13 – 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

Sumário

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

Dicionário

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

Dicionário

- 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 int hash_code(<T> 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?!
- tratamento de colisão;
  ¿É festa? Todo mundo com a mesma chave?!

TAD

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;
  ☺Definir tamanho da tabela *hash*: com 10^8 posições não há colisão, rs

- Armazenar nota de 9 alunos, sendo a chave seus respectivos nomes;
  ☹Até com 10^{100} posições pode ter colisão: nomes iguais.

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.

`http://www.cs.unm.edu/~saia/numtheory.html`

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: $\alpha = n/m$

Tipos de *Hashing*

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.

Hashing Estático Aberto

- Os elementos NÃO são armazenados na própria tabela *hash*:
  ✓cada posição da tabela *hash* possui um ponteiro para uma lista encadeada;

- Tratamento de colisões
  ☺inserir novo elemento: $\mathcal{O}(1)$:
  ☹buscar elemento, no pior caso: $\mathcal{O}(n)$

Pode-se reduzir o tempo (tempo != complexidade de tempo) para buscar uma chave?

Hashing Estático Aberto

¿Compensa?!

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

Hashing Estático Aberto

¿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?

Hashing Estático Aberto

☺Vantagem:

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

☹Desvantagens:

- Espaço extra para as listas;
- Listas longas ⇒ 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).

Hashing Estático Fechado

- Os elementos são armazenados na própria tabela *hash*:
  a isso normalmente chamamos de **endereçamento aberto**;

- Tratamento de colisões $\longrightarrow$ aplicar técnicas de *rehash*:
  1. *overflow* progressivo;
  2. segunda função *hash*.

Hashing Estático Fechado

- ***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?

Overflow progressivo

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*?

Diferenciar posição não ocupada de posição que sofreu remoção dos dados.

Overflow progressivo

☺Vantagem:

- Fácil de implementar: nada de lista encadeada, ponteiros..

☹Desvantagens:

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

Bibliografia I

SHEWCHUK, J.
Hash Table – Data Structures University of California, Berkeley..
Disponível em: `http://www.youtube.com/watch?v=UPo-M8bzRrc`

ZIVIANI, N.
Projeto de Algoritmos (Capítulo 5). 3.ed..
Cengage Learning

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C.
Introduction to Algorithms.
MIT Press, 2001

LEVITIN, A. V.
Introduction to the Design and Analysis of Algorithms (Chapter 7). 1.ed..
Addison-Wesley Longman Publishing Co., Inc., 2002.

Bibliografia II

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`