Introdução à Ciência da Computação II

Hashing Pt. I:
Funções Hash

Prof. Ricardo J. G. B. Campello

Aula de Hoje

- Mapas e Dicionários
- Tabelas Hash
- Funções Hash
  - Códigos Hash
  - Funções de Compressão

Mapas e Dicionários

- Um **mapa** ou **dicionário** é um modelo computacional genérico de uma coleção de itens do tipo chave-informação
  - uma coleção de elementos com chaves de busca
- O que diferencia mapas de dicionários é que os primeiros operam apenas com chaves primárias
  - mapas não admitem múltiplos itens com uma mesma chave
- As principais operações de mapas e dicionários são:
  - **busca**, **inserção** e **remoção** de itens
- Mapas e Dicionários são tipos abstratos de dados que podem ser implementados via diferentes estruturas de dados
    - listas, árvores, tabelas hash ...

3

Motivação

- Árvores AVL permitem realizar as operações básicas de busca, inserção e remoção de itens em tempo $O(\log n)$, onde n é o no. de itens
- É possível conseguir desempenho ainda melhor que este ???

4

Hashing

- Uma abordagem para tentar obter desempenho superior àquele das árvores AVL é conhecida como **hashing**
- Essa abordagem utiliza uma estrutura de dados denominada **tabela hash**
  - também denominada **tabela de dispersão**

5

Tabela Hash

- Uma **tabela hash** T consiste de:
  - Uma **função hash** h
  - Um vetor (tabela) V de tamanho N
- Uma função hash h mapeia chaves de um dado tipo em inteiros em um intervalo fixo $[0, N-1]$
  - O valor inteiro $h(k) \in [0, N-1]$ é chamado de **valor hash** da chave k

6

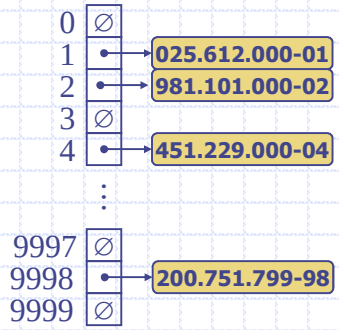
Tabela Hash

- Aplicação em Mapas / Dicionários:
  - item com chave k é armazenado no índice $i = h(k)$ da tabela
  - o índice é **calculado**, não procurado !

7

Tabela Hash – Exemplo Simples

- Tabela hash para um mapa de itens de Dados Pessoais com chave dada pelo CPF:
  - inteiro positivo com 11 dígitos
- No exemplo ao lado tem-se vetor de tamanho $N = 10.000$ e a seguinte função hash:
  - $h(k) = 4$ últimos dígitos de k
- Pode haver conflitos...
  - discutiremos posteriormente...



8

Funções Hash

- Uma função hash é composta das seguintes sub-funções:

  Código Hash (hash code):
  h_1: chaves $\rightarrow$ inteiros

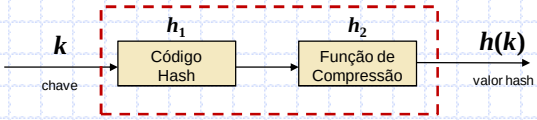
  Função de Compressão:
  h_2: inteiros $\rightarrow [0, N-1]$

- Usualmente, quando se assume que já se dispõe de uma codificação inteira das chaves, refere-se à função de compressão sozinha como "função hash"

- O código hash (quando necessário) é aplicado primeiro; em seguida a função de compressão é aplicada ao resultado:

  $h(k) = h_2(h_1(k))$

- A meta da função hash é "dispersar" as chaves de forma que essas ocupem a tabela da forma mais uniforme possível



Funções Hash e Colisões

- Colisões ocorrem quando diferentes chaves são mapeadas sem distinção na mesma célula da tabela
- Colisões podem ocorrer tanto na fase de codificação das chaves (h_1) como na fase de compressão (h_2)
- Ocorrendo na fase de codificação, não há como serem revertidas na fase de compressão. Por exemplo:
  - suponha que duas chaves string, k_1 = "Joao" e k_2 = "Maria", sejam ambas codificadas como $h_1(k_1) = h_1(k_2) = 3546$
  - qualquer que seja a função de compressão, ter-se-á:
    - $h(k_1) = h_2(h_1(k_1)) = h_2(3546) = h_2(h_1(k_2)) = h(k_2)$

10

Funções Hash e Colisões

- Funções hash devem ser simples e rápidas de calcular, minimizando ao máximo colisões
- Colisões requerem tratamentos *a posteriori* que, por sua vez, demandam esforço computacional
  - vide próxima aula...
- Note que é impossível evitar completamente colisões se o **fator de carga** λ de uma tabela hash for $\lambda > 1$
- Esse fator é definido como a razão entre o número de chaves armazenadas, n, pelo tamanho da tabela, N
  - Por exemplo, tabela de tamanho 10000 com 1000 chaves:
    - $\lambda = n / N = 0.1$

11

Códigos Hash

- Casting
  - Interpretação dos bits da chave como um inteiro
  - Por exemplo, para chaves numéricas reais:
    - $k = m*10^{-e}$ (representação em ponto flutuante)
  - Uma alternativa é somar os inteiros correspondentes à mantissa e o expoente do número em representação de ponto flutuante
    - $h_1(k) = m + e$
  - Exemplo: no quadro...

12

Códigos Hash

Soma de Componentes

- A idéia de somar inteiros correspondentes a partes de uma representação pode ser estendida para qualquer chave k que possa ser representada por uma série $k_0, k_1, \ldots, k_{m-1}$ de inteiros
- Em outras palavras:

$$h_1(k) = \sum_{i=0}^{m-1} k_i$$

- Útil em muitos casos, mas incapaz de distinguir chaves distintas que diferem entre si apenas pela ordem das componentes k_i
- Esse é o caso de chaves tipo **string**, representadas como séries de inteiros dados pelos códigos ASCII de cada um dos seus caracteres (ou pela ordem alfabética do caractere)

13

Códigos Hash

$$h_1(k) = \sum_{i=0}^{m-1} k_i$$

Soma de Componentes

- Exemplos:
  - k = "Joao" $\Rightarrow$ $h_1(k) = 74 + 111 + 97 + 111 = 393$
  - k = "cara" $\Rightarrow$ $h_1(k) = 99 + 97 + 114 + 97 = 407$
  - k = "arca" $\Rightarrow$ $h_1(k) = 97 + 114 + 99 + 97 = 407$

14

Códigos Hash

Acumulação Polinomial

- Ao invés de uma soma simples, utiliza-se o seguinte polinômio:

  $h_1(k) = k_0 + k_1 a + k_2 a^2 + \ldots + k_{m-1} a^{m-1}$

  para um valor fixo de a

- Funciona bem, em especial para strings
  - p. ex. as escolhas empíricas $a = 33$, 37, 39 ou 41 geram em torno de apenas 6 colisões para 50.000 palavras em Inglês !
  - Muitas vezes basta tomar apenas um subconjunto dos m (p. ex. 10) primeiros / últimos caracteres da string

15

Códigos Hash

$$h_1(k) = \sum_{i=0}^{m-1} k_i a^i$$

Acumulação Polinomial (Exemplos):

- k = "Joao" (a = 33)
  - $h_1(k) = 74 + 111*33 + 97*33^2 + 111*33^3 = 4098377$
- k = "cara" (a = 33)
  - $h_1(k) = 99 + 97*33 + 114*33^2 + 97*33^3 = 3613335$
- k = "arca" (a = 33)
  - $h_1(k) = 97 + 114*33 + 99*33^2 + 97*33^3 = 3597559$

16

Códigos Hash

$$h_1(k) = \sum_{i=0}^{m-1} k_i a^i$$

- Acumulação Polinomial (Implementação Ingênua):

```
#include <math.h>

int Hash_Code(char chave[], int m_max, int a){
   int codigo_hash = 0, i = 0;
   while (i<=m_max-1 && chave[i]!='\0'){
         codigo_hash = codigo_hash + chave[i]*pow(a,i);
         i++;
   }
   return codigo_hash;
}
```

17

Códigos Hash

$$h_1(k) = \sum_{i=0}^{m-1} k_i a^i$$

- Acumulação Polinomial (Implementação Eficiente):

```
int Hash_Code(char chave[], int m_max, int a){
   int codigo_hash = 0; i = 0, a_i = 1;
   while (i<=m_max-1 && chave[i]!='\0'){
         codigo_hash = codigo_hash + chave[i]*a_i;
         i++;
         a_i = a_i*a;
   }
   return codigo_hash;
}
```

18

Funções de Compressão

- Resto da Divisão
  - $h_2(y) = \text{abs}(y) \mod N$
  - Exemplo:
    - Códigos hash: $y = h_1(k)$ = [200, 205, 210, 215, ..., 595, 600]
    - Para N = 101, tem-se os seguintes valores hash:
      - $h_2(y)$ = [99, 3, 8, 13, ..., 90, 95]
      - pois os múltiplos de 101 são 101, 202, 303, 404, 505, ...
    - Note que não ocorre qualquer colisão nesse caso

* OBS. mod (% em C) é o **resto da divisão**, definido matematicamente como x mod z = x – piso(x/z)*z

Funções de Compressão

- Resto da Divisão (Implementação)

```
#include <math.h>

int F_Compressao(int codigo_hash, int N){
   return (abs(codigo_hash) % N);
}
```

  - Nota:
    - o valor absoluto permite lidar com códigos hash negativos...
    - mas produz colisões com suas contrapartidas positivas...

Funções de Compressão

- Resto da Divisão (cont.)
  - O tamanho N da tabela é usualmente escolhido número primo
    - A razão formal está relacionada com a teoria dos números
    - Informalmente, padrões do tipo $y = pN+q$ para q inteiro positivo e $p = 0,1,2,...$ são menos comuns em códigos hash para N primo
    - Note que qualquer código hash seguindo este padrão possui o mesmo valor hash $h_2(y)$!
  - Exemplo: y = [200, 205, 210, 215, ..., 595, 600]
    - Para $N = 101$, já vimos que não ocorre qualquer colisão
    - Já para $N = 100$, cada código terá o mesmo valor hash de ao menos outros 3 códigos !

21

Funções de Compressão

- Resto da Divisão (cont.)
  - Heurística:
    - Estimar a quantidade de chaves n e definir N como o no. primo que mais aproxima o fator de carga $\lambda = n/N$ desejado
  - Exemplo:
    - $n = 2000$ chaves e fator de carga desejado $\lambda \cong 3$
    - escolhe-se $N = 701$ como no. primo mais próximo de 2000/3

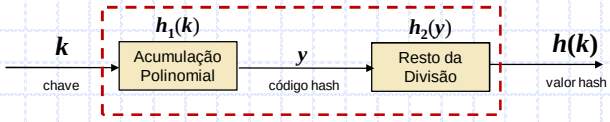
22

Funções de Compressão

- Existem outras funções de compressão:
  - Multiplicação-Divisão
  - Multiply, Add and Divide (MAD)
  - Funções Universais
- De formas distintas, essas funções buscam reduzir a sensibilidade à escolha de N e/ou minimizar a probabilidade de colisões

23

Exemplo de Função Hash




```
#define TAMANHO_MAX_CHAVE 10
#define CTE_POLINOMIAL 33

int Funcao_Hash(char chave[], int N){
   int codigo_hash = Hash_Code(chave, TAMANHO_MAX_CHAVE, CTE_POLINOMIAL);
   return F_Compressao(codigo_hash, N);
}
```

24

Exercícios

- Calcule códigos hash para as chaves numéricas reais abaixo utilizando a técnica de casting:
  - $135*10^{-8}$, $563*10^{-2}$, $864*10^{-30}$, $298*10^{-5}$
- Calcule códigos hash para as chaves string abaixo utilizando a técnica da soma de componentes:
  - "orca", "arco", "caro", "carro", "carrossel", "carroceria"
- Repita o exercício acima utilizando a técnica da acumulação polinomial com constante $a = 3$
- Obtenha os valores hash a partir dos códigos hash resultantes de ambos os exercícios anteriores através do método do resto da divisão com $N = 71$

Exercícios

- Repita o exercício anterior para $N = 6$ (igual ao número de chaves) e verifique quantas colisões ocorrem em cada técnica de codificação hash
- Pratique os exercícios anteriores para outras chaves e outros parâmetros a e N
- Suponha que se queira armazenar 100 chaves em uma tabela hash com fator de carga $\lambda \cong 4$. Escolha o tamanho apropriado N da tabela segundo a heurística do número primo mais próximo

Para saber mais...

- Capítulo 8 (Goodrich & Tamassia, 2002)
- Capítulo 8 (Szwarcfiter & Markenzon, 1994)
- Capítulo 5 (Ziviani, 2004)

27

Bibliografia

- M. T. Goodrich & R. Tamassia, *Data Structures and Algorithms in C++/Java*, John Wiley & Sons, 2002/2005
- M. T. Goodrich & R. Tamassia, *Estruturas de Dados e Algoritmos em Java*, Bookman, 2002
- J. L. Szwarcfiter & L. Markenzon, *Estruturas de Dados e seus Algoritmos*, LTC, 1994
- N. Ziviani, *Projeto de Algoritmos*, Thomson, 2a. Edição, 2004
- T. H. Cormen et al., *Introduction to Algorithms*, MIT Press, 2nd Edition, 2001

28