

Arquivos de Acesso Direto

Vanessa Braganholo

Arquivo de Acesso Direto

- ▶ Definição: arquivo em que o acesso a um registro pode ser feito diretamente, sem ter que ler todos os registros que vêm antes dele
- ▶ Implicação prática: não é mais necessário que o arquivo esteja ordenado
- ▶ Basta que saibamos o endereço do registro que queremos acessar

Endereço?

- ▶ Como saber o endereço de um determinado registro?
- ▶ Definido sempre como um deslocamento em relação ao primeiro registro
- ▶ $\text{EndereçoReg}(i) = (i - 1) * \text{tamanhoReg}$

onde tamanhoReg é o tamanho dos registros do arquivo, em bytes

Exemplo

CodCli	Nome	DataNascimento
10	Joao	02/12/1990
02	Maria	04/10/1976
15	Carlos	30/06/1979
04	Carolina	14/05/2000
01	Murilo	23/10/1988

▶ Tamanho Registro:

- ▶ CodCli = 4 bytes
- ▶ Nome = 10 bytes
- ▶ DataNascimento = 12 bytes

- ▶ **Total:** 26 bytes por registro

Exemplo

	CodCli	Nome	DataNascimento
0	10	Joao	02/12/1990
26	02	Maria	04/10/1976
52	15	Carlos	30/06/1979
78	04	Carolina	14/05/2000
104	01	Murilo	23/10/1988

- ▶ Tamanho Registro:
 - ▶ CodCli = 4 bytes
 - ▶ Nome = 10 bytes
 - ▶ DataNascimento = 12 bytes
- ▶ **Total:** 26 bytes por registro

- ▶ Endereço do Registro 3 = $(3-1) * 26 = 52$

Para ler o registro 3

1. Abrir o arquivo
 2. Calcular o endereço do registro 3
 3. Avançar o cursor (*seek*) para o endereço calculado
 4. Ler o registro
- ▶ Ao terminar de ler o registro, o cursor estará posicionado no registro 4

Mas...

- ▶ A operação de leitura normalmente é guiada pelo valor da chave do registro que estamos procurando, e não pela posição do registro
- ▶ Ao invés de ler registro 3, ler o registro de chave 55
- ▶ Como fazer para saber em que endereço está um registro que possui uma determinada chave?

Técnicas para localizar registros

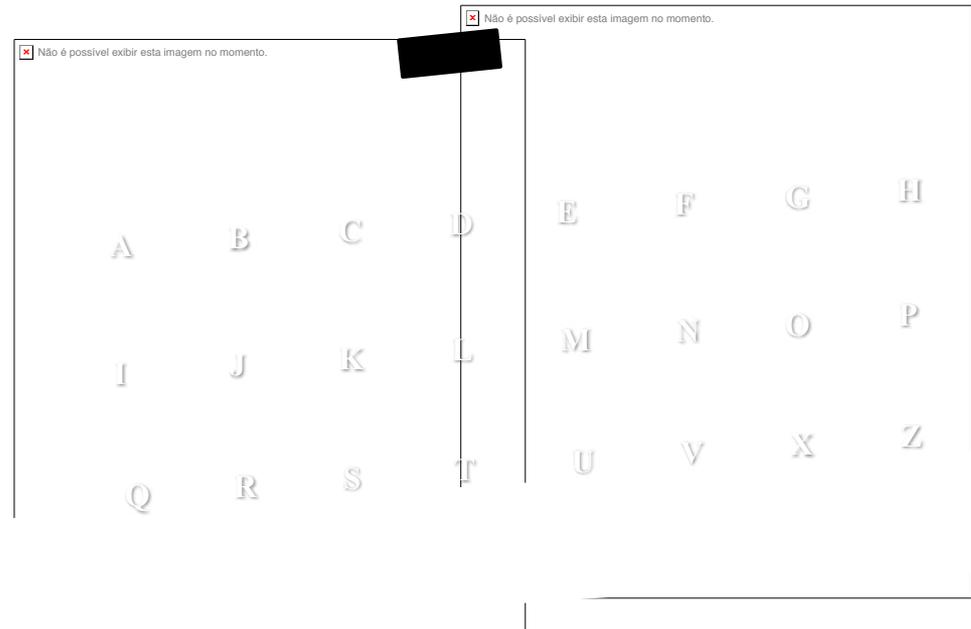
- ▶ Como fazer para saber em que endereço está um registro que possui uma determinada chave?
 - ▶ Cálculos computacionais: uso de funções de cálculo de endereço a partir do valor da chave (hashing)
 - ▶ Indexação: uso de uma estrutura de dados auxiliar (ex. Árvore B, ...)

Hashing (Tabelas de Dispersão)

Fonte de consulta: Szwarcfiter, J.; Markezon, L. Estruturas de Dados e seus Algoritmos, 3a. ed. LTC. Cap. 10

Exemplo Motivador

- ▶ Distribuição de correspondências de funcionários numa empresa
 - ▶ Um escaninho para cada inicial de sobrenome
 - ▶ Todos os funcionários com a mesma inicial de sobrenome procuram sua correspondência dentro do mesmo escaninho
 - ▶ Pode haver mais de uma correspondência dentro do mesmo escaninho



Hashing: Princípio de Funcionamento

- ▶ Suponha que existem **n** chaves a serem armazenadas numa tabela de comprimento **m**
 - ▶ Em outras palavras, a tabela tem **m** compartimentos
 - ▶ Endereços possíveis: [**0, m-1**]
- ▶ Situações possíveis: cada compartimento da tabela pode armazenar **x** registros
- ▶ Para simplificar, assumimos que **x = 1** (cada compartimento armazena apenas **1** registro)

Como determinar m ?

- ▶ Uma opção é determinar **m** em função do número de valores possíveis das chaves a serem armazenadas

Hashing: Princípio de Funcionamento

- ▶ Se os valores das chaves variam de $[0, m-1]$, então podemos usar o valor da chave para definir o endereço do compartimento onde o registro será armazenado

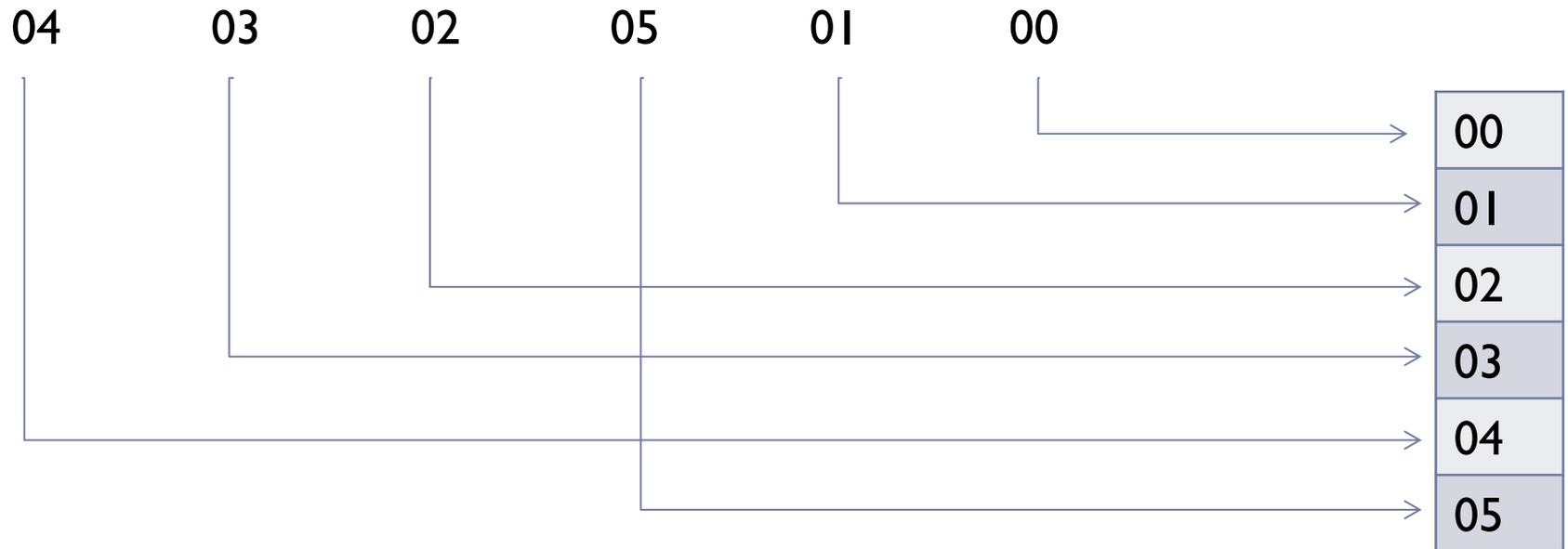
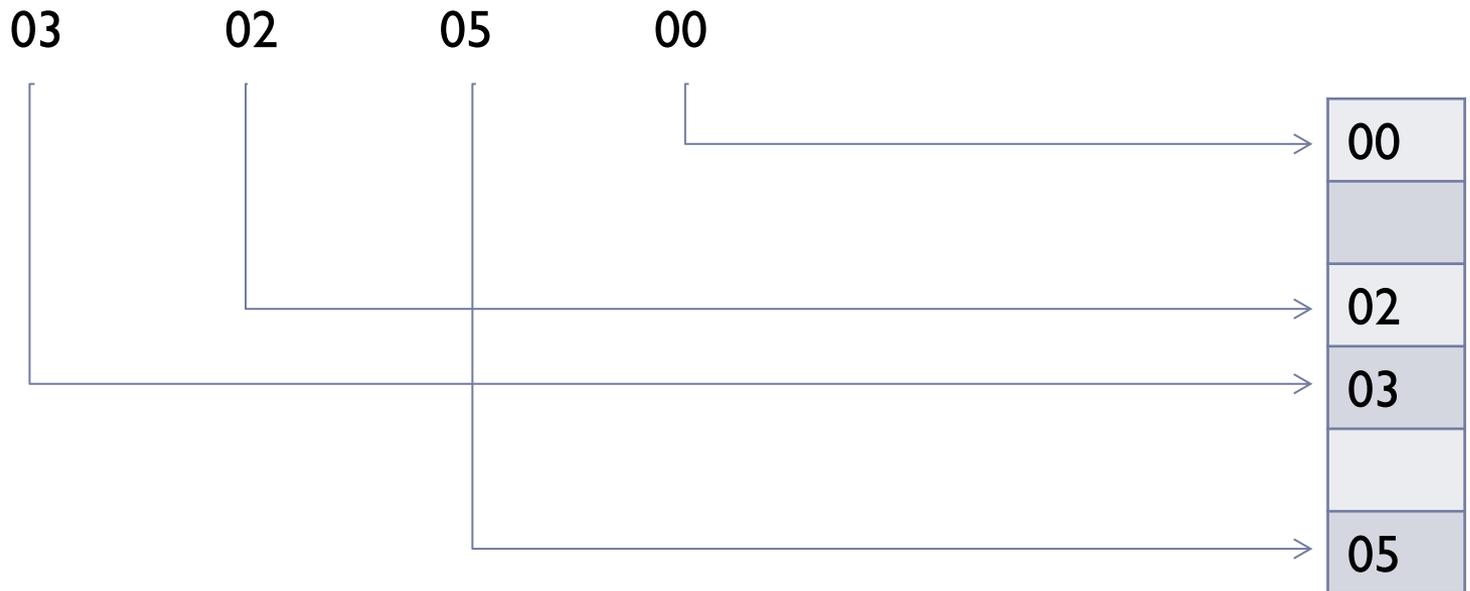


Tabela pode ter espaços vazios

- ▶ Se o número **n** de chaves a armazenar é menor que o número de compartimentos **m** da tabela



Mas...

- ▶ Se o intervalo de valores de chave é muito grande, **m** é muito grande
- ▶ Pode haver um número proibitivo de espaços vazios na tabela se houver poucos registros

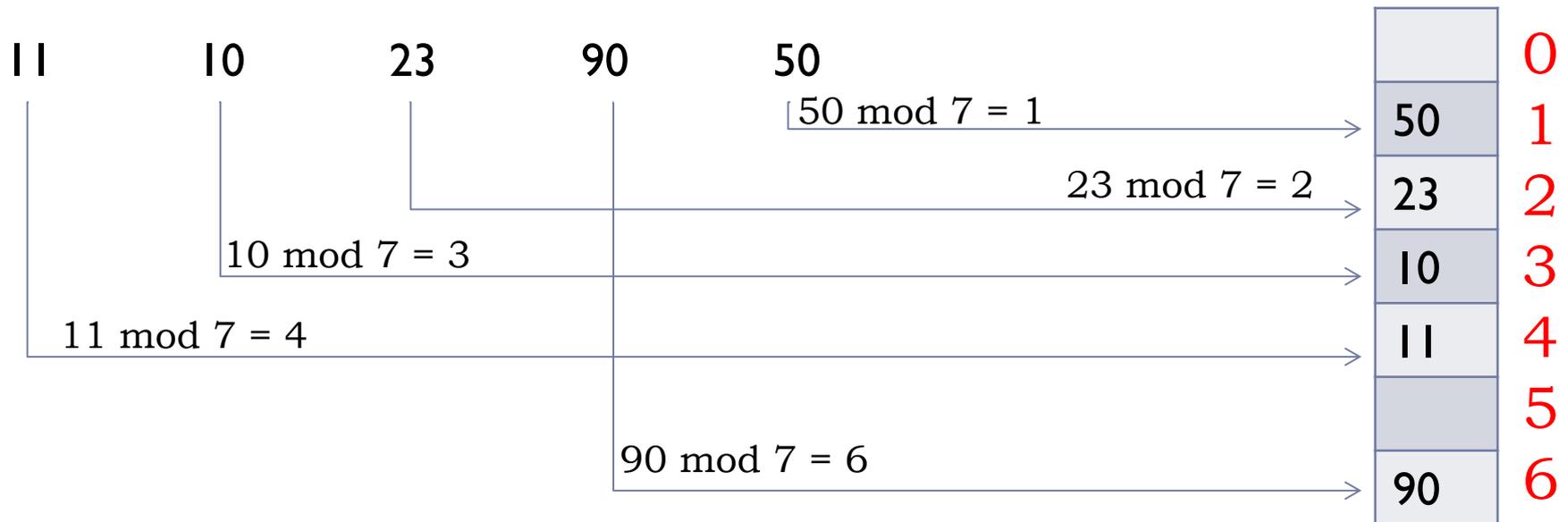
- ▶ Exemplo: armazenar 2 registros com chaves 0 e 999.999 respectivamente
 - ▶ **m** = 1.000.000
 - ▶ tabela teria 999.998 compartimentos vazios

Solução

- ▶ Definir um valor de **m** menor que os valores de chaves possíveis
- ▶ Usar uma função hash **h** que mapeia um valor de chave **x** para um endereço da tabela
- ▶ Se o endereço **$h(x)$** estiver livre, o registro é armazenado no compartimento apontado por **$h(x)$**
- ▶ Diz-se que **$h(x)$** produz um **endereço-base** para **x**

Exemplo

► $h(x) = x \bmod 7$

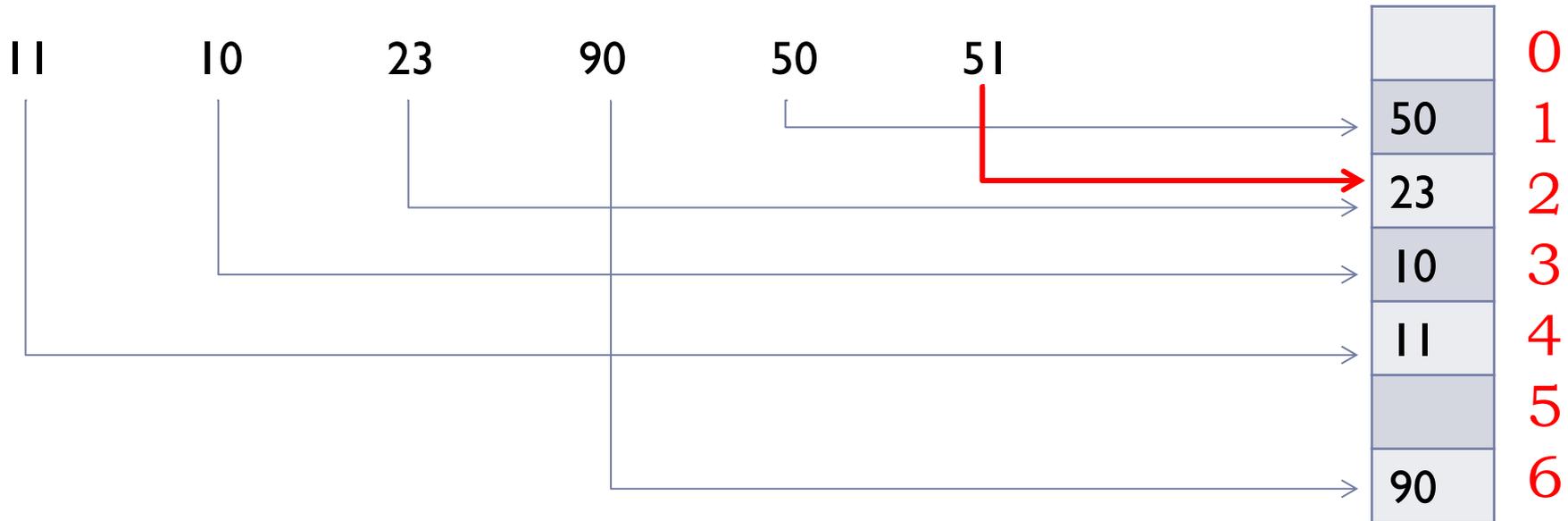


Função hash h

- ▶ Infelizmente, a função pode não garantir injetividade, ou seja, é possível que $x \neq y$ e $h(x) = h(y)$
- ▶ Se ao tentar inserir o registro de chave x o compartimento de endereço $h(x)$ já estiver ocupado por y , ocorre uma **colisão**
 - ▶ Diz-se que x e y são sinônimos em relação a h

Exemplo: Colisão

► $h(x) = x \bmod 7$



A chave 51 colide com a chave 23 e não pode ser inserida no endereço 2!

Solução: uso de um procedimento especial para armazenar a chave 51 (tratamento de colisões)

Características desejáveis das funções de hash

- ▶ Produzir um número baixo de colisões
- ▶ Ser facilmente computável
- ▶ Ser uniforme

Características desejáveis das funções de hash

- ▶ **Produzir um número baixo de colisões**
 - ▶ Difícil, pois depende da distribuição dos valores de chave
 - ▶ Exemplo: Pedidos que usam o ano e mês do pedido como parte da chave
 - ▶ Se a função **h** realçar estes dados, haverá muita concentração de valores nas mesmas faixas

Características desejáveis das funções de hash

- ▶ **Ser facilmente computável**

- ▶ Se a tabela estiver armazenada em disco (nosso caso), isso não é tão crítico, pois a operação de I/O é muito custosa, e dilui este tempo
- ▶ Das 3 condições, é a mais fácil de ser garantida

- ▶ **Ser uniforme**

- ▶ Idealmente, a função **h** deve ser tal que todos os compartimentos possuam a mesma probabilidade de serem escolhidos
- ▶ Difícil de testar na prática

Exemplos de Funções de Hash

- ▶ Algumas funções de hash são bastante empregadas na prática por possuírem algumas das características anteriores:
 - ▶ Método da Divisão
 - ▶ Método da Dobra
 - ▶ Método da Multiplicação

Exemplos de Funções de Hash

- ▶ **Método da Divisão**
- ▶ Método da Dobra
- ▶ Método da Multiplicação

Método da Divisão

- ▶ Uso da função mod:

$$h(x) = x \bmod m$$

onde m é a dimensão da tabela

- ▶ Alguns valores de m são melhores do que outros
 - ▶ Exemplo: se m for par, então $h(x)$ será par quando x for par, e ímpar quando x for ímpar → indesejável

Método da divisão

- ▶ Estudos apontam bons valores de **m**:
 - ▶ Escolher **m** de modo que seja um número primo não próximo a uma potência de 2; ou
 - ▶ Escolher **m** tal que não possua divisores primos menores do que 20

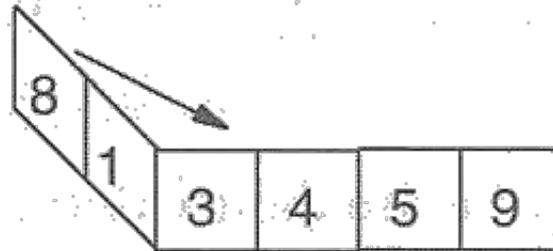
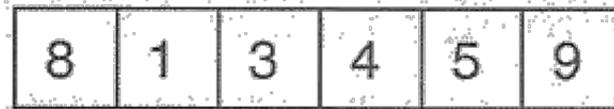
Exemplos de Funções de Hash

- ▶ Método da Divisão
- ▶ **Método da Dobra**
- ▶ Método da Multiplicação

Método da Dobra

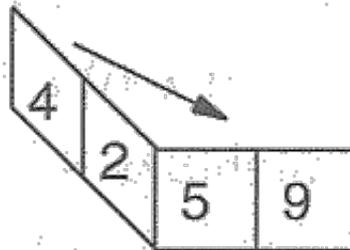
- ▶ Suponha a chave como uma sequência de dígitos escritos em um pedaço de papel
- ▶ O método da dobra consiste em “dobrar” este papel, de maneira que os dígitos se superponham
- ▶ Os dígitos então devem ser somados, sem levar em consideração o “vai-um”

Exemplo: Método da Dobra



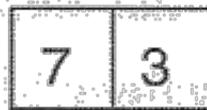
$$8+4=12$$

$$1+3=4$$



$$4+9=13$$

$$2+5=7$$



Método da Dobra

- ▶ A posição onde a dobra será realizada, e quantas dobras serão realizadas, depende de quantos dígitos são necessários para formar o endereço base
- ▶ O tamanho da dobra normalmente é do tamanho do endereço que se deseja obter

Exercício

- ▶ Escreva uma função em C que implementa o método da dobra, de forma a obter endereços de 2 dígitos
- ▶ Assuma que as chaves possuem 6 dígitos

Exemplos de Funções de Hash

- ▶ Método da Divisão
- ▶ Método da Dobra
- ▶ **Método da Multiplicação**

Método da Multiplicação

- ▶ Multiplicar a chave por ela mesma
- ▶ Armazenar o resultado numa palavra de **b** bits
- ▶ Descartar os bits das extremidades direita e esquerda, um a um, até que o resultado tenha o tamanho de endereço desejado

Método da Multiplicação

- ▶ **Exemplo: chave 12**
 - ▶ $12 \times 12 = 144$
 - ▶ 144 representado em binário: 10010000
 - ▶ Armazenar em 10 bits: 0010010000
 - ▶ Obter endereço de 6 bits (endereços entre 0 e 63)

0 0 1 0 0 1 0 0 0 0

Método da Multiplicação

- ▶ **Exemplo: chave 12**
 - ▶ $12 \times 12 = 144$
 - ▶ 144 representado em binário: 10010000
 - ▶ Armazenar em 10 bits: 0010010000
 - ▶ Obter endereço de 6 bits (endereços entre 0 e 63)

0 0 1 0 0 1 0 0 0 0



Método da Multiplicação

- ▶ **Exemplo: chave 12**
 - ▶ $12 \times 12 = 144$
 - ▶ 144 representado em binário: 10010000
 - ▶ Armazenar em 10 bits: 0010010000
 - ▶ Obter endereço de 6 bits (endereços entre 0 e 63)



Método da Multiplicação

- ▶ **Exemplo: chave 12**
 - ▶ $12 \times 12 = 144$
 - ▶ 144 representado em binário: 10010000
 - ▶ Armazenar em 10 bits: 0010010000
 - ▶ Obter endereço de 6 bits (endereços entre 0 e 63)



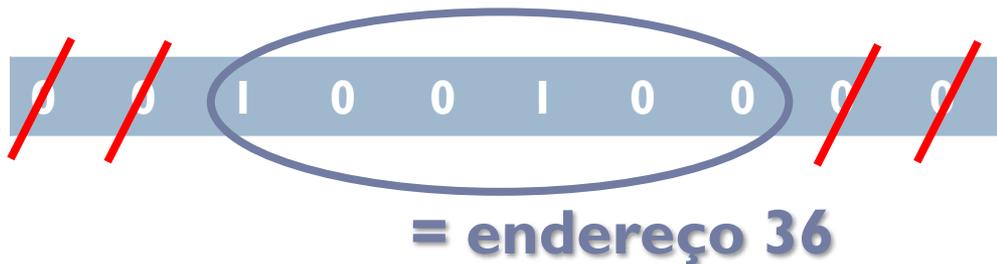
Método da Multiplicação

- ▶ **Exemplo: chave 12**
 - ▶ $12 \times 12 = 144$
 - ▶ 144 representado em binário: 10010000
 - ▶ Armazenar em 10 bits: 0010010000
 - ▶ Obter endereço de 6 bits (endereços entre 0 e 63)



Método da Multiplicação

- ▶ Exemplo: chave 12
 - ▶ $12 \times 12 = 144$
 - ▶ 144 representado em binário: 10010000
 - ▶ Armazenar em 10 bits: 0010010000
 - ▶ Obter endereço de 6 bits (endereços entre 0 e 63)



Uso da função de hash

- ▶ A mesma função de hash usada para inserir os registros é usada para buscar os registros

Exemplo: busca de registro por chave

▶ **$h(x) = x \bmod 7$**

▶ Encontrar o registro de chave 90

▶ $90 \bmod 7 = 6$

▶ Encontrar o registro de chave 7

▶ $7 \bmod 7 = 0$

▶ Compartimento 0 está vazio: registro não está armazenado na tabela

▶ Encontrar o registro de chave 8

▶ $8 \bmod 7 = 1$

▶ Compartimento 1 tem um registro com chave diferente da chave buscada, e não existem registros adicionais: registro não está armazenado na tabela

0	
1	50
2	23
3	10
4	11
5	
6	90

Tratamento de Colisões

Fator de Carga

- ▶ O fator de carga de uma tabela hash é $\alpha = n/m$, onde n é o número de registros armazenados na tabela
 - ▶ O número de colisões cresce rapidamente quando o fator de carga aumenta
 - ▶ Uma forma de diminuir as colisões é diminuir o fator de carga
 - ▶ Mas **isso não resolve o problema**: colisões sempre podem ocorrer

- ▶ Como tratar as colisões?

Tratamento de Colisões

- ▶ Por Encadeamento
- ▶ Por Endereçamento Aberto

Tratamento de Colisões

- ▶ **Por Encadeamento**
- ▶ **Por Endereçamento Aberto**

Tratamento de Colisões por Encadeamento

- ▶ **Encadeamento Exterior**
- ▶ Encadeamento Interior

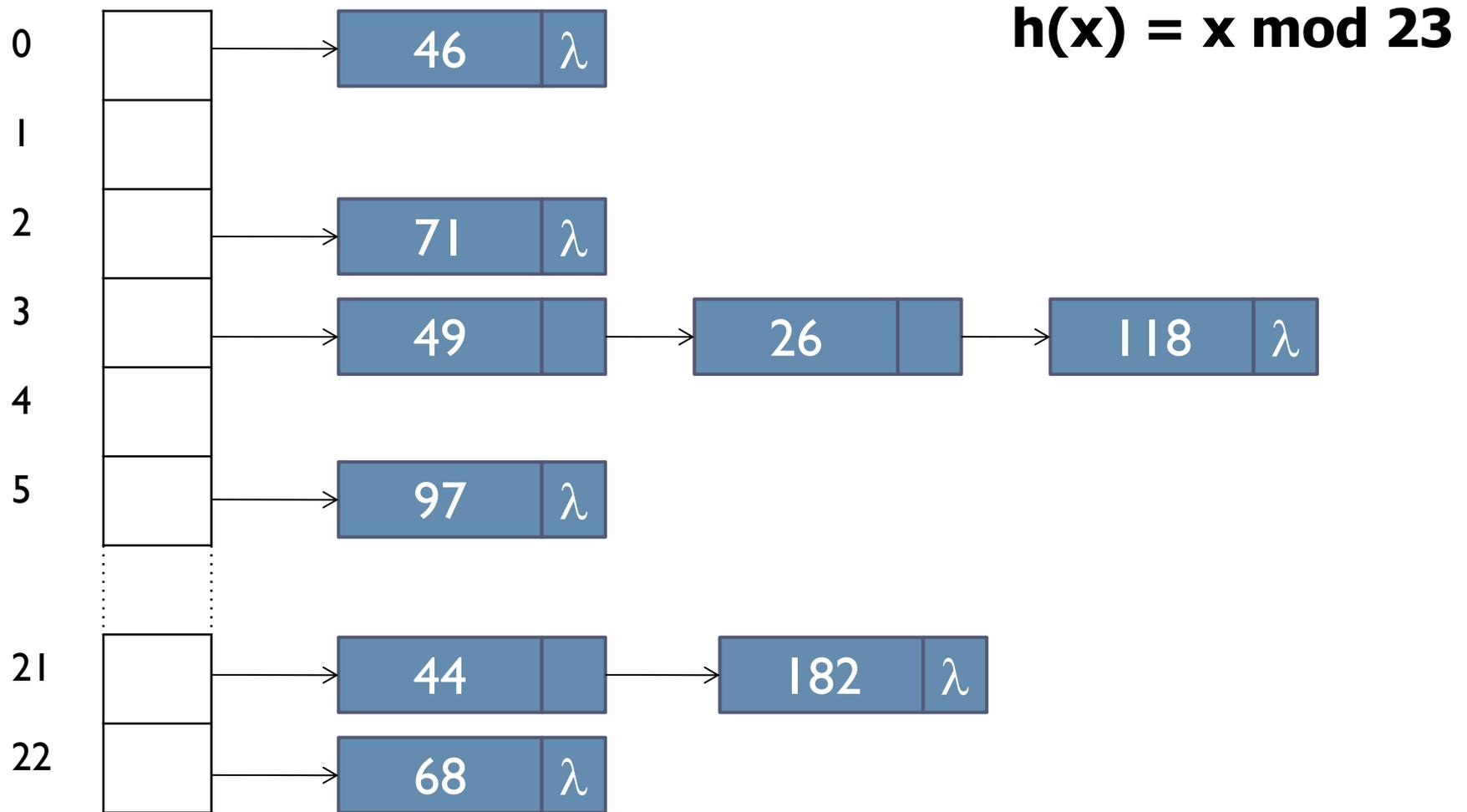
Encadeamento Exterior

- ▶ Manter **m** listas encadeadas, uma para cada possível endereço base
- ▶ A tabela base não possui nenhum registro, apenas os ponteiros para as listas encadeadas
- ▶ Por isso chamamos de encadeamento **exterior**: a tabela base não armazena nenhum registro

Nós da lista Encadeada

- ▶ Cada nó da lista encadeada contém:
 - ▶ um registro
 - ▶ um ponteiro para o próximo nó

Exemplo: Encadeamento Exterior



Encadeamento Exterior

- ▶ Busca por um registro de chave x :
 1. Calcular o endereço aplicando a função $h(x)$
 2. Percorrer a lista encadeada associada ao endereço
 3. Comparar a chave de cada nó da lista encadeada com a chave x , até encontrar o nó desejado
 4. Se final da lista for atingido, registro não está lá

Encadeamento Exterior

- ▶ Inserção de um registro de chave x
 1. Calcular o endereço aplicando a função $h(x)$
 2. Buscar registro na lista associada ao endereço $h(x)$
 3. Se registro for encontrado, sinalizar erro
 4. Se o registro não for encontrado, inserir no final da lista

Encadeamento Exterior

▶ Exclusão de um registro de chave x

1. Calcular o endereço aplicando a função $h(x)$
2. Buscar registro na lista associada ao endereço $h(x)$
3. Se registro for encontrado, excluir registro
4. Se o registro não for encontrado, sinalizar erro

Complexidade no Pior Caso

- ▶ É necessário percorrer uma lista encadeada até o final para concluir que a chave não está na tabela
- ▶ Comprimento de uma lista encadeada pode ser $O(n)$
- ▶ Complexidade no pior caso: $O(n)$

Complexidade no Caso Médio

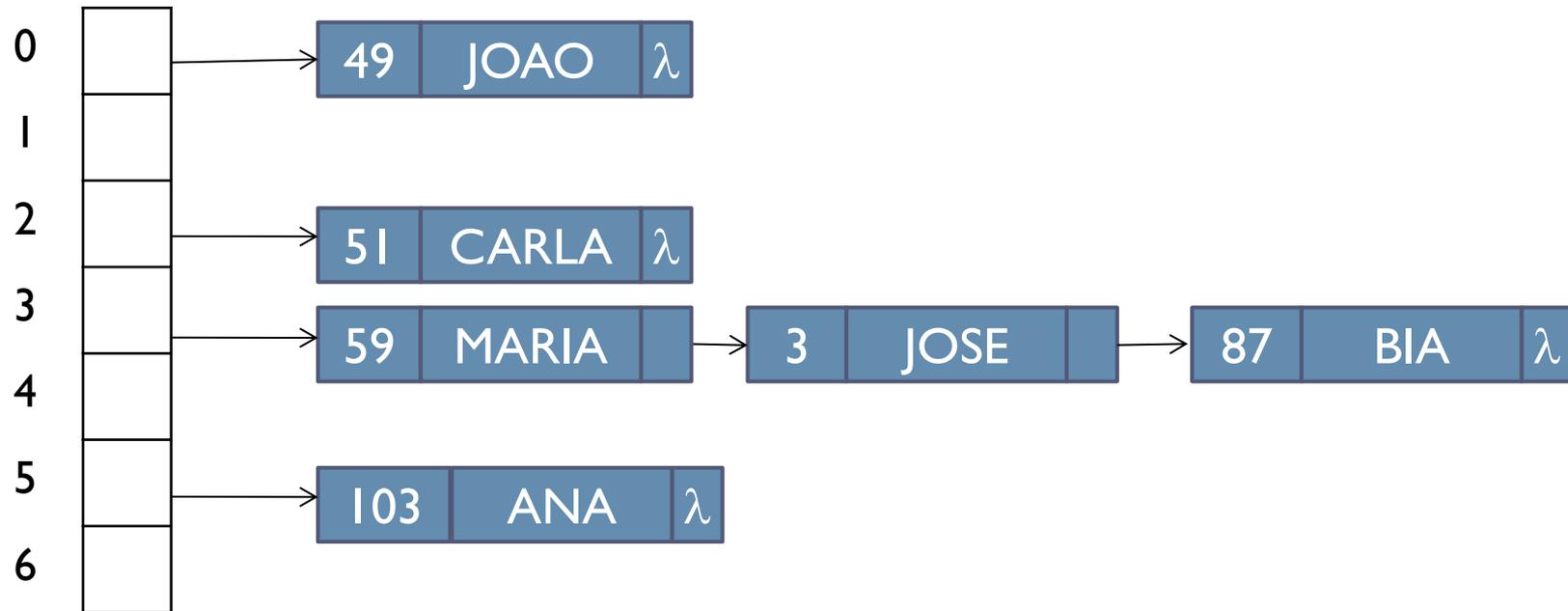
- ▶ Assume que função hash é uniforme
- ▶ Número médio de comparações feitas na **busca sem sucesso** é igual ao fator de carga da tabela $\alpha = n/m$
- ▶ Número médio de comparações feitas na **busca com sucesso** também é igual a $\alpha = n/m$

- ▶ Se assumirmos que o número de chaves **n** é proporcional ao tamanho da tabela **m**
 - ▶ $\alpha = n/m = O(1)$
 - ▶ **Complexidade constante!**

Implementação

- ▶ Normalmente, usa-se um arquivo para armazenar os compartimentos da tabela, e outro para armazenar as listas encadeadas
- ▶ Ponteiros para NULL são representados por -1

Exemplo



Estrutura dos arquivos

Arquivo tabHash.dat
(compartimento_hash)

0	0
1	-1
2	4
3	1
4	-1
5	2
6	-1

$m = 7$

Arquivo clientes.dat (cliente)

	CodCliente	Nome	Prox	Status
0	49	JOAO	-1	FALSE
1	59	MARIA	3	FALSE
2	103	ANA	-1	FALSE
3	3	JOSE	5	FALSE
4	51	CARLA	-1	FALSE
5	87	BIA	-1	FALSE
6				
7				
8				
...				

Uso de Flag indicador de Status

- ▶ Para facilitar a manutenção da lista encadeada, pode-se adicionar um flag indicador de **status** a cada registro
- ▶ O flag pode ter os seguintes valores:
 - ▶ OCUPADO: quando o compartimento tem um registro
 - ▶ LIBERADO: quando o registro que estava no compartimento foi excluído

Reflexão:

- ▶ Como seriam os procedimentos para inclusão e exclusão?

Implementação de Exclusão

- ▶ Ao excluir um registro, marca-se o flag de status como LIBERADO

Implementação de Inserção (Opção 1)

- ▶ Para inserir novo registro
 - ▶ Inserir o registro no final da lista encadeada, se ele já não estiver na lista
 - ▶ De tempos em tempos, rearrumar o arquivo para ocupar as posições onde o flag de status é LIBERADO

Implementação de Inserção (Opção 2)

- ▶ Para inserir novo registro
 - ▶ Ao passar pelos registros procurando pela chave, guardar o endereço **p** do primeiro nó marcado como LIBERADO
 - ▶ Se ao chegar ao final da lista encadeada, a chave não for encontrada, gravar o registro na posição p
 - ▶ Atualizar ponteiros
 - ▶ Nó anterior deve apontar para o registro inserido
 - ▶ Nó inserido deve apontar para nó que era apontado pelo nó anterior

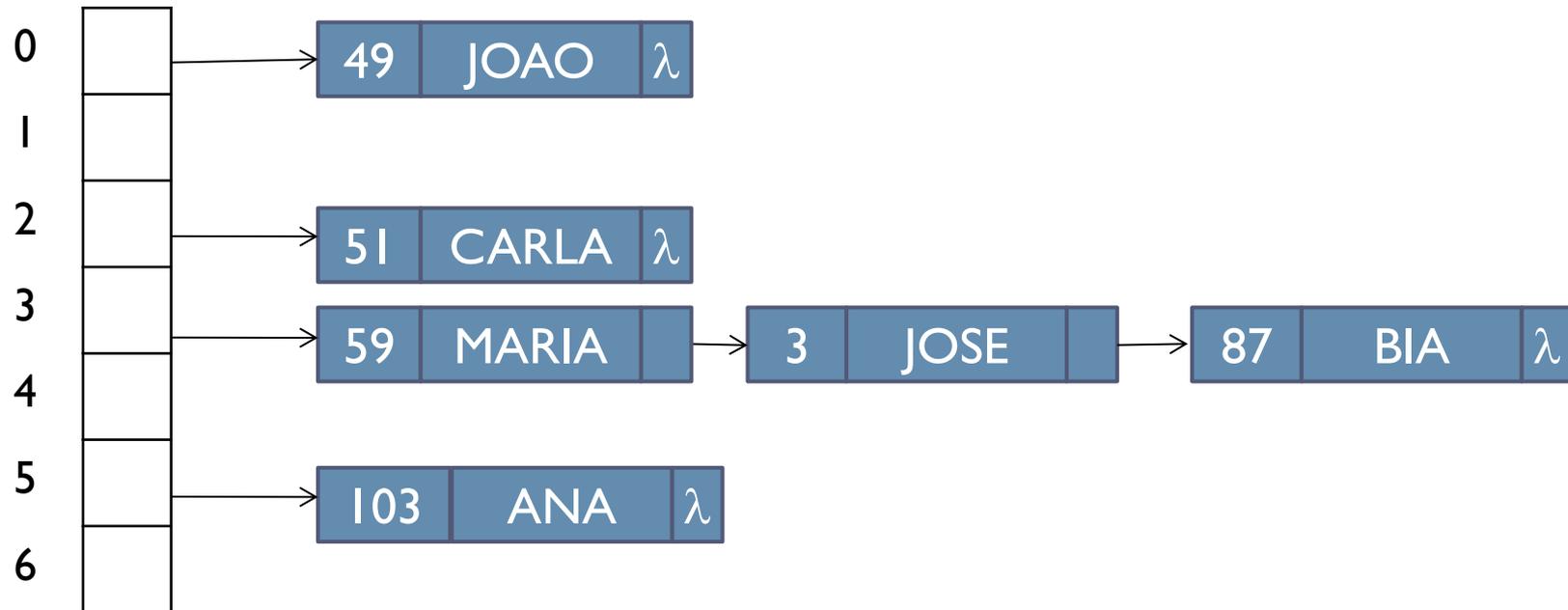
Exercício em Grupo

- ▶ Reúnam-se em grupos
- ▶ Implementar o Encadeamento Exterior
 - ▶ Tamanho da tabela: $m = 7$
 - ▶ Função de hash: $h(x) = x \bmod 7$
 - ▶ Registros a inserir: Clientes (codCliente (inteiro) e nome (String de 100 caracteres))

Estrutura da Implementação

- ▶ **Uso de dois arquivos:**
 - ▶ tabHash.dat (modelado por compartimento_hash.h)
 - ▶ clientes.dat (modelado por cliente.h)

Exemplo



Estrutura dos arquivos

Arquivo tabHash.dat
(compartimento_hash)

0	0
1	-1
2	4
3	1
4	-1
5	2
6	-1

$m = 7$

Arquivo clientes.dat (cliente)

	CodCliente	Nome	Prox	Status
0	49	JOAO	-1	FALSE
1	59	MARIA	3	FALSE
2	103	ANA	-1	FALSE
3	3	JOSE	5	FALSE
4	51	CARLA	-1	FALSE
5	87	BIA	-1	FALSE
6				
7				
8				
...				



FALSE = OCUPADO
TRUE = LIBERADO

Modo de abertura do arquivo

- ▶ Usar modo **r+b** para abrir o arquivo, pois esse modo permite ler e escrever em qualquer ponto do arquivo

Tratamento de Colisões por Encadeamento

- ▶ Encadeamento Exterior
- ▶ **Encadeamento Interior**

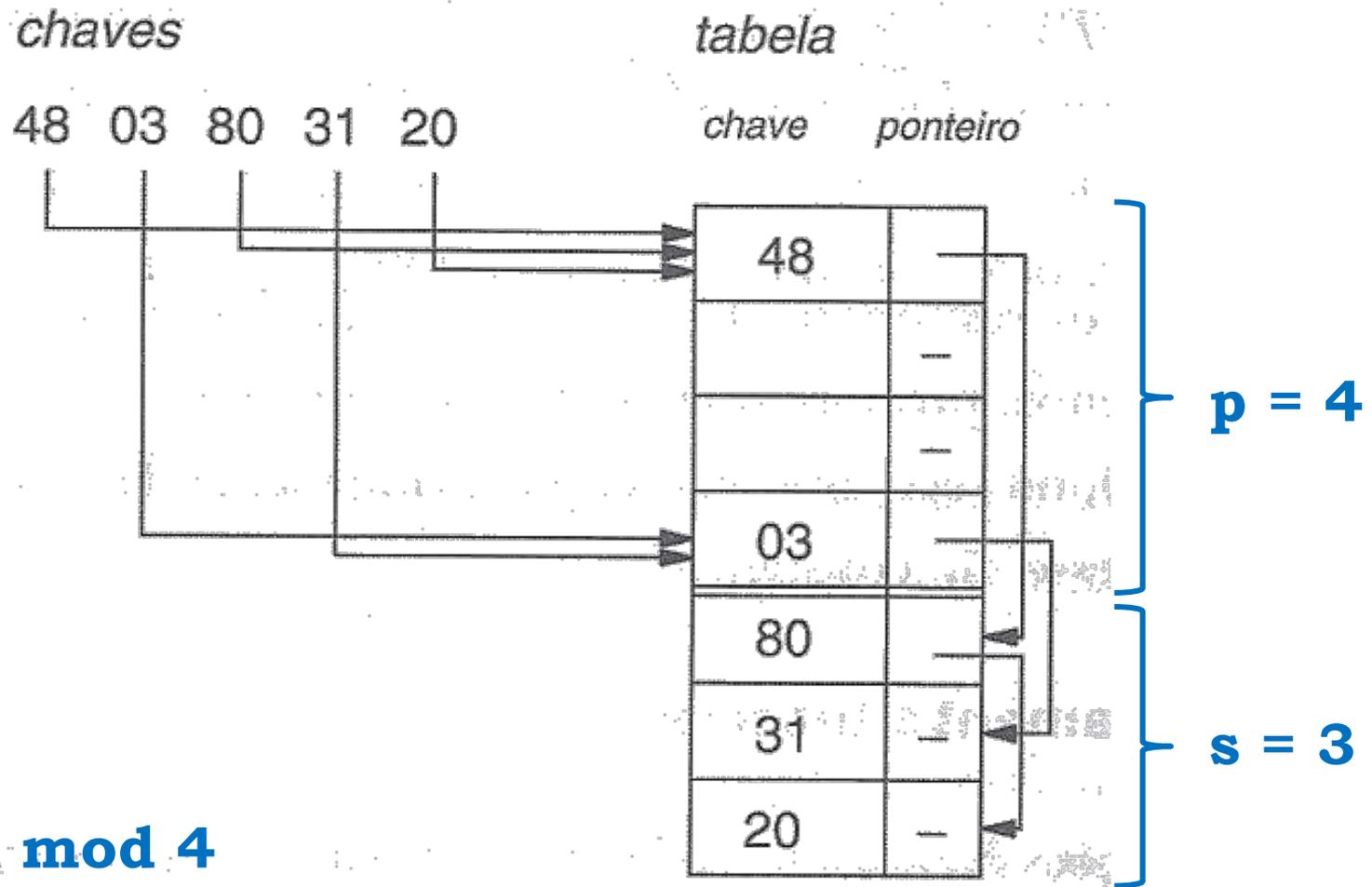
Encadeamento Interior

- ▶ 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



$$h(x) = x \bmod 4$$

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:
 - ▶ $p = 1; s = m - 1$
 - ▶ $p = m - 1; s = 1$

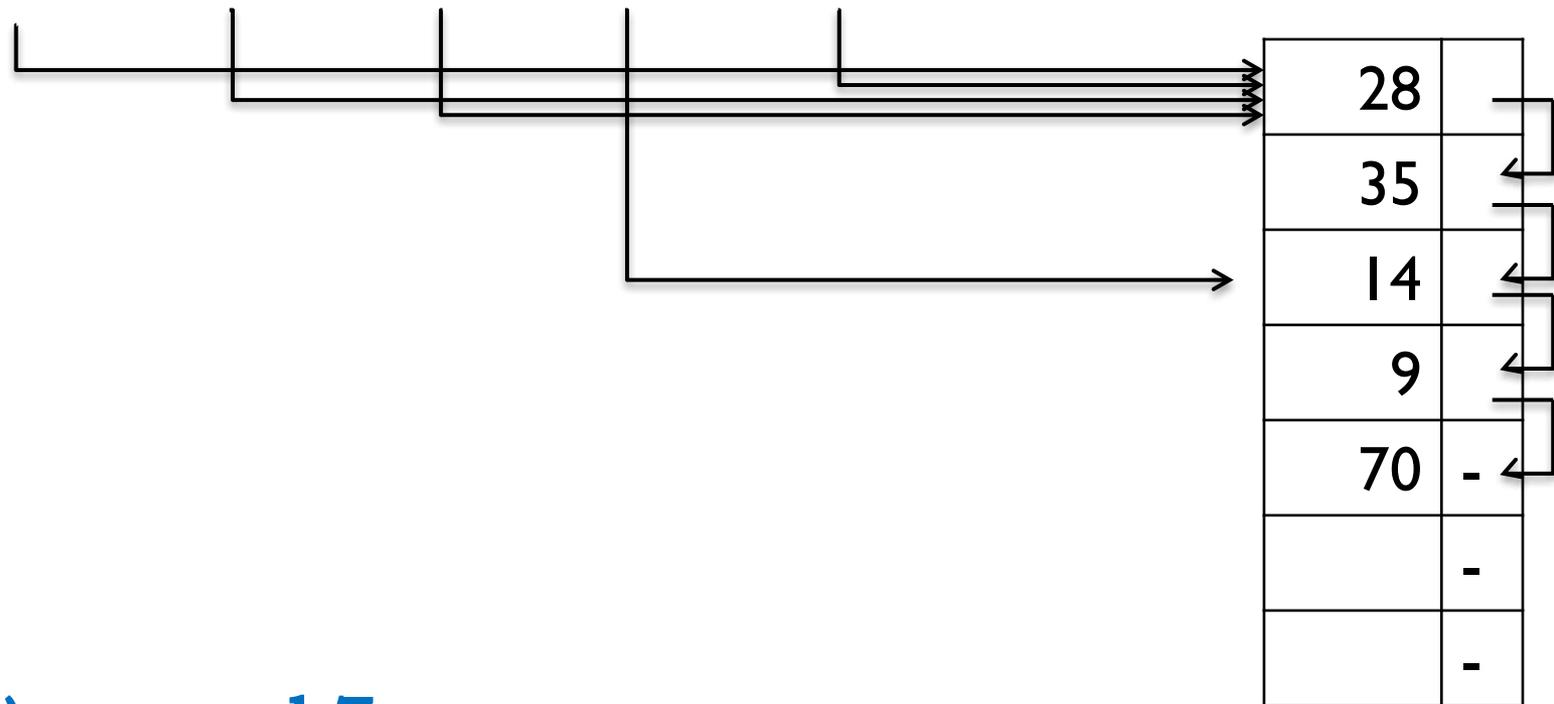
Encadeamento Interior **sem** Zona de Colisões

- ▶ 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

Chaves

28 35 14 9 70



$$h(x) = x \bmod 7$$

-
- 75 Note que a Fig. 10.7, pag 243 do livro busca compartimentos livres de baixo para cima

Procedimento: Busca por Encadeamento Interior

/* Procedimento assume que a tabela tenha sido inicializada da seguinte maneira: $T[i].estado = \text{liberado}$, e $T[i].pont = i$, para $0 < i < m-1$

RETORNO:

Se chave x for encontrada, $a = 1$,
 $end = \text{endereço onde } x \text{ foi encontrada}$

Se chave x não for encontrada, $a = 2$, e há duas possibilidades para o valor de end :

$end = \text{endereço de algum compartimento livre, encontrado na lista encadeada associada a } h(x)$

$end = \lambda$ se não for encontrado endereço livre

*/

Procedimento: Busca por Encadeamento Interior

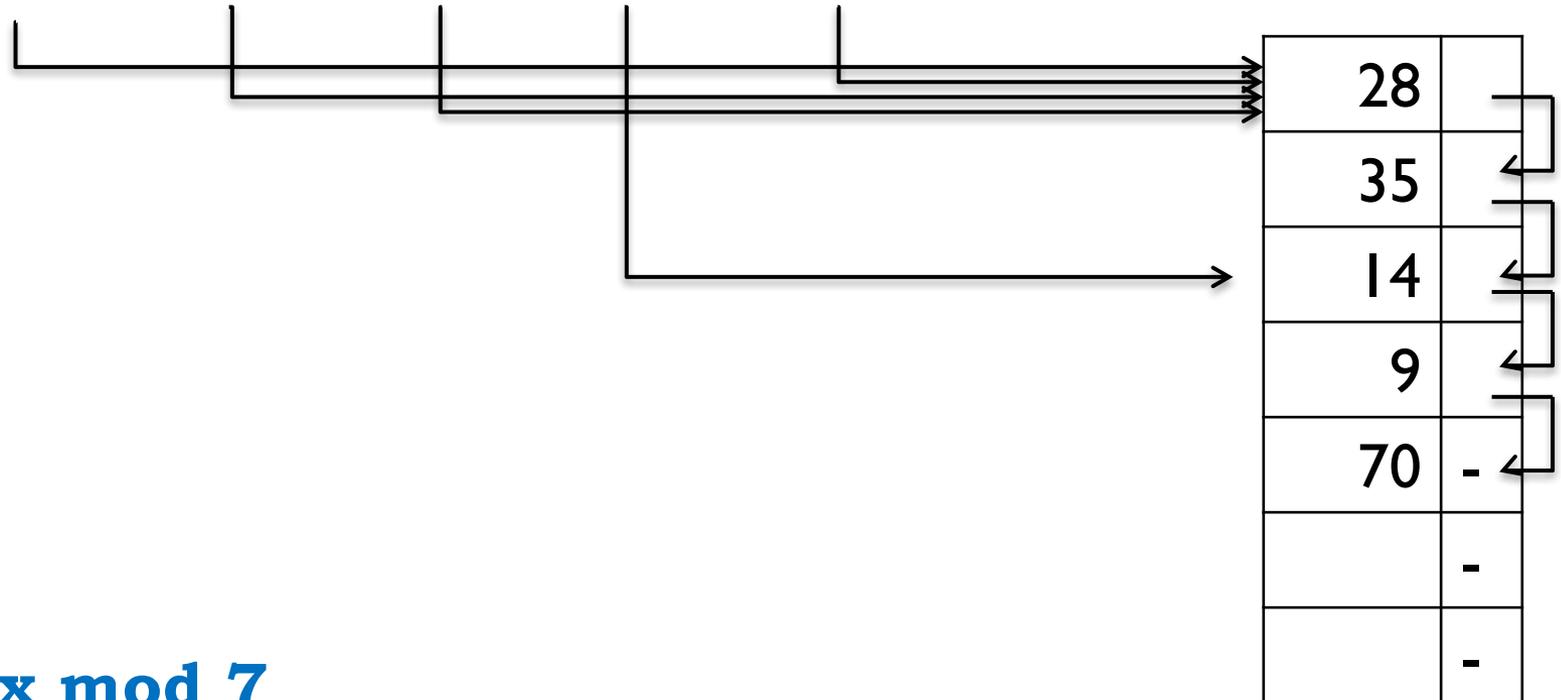
```
procedimento busca(x, end, a)
  a:= 0; end:= h(x); j:=  $\lambda$ ;
  enquanto a = 0 faça
    se T[end].estado = liberado então j := end
    se T[end].chave = x e T[end].estado = ocupado então
      a := 1 % chave encontrada
    senão
      se end = T[end].pont então
        a := 2; end := j % chave não encontrada
      senão end := T[end].pont
  fim enquanto
```

Exercício: Simular a execução do algoritmo de busca

- ▶ Procurar chave 9

Chaves

28 35 14 9 70



$$h(x) = x \bmod 7$$

Procedimento: Inserção por Encadeamento Interior

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

*/

Procedimento: Inserção por Encadeamento Interior

procedimento insere(x)

 busca(x, end, a)

 se $a \neq 1$ então

 se $end \neq \lambda$ então $j := end$

 senão $i := 1; j := h(x)$

 enquanto $i \leq m$ faça

 se $T[j].estado = \text{ocupado}$ então

$j := (j + 1) \bmod m$

$i := i + 1$

 senão $i := m + 2$ % comp. não ocupado

 se $i = m + 1$ então "inserção inválida: overflow"; pare

 temp := $T[h(x)].pont$ % fusão de listas

$T[h(x)].pont := j$

$T[j].pont := temp$

$T[j].chave := x$ % inserção de x

$T[j].estado := \text{ocupado}$

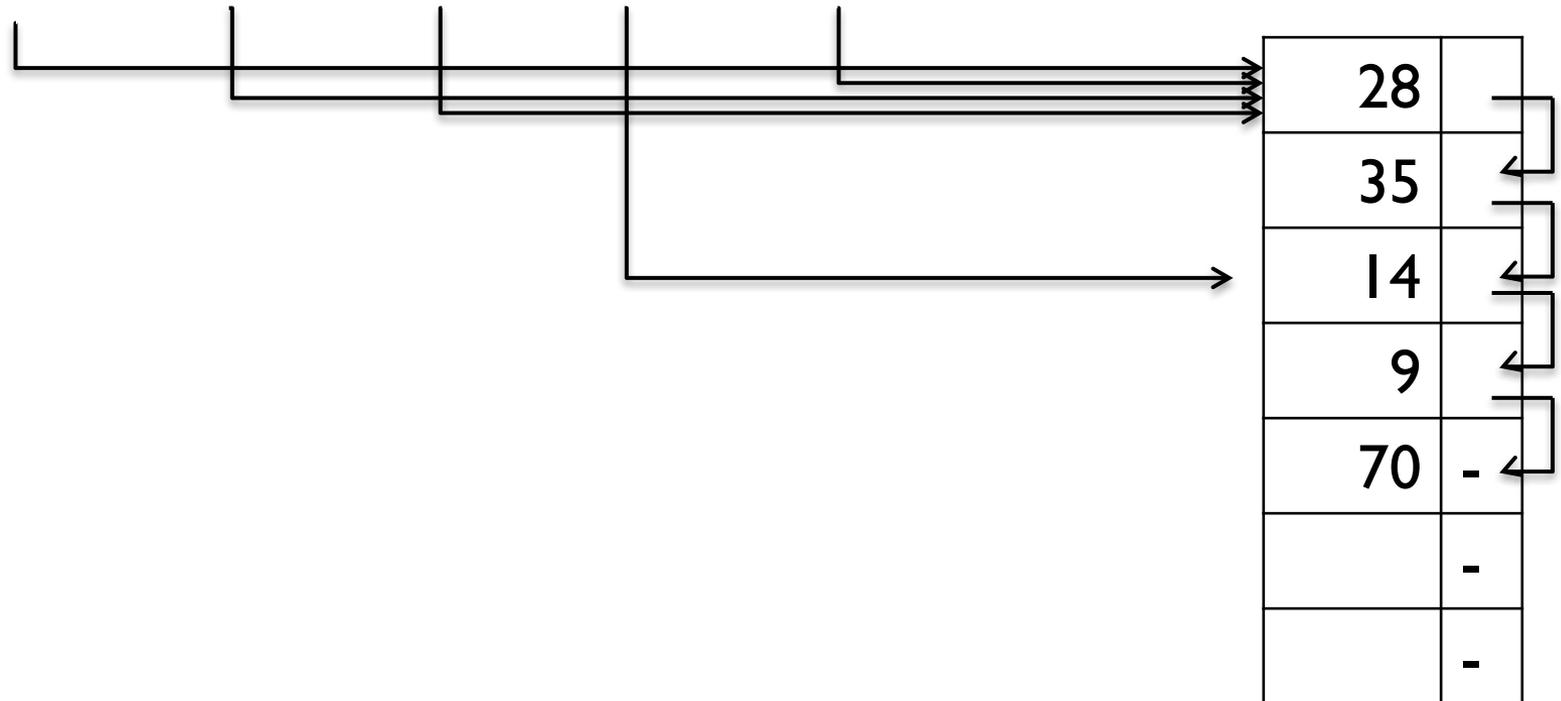
 senão "inserção inválida: chave já existente"

Exercício: Simular a execução do algoritmo de inserção

- ▶ Inserir chave 21

Chaves

28 35 14 9 70



Procedimento: Remoção por Encadeamento Interior

procedimento remove(x)

 busca(x, end, a)

 se a = 1 então T [end].estado: = liberado

 senão "exclusão inválida: chave não existente"

Tratamento de Colisões

- ▶ Por Encadeamento
- ▶ **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 $k = 0, \dots, m-1$
- ▶ 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 $h(x,0), h(x,1), \dots, h(x, m-1)$ 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

Procedimento: Busca por Endereçamento Aberto

```
/* Tabela deve ser inicializada com T[i].chave =
    $\lambda$  .Se a = 1, chave foi encontrada. Se a = 2
   ou 3, a chave não foi encontrada pq
   encontrou uma posição livre (a=2) ou pq a
   tabela foi percorrida até o final (a=3)
*/
procedimento busca-aberto(x, end, a)
    a:=3; k:=0
    enquanto k < m faça
        end:= h(x, k)
        se T[end].chave = x então
            a:= 1  % chave encontrada
            k:= m
        senão se T[end].chave =  $\lambda$  então
            a:= 2  % posição livre
            k:= m
        senão k:= k+ 1
```

Função hash

- ▶ Exemplos de funções hash p/ gerar sequência de tentativas
 - ▶ Tentativa Linear
 - ▶ Tentativa Quadrática
 - ▶ Dispersão Dupla

Função hash

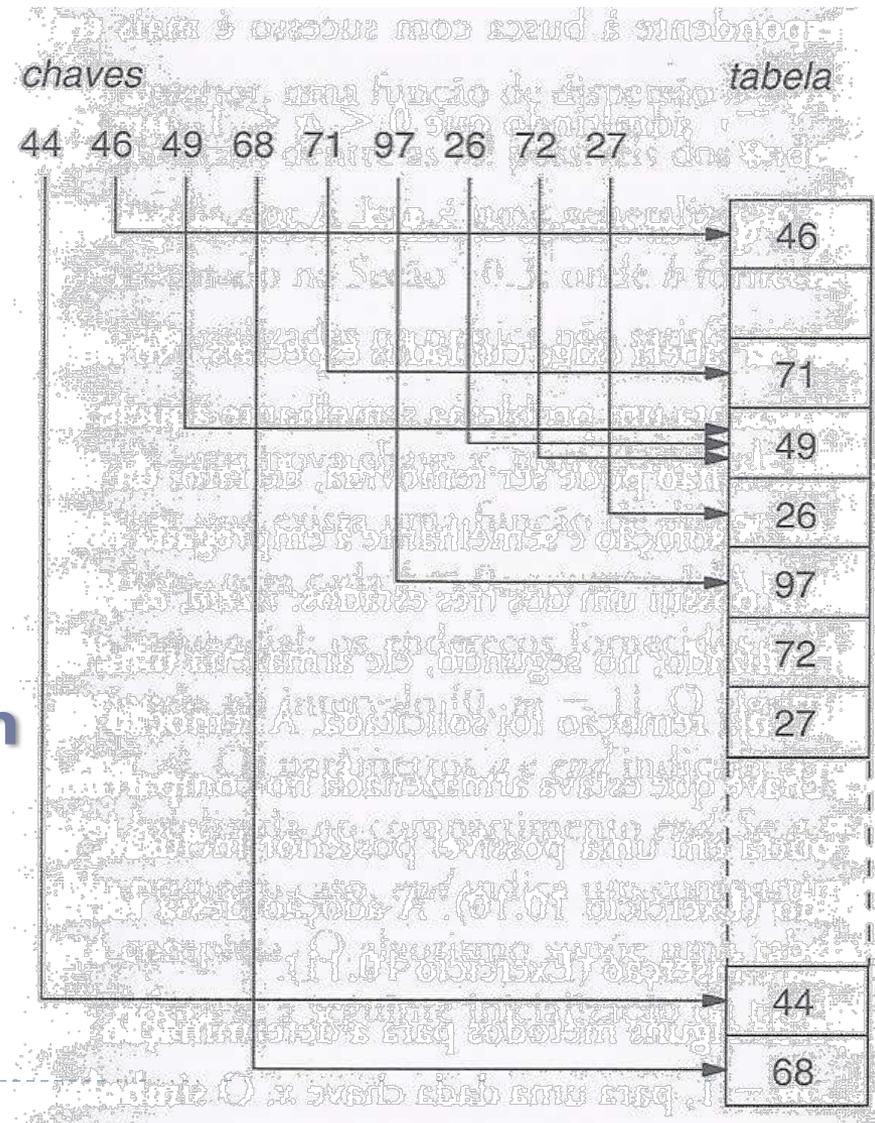
- ▶ Exemplos de funções hash p/ gerar sequência de tentativas
 - ▶ **Tentativa Linear**
 - ▶ Tentativa Quadrática
 - ▶ Dispersão Dupla

Tentativa Linear

- ▶ Suponha que o endereço base de uma chave x é $h'(x)$
- ▶ Suponha que já existe uma chave y ocupando o endereço $h'(x)$
- ▶ Idéia: 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
- ▶ $h(x, k) = (h'(x) + k) \bmod m, 0 \leq k \leq m-1$

Exemplo Tentativa Linear

- ▶ Observem a tentativa de inserir chave 26
- ▶ Endereço já está ocupado: inserir no próximo endereço livre
- ▶ $h(x, k) = (h'(x) + k) \bmod m$
- ▶ $h'(x) = x \bmod 23$



Quais são as desvantagens?

Quais são as desvantagens?

- ▶ 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 compartimento primário para $j + l$
 - ▶ 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 p/ 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
- ▶ $h(x,k) = (h'(x) + c_1 k + c_2 k^2) \bmod m$,
onde c_1 e c_2 são constantes, $c_2 \neq 0$ e $k = 0, \dots, m-1$

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

- ▶ Valores de **m**, **c₁** e **c₂** precisam ser escolhidos de forma a garantir que todos os endereços-base serão percorridos
- ▶ Exemplo:
 - ▶ **$h(x,0) = h'(x)$**
 - ▶ **$h(x,k) = (h(x,k-1) + k) \bmod m$, para $0 < k < m$**
 - ▶ Essa função varre toda a tabela se **m** for potência de 2

Comparação: Tentativa Linear x Tentativa Quadrática

tentativa linear:

endereço-base 0



endereço-base 1

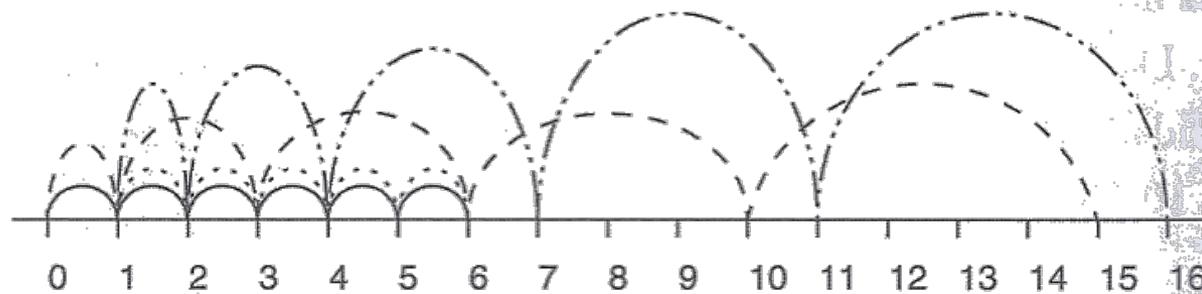


tentativa quadrática:

endereço-base 0



endereço-base 1



Função hash

- ▶ Exemplos de funções hash p/ gerar sequência de tentativas
 - ▶ Tentativa Linear
 - ▶ Tentativa Quadrática
 - ▶ **Dispersão Dupla**

Dispersão Dupla

- ▶ Utiliza duas funções de hash, $h'(x)$ e $h''(x)$
- ▶ $h(x,k) = (h'(x) + k \cdot h''(x)) \bmod m$, 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 $h''(x) = h''(y)$

Tabelas de Dimensão Dinâmica (Tabelas Extensíveis)

Seção 10.6 do livro “Estruturas de Dados e Seus Algoritmos”

Tabelas Extensíveis

- ▶ O que fazer quando o fator de carga da tabela aumenta muito?
 - ▶ Deveria ser possível aumentar o tamanho **m** da tabela, de forma a equilibrar o fator de carga
 - ▶ Quais são os impactos de se aumentar o tamanho **m** da tabela?

Impactos

- ▶ Impactos de se aumentar o tamanho **m** da tabela:
 1. A função hash tem que mudar
 2. Com isso, todos os endereços dos registros armazenados precisam ser recalculados, e os registros movidos

Solução

- ▶ Alterar apenas parte dos endereços já alocados
- ▶ Método: Dispersão linear

Dispersão Linear

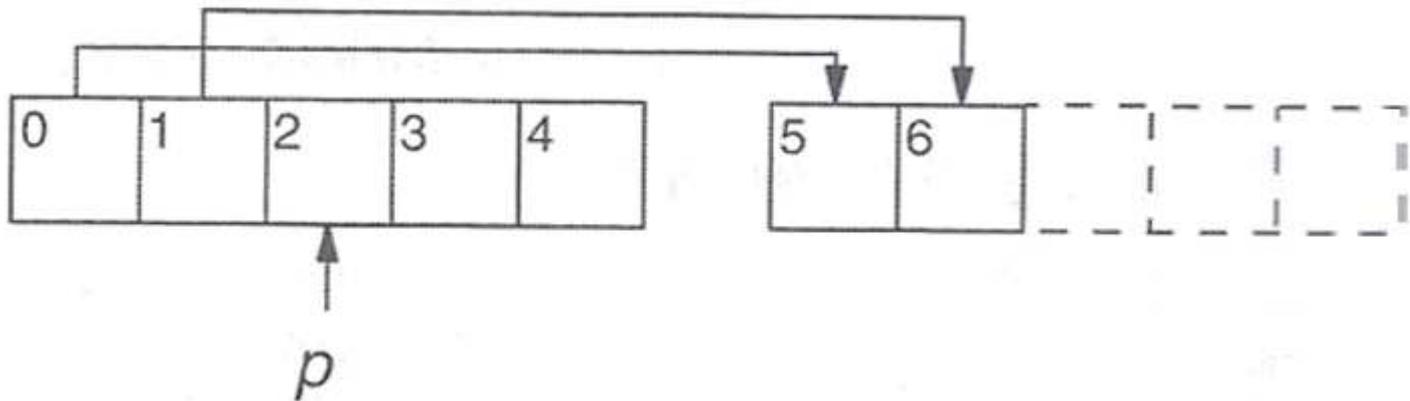
- ▶ Situação inicial: Tabela com **m** compartimentos **0, ..., m-1**
- ▶ Expandir inicialmente o compartimento 0, depois o compartimento 1, e assim sucessivamente
- ▶ Expandir um compartimento **p** significa criar um novo compartimento **q** no final da tabela, denominado expansão de **p**
- ▶ O conjunto de chaves sinônimas, originalmente com endereço-base **p** é distribuído entre os compartimentos **p** e **q** de forma conveniente

Dispersão Linear

- ▶ Quando todos os compartimentos tiverem sido expandidos, o tamanho da tabela terá sido dobrado
- ▶ Nesse ponto o processo poderá ser recomeçado, se necessário

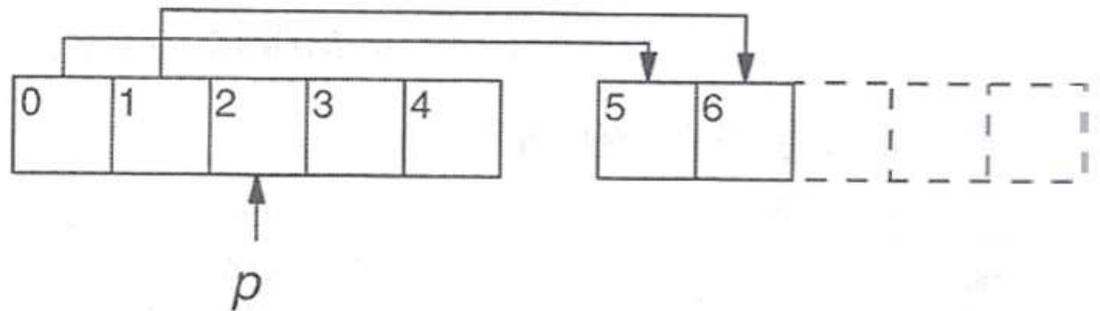
Exemplo ($m = 5$)

- ▶ **p** indica o próximo compartimento a ser expandido



Manutenção dos endereços ao longo do processo

- ▶ Quando a tabela possui tamanho **$m = 5$** , os endereços-base podem ser encontrados com a função **$h_0(x) = x \bmod 5$** , por exemplo
- ▶ Quando a tabela tiver dobrado de tamanho, as chaves serão endereçadas com **$h_1(x) = x \bmod 10$**
- ▶ Mas e antes do processo terminar? Como calcular os endereços?



Manutenção dos endereços ao longo do processo

- ▶ Segundo a função **$h_0(x) = x \bmod 5$** :
 - ▶ para pertencer ao compartimento 0, último dígito da chave deve ser 0 ou 5
- ▶ Segundo a função **$h_1(x) = x \bmod 10$** :
 - ▶ chaves com último dígito 0 continuam a pertencer ao compartimento 0
 - ▶ chaves com último dígito 5 serão alocadas ao novo compartimento
 - ▶ Nenhum dos registros que iriam para os compartimentos 1, 2, 3 ou 4 sofre alterações

Cálculo dos Endereços

- ▶ Dada uma chave \mathbf{x} , computa-se $\mathbf{h}_0(\mathbf{x})$
- ▶ Seja \mathbf{p} o menor compartimento ainda não expandido
- ▶ Se $\mathbf{h}_0(\mathbf{x}) < \mathbf{p}$, o compartimento correspondente já foi expandido
 - ▶ Usar $\mathbf{h}_1(\mathbf{x})$ para recalcular o endereço correto

Cálculo dos Endereços: Caso Geral

- ▶ É preciso conhecer **l**: número de vezes que a tabela já foi expandida
- ▶ Função de hash: **$h_l = x \bmod (m * 2^l)$**

Procedimento mapear

```
/* l indica o número de vezes que a tabela foi expandida a partir
   de seu tamanho mínimo m
   p indica o próximo compartimento a ser expandido
   inicialmente p e l são iguais a zero
*/
procedimento mapear (x, ender, p, l)
    ender :=  $h_1(x)$ 
    se ender < p então
        ender :=  $h_{l+1}(x)$ 
```

Tratamento de colisões

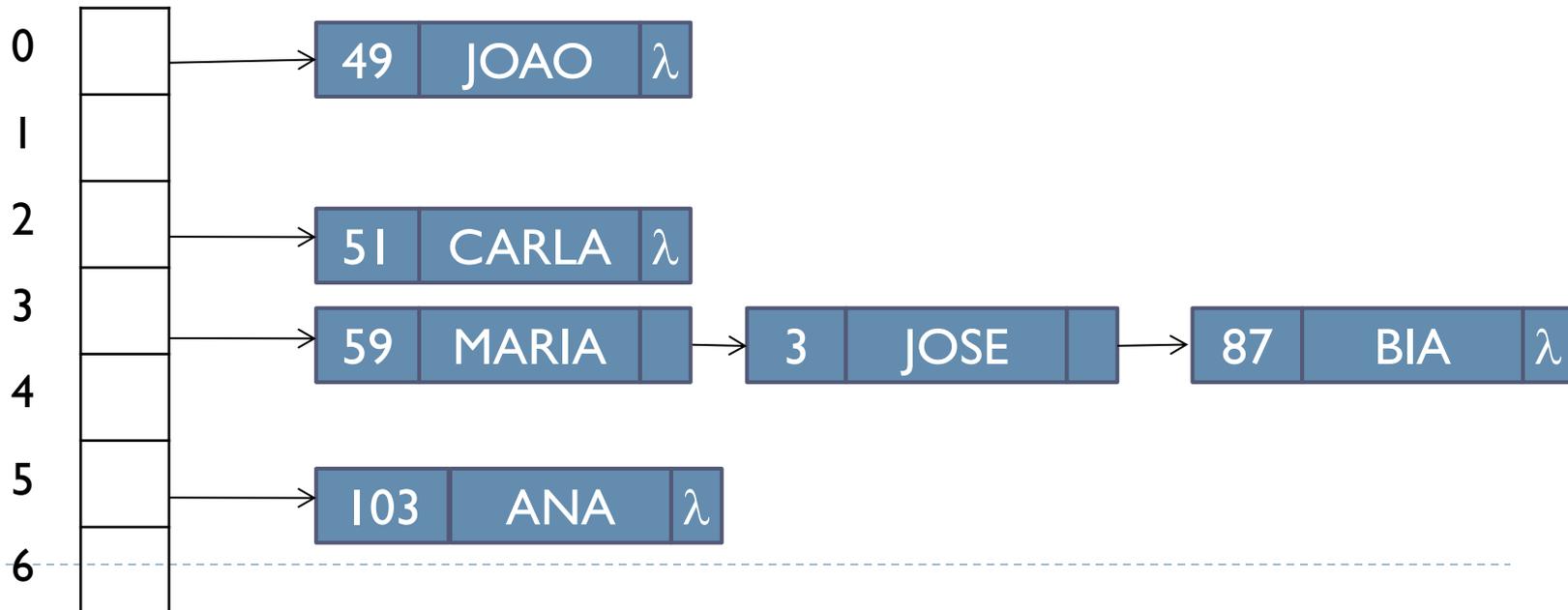
- ▶ **Feito por Encadeamento Exterioror**
 - ▶ Ao expandir um compartimento, é necessário apenas ajustar os ponteiros da lista de nós
 - ▶ Não é necessário mover registros fisicamente

Quando iniciar um processo de expansão?

- ▶ Quando o fator de carga atingir um determinado limite máximo
- ▶ Da mesma forma, pode-se “encolher” a tabela quando o fator de carga atingir um valor muito baixo

Exercício

- Suponha o cenário abaixo, onde $m = 7$, $h_i = x \bmod (m * 2^i)$; $l = 0$; $p = 0$. Suponha que o a tabela deve ser expandida sempre que o fator de carga for maior ou igual a 1. Mostre o estado da tabela após a inserção de cada um dos registros a seguir: (7, CARLOS), (8, JOAO), (14, CAROL), (15, PEDRO)



Discussão

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

Exercício em Grupo

- ▶ Reúnam-se em grupos
- ▶ Implementar o Encadeamento Interior
 - ▶ Tamanho da tabela: $m = 7$
 - ▶ Função de hash: $h(x) = x \bmod 7$
 - ▶ Registros a inserir: Clientes (codCliente (inteiro) e nome (String de 100 caracteres))

Estrutura da Implementação

- ▶ **Uso de um arquivo**
 - ▶ tabHash.dat (cliente.h)

Estrutura do arquivo

Arquivo tabHash.dat

	CodCliente	Nome	Prox	Flag
0	49	JOAO	0	1
1	-1		1	0
2	51	ANA	2	1
3	59	MARIA	4	1
4	10	JANIO	4	1
5	103	PEDRO	5	1
6	-1		6	0

$m = 7$



1 = OCUPADO
0 = LIBERADO

$h(x) = x \bmod 7$

