PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

(baseado nas notas de aula do prof. Túlio A. M. Toffolo)
Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Mapas

Descrição

Mapas, ou **dicionários**, são estruturas de dados associativas, ou seja, armazenam elementos formados por pares (chave, valor).

As chaves são utilizadas para indexar os valores, portanto, um mapa "mapeia chaves para valores".

Chaves e valores podem ser qualquer tipo de dados.

Justificativa

O endereçamento direto de elementos é especialmente útil para acessarmos um elemento arbitrário em tempo $O(1)$. Nesse caso, o valor da chave é seu endereço em um vetor.

Todavia, se o universo de chaves é grande, o armazenamento de uma tabela de tamanho idêntico pode ser impraticável.

Por exemplo, como armazenar as chaves 0, 100 e 9.999? Um vetor de 10.000 posições?

Além disso, o subconjunto de chaves realmente utilizadas pode ser muito pequeno, causando grande desperdício de espaço.

Mapas

Operações Válidas

- Inserir elementos;
- Remover elementos;
- Percorrer os elementos;
- Pesquisar elementos;
- Obter o número de elementos.

Operações Inválidas

- Inserir elementos repetidos (possível apenas em multimapas);
- Ordenar;
- Recuperar o elemento sucessor ou antecessor a outro.

Implementação

Um mapa pode manter suas chaves ordenadas ou não, sendo esta a principal diferença quanto à implementação:

Mapa ordenado Árvores Binárias Balanceadas.

Mapa não ordenado Tabelas Hash.

Tabelas Hash

Hash significa:

- Fazer picadinho de carne e vegetais para cozinhar.
- Fazer uma bagunça.
- Espalhar, transformar (na computação).

Em uma **Tabela Hash**, os registros armazenados em um vetor são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.

Tabelas Hash

Quando o conjunto K de chaves armazenadas é menor que o universo de todas as chaves possíveis, Tabelas Hash podem reduzir os requisitos de armazenamento a $\Theta(|K|)$.

Em tabelas hash, ao invés de inserir o elemento k na posição k, o elemento é inserido na posição $h(k)$.

A chamada Função Hash é utilizada para calcular a posição na Tabela Hash. Sua função é diminuir o intervalo de índices necessários.

Tabelas Hash

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.

Colisões

Qualquer que seja a função de transformação, algumas colisões irão ocorrer fatalmente, e 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 aleatórias em uma tabela de tamanho 365, a probabilidade de que haja colisões é maior do que 50%.

Colisões

A probabilidade p de se inserir n itens consecutivos sem colisão em uma tabela de tamanho m é:

$$
\begin{aligned}
p &= \frac{m-1}{m} \times \frac{m-2}{m} \times \ldots \times \frac{m-n-1}{m} \\
&= \prod_{i=1}^{n} \frac{m-i-1}{m} \\
&= \frac{m!}{(m-n)!m^n}
\end{aligned}
$$

Colisões

Consideremos alguns valores de p para valores de n, com $m = 365$.

Para n pequeno, a probabilidade p pode ser aproximada por $p \approx n(n-1)/730$.

Por exemplo, para $n = 10$ então $p \approx 87{,}7\%$.

n	p
10	0,883
22	0,524
23	0,493
30	0,303

Função Hash

Uma função de transformação deve mapear chaves para valores inteiros dentro do intervalo $[0 \ldots m-1]$, em que m é o tamanho da tabela de armazenamento.

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.

Funções Hash Comuns

Existem diversas técnicas para o cálculo, algumas mais simples e rápidas, outras mais eficientes e mais lentas.

Basicamente, dois métodos são muito utilizados: **Método de Divisão e Método de Multiplicação**.

O **Método da Divisão** utiliza o resto da divisão por m: $h(k) = k \bmod m$, em que k é um inteiro correspondente à chave.

Método da Divisão

É necessário tomar cuidado na escolha do valor de m, que deve preferencialmente ser um número primo, mas não qualquer primo.

Devem ser evitados os números primos obtidos a partir de $b \times i \pm j$, em que b é a base do conjunto de caracteres.

Geralmente, $b = 64$ para BCD, 128 para ASCII, 256 para EBCmp, ou 100 para alguns códigos decimais, e i e j são pequenos inteiros.

Transformações de Chaves não numéricas

As chaves não numéricas devem ser transformadas em números:

$$k = \sum_{i=1}^{n} Chave[i] \times p[i] \quad (1)$$

- n é o número de caracteres da chave.
- Chave[i] corresponde à representação ASCII do i-ésimo caractere da chave.
- p[i] é um inteiro de um conjunto de pesos gerados aleatoriamente para $1 \leq i \leq n$.

Transformações de Chaves não numéricas

A utilização de pesos fornece uma vantagem considerável:

Dois conjuntos diferentes de pesos $p_1[i]$ e $p_2[i]$, $1 \leq i \leq n$, 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 aleatorios entre 1 e 10.000
void GeraPesos(int p[], int n) {
     int i;
     // utiliza o tempo como semente de numeros aleatorios
     srand(time(NULL));
     for (i = 0; i < n; i++)
          p[i] = 1 + (int) (10000.0*rand() / RAND_MAX);
}
```

Exemplo

Transformações de Chaves não numéricas

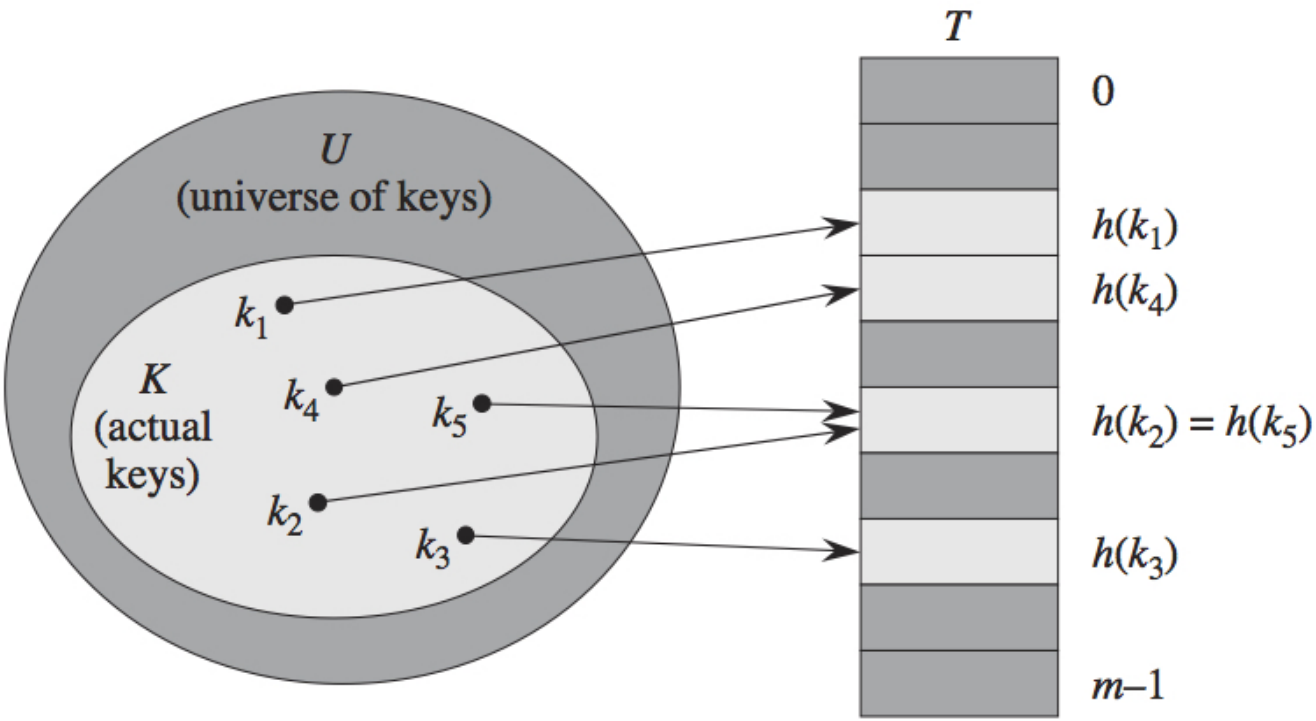
Implementação da função de transformação.

```
// Funcao hash que retorna o indice (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

Há duas formas básicas para resolução de colisões:

- Listas Encadeadas;
- Endereçamento Aberto.



Listas Encadeadas

Uma das formas de resolver as colisões é simplesmente construir uma lista linear encadeada para cada endereço da tabela.

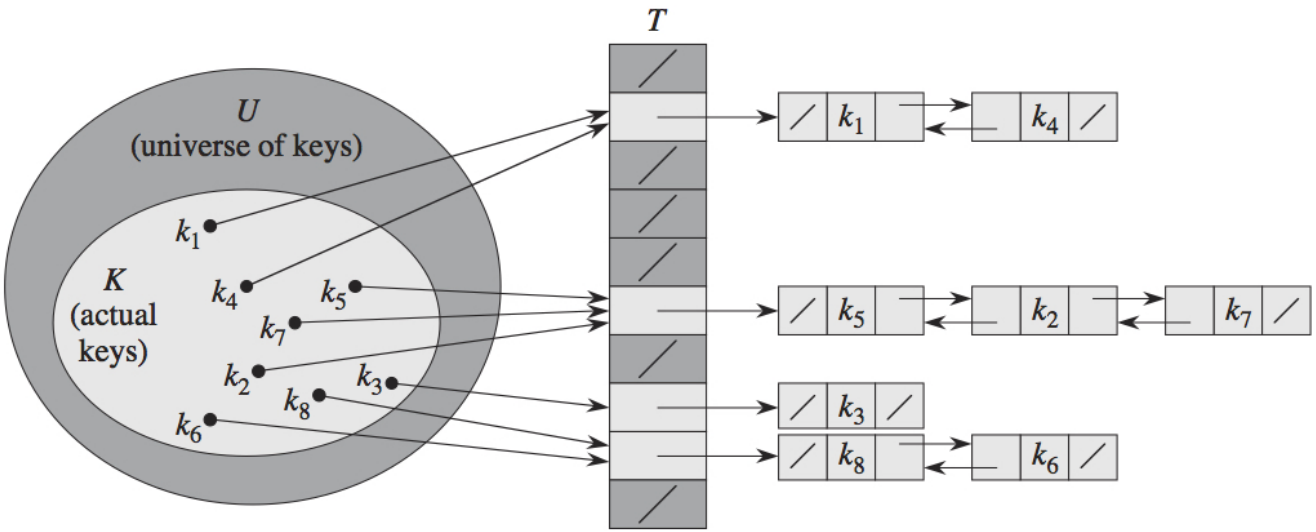
Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.

Tabelas Hash Usando Listas Encadeadas

Cada posição da tabela hash $T[j]$ contém uma lista encadeada com todas as chaves cujo hash seja j.

Por exemplo, $h(k_1) = h(k_4)$ e $h(k_5) = h(k_7) = h(k_2)$.

Se listas duplamente encadeadas forem utilizadas, a remoção de elementos se torna mais rápida.



Tabelas 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;
```

Tabelas Hash Usando Listas Encadeadas

```c
#include "lista.h"

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

Tabelas Hash Usando Listas Encadeadas

```
// Funcao de hash
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);
}
```

Tabelas Hash Usando Listas Encadeadas

```
// Inicializa a tabela hash.
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;
}
```

Tabelas Hash Usando Listas Encadeadas

```
// Retorna o ponteiro apontando para a celula anterior
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
}
```

Tabelas Hash Usando Listas Encadeadas

```
// Retorna se a pesquisa foi bem sucedida e o item (x)
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;
}
```

Tabelas 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;
}
```

Fator de Carga

Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada de T, então o fator de carga, dado pelo comprimento esperado de cada lista encadeada é n/m, em que:

- n representa o número de registros na tabela.
- m representa o tamanho da tabela.

Tabelas Hash Usando Listas Encadeadas

Melhor Caso

Se os valores forem bem distribuídos (poucas colisões) e n for proporcional a m, teremos que as operações Pesquisar, Inserir e Remover terão complexidade $O(1)$.

Pior Caso

Se os valores forem mal distribuídos, com todas as chaves possuindo o mesmo endereçamento, teremos uma única lista de tamanho n. Portanto, teremos que as operações Pesquisar, Inserir e Remover terão complexidade $\Theta(n)$.

Caso Médio

As operações Pesquisar, Inserir e Remover custam $\Theta(n/m)$, ou $1 + n/m$ operações em média, em que a constante 1 representa o tempo para encontrar a entrada na tabela e n/m o tempo para percorrer a lista.

Tabelas Hash Usando Endereçamento Aberto

Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, então não há 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 as posições vazias na própria tabela para resolver as colisões. (Knuth, 1973, p.518)

Desta forma, a memória é melhor utilizada.

Tabelas Hash Usando Endereçamento Aberto

Para pesquisar um determinado elemento na tabela hash usando endereçamento aberto, analisamos (ou sondamos) sistematicamente as posições da tabela até encontrarmos o elemento ou até concluirmos por sua inexistência na tabela.

Para inserirmos um novo elemento usando o endereçamento aberto, novamente sondamos as posições da tabela até encontrarmos uma posição livre para a inserção.

Os métodos de sondagem incluem sondagem linear, quadrática e hash duplo.

Sondagem Linear

Dada uma função hash auxiliar comum h' a sondagem linear usa a função hash

$$h(k, i) = (h'(k) + i) \text{ mod } m$$

em que $i = 0, 1 \ldots m - 1$

A primeira tentativa de inserção é realizada na posição $h'(k)$, e as demais possuem um deslocamento de uma unidade ($h'(k)$+1) e assim por diante até a última posição.

Este método sofre do problema de agrupamento primário, em que longas sequências de posições ocupadas se acumulam, aumentando o tempo de pesquisa.

Exemplo

Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação $h(\text{chave}) = \text{chave mod } m$ é utilizada para $m = 7$, teremos:

- $h(\text{L}) = h(12) = 5$
- $h(\text{U}) = h(21) = 0$
- $h(\text{N}) = h(14) = 0$
- $h(\text{E}) = h(5) = 5$
- $h(\text{S}) = h(19) = 5$.

Tabelas Hash Usando Endereçamento Aberto

Exemplo

- $h(\mathsf{L}) = h(12) = 5$,
- $h(\mathsf{U}) = h(21) = 0$,
- $h(\mathsf{N}) = h(14) = 0$,
- $h(\mathsf{E}) = h(5) = 5$,
- $h(\mathsf{S}) = h(19) = 5$.

	T
0	*U*
1	*N*
2	*S*
3	
4	
5	*L*
6	*E*

Análise

Conforme demonstrado por Knuth (1973), o custo de uma pesquisa com sucesso é:

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

Em que $\alpha = \frac{n}{m}$ representa o fator de carga da tabela.

Análise

Apesar da sondagem linear ser um método relativamente pobre para resolver colisões, os resultados apresentados são bons.

O melhor caso para as operações de pesquisa, inserção e remoção, assim como o caso médio, possui complexidade $O(1)$.

O pior caso ocorre quando é necessário percorrer toda a tabela para encontrar uma chave, pelo excesso de colisões possui complexidade $O(n)$.

Sondagem Quadrática

A sondagem quadrática usa a função hash

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Em que h' é uma função hash auxiliar comum, c_1 e c_2 são constantes e $i = 0, 1 \ldots m-1$.

A primeira tentativa de inserção é realizada na posição $h'(k)$, e as demais possuem um deslocamento que dependem de modo quadrático do valor de i.

Este método sofre do problema de agrupamento secundário, uma forma mais branda do agrupamento primário que também aumenta o tempo de pesquisa.

Sondagem por hashing duplo

Um dos melhores métodos para sondagem endereçamento aberto é o hashing duplo, que possui várias das características de permutações geradas aleatoriamente.

É utilizada uma função do tipo

$$h(k,i) = (h_1(k) + ih_2(k)) \text{ mod } m$$

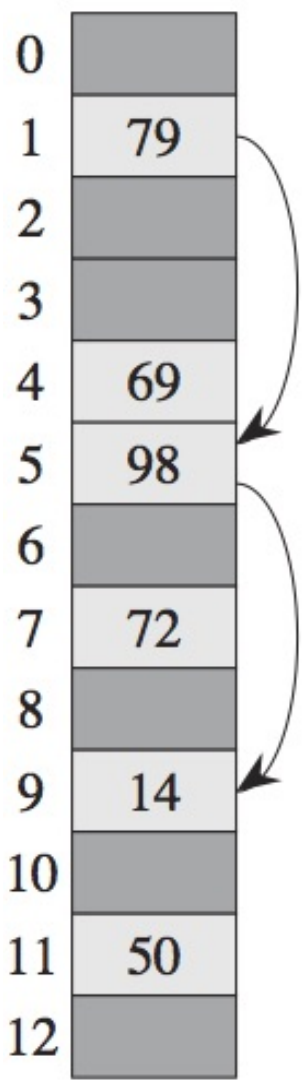
em que ambas h_1 e h_2 são funções hash auxiliares.

A primeira tentativa de inserção é realizada na posição $h_1(k)$, e as demais possuem um deslocamento dado por $h_2(k)$ mod m.

Sondagem por Hashing Duplo

Temos uma tabela hash de tamanho 13 com $h_1(k) = k$ mod 13 e $h_2(k)$= 1 + (k mod 11).

Uma vez que $h_1(14) \equiv 1$ e $h_2(14)$ mod $m \equiv 4$ inserimos a chave 14 na posição 9, depois de examinar as posições 1 e 5, ocupadas.



Exemplo - Sondagem Linear

```
#define VAZIO "!!!!!!!!!\0"
#define n 10 // tamanho da chave (string)

typedef char TChave[n];

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

typedef TItem TMapa[m];
```

Exemplo - Sondagem Linear

```
void TMapa_Inicia(TMapa mp){
    int i;
    for (i = 0; i < m; i++)
        memcpy(mp[i].chave, VAZIO, n);
}
```

Exemplo - Sondagem Linear

```
int TMapa_Pesquisa(TMapa mp, TChave chave, int p){
    int i = 0;//varia de acordo com o tipo de sondagem
    int ini = h(chave, p);

    while (strcmp(mp[(ini + i) % m].chave, VAZIO) != 0 &&
           strcmp(mp[(ini + i) % m].chave, chave) != 0 &&
           i < m)
        i++;//varia de acordo com o tipo de sondagem

    if (strcmp(mp[(ini + i) % m].chave, chave) == 0)
        return (ini + i) % m;
    return -1; // pesquisa sem sucesso
}
```

Exemplo - Sondagem Linear

```
int TMapa_Insere(TMapa mp, TItem x, int p){
    if (TMapa_Pesquisa(mp, x.chave, p) >= 0)
        return 0; // chave ja existe no mapa

    int i = 0;//varia de acordo com o tipo de sondagem
    int ini = h(x.chave, p);
    while (strcmp(mp[(ini + i) % m].chave, VAZIO) != 0 &&
           i < m)
        i++;//varia de acordo com o tipo de sondagem

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

Exemplo - Sondagem Linear

```
int TMapa_Remove(TMapa mp, TItem *x, int p){
    int i = TMapa_Pesquisa(mp, x->chave, p);
    if (i == -1)
        return 0; // chave nao encontrada

    memcpy(mp[i].chave, VAZIO, n);
    return 1;
}
```

Tabelas Hash: Vantagens

- Alta eficiência no custo de pesquisa, que é $O(1)$ no caso médio;
- Simplicidade de implementação.

Tabelas Hash: Desvantagens

- O custo para recuperar os registros ordenados pela chave é alto, sendo necessário ordenar toda a tabela;
- Pior caso é O(n).

Mapas

Listas Encadeadas

- Melhor caso: $O(1)$, caso médio $\Theta(n/m)$, pior caso $\Theta(n)$.

Endereçamento Aberto

- Melhor caso: $O(1)$, caso médio $O(1)$, pior caso $O(n)$;
- Sondagem Linear: agrupamento primário;
- Sondagem Quadrática: agrupamento secundário;
- Sondagem por Hashing Duplo: espalha melhor os elementos, mas possui o mesmo pior caso dos outros métodos.

Exercício

Considere uma tabela com m=11, $h_1(k) = k\%m$ e $h_2(k) = 3$.

- Insira os elementos 25, 37, 48, 59, 32, 44, 70 e 18, nesta ordem;
- A função h_2 é adequada para uso no hashing duplo? Justifique.

Dúvidas?