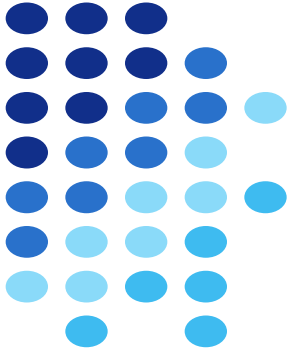
Universidade Federal de Sergipe
Departamento de Sistemas de Informação

SINF0007 – Estrutura de Dados II

Tabelas Hash (hash tables, hashing ou tabela de espalhamento)



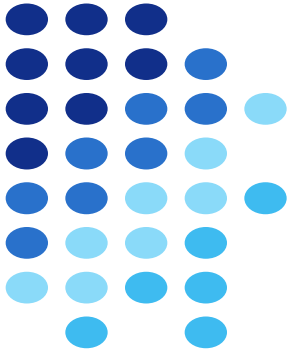
5.1

Prof. Dr. Raphael Pereira de Oliveira
raphael.oliveira@academico.ufs.br

Tratamento de Colisões

- **Por Encadeamento (exterior e interior)**
- Por Endereçamento Aberto

Tratamento de Colisões por Encadeamento Interior



Encadeamento Interior

- Em algumas aplicações não é desejável manter uma estrutura externa à tabela **hash**, ou seja, não se pode permitir que o espaço de registros cresça indefinidamente
- Nesse caso, ainda assim pode-se fazer tratamento de colisões

Encadeamento Interior COM Zona de Colisões

- Dividir a tabela em duas zonas
  - Uma de endereços-base, de tamanho **`p`**
  - Uma de colisão, de tamanho **`s`**
  - **`p + s = m`**
  - Função de **hash** deve gerar endereços no intervalo **`[0, p-1]`**
  - Cada nó tem a mesma estrutura utilizada no Encadeamento Exterior (tabela de dados)

Exemplo: Encadeamento Interior COM Zona de Colisões

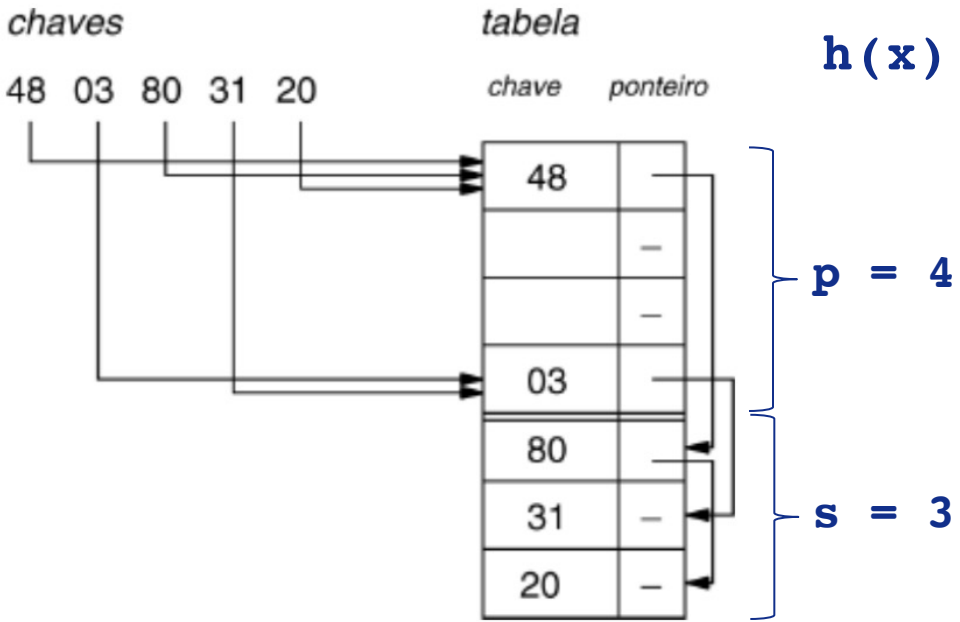


FIGURA 10.6 Tratamento de colisões por encadeamento interior.

*Livro: Jayme Luiz Szwarcfiter e Lilian Markenzon - **Estruturas de Dados e seus Algoritmos** (2010)*

Overflow

- Em um dado momento, pode acontecer de não haver mais espaço para inserir um novo registro

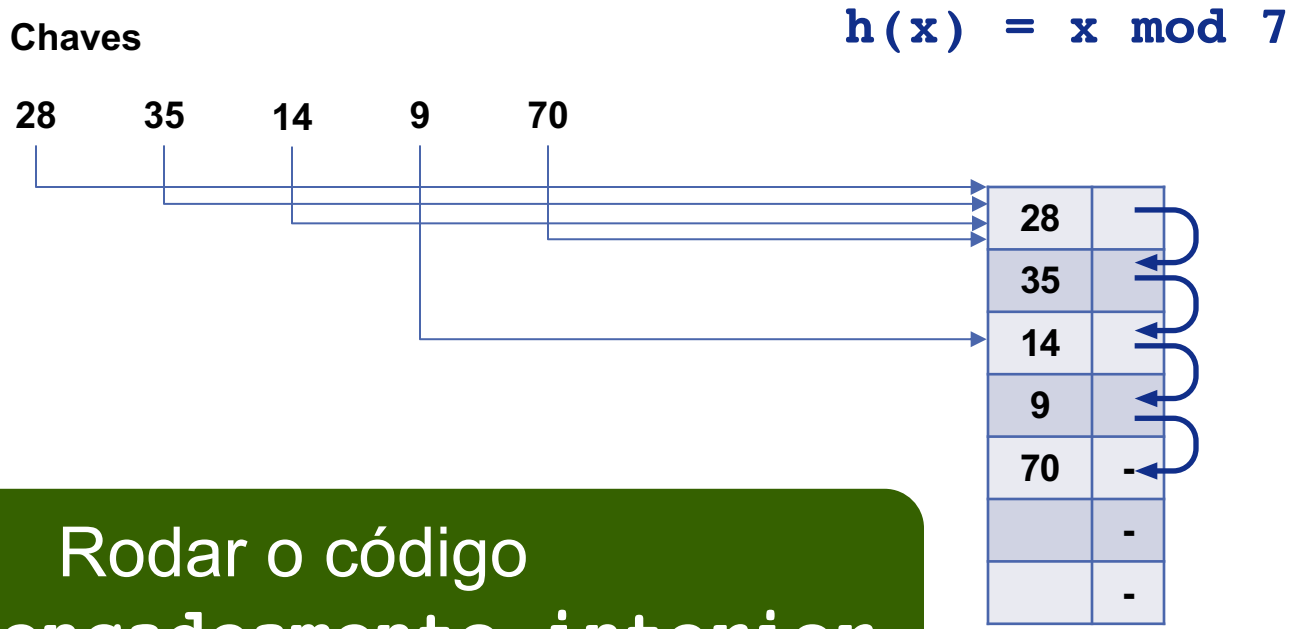
Reflexões

- Qual deve ser a relação entre o tamanho de **p** e **s**?
  - O que acontece quando **p** é muito grande, e **s** muito pequeno?
  - O que acontece quando **p** é muito pequeno, e **s** muito grande?
  - Pensem nos casos extremos:
    - `p = 1; s = m - 1`
    - `p = m - 1; s = 1`

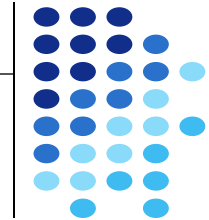
Encadeamento Interior SEM Zona de Colisões

- Outra opção de solução é não separar uma zona específica para colisões
  - Qualquer endereço da tabela pode ser de base ou de colisão
  - Quando ocorre colisão a chave é inserida no **primeiro compartimento vazio** a partir do compartimento em que ocorreu a colisão
  - Efeito indesejado: **colisões secundárias**
    - Colisões secundárias são provenientes da coincidência de endereços para chaves que não são sinônimas

Exemplo: Encadeamento Interior SEM Zona de Colisões



Rodar o código
3.**hash-encadeamento-interior**



Implementação em Memória Principal

```
#define LIBERADO 0
#define OCUPADO 1

typedef struct aluno {
    int matricula;
    float cr;
    int prox;
    int ocupado;
} TAluno;

//Hash é um vetor que será alocado dinamicamente
typedef TAluno *Hash;
```

Inicialização

```
TAluno *aloca(int mat, float cr, int status, int prox) {
    TAluno *novo = (TAluno *) malloc(sizeof(TAluno));
    novo->matricula = mat;
    novo->cr = cr;
    novo->ocupado = status;
    novo->prox = prox;
    return novo;
}
void inicializa(Hash *tab, int m) {
    int i;
    for (i = 0; i < m; i++) {
        tab[i] = aloca(-1, -1, LIBERADO, -1);
    }
}
```

Busca em Encadeamento Interior

```
/*
    Função busca assume que a tabela tenha sido inicializada
    da seguinte maneira:
        T[i].ocupado = LIBERADO, e
        T[i].prox = -1, para 0 < i < m-1

    RETORNO:
    Se chave x for encontrada, achou = 1,
    função retorna endereço onde x foi encontrada

    Se chave x não for encontrada, achou = 0, e há duas
    possibilidades para valor retornado pela função:
        endereço de algum compartimento livre, encontrado
    na lista encadeada associada a h(mat)
        -1 se não for encontrado endereço livre
*/
```

```c
int busca(Hash *tab, int m, int mat, int *achou) {
     *achou = -1;
     int temp = -1;
     int end = hash(mat, m);
     while (*achou == -1) {
          TAluno *aluno = tab[end];
          if (!aluno->ocupado) {//achou compartimento livre -- guarda para retorná-lo caso chave não seja
                                //encontrada
               temp = end;
          }
          if (aluno->matricula == mat && aluno->ocupado) {
               //achou chave procurada
               *achou = 1;
          } else {
               if (aluno->prox == -1) {
                    //chegou no final da lista encadeada
                    *achou = 0;
                    end = temp;
               } else {
                    //avança para o próximo
                    end = aluno->prox;
               }
          }
     }
     return end;
}
```

Inserção em Encadeamento Interior

```
/*
   Função assume que pos é o endereço onde
   será efetuada a inserção. Para efeitos de
   escolha de pos, a tabela foi considerada
   como circular, isto é, o compartimento 0 é
   o seguinte ao m-1
*/
```

Exclusão em Encadeamento Interior

```
void exclui(Hash *tab, int m, int mat) {
    int achou;
    int end = busca(tab, m, mat, &achou);
    if (achou) {
        //remove marcando flag para liberado
        tab[end]->ocupado = LIBERADO;
    } else {
        printf("Matrícula não encontrada. Remoção não realizada!");
    }
}
```

Exercícios

1. Desenhe a tabela **hash** (em disco) resultante das seguintes operações (cumulativas) usando o algoritmo de **inserção em Tabela Hash com Encadeamento Interior SEM zona de colisão**. Considere que a tabela tem tamanho 7 e a função de **hash** usa o método da divisão.
   - (a) Inserir as chaves 10, 3, 5, 7, 12, 6, 14
   - (b) Inserir as chaves 4, 8 2.

2. Repita o exercício anterior usando **Tabela Hash com Encadeamento Interior COM zona de colisão**. Considere que a zona de colisão tem tamanho 3.

3. Repita o exercício 1 usando **Tabela Hash com Encadeamento Exterior**.

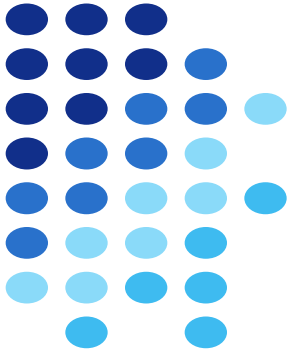
Tratamento de Colisões

- Por Encadeamento (exterior e interior)
- Por Endereçamento Aberto

Tratamento de Colisões

- Por Encadeamento (exterior e interior)
- **Por Endereçamento Aberto**

Tratamento de Colisões por Endereçamento Aberto

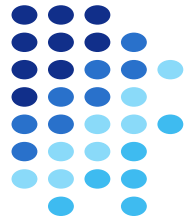


Tratamento de Colisões por Endereçamento Aberto

- **Motivação**: as abordagens anteriores utilizam ponteiros nas listas encadeadas
  - Aumento no consumo de espaço
- **Alternativa**: armazenar apenas os registros, sem os ponteiros
- Quando houver colisão, determina-se, por cálculo de novo endereço, o próximo compartimento a ser examinado

Funcionamento

- Para cada chave **x**, é necessário que todos os compartimentos possam ser examinados
- A função **h(x)** deve fornecer, ao invés de um único endereço, um conjunto de **m** endereços base
- Nova forma da função: **h(x,k)**, onde **k = 0, …, m-1**
- Para encontrar a chave **x** deve-se tentar o endereço base **h(x,0)**
- Se estiver ocupado com outra chave, tentar **h(x,1)**, e assim sucessivamente

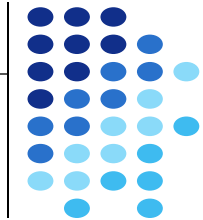


Sequência de Tentativas

- A sequência **`h(x,0), h(x,1), …, h(x, m-1)`** é denominada sequencia de tentativas
- A sequencia de tentativas é uma **permutação** do conjunto **`{0, m-1}`**
- **Portanto**: para cada chave **`x`** a função **`h`** deve ser capaz de fornecer uma permutação de endereços base

Função Hash

- Exemplos de funções **hash** para gerar sequência de tentativas
  - `Tentativa Linear`
  - `Tentativa Quadrática`
  - `Dispersão Dupla`



Função Hash

- Exemplos de funções **hash** para gerar sequência de tentativas
  - **Tentativa Linear**
  - Tentativa Quadrática
  - Dispersão Dupla

Tentativa Linear

- Suponha que o endereço base de uma chave **x** é **h'(x)**
- Suponha que já existe uma chave **y** ocupando o endereço **h'(x)**
- **Ideia**: tentar armazenar **x** no endereço consecutivo a **h'(x)**. Se já estiver ocupado, tenta-se o próximo e assim sucessivamente
- Considera-se uma tabela circular

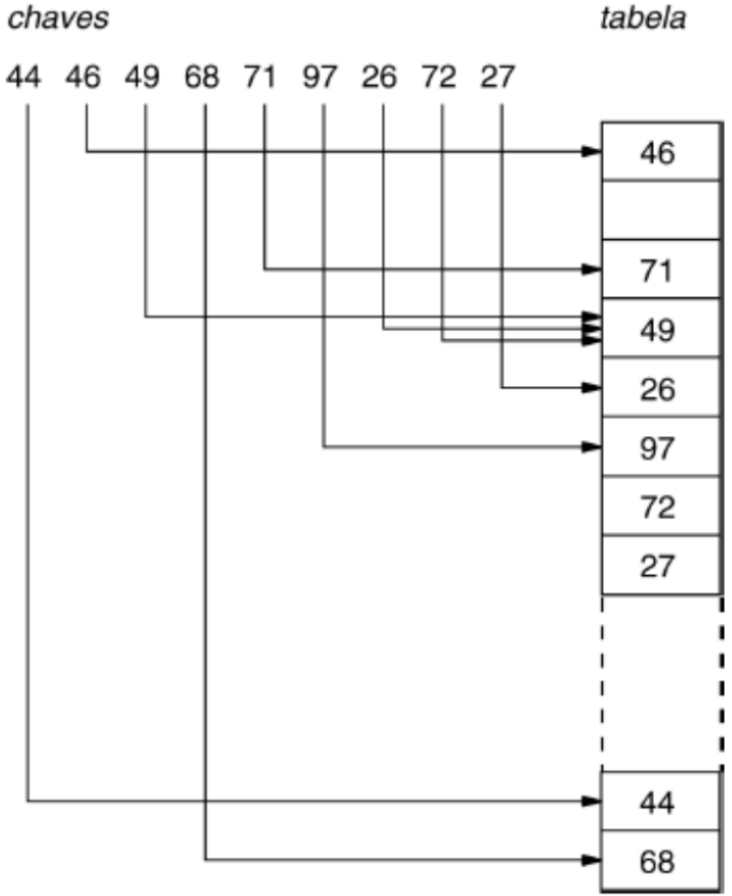
$$h(x, k) = (h'(x) + k) \bmod m, \quad 0 \leq k \leq m-1$$

Exemplo Tentativa Linear

- Observe a tentativa de inserir chave **26**
- Endereço já está ocupado: inserir no próximo endereço livre

$$h(x, k) = (h'(x) + k) \bmod m$$

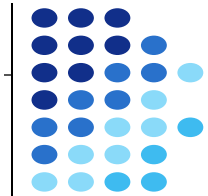
$$h'(x) = x \bmod 23$$



Livro: Jayme Luiz Szwarcfiter e Lilian Markenzon - ***Estruturas de Dados e seus Algoritmos*** *(2010)*

FIGURA 10.8 Tentativa linear.

Implementação Endereçamento Aberto (em memória principal)



```c
typedef struct aluno {
    int matricula;
    float cr;
} TAluno;

typedef TAluno *Hash; //Hash é um vetor que será alocado dinamicamente

void inicializa(Hash *tab, int m) {
    int i;
    for (i = 0; i < m; i++) {
        tab[i] = NULL;
    }
}
```

Rodar o código
4.hash-encadeamento-aberto

Busca por Endereçamento Aberto

```c
int hash_linha(int mat, int m) {
        return mat % m;
}

int hash(int mat, int m, int k) {
        return (hash_linha(mat, m) + k) % m;
}
/*
    * Função busca
    RETORNO:
    Se chave mat for encontrada, achou = 1,
    função retorna endereço onde mat foi encontrada
    Se chave mat não for encontrada, achou = 0, e há duas
    possibilidades para valor retornado pela função:
    endereço de algum compartimento livre encontrado durante a busca
    -1 se não for encontrado endereço livre (tabela foi percorrida até o final)
*/
```

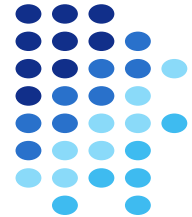
```c
int busca(Hash *tab, int m, int mat, int *achou) {
     *achou = 0;
     int end = -1;
     int pos_livre = -1;
     int k = 0;
     while (k < m) {
          end = hash(mat, m, k);
          if (tab[end] != NULL && tab[end]->matricula == mat) {//encontrou chave
               *achou = 1;
               k = m; //força saída do loop
          } else {
               if (tab[end] == NULL) {//encontrou endereço livre
                    //se for o primeiro, registra isso
                    if (pos_livre == -1)
                        pos_livre = end;
               }
               k = k + 1; //continua procurando
          }
     }
     if (*achou)
          return end;
     else
          return pos_livre;
}
```

Inserção em Endereçamento Aberto

```
// Função insere assume que end é o endereço onde será efetuada a inserção
void insere(Hash *tab, int m, int mat, float cr) {
    int achou;
    int end = busca(tab, m, mat, &achou);
    if (!achou) {
        if (end != -1) {//Não encontrou a chave, mas encontrou posição livre
                //Inserção será realizada nessa posição
                tab[end] = aloca(mat, cr);
        } else {
            //Não foi encontrada posição livre durante a busca: overflow
            printf("Ocorreu overflow. Inserção não realizada!\n");
        }
    } else {
        printf("Matricula já existe. Inserção inválida! \n");
    }
}
```

Exclusão em Endereçamento Aberto

```
void exclui(Hash *tab, int m, int mat) {
    int achou;
    int end = busca(tab, m, mat, &achou);
    if (achou) {
        //remove
        free(tab[end]);
        tab[end] = NULL;
    } else {
        printf("Matricula não encontrada. Remoção não realizada!");
    }
}
```



Discussão do Algoritmo

- Na presença de remoções, a inserção precisa que a busca percorra toda a tabela até ter certeza de que o registro procurado não existe

- Em situações onde não há remoção, a busca pode parar assim que encontrar um compartimento livre (se a chave existisse, ela estaria ali)

Quais são as desvantagens da Tentativa Linear?

- Suponha um trecho de `j` compartimentos consecutivos ocupados (chama-se **agrupamento primário**) e um compartimento `L` vazio imediatamente seguinte a esses

- Suponha que uma chave `x` precisa ser inserida em um dos `j` compartimentos

  - `x` será armazenada em `L`

  - isso aumenta o tamanho do **agrupamento primário** para `j + 1`

  - Quanto maior for o tamanho de um agrupamento primário, maior a probabilidade de aumentá-lo ainda mais mediante a inserção de uma nova chave

Função Hash

- Exemplos de funções **hash** para gerar sequência de tentativas
  - Tentativa Linear
  - **Tentativa Quadrática**
  - Dispersão Dupla

Tentativa Quadrática

- Para mitigar a formação de agrupamentos primários, que aumentam muito o tempo de busca:
  - Obter sequências de endereços para endereços-base próximos, porém diferentes
  - Utilizar como incremento uma **função quadrática de k**
  - $h(x,k) = (h'(x) + c1\ k + c2\ k^2) \bmod m$,

onde $c1$ e $c2$ são constantes, $c2 \neq 0$ e $k = 0, \ldots, m-1$

Tentativa Quadrática

- Método evita agrupamentos primários
- Mas… se duas chaves tiverem a mesma tentativa inicial, vão produzir sequências de tentativas idênticas: **agrupamento secundário**

Tentativa Quadrática

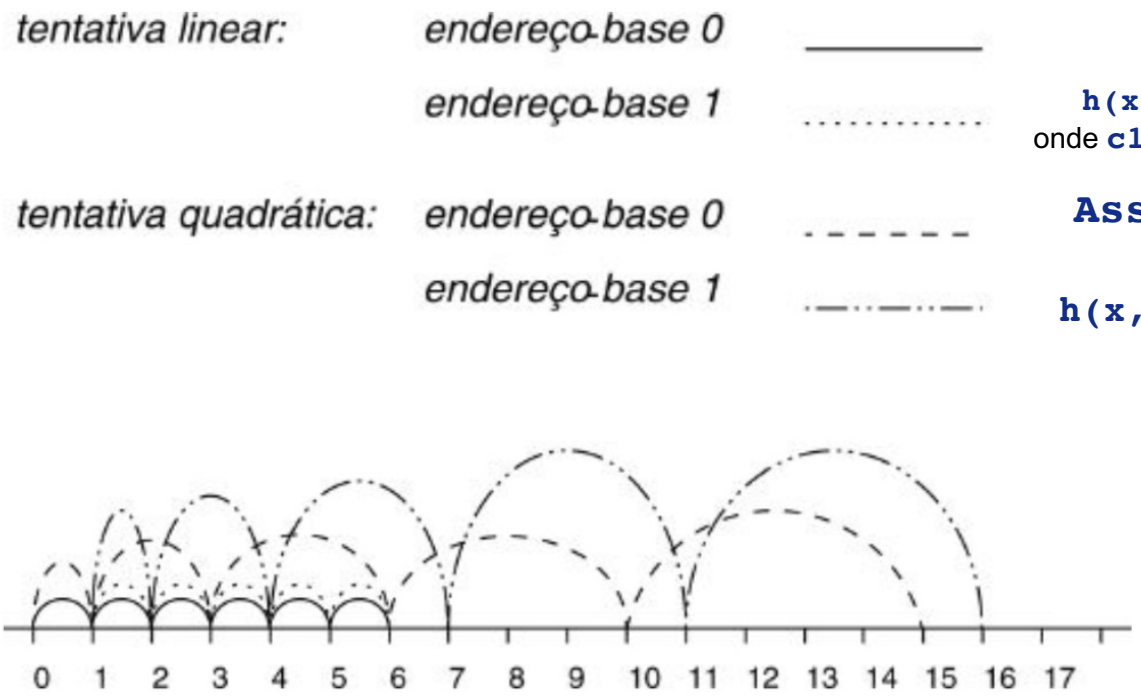


FIGURA 10.9 Comparação entre tentativas linear e quadrática.

$h(x,k) = (h'(x) + c1\ k + c2\ k^2) \bmod m$,
onde $c1$ e $c2$ são constantes, $c2 \neq 0$ e $k = 0, \ldots, m-1$

Assumindo c1 = 0,5 e c2 = 0,5

Endereço base 0

$h(x,k) = (0 + 0{,}5\ k + 0{,}5\ k^2) \bmod m$

$k = 0$	$h(x,k) = 0$
$k = 1$	$h(x,k) = 1$
$k = 2$	$h(x,k) = 3$
$k = 3$	$h(x,k) = 6$
$k = 4$	$h(x,k) = 10$
$k = 5$	$h(x,k) = 15$

*Livro: Jayme Luiz Szwarcfiter e Lilian Markenzon - **Estruturas de Dados e seus Algoritmos** (2010)*

Função Hash

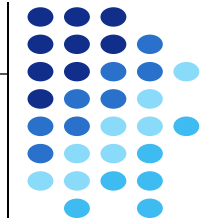
- Exemplos de funções **hash** para gerar sequência de tentativas
  - `Tentativa Linear`
  - `Tentativa Quadrática`
  - **`Dispersão Dupla`**

Dispersão Dupla

- Utiliza duas funções de hash, **h'(x)** e **h''(x)**

$$h(x,k) = (h'(x) + k.h''(x)) \bmod m, \text{ para } 0 \leq k < m$$

- Método distribui melhor as chaves do que os dois métodos anteriores
  - Se duas chaves distintas **x** e **y** são sinônimas **(h'(x) = h'(y))**, os métodos anteriores produzem exatamente a mesma sequência de tentativas para **x** e **y**, ocasionando concentração de chaves em algumas áreas da tabela
  - No método da dispersão dupla, isso só acontece se **h'(x) = h'(y)** e **h''(x) = h''(y)**



Discussão

- A técnica de hashing é mais utilizada nos casos em que existem muito mais buscas do que inserções de registros

Exercício

1. Desenhe a tabela **hash** (em disco) resultante das seguintes operações (cumulativas) usando o algoritmo de **inserção Tabela Hash por Endereçamento Aberto**. A tabela tem tamanho 7.

   a) Inserir as chaves 10, 3, 5, 7, 12, 6, 14, 4, 8. Usar a função de tentativa linear $h(x, k) = (h'(x) + k) \bmod 7$, $0 \leq k \leq m-1$, e $h'(x) = x \bmod 7$

   b) Repita o exercício anterior, mas agora usando dispersão dupla $h(x,k) = (h'(x) + k.h''(x)) \bmod 7$, sendo $h'(x) = x \bmod 7$ e $h''(x) = x + 1$

Referências

- Material baseado nos slides de **Vanessa Braganholo**, Disciplina de Estruturas de Dados e Seus Algoritmos. Instituto de Computação. Universidade Federal Fluminense (UFF), Niterói, Brasil.
- Szwarcfiter, J.; Markezon, L. Estruturas de Dados e seus Algoritmos, 3a. ed. LTC. Cap. 10
- Inhaúma Neves Ferraz. Programação Com Arquivos. 2003. Editora: manole.
- Schildt, H. C Completo e Total. Ed. McGraw-Hill.