códigos ASCII de cada caractere da *string* como uma sequência de números

LOWELL	=		L		O		W		E		L		L	
Cód. ASCII:			76		79		87		69		76		76	

767987697676

- Utilizar todos os caracteres da chave aumenta a chance de produzir endereços diferentes.
- Outra forma de converter uma *string* em um número é considerar que cada caractere é um valor entre 0 e 255. Portanto, uma cadeia não-vazia pode ser interpretada como um número na base 256.

Exemplo de função *hash*

2. Divida o número em partes e some as partes
 - Vamos considerar que cada parte é uma sequência formada por dois bytes; some duas partes e divida o resultado por um número primo; utilize o resto da divisão como parcela para a próxima soma

7679 | 8769 | 7676

$$7.679 + 8.769 = 16.448 \rightarrow 16.448 \bmod 19.937 = 16.448$$

$$16.448 + 7.676 = 24.124 \rightarrow 24.124 \bmod 19.937 = 4.187$$

- Neste exemplo, a função módulo está sendo utilizada para garantir que o resultado da soma não extrapole o valor máximo que pode ser armazenado em um inteiro de dois bytes $\rightarrow 32.767$
- O valor 19.937 foi utilizado como parâmetro da função módulo por ser um número primo
 - A divisão por um primo costuma produzir uma distribuição mais uniforme do que a divisão por um não primo

Exemplo de função *hash*

3. Dividir o resultado final pelo tamanho do espaço de endereços (preferencialmente um número primo) e usar o resto como o endereço “e”:
 - $e = \text{resultado_soma} \bmod \text{end_max}$
 - O espaço de endereços vai de 0 a $\text{end_max}-1$

$$\begin{aligned}\text{end_max} &= 101 \text{ (0 a 100)} \\ e &= 4187 \bmod 101 \\ &= 46\end{aligned}$$

Para um arquivo com 75 registros, $N = 101$ seria uma boa escolha, pois preencheria 74,2% do arquivo

A chave LOWELL ficaria no endereço 46

O objetivo do passo 3 é reduzir a magnitude do número produzido no passo 2 para que ele fique dentro do espaço de endereços projetado

Redução de colisões

- Como reduzir as colisões?
 - **Melhorar o espalhamento dos registros**
 - Dado o espaço de endereços, encontrar uma função *hash* que mapeie as chaves do modo mais uniforme possível, evitando funções que gerem “agrupamentos” em certas regiões do espaço de endereços
 - **Aumentar o espaço de endereços → Fator de carga**
 - É “mais fácil” espalhar 75 registros em 1.000 endereços do que em 100
 - *Trade-off* entre espaço e desempenho
 - **Reservar espaço para mais de um registro por endereço → Buckets**
 - Cada endereço armazenará um bloco de registros de tamanho fixo
 - Esse bloco de registros é chamado de *bucket*

Fator de carga (*Packing density*)

- O **fator de carga** de um *hashing* é a razão r/N
 - r = quantidade de chaves armazenada no *hashing*
 - N = número de endereços disponíveis
 - Mede o uso real do arquivo
- Supondo uma distribuição aleatória das chaves, quanto menor for o fator de carga, menor é a chance de colisão
 - Quanto mais denso for o arquivo (r/N mais próximo a 1), maior é a chance de colisão
 - Por isso, normalmente o espaço de endereços é projetado para ser maior do que o número de endereços que serão realmente ocupados

Fator de carga vs. colisões

- Efeito do fator de carga na % de colisões supondo uma função *hashing* que gera uma **distribuição aleatória**

Fator de carga (%)	Colisão (%)
10	4,8
20	9,4
30	13,6
40	17,6
50	21,4
60	24,8
70	28,1
80	31,2
90	34,1
100	36,8

Uma taxa de colisão de 4,8% parece muito boa, mas perceba que para isso teremos um arquivo muito maior que o necessário, pois o fator de carga é 10% → para cada endereço ocupado no arquivo, teremos 9 endereços sem utilização.

Redução de colisões

- Como reduzir as colisões?
 - **Melhorar o espalhamento dos registros**
 - Dado o espaço de endereços, encontrar uma função *hash* que mapeie as chaves do modo mais uniforme possível, evitando funções que gerem “agrupamentos” em certas regiões do espaço de endereços
 - **Aumentar o espaço de endereços → Fator de carga**
 - É “mais fácil” espalhar 75 registros em 1.000 endereços do que em 100
 - *Trade-off* entre espaço e desempenho
 - **Reservar espaço para mais de um registro por endereço → Buckets**
 - Cada endereço armazenará um bloco de registros de tamanho fixo
 - Esse bloco de registros é chamado de *bucket*

Buckets

- *Hashing com buckets*
 - É uma variação do *hashing* na qual mais de um registro pode ser armazenado no mesmo endereço
 - Um *bucket* é um bloco de tamanho fixo que estará associado um endereço único do arquivo *hash*
 - A unidade de leitura e escrita (do arquivo para RAM e vice-versa) passa a ser um *bucket* e não mais um único registro
- Essa é a **forma convencional para arquivos *hash***

Buckets

- Supondo *buckets* de tamanho 3, até 3 sinônimos poderão ser armazenados no mesmo endereço
 - Só teremos colisão a partir do 4º sinônimo

Chave	ASCII das iniciais	Produto	End. base
BALL	66, 65	$66 * 65 = 4.290$	290
LOWELL	76, 79	$76 * 79 = 6.004$	004
TREE	84, 82	$84 * 82 = 6.888$	888
OLIVER	76, 79	$76 * 79 = 6.004$	004
BALEY	66, 65	$66 * 65 = 4.290$	290
LOVELY	76, 79	$76 * 79 = 6.004$	004



RRN	Arquivo de buckets		
000			
001			
⋮	⋮	⋮	⋮
004	LOWELL	OLIVER	LOVELY
⋮	⋮	⋮	⋮
290	BALL	BALEY	
⋮	⋮	⋮	⋮
888	TREE		
⋮	⋮	⋮	⋮

Buckets

- Estimativa do número de colisões supondo uma distribuição aleatória das chaves com diferentes tamanhos de *buckets*

	Tamanho do <i>bucket</i>					
Fator de Carga	1	2	5	10	100	
100%	36,8%	27,1%	17,6%	12,5%	4,0%	(% de colisões)

Conforme o tamanho do *bucket* aumenta, diminui o número de colisões