2013-02 – Aulas 22 e 23
Adaptado por Reinaldo Fortes para o curso de 2013-02
Arquivo original: 24._hashing

Tabelas Hash

Prof. Túlio Toffolo
http://www.toffolo.com.br

BCC202 – Aula 21
Algoritmos e Estruturas de Dados I

Pesquisa em Memória Primária

- Introdução - Conceitos Básicos
- Pesquisa Sequencial
- Pesquisa Binária
- Árvores de Pesquisa
  - Árvores Binárias de Pesquisa
  - Árvores AVL
- **Transformação de Chave (Hashing)**
  - **Listas Encadeadas**
  - **Endereçamento Aberto**
  - **Hashing Perfeito**

Transformações de Chaves

FUNÇÃO DE HASH

Transformação de Chave (Hashing)

- Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.

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

Transformação de Chave (Hashing)

- Um método de pesquisa com o uso da transformação de chave é constituído de duas etapas principais:
  1. Computar o valor da **função de transformação**, a qual transforma a chave de pesquisa em um endereço da tabela.
  2. Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método para lidar com **colisões**.

Transformação de Chave (Hashing)

- Qualquer que seja a função de transformação, algumas colisões fatalmente ocorrerão.
- Tais colisões têm de ser resolvidas de alguma forma.
- Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entradas da tabela, existe uma alta probabilidade de haver colisões.

Colisões

- O paradoxo do aniversário (Feller, 1968, p. 33)
  - Em um grupo de 23 ou mais pessoas, existe uma chance maior do que 50% de que 2 pessoas comemorem o aniversário no mesmo dia.

- Assim, se for utilizada uma função de transformação uniforme que enderece 23 chaves randômicas em uma tabela de tamanho 365, a probabilidade de que haja colisões é maior do que 50%.

Transformação de Chave (Hashing)

- A probabilidade p de se inserir N itens consecutivos sem colisão em uma tabela de tamanho M é:

$$p = \frac{M-1}{M} \times \frac{M-2}{M} \times \ldots \times \frac{M-N+1}{M} = \prod_{i=1}^{N} \frac{M-i+1}{M} = \frac{M!}{(M-N)!M^N}$$

Transformação de Chave (Hashing)

- Alguns valores de p para valores de N, com M = 365.

N	P
10	0,883
22	0,524
23	0,493
30	0,303

Funções de Transformação

- Uma função de transformação deve mapear chaves em inteiros dentro do intervalo [0...M – 1], onde M é o tamanho da tabela.
- A função de transformação ideal é aquela que:
  - Seja **simples** de ser computada.
  - Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.

Transformações de Chaves NUMÉRICAS

- Usa o resto da divisão por M .

$$h(K) = K \% M \quad \text{(em linguagem C)}$$

onde K é um inteiro correspondente à chave.

- Cuidado na escolha do valor de M:
  - Potências de dois devem ser evitadas.
  - Deve ser um número primo distante de pequenas potências de dois (grande).

Transformações de Chaves NÃO NUMÉRICAS

- As chaves não numéricas devem ser transformadas em números:

$$K = \sum_{i=1}^{n} \text{Chave}[i] \times p[i],$$

  - n é o número de caracteres da chave.
  - Chave[i] corresponde à representação do i-ésimo caractere da chave na codificação usada.
  - p[i] é um inteiro de um conjunto de pesos gerados randomicamente para 1 ≤ i ≤ n.

Transformações de Chaves NÃO NUMÉRICAS

- Vantagem de se usar pesos:
  - Dois conjuntos diferentes de pesos p1[i] e p2[i], $1 \leq i \leq n$, levam a duas funções de transformação $h_1(K)$ e $h_2(K)$ diferentes.

Transformações de Chaves NÃO NUMÉRICAS

- Programa que gera um peso para cada caractere de uma chave constituída de n caracteres:

```
/* Gera valores randomicos entre 1 e 10.000 */
void GeraPesos(int p[], int n) {
    int i;
    // utilizando o tempo como semente de números aleatórios
    srand(time(NULL));
    for (i = 0; i < n; i++)
        p[i] = 1 + (int) (10000.0*rand() / RAND_MAX);
}
```

Transformações de Chaves NÃO NUMÉRICAS

- Implementação da função de transformação:

```
/* Função de hash que retorna o índice (número inteiro)
 * de uma chave (string) */
int h(char *chave, int p[], int m, int tam_p){
    int i;
    unsigned int soma = 0;
    int comp = strlen(chave);

    for (i = 0; i < comp; i++)
        soma += (unsigned int) chave[i] * p[i%tam_p];

    return (soma % m);
}
```

Resolvendo colisões

- Listas Encadeadas
- Endereçamento Aberto

HASH – Tratamento de Colisões

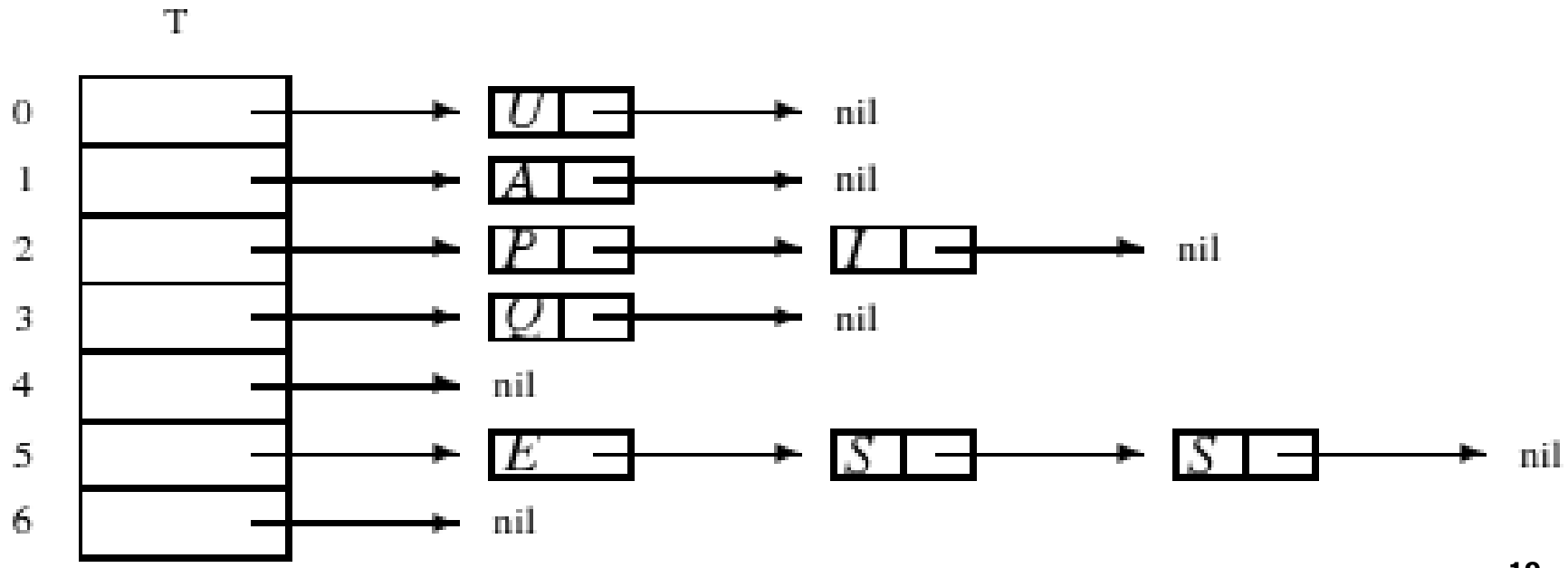
LISTAS ENCADEADAS

Listas Encadeadas

- Uma das formas de resolver as **colisões** é simplesmente construir uma lista linear encadeada para cada endereço da tabela.
- Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.

Listas Encadeadas

- **Exemplo:** Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação h(chave) = chave mod M é utilizada para M = 7, o resultado da inserção das chaves P E S Q U I S A na tabela é o seguinte:
  - h(A) = h(1) = 1, h(E) = h(5) = 5, h(S) = h(19) = 5, …



Hash Usando Listas Encadeadas

```
#define N 16 // tamanho da chave (string)

typedef char TChave[N];

typedef struct {
    /* outros componentes */
    TChave chave;
} TItem;

typedef struct celula {
    struct celula *pProx;
    TItem item;
} TCelula;

typedef struct {
    TCelula *pPrimeiro, *pUltimo;
} TLista;
```

Hash Usando Listas Encadeadas

```
#include "lista.h"

typedef struct {
    int n; // numero de itens na hash
    int nro_listas;  // tamanho do array de listas
    int nro_pesos; // tamanho do array de pesos
    int *p; // array de pesos
    TLista *v; // array de listas
} THash;

void THash_Inicia(THash *hash, int nro_listas, int nro_pesos);
int THash_H(THash *hash, TChave chave);
int THash_Pesquisa(THash *hash, TChave chave, TItem *x);
TCelula *THash_PesquisaCelula(THash *hash, TChave chave);
int THash_Insere(THash *hash, TItem x);
int THash_Remove(THash *hash, TItem *x);
```

Dicionário Usando Listas Encadeadas

```
/* Função de hash que retorna o índice (número inteiro)
 * de uma chave (string) */
int THash_H(THash *hash, TChave chave) {
    int i;
    unsigned int soma = 0;
    int comp = strlen(chave);

    for (i = 0; i < comp; i++)
        soma += (unsigned int) chave[i] *
                hash->p[i % hash->nro_pesos];

    return (soma % hash->nro_listas);
}
```

Dicionário Usando Listas Encadeadas

```
/* Inicializa a hash. Parametros: p = nro de pesos
 *                               m = tamanho vetor de listas */
void THash_Inicia(THash *hash, int nro_listas, int nro_pesos) {
    int i;
    hash->n = 0;
    hash->nro_listas = nro_listas;
    hash->nro_pesos = nro_pesos;

    // inicializando as listas
    hash->v = (TLista*) malloc(sizeof(TLista) * nro_listas);
    for (i = 0; i < nro_listas; i++)
        TLista_Inicia(&hash->v[i]);

    // inicializando os pesos
    hash->p = (int*) malloc(sizeof(int) * nro_pesos);
    for (i = 0; i < nro_pesos; i++)
        hash->p[i] = rand() % 100000;
}
```

Hash Usando Listas Encadeadas

```
/* Retorno do ponteiro para a celula ANTERIOR da lista */
TCelula *THash_PesquisaCelula(THash *hash, TChave chave) {
    int i = THash_H(hash, chave);
    TCelula *aux;

    if (TLista_EhVazia(&hash->v[i]))
        return NULL; // pesquisa sem sucesso

    aux = hash->v[i].pPrimeiro;
    while (aux->pProx->pProx != NULL && strcmp(chave, aux->pProx->item.chave) != 0)
        aux = aux->pProx;

    if (!strncmp(chave, aux->pProx->item.chave, sizeof(TChave)))
        return aux;
    else
        return NULL; // pesquisa sem sucesso
}
```

Hash Usando Listas Encadeadas

```
/* Retorna se a pesquisa foi bem sucedida e o item (x) por meio
 * de passagem por referência */
int THash_Pesquisa(THash *hash, TChave chave, TItem *x) {
    TCelula *aux = THash_PesquisaCelula(hash, chave);
    if (aux == NULL)
        return 0;
    *x = aux->pProx->item;
    return 1;
}
```

Hash Usando Listas Encadeadas

```
int THash_Insere(THash *hash, TItem x) {
    if (THash_PesquisaCelula(hash, x.chave) == NULL) {
        TLista_Insere(&hash->v[THash_H(hash, x.chave)], x);
        hash->n++;
        return 1;
    }
    return 0;
}

int THash_Remove(THash *hash, TItem *x) {
    TCelula *aux = THash_PesquisaCelula(hash, x->chave);

    if (aux == NULL)
        return 0;
    TLista_Remove(&hash->v[THash_H(hash, x->chave)], aux, x);
    hash->n--;
    return 1;
}
```

HASH – Tratamento de Colisões

LISTAS ENCADEADAS

ANÁLISE

Análise

- Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada de T, então o comprimento esperado de cada lista encadeada é N/M, onde:
  - N representa o número de registros na tabela
  - M representa o tamanho da tabela.
- Logo: as operações Pesquisa, Insere e Retira custam O(1 + N/M) operações em média, onde a constante 1 representa o tempo para encontrar a entrada na tabela e N/M o tempo para percorrer a lista.

Análise

- Se os valores forem bem distribuídos (poucas colisões) e N for igual a M, teremos que:
  - Pesquisa, inserção e remoção serão O(1)

HASH – Tratamento de Colisões

ENDEREÇAMENTO ABERTO

Endereçamento Aberto

- Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, então não haverá necessidade de usar apontadores para armazenar os registros.
- Existem vários métodos para armazenar N registros em uma tabela de tamanho M > N , os quais utilizam os lugares vazios na própria tabela para resolver as colisões. (Knuth, 1973, p.518)

Endereçamento Aberto

- **No Endereçamento aberto** todas as chaves são armazenadas na própria tabela, sem o uso de ponteiros explícitos.
- Existem várias propostas para a escolha de localizações alternativas. A mais simples é chamada de **hashing linear**, onde a posição h_j na tabela é dada por:

$$h_j = (h(x) + j) \underline{\text{mod}}\ M, \text{ para } 0 \leq j \leq M - 1.$$

Exemplo

- Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação

$$h(\text{chave}) = \text{chave } \underline{\text{mod}}\ M$$

é utilizada para M = 7, teremos:

- h(L) = h(12) = 5
- h(U) = h(21) = 0
- h(N) = h(14) = 0
- h(E) = h(5) = 5
- h(S) = h(19) = 5.

Exemplo

- Por exemplo:

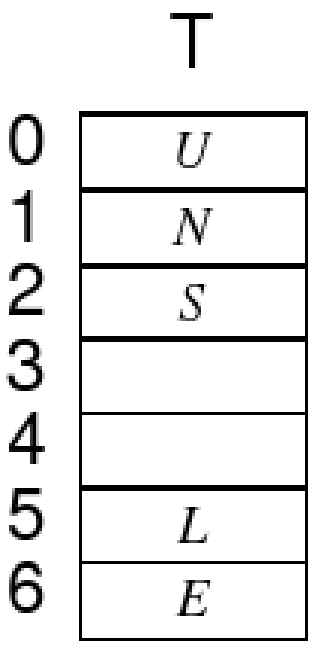
  h(L) = h(12) = 5,

  h(U) = h(21) = 0,

  h(N) = h(14) = 0,

  h(E) = h(5) = 5,

  h(S) = h(19) = 5.



Dicionário usando Endereçamento Aberto

```
#define VAZIO "!!!!!!!!!\0"
#define N 10 // tamanho da chave (string)
#define M 100 // tamanho da tabela

typedef char TChave[N];

typedef struct {
  /* outros componentes */
  TChave Chave;
} TItem;

typedef TItem TDicionario[M];
```

Dicionário usando Endereçamento Aberto

```
void TDicionario_Inicia(TDicionario dic){
    int i;
    for (i = 0; i < M; i++)
       memcpy(dic[i].chave, VAZIO, N);
}
```

Dicionário usando Endereçamento Aberto

```
int TDicionario_Pesquisa(TDicionario dic, TChave chave, int *p) {
    int i = 0;
    int ini = h(chave, p);

    while (strcmp(dic[(ini + i) % M].chave, VAZIO) != 0 &&
           strcmp(dic[(ini + i) % M].chave, chave) != 0 && i < M)
        i++;

    if (strcmp(dic[(ini + i) % M].chave, chave) == 0)
        return (ini + i) % M;

    return -1; // pesquisa sem sucesso
}
```

Dicionário usando Endereçamento Aberto

```
int TDicionario_Insere(TDicionario dic, TItem x, int *p){
    if (TDicionario_Pesquisa(dic, x.chave, p) >= 0)
        return 0; // chave já existe no dicionário

    int i = 0;
    int ini = h(x.chave, p);
    while (strcmp(dic[(ini + i) % M].chave, VAZIO) != 0 && i < M)
        i++;

    if (i < M) {
        dic[(ini + i) % M] = x;
        return 1;
    }
    return 0;
}
```

Dicionário usando Endereçamento Aberto

```
int TDicionario_Retira(TDicionario dic, TItem *x, int *p) {
    int i = TDicionario_Pesquisa(dic, x->chave, p);
    if (i == -1)
        return 0; // chave não encontrada

    (*p) = dic[i];
    memcpy(dic[i].chave, VAZIO, N);
    return 1;
}
```

HASH – Tratamento de Colisões

ENDEREÇAMENTO ABERTO

ANÁLISE

Análise

- Seja α = N/M o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa com sucesso é:

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

- O hashing linear sofre de um mal chamado agrupamento (clustering) (Knuth, 1973, pp.520–521).
  - Este fenômeno ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas.

Análise

- Apesar do hashing linear ser um método relativamente pobre para resolver colisões os resultados apresentados são bons.
- O melhor caso, assim como o caso médio, é O(1).
- Qual o pior caso?
  - Quando é necessário percorrer toda a tabela para encontrar uma chave (excesso de colisões).
  - Complexidade do pior caso: O(n)

TABELAS HASH

VANTAGENS E DESVANTAGENS

Vantagens e Desvantagens

- Vantagens:
  - Alta eficiência no custo de pesquisa, que é **O(1)** para o caso médio.
  - Simplicidade de implementação.
- Desvantagens:
  - Custo para recuperar os registros ordenados pela chave é alto, sendo necessário ordenar toda a tabela.
  - Pior caso para a busca é **O(n)**.

Perguntas?