

12 – Hashing (parte 1)

SCC201 - Introdução à Ciência de Computação II

PAE Pâmela Cândida
pamela@icmc.usp.br
Prof Moacir Ponti Jr.
www.icmc.usp.br/~moacir

Instituto de Ciências Matemáticas e de Computação – USP

2010/2



- 1 Motivação
- 2 Definições
- 3 Exemplo
- 4 Funções Hash
- 5 Tipos de Hashing



- conjunto de palavras e suas definições
- palavra \rightarrow chave
- definição \rightarrow conteúdo do endereço devolvido pela chave



- Supondo palavras de apenas 2 letras
- nosso alfabeto: 26 letras
- portanto, teríamos 26^2 posições de memória.

Segundo a ABL temos 390 mil palavras. Imagine se quiséssemos fazer busca binária para encontrar um vocábulo. Quanto custaria para inserirmos uma nova palavra? E no caso de *hash*?



Definições

hashing espalhamento;

hash table vetor[0..m-1] que armazena os registros;

hash function $\text{int hash_code}(\langle T \rangle \text{ chave});$

hash address endereço $i \in [0..m - 1]$, devolvido pela *hash function*.

Para implementar um TAD *hash* só precisamos da função *hash_code*?



Exemplo: exemplo1.c

- funções necessárias;
- boa função hash?!
- fator de carga;
- colisão.



Funções que devem ser implementadas:

- `void hash(< E > *vetor, int tamanho);`
- `int hash_code(< T > chave);`
- `void insert(< E > *vetor, < T > chave, < E > & elemento);`
- `< E > & remove(< E > *vetor, < T > chave);`
- `< E > & find(< E > *vetor, < T > chave);`



Exemplos de Funções *Hash*

- Armazenar nota de 9 alunos, sendo a chave seus respectivos números USP;
- Armazenar nota de 9 alunos, sendo a chave seus respectivos nomes;



Por que...

... usar função *hash* que retorna o resto da divisão pelo tamanho da *tabela hash*?

... usamos um número primo para o tamanho da *tabela hash*?



Por que...

... usar função hash que retorna o resto da divisão pelo tamanho da *tabela hash*?

O *endereço hash* é sempre um número entre 0 e $tamanho - 1$.

... usamos um número primo para o tamanho da *tabela hash*?

Menor número de colisões.



Fator de Carga

Tabela *hash* de tamanho m para armazenar n elementos.

Qual o número esperado de elementos em cada posição da tabela?

Fator de carga: $\alpha = n/m$



1 Estático

- Espaço de endereçamento não muda: sempre a mesma quantidade de posições disponíveis na tabela *hash*.
- Há dois tipos:
 - 1 Aberto: permite armazenar um conjunto potencialmente ilimitado de elementos.
 - 2 Fechado: permite armazenar um conjunto limitado de elementos;

2 Dinâmico

- Espaço de endereçamento pode aumentar.



- Os elementos não são armazenados na própria tabela *hash*:
 - ✓ cada posição da tabela *hash* possui um ponteiro para uma lista encadeada;
 - ¿ É festa? Todo mundo com a mesma chave?!
- Tratamento de colisões
 - ☺ inserir novo elemento: $\mathcal{O}(1)$;
 - ☹ buscar elemento, no pior caso: $\mathcal{O}(n)$

Pode-se reduzir o tempo (tempo \neq complexidade de tempo) para buscar uma chave?



¿Compensa?!

Mantendo ordenada a lista encadeada, reduz-se o tempo na busca.



¿Compensa?!

Mantendo ordenada a lista encadeada, reduz-se o tempo na busca.

¿Compensa?!

☺☹Depende! Remoção também implica em busca. Mas qual operação será realizada na maioria das vezes?



☺Vantagem:

- A tabela pode receber mais itens mesmo quando uma posição já foi ocupada:
 - ✓ permite armazenar um conjunto limitado de elementos;

☹Desvantagens:

- Espaço extra para as listas;
- Listas longas \Rightarrow muito tempo gasto na operação de busca!
 - ☺Se as listas estiverem ordenadas, reduz-se o tempo de busca,
 - ☹mas tem o custo extra para manter ordenado (durante a inserção).



- Os elementos são armazenados na própria tabela *hash*:
a isso normalmente chamamos de **endereçamento aberto**;
- Tratamento de colisões → aplicar técnicas de *rehash*:
 - 1 *overflow* progressivo;
 - 2 segunda função *hash*.



- **Overflow progressivo:**

$$rh(h(chave)) = (h(chave) + i) \% m, i \in [1..m - 1]$$

✓ i é incrementado a cada tentativa de “inserir” a mesma chave

¿ Como saber que a informação procurada não está armazenada?



Como saber que a informação procurada não está armazenada?

☹ Tem que procurar por toda a tabela!

Mas e se removermos um elemento de mesmo endereço *hash*?



- $rh(h(chave)) = (h(chave) + i)\%m, i \in [1..m - 1]$

é chamada **sondagem linear**, pois todas as posições da tabela são checadas, uma após a outra.

- $rh(h(chave)) = (h(chave) + c_1 * i + c_2 * i^2)\%m, i \in [1..m - 1]$ e c_1, c_2 constantes

é chamada **sondagem quadrática**, considerada melhor que a linear, pois evita o agrupamento de dados.



Overflow progressivo

☺ Vantagem:

- Simplicidade.

☹ Desvantagens:

- Busca fica muito lenta quando fator de carga cresce;
- Isso dificulta também inserções e remoções!



- LEVITIN, A. V. **Introduction to the Design and Analysis of Algorithms**. (Chapter 7). 1.ed. Addison-Wesley Longman Publishing Co., Inc., 2002.
- SHEWCHUK, J. **Hash Table – Data Structures University of California, Berkeley**. Disponível em: <http://www.youtube.com/watch?v=UPo-M8bzRrc>.
- ROSA, J. L. G. **Métodos de Busca**. Slides de aula SCC-201, ICMC-USP.
- COELHO, C. J. **Estrutura de Dados II – Tabela *hash***. (Capítulo 7). Disponível em: <http://www.lncc.br/~rogerio/ed/Tabela%20Hash/Tabela%20Hash.pdf>.

