ACH2024

Aula 20

HASHING

Hashing estático – endereçamento fechado (interno)

Profa. Ariane Machado Lima

Aula anterior

Motivação e Conceitos Básicos

Formas de organizar os registros/elementos/objetos em uma estrutura (baseado em seu campo chave):

- Pelo valor relativo das chaves
  - Ex: vetor ordenado, árvore de busca, árvore B/B+, etc...
- Pelo valor absoluto:
  - Hashing: o endereço base depende apenas do valor absoluto da chave

  Obs: *hash* em inglês: cortar em pequenos pedaços

OBSERVAÇÕES INICIAIS

VAMOS PENSAR A PRINCÍPIO EM MEMÓRIA PRINCIPAL

(HASHING INTERNO)

Vamos falar em armazenar chave como sinônimo de armazenar um registro/objeto/elemento

Motivação e Conceitos Básicos

Você quer armazenar 6 chaves contendo valores de 0 a 5

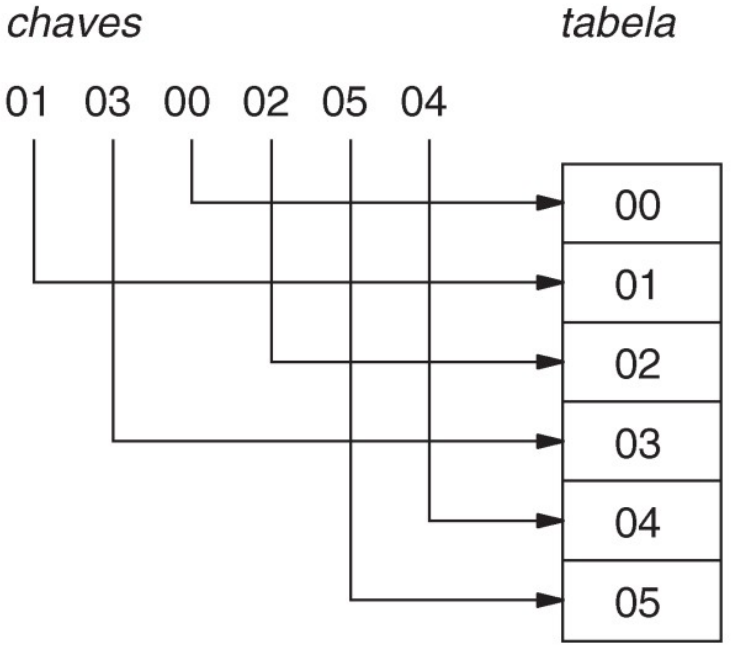
- Que estrutura de dados usaria? Onde armazenaria cada chave?

Motivação e Conceitos Básicos

Você quer armazenar 6 chaves contendo valores de 0 a 5

- Que estrutura de dados usaria? Onde armazenaria cada chave?

Vantagens:



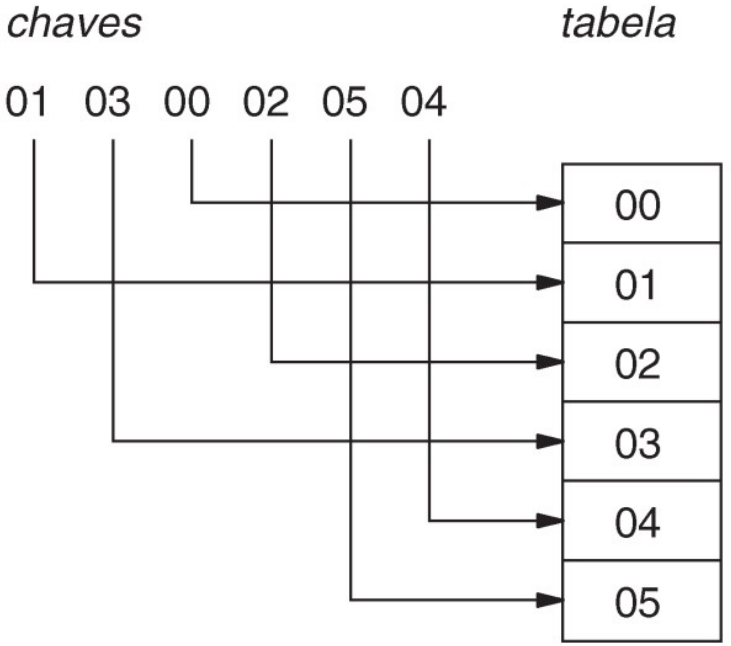
Motivação e Conceitos Básicos

Você quer armazenar 6 chaves contendo valores de 0 a 5

- Que estrutura de dados usaria? Onde armazenaria cada chave?

Vantagens:

- ACESSO DIRETO!!! → Inserção/remoção/busca em tempo constante (O(1))



Motivação e Conceitos Básicos

Agora você quer armazenar 6 chaves contendo valores {0, 1, 3, 4, 5, 7}

- Que estrutura de dados usaria? Onde armazenaria cada chave?

Motivação e Conceitos Básicos

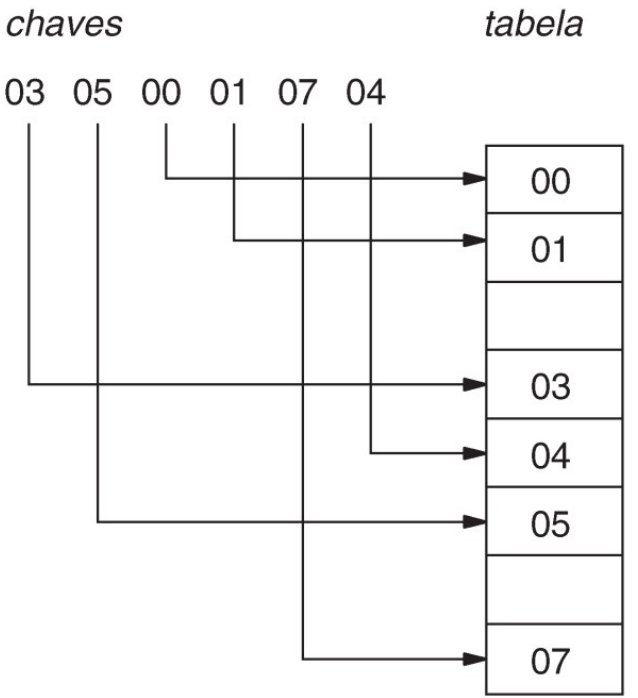
Agora você quer armazenar 6 chaves contendo valores {0, 1, 3, 4, 5, 7}

- Que estrutura de dados usaria? Onde armazenaria cada chave?

Vantagens:

- ACESSO DIRETO!!! → Inserção/remoção/busca em tempo constante (O(1))

Desvantagem:



Motivação e Conceitos Básicos

Agora você quer armazenar 6 chaves contendo valores {0, 1, 3, 4, 5, 7}

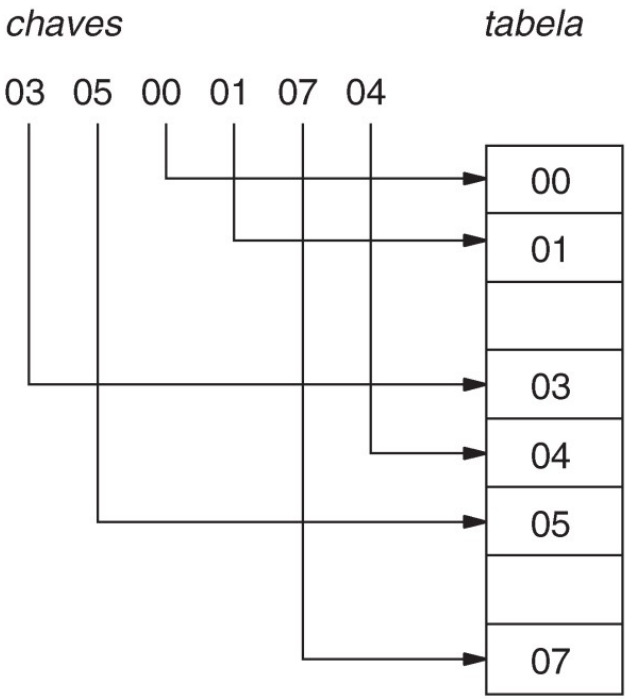
- Que estrutura de dados usaria? Onde armazenaria cada chave?

Vantagens:

- ACESSO DIRETO!!! → Inserção/remoção/busca em tempo constante (O(1))

Desvantagem:

- Pequeno desperdício de espaço (parece valer a pena)



Motivação e Conceitos Básicos

Agora você quer armazenar 6 chaves contendo valores {0, 7, 15, 367, 4067, 50876}

- Que estrutura de dados usaria? Onde armazenaria cada chave?

Motivação e Conceitos Básicos

Agora você quer armazenar 6 chaves contendo valores {0, 7, 15, 367, 4067, 50876}

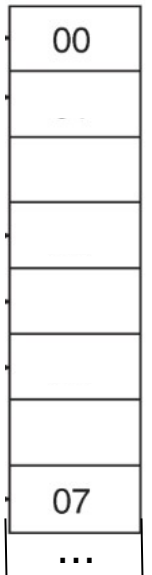
- Que estrutura de dados usaria? Onde armazenaria cada chave?

Vantagens:

- ACESSO DIRETO!!! → Inserção/remoção/busca em tempo constante (O(1))

Desvantagem:

- Desperdício de espaço já começa a não valer a pena quando chave máxima > > número de chaves



Motivação e Conceitos Básicos

Agora você quer armazenar 6 chaves contendo valores {0, 7, 15, 367, 4067, 50876}

- Que estrutura de dados usaria? Onde armazenaria cada chave?

IDEIA:

1) utilizar um vetor (tabela) de tamanho m
2) aplicar uma função que mapeie cada chave a um número de 0 a m-1 (Ex?)

Motivação e Conceitos Básicos

Agora você quer armazenar 6 chaves contendo valores {0, 7, 15, 367, 4067, 50876}

- Que estrutura de dados usaria? Onde armazenaria cada chave

IDEIA:

1) utilizar um vetor (tabela) de tamanho m
2) aplicar uma função que mapeie cada chave a um número de 0 a m-1

Ex: m = 10 e pegar o primeiro dígito (ou letra)

Hashing: picar/dividir o conjunto em **slots**

Tabela de armazenamento: **tabela de hash**

Função de mapeamento: **função de hash**

Endereço calculado: **endereço-base**

Motivação e Conceitos Básicos

Agora você quer armazenar 6 chaves contendo valores {0, 7, 15, 367, 4067, 50876}

- Que estrutura de dados usaria? Onde armazenaria cada chave

IDEIA:

1) utilizar um vetor (tabela) de tamanho m
2) aplicar uma função que mapeie chave a um número de 0 a m-1

Ex: m = 10 e pegar o primeiro dígito (ou letra)

Hashing: picar/dividir o conjunto em **slots**

Tabela de armazenamento: **tabela de dispersão**

Função de mapeamento: **função de dispersão**

Endereço calculado: **endereço-base**

Motivação e Conceitos Básicos

Agora você quer armazenar 6 chaves contendo valores {0, 7, 15, 367, 4067, 50876}

- Que estrutura de dados usaria? Onde armazenaria cada chave

IDEIA:

1) utilizar um vetor (tabela) de tamanho m
2) aplicar uma função que mapeie chave a um número de 0 a m-1

Ex: m = 10 e pegar o primeiro dígito (ou letra)

Hashing: picar/dividir o conjunto em **slots**

Tabela de armazenamento: **tabela de espalhamento**

Função de mapeamento: **função de espalhamento**

Endereço calculado: **endereço-base**

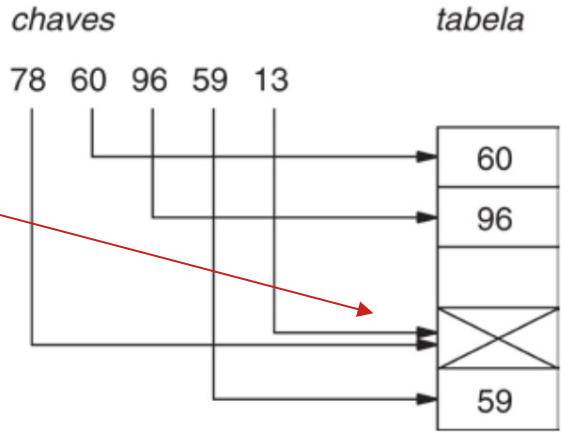
Motivação e Conceitos Básicos

Questões que podem surgir?

Motivação e Conceitos Básicos

Questões que podem surgir:

- O que fazer quando duas chaves caem na mesma posição? (colisão)
  - Tratamento de colisões
- Qual função de hash utilizar? Como ela impacta na ocorrência de colisões?



Aula de hoje

- Funções de hash
- Tratamento de colisões (hashing estático)

Funções de hash

Definição: considerando:

uma tabela de tamanho m (m slots)

Um domínio C de valores de chaves (strings, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, ...)

um função de hash é uma função h: C $\rightarrow$ {0, 1, …, m-1}

Ou seja, se x $\in$ C é uma chave, h(x) retorna o endereço-base de x (ou seja, seu índice na tabela de hash)

Ex: h(x) = x (primeiro exemplo)

h(x) = dígito mais significativo (segundo exemplo)

Funções de hash

Propriedades desejáveis:

Funções de hash

Propriedades desejáveis:

1) Poucas colisões

2) Ser rapidamente calculada (O(1), senão estraga vantagem do hashing)

3) Distribuição uniforme:

- idealmente se há m slots, P(h(x)) = 1/m ∀x

  (a probabilidade de qualquer endereço-base deve ser 1/m)

- importante para minimizar colisões (de pior caso)

- difícil de ser testada, mas bom senso pode ajudar. Ex: dígito mais significativo seria uma boa?

Funções de hash

Propriedades desejáveis:

1) Poucas colisões

2) Ser rapidamente calculada (O(1), senão estraga vantagem do hashing)

3) Distribuição uniforme:

- idealmente se há m slots, P(h(x)) = 1/m ∀x

  (a probabilidade de qualquer endereço-base deve ser 1/m)

- importante para minimizar colisões (de pior caso)

- difícil de ser testada, mas bom senso pode ajudar. Ex: dígito mais significativo seria uma boa? – péssima idea

Funções de hash

Principais métodos de funções de hash:

1) Método da divisão
2) Método da dobra
   - baseado em soma
   - baseado em ou-exclusivo
3) Método da multiplicação
4) Método da análise de dígitos

Funções de hash
Método da divisão

Considerando uma tabela de m slots:

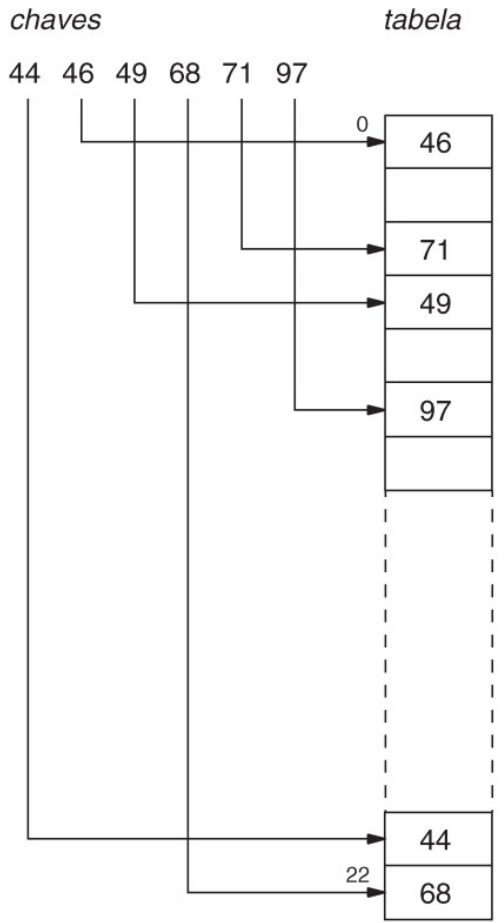
h(x) = x mod m (mod: resto da divisão inteira)

Ex: h(x) = x mod 23

Alguns valores de m são melhores que outros…

Ex de m ruins:

- potência de 2 ou de 10: depende apenas de alguns bits/dígitos (no caso os menos significativos → muito ruim em casos em que chaves podem ser concatenações de outras)
- par: h(x) par <=> x par (ruim se par/ímpar for desbalanceado)



Funções de hash
Método da divisão

Considerando uma tabela de m slots:

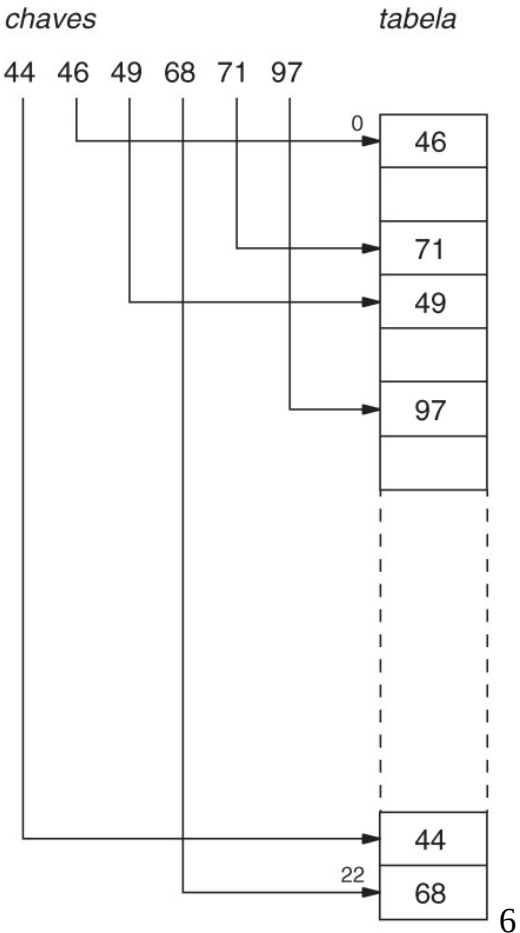
h(x) = x mod m (mod: resto da divisão inteira)

Ex: h(x) = x mod 23

Alguns valores de m são melhores que outros…

Ex de m **bons**:

- Números primos (de preferência não próximo de potência de 2)



Funções de hash
Método da dobra baseado em soma

Método da dobra: sucessivas dobras de trechos do número e efetuar uma operação (ex: soma ou ou-exclusivo)

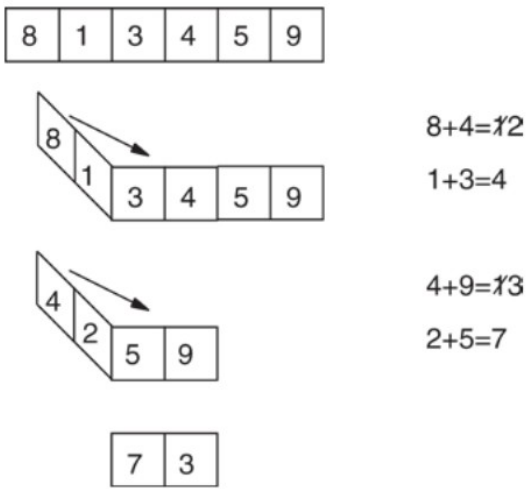
Método da dobra baseado em soma: (obtém h(x) com j dígitos)

x = número descrito com k dígitos decimais : $d_1, d_2, \ldots, d_k$ (ex: k = 6)

Dobras de tamanho j (ex: j = 2) =>

$d_1, \ldots, d_k \rightarrow d'_1, \ldots, d'_j, d_{2j+1}, \ldots, d_k$

Sendo cada d'_i o dígito menos significativo da soma $d_i + d_{2j-i+1}$



Funções de hash
Método da dobra baseado em ou-exclusivo

Método da dobra: sucessivas dobras de trechos do número e efetuar uma operação (ex: soma ou ou-exclusivo)

Método da dobra baseado em ou-exclusivo: (obtém h(x) com J BITS)

x = número binário descrito com K BITS : $d_1, d_2, \ldots, d_K$ (ex: K = 10)

Dobras de tamanho J (ex: J = 5) =>

$d_1, \ldots, d_k \rightarrow d'_1, \ldots, d'_J, d_{2j+1}, \ldots, d_k$

Sendo cada d'_i o bit resultante do ou-exclusivo entre d_i e d_{i+J}

Ex: x = 71 = 00010 00111 → h(x) = 00010 **ouex** 00111 = 00101 = 5

x = 46 = 00001 01110 → h(x) = 00001 **ouex** 01110 = 01111 = 15

Funções de hash
Método da dobra baseado em ou-exclusivo

Método da dobra: sucessivas dobras de trechos do número e efetuar uma operação (ex: soma ou ou-exclusivo)

Método da dobra baseado em ou-exclusivo: (obtém h(x) com J BITS)

x = número binário descrito com K BITS : $d_1, d_2, \ldots, d_K$ (ex: K = 10)

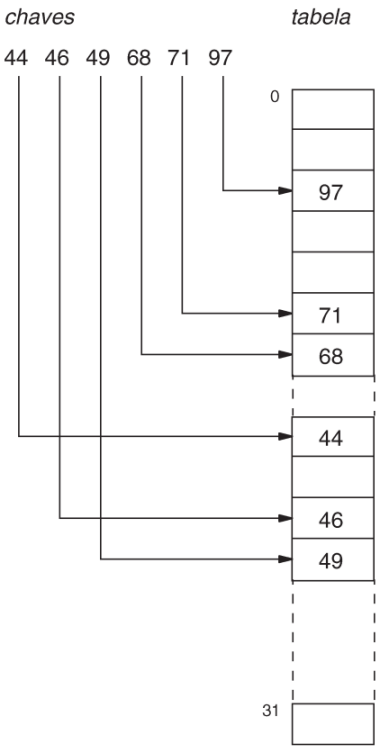
Dobras de tamanho J (ex: J = 5) =>

$d_1, \ldots, d_k \rightarrow d'_1, \ldots, d'_J, d_{2j+1}, \ldots, d_k$

Sendo cada d'_i o bit resultante do ou-exclusivo entre d_i e d_{i+J}

Ex: x = 71 = 00010 00111 → h(x) = 00010 **ouex** 00111 = 00101 = 5

x = 46 = 00001 01110 → h(x) = 00001 **ouex** 01110 = 01111 = 15



Funções de hash
Método da Multiplicação

Muitas variações, a mais conhecida:

Método do "meio do quadrado": (obtém h(x) com b bits – Ex: b = 8)

x = número binário descrito com K BITS : $d_1, d_2, \ldots, d_K$ (ex: K = 32)

=> pegar o b bits centrais do número binário n^2

Ex: x = 157 = 0000 0000 0000 0000 0000 0000 1001 0101

x^2 = 24649 = 0000 0000 0000 0000 0110 0000 0100 1001

h(x) = 0000 0110 = 6

Funções de hash
Método da Análise de Dígitos

Diferentemente dos demais, precisa analisar ANTES o conjunto de **n chaves** a serem armazenadas

Ideia básica: utilizar as d posições de dígitos (decimais) que mais se aproximam da distribuição uniforme (isto é, com menor desvio):

- $n(j)_i$ = número de chaves contendo o dígito i na posição j

- Se a posição j apresenta-se perfeitamente uniforme, cada valor i apareceria n/10 vezes nessa posição (potencialmente com algum desvio)

Procedimento: para números de até k dígitos (posições), calcule o desvio (da dist. uniforme) para a posição j = 1 .. k

$desvio(j) = \sum_{i=0..9} | n(j)_i - n/10|$ ou $\sum_{i=0..9} (n(j)_i - n/10)^2$

E escolha os d dígitos com menor desvio

Ex: chaves = {44, 46, 49, 68, 71, 97}, k = 2, d =1

desvio(1) =7.2 ; desvio(2) = 4.1 => h(x) deve pegar o dígito 2

h(44) = 4; h(46) = 6; h(49) = 9; ...

Tratamento de colisões

Tratamento de colisões

Colisão: quando $x \neq y$ mas $h(x) = h(y)$

Fator de carga: $\alpha = n/m$ (m = nr de slots da tabela de hash, n = nr de chaves a serem inseridas)

Qual a relação de α com o nr de colisões ?

Tratamento de colisões

Colisão: quando $x \neq y$ mas $h(x) = h(y)$

Fator de carga: $\alpha = n/m$ (m = nr de slots da tabela de hash, n = nr de chaves a serem inseridas)

Maior $\alpha \rightarrow$ maior o nr de colisões

Mas $\alpha < 1$ não garante ausência de colisões… $\rightarrow$ tem que tratar

Tratamento de colisões

Estratégias:

A) Hashing estático (tamanho da tabela é constante)

1) Encadeamento ou endereçamento fechado – colisões vão para uma lista ligada

1.1) Encadeamento exterior (fora da tabela)

1.2) Encadeamento interior (dentro da tabela)

2) Endereçamento aberto (chaves dentro da tabela, sem ponteiros)

2.1) Tentativa/Sondagem linear

2.2) Tentativa/Sondagem quadrática

2.3) Dispersão dupla / Hash duplo

B) Hashing dinâmico (tabela pode expandir/encolher)

3) Hashing extensível (estrutura de dados adicional)

4) Hashing linear

Tratamento de colisões

Estratégias:

A) Hashing estático (tamanho da tabela é constante)

1) Encadeamento ou endereçamento fechado – colisões vão para uma lista ligada

1.1) Encadeamento exterior (fora da tabela)

1.2) Encadeamento interior (dentro da tabela)

2) Endereçamento aberto (chaves dentro da tabela, sem ponteiros)

2.1) Tentativa/Sondagem linear

2.2) Tentativa/Sondagem quadrática

2.3) Dispersão dupla / Hash duplo

B) Hashing dinâmico (tabela pode expandir/encolher)

3) Hashing extensível (estrutura de dados adicional)

4) Hashing linear

Tudo isso para hashing interno (em memória) quanto para externo (em disco).
Primeiro assumiremos hashing interno e depois discutiremos mudanças para hashing externo.

Hashing estático

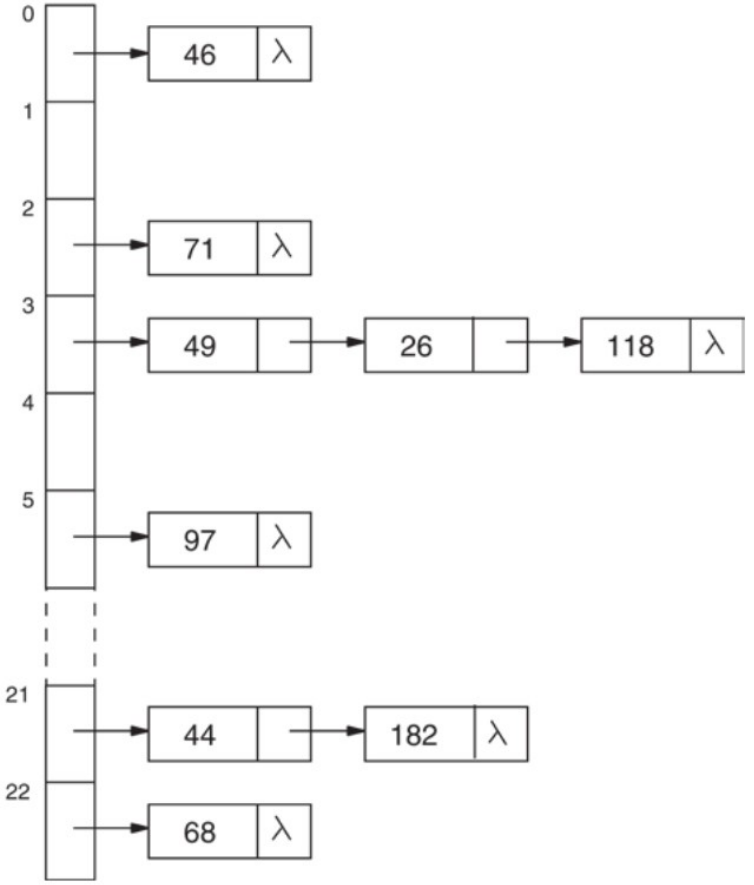
Tratamento de colisões - 1) Endereçamento fechado
1.1) Encadeamento exterior

A tabela de hash é um vetor de m listas ligadas, uma para cada endereço base

- chaves ficam fora do espaço da tabela
- T[i] guarda o ponteiro para o início da lista de chaves com endereço-base = i

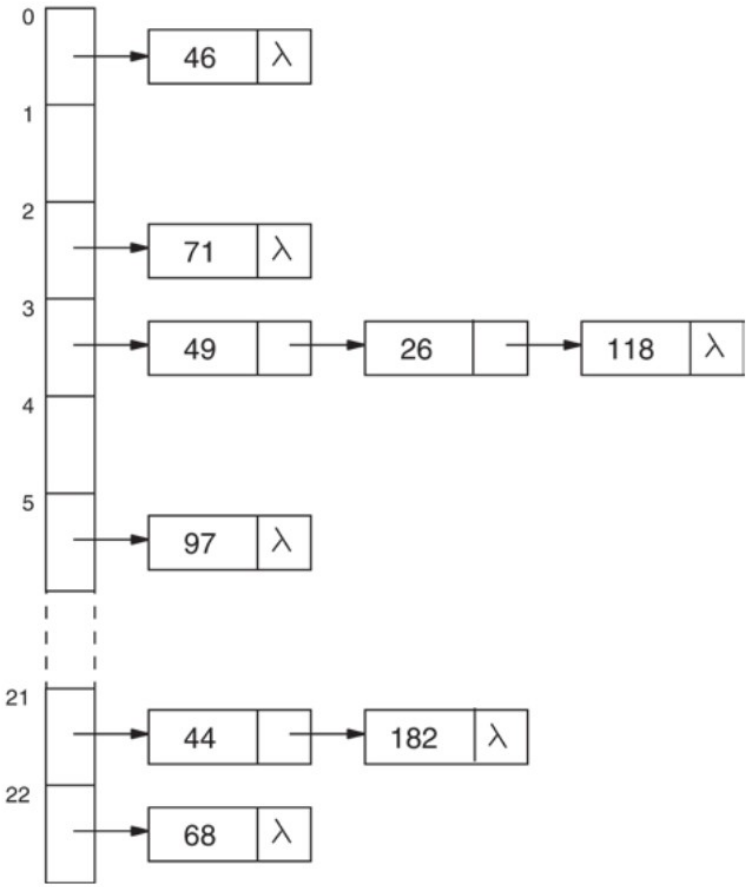
Busca/inserção/remoção: em listas ligadas

Se tiver que impedir duplicação de chaves, pode inserir no final



Tratamento de colisões - 1) Endereçamento fechado
1.1) Encadeamento exterior

Complexidade de busca/inserção/remoção:



Tratamento de colisões - 1) Endereçamento fechado
1.1) Encadeamento exterior

Complexidade de busca/inserção/remoção:

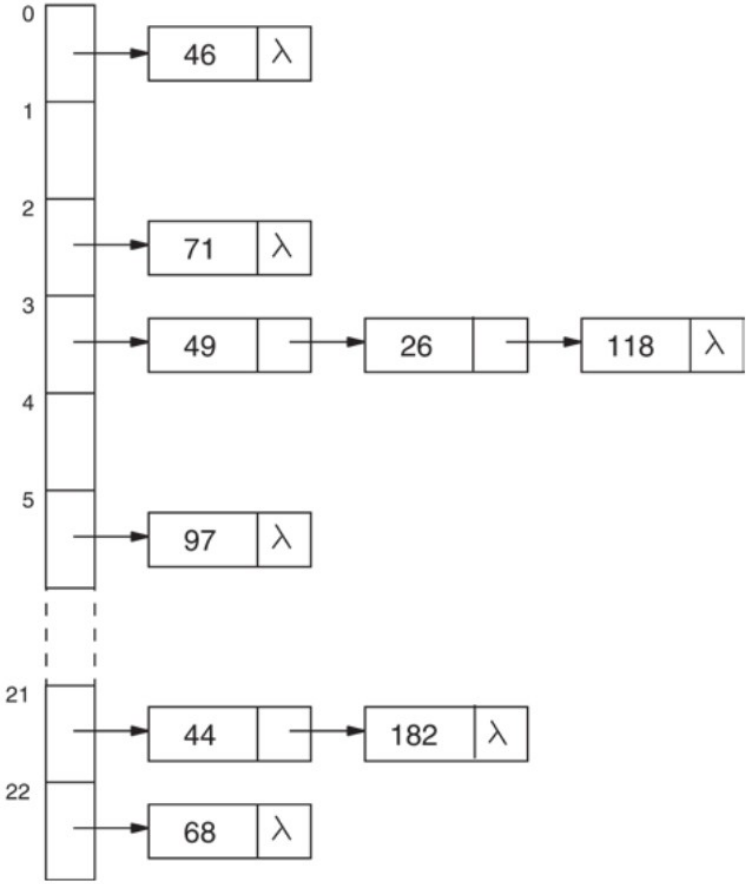
- pior caso: O(n)

- caso médio (assumindo hash uniforme):

  - sem sucesso (chave não existe): tamanho médio da lista = α

    = média ponderada (pela prob. de i) do nr de elementos em cada lista L_i

    = $1/m \sum_{i=1..m} |L_i| = n/m = \alpha$



Tratamento de colisões - 1) Endereçamento fechado
1.1) Encadeamento exterior

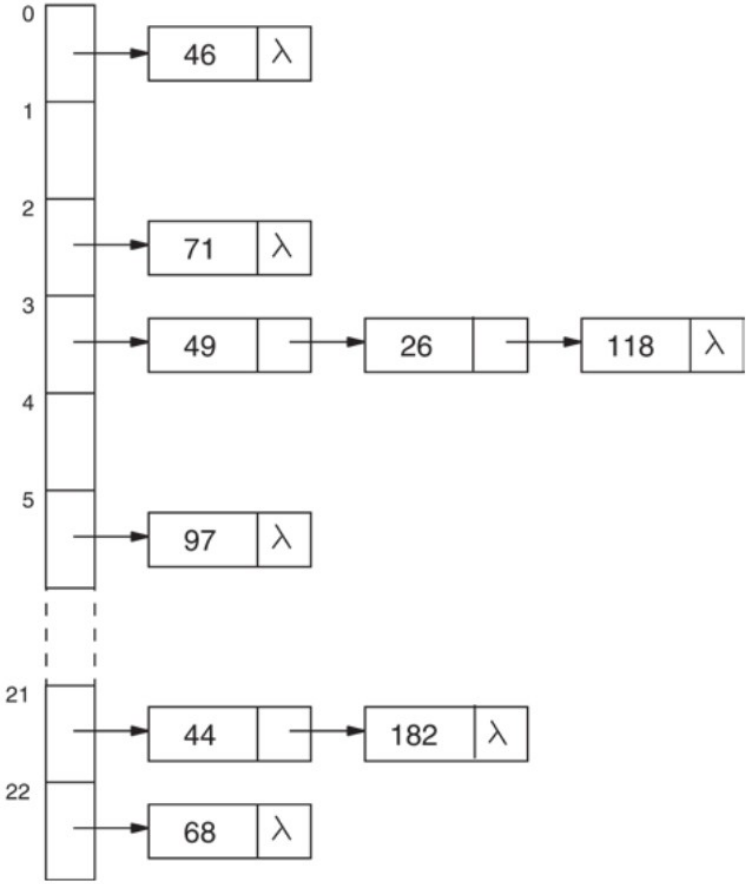
Complexidade de busca/inserção/remoção:

- pior caso: O(n)
- caso médio (assumindo hash uniforme):
  - sem sucesso: α
  - com sucesso: 1 + α/2 - 1/2m

= nr médio de comparações assumindo que x é (j+1)-ésima chave a ser inserida, no final, sem remoções (posição fixa), $|L_i|_{médio} = j/m$

$= 1/n \sum_{j=0..n-1} (1+ j/m) = 1+ n(n-1)/2nm$

$= 1+ (n^2-n)/2nm = 1 + α/2 – 1/2m$



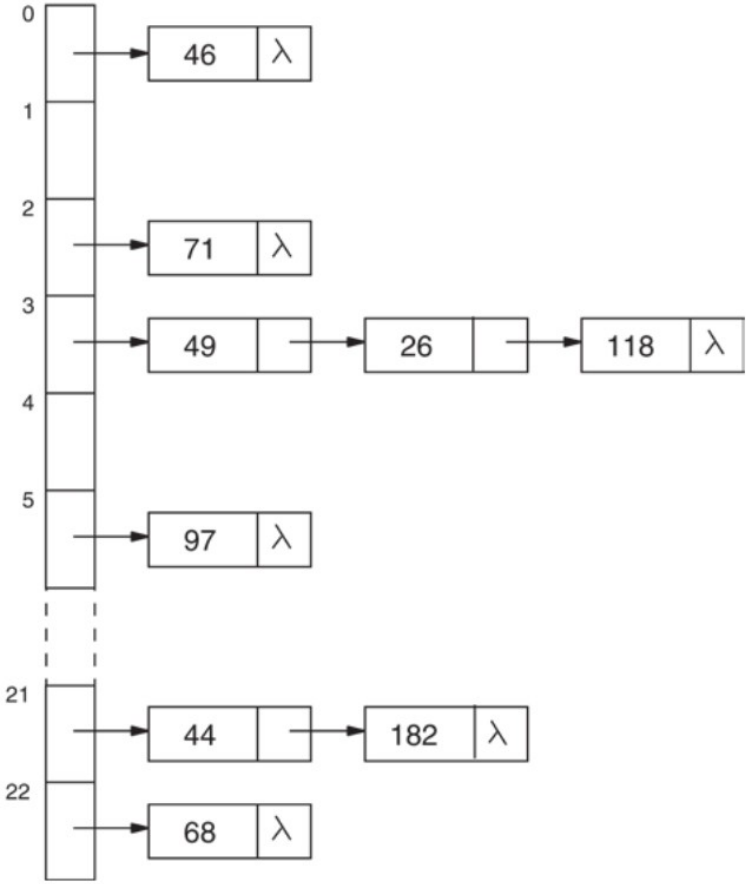
Tratamento de colisões - 1) Endereçamento fechado
1.1) Encadeamento exterior

Complexidade de busca/inserção/remoção:

- pior caso: O(n)
- caso médio (assumindo hash uniforme):
  - sem sucesso: α
  - com sucesso: 1 + α/2 - 1/2m

Isto é, mais rápido conforme:

- α menor (m maior) - linear
- m maior



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior

Listas ligadas ficam dentro da tabela:

T[i] = (chave, slot do próximo)

Exige $\alpha = m/m \leq 1$

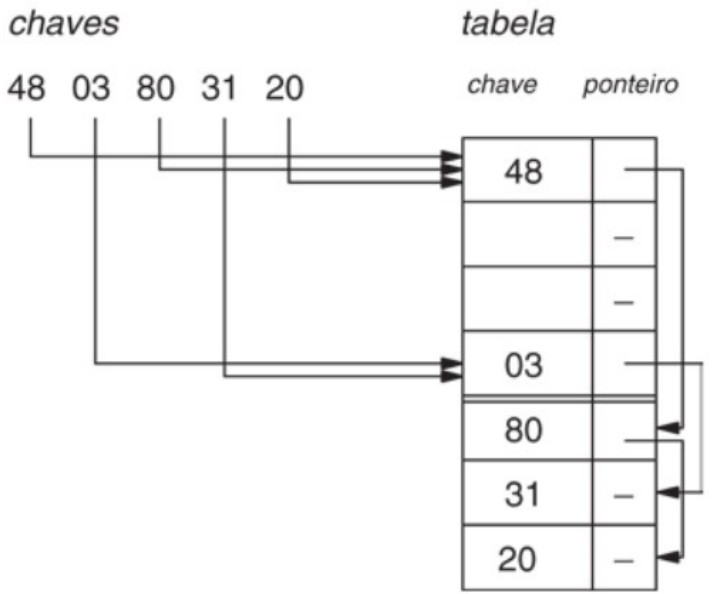
Opção 1: Tabela T dividida em duas áreas (m=p+s)

- p slots para endereços-base
- s slots para **sinônimos**
- h(x) → [0, p-1]
- ponteiros sempre apontam para um valor em [p, m-1]

Problema: área de colisão pode ficar lotada mesmo havendo espaço na área de endereços-base

Possibilidade: aumentar s e diminuir p (mas diminui poder de espalhamento

- no limite p = 1 e s = m-1 → O(n) para buscas



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior

Listas ligadas ficam dentro da tabela:

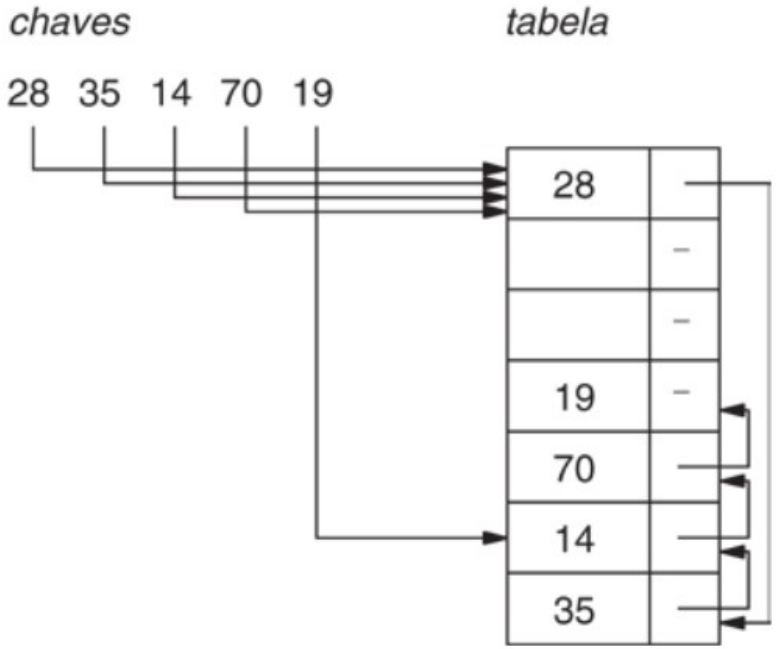
T[i] = (chave, slot do próximo)

Exige $\alpha = n/m \leq 1$

Opção 2: Tabela T mistura endereços-base e sinônimos

- h(x) → [0, m-1]
- se h(x) estiver vago armazena chave x lá

  Senão armazena x na próxima posição livre (a partir de h(x) ou a partir do fim da tabela)



Problema: colisões secundárias: h(y) já está ocupada por uma chave x em que h(x) ≠ h(y)

→ fusão das listas encadeadas de h(x) e h(y)

→ diminuição da eficiência

Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

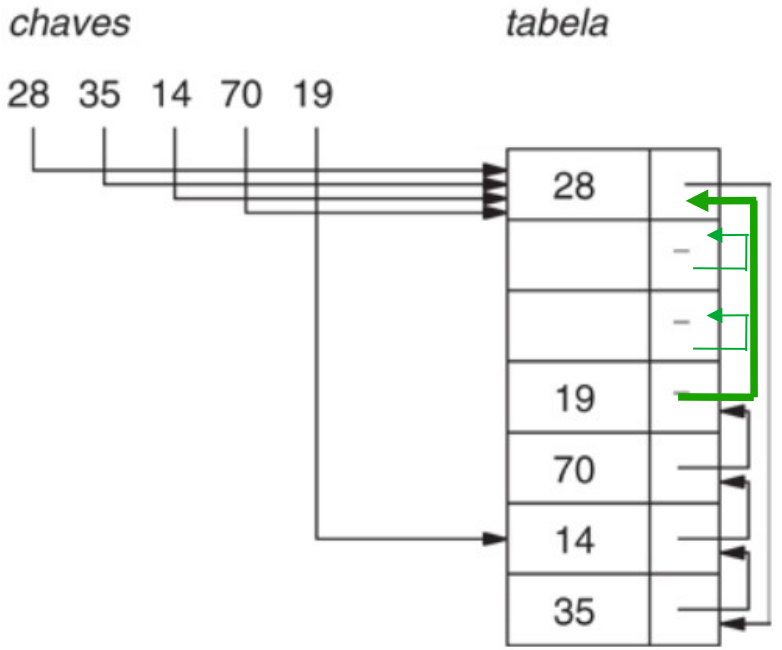
Operações:

Inserção e busca seriam tranquilas, mas e a remoção? (ex: remove 14)

- Não podemos abrir buracos no encadeamento…
- Reorganizar a tabela a cada remoção é normalmente inviável...

Solução:

- cada slot pode ter 1 campo adicional:
  - ocupado: tem chave
  - não ocupado: vazio (sem chave) ou liberado (tinha chave mas não tem mais - pode estar no meio de um encadeamento)
- listas serão circulares (o último da lista de h(x) volta para h(x)



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Operações:

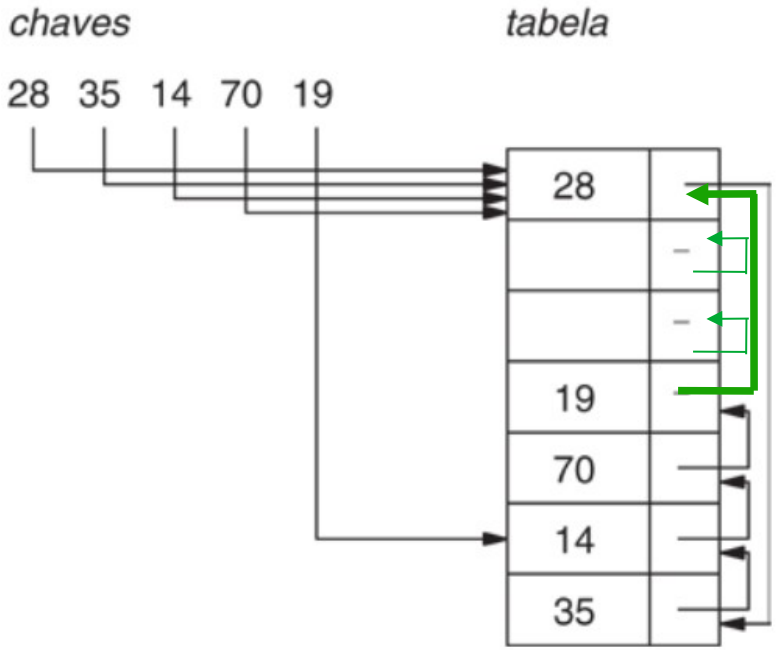
Inicialização: para i = 0 a m-1

T[i].estado = não ocupado

T[i].prox = i (listas serão circulares)

Remoção em T[i] deverá fazer

T[i].estado = não ocupado

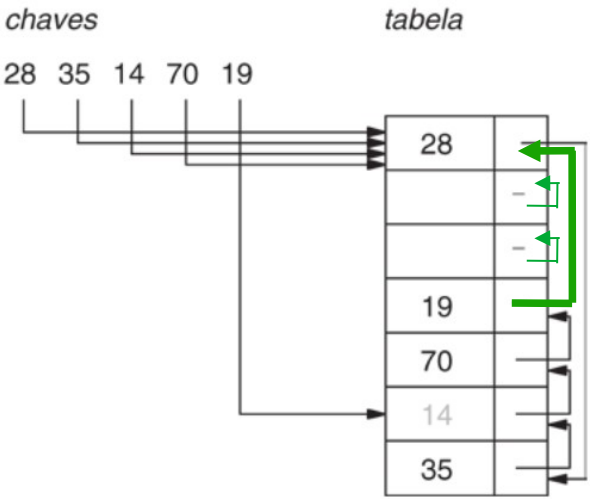


Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Busca (T, x, end, a): **a** indica presença/ausência da chave; **end** auxiliará a inserção

/* se a = 1 então chave encontrada no slot end ; se a = 2 a chave não está na tabela e end tem 2 possibilidades:
1) end = -1 => lista h(x) está vazia ou completa; 2) end = j ≥ 0 => slot j é da lista h(x) e está desocupado */

```
a ← 0; end ← h(x); j ← -1;   /* a = 0 significa que ainda não sei */
enquanto a = 0
   se T[end].estado = não ocupado
        j ←end
   se T[end].chave = x e T[end].estado = ocupado
        a ← 1   /* chave encontrada */
   senão
        end ← T[end].pont
        se end = h(x)    /* voltei para o início da lista circular */
              a←2; end ← j      /* chave não encontrada */
```



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

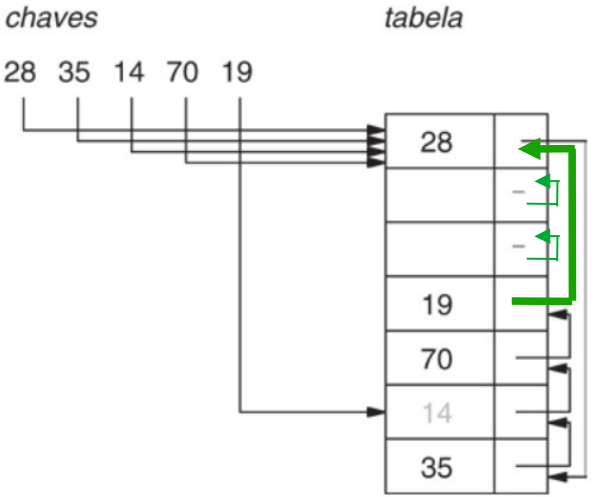
Busca (T, x, end, a): **a** indica presença/ausência da chave; **end** auxiliará a inserção

/* se a = 1 então chave encontrada no slot end ; se a = 2 a chave não está na tabela e end tem 2 possibilidades:
1) end = -1 => lista h(x) está vazia ou completa; 2) end = j ≥ 0 => slot j é da lista h(x) e está desocupado */

```
a ← 0; end ← h(x); j ← -1;   /* a = 0 significa que ainda não sei */
enquanto a = 0
   se T[end].estado = não ocupado
        j ←end
   se T[end].chave = x e T[end].estado = ocupado
        a ← 1   /* chave encontrada */
   senão
        end ← T[end].pont
        se end = h(x)    /* voltei para o início da lista circular */
              a←2; end ← j      /* chave não encontrada */
```

Ex: busca(20, end, a):
a = 0
end = 0 (h(x) = 0)
j = -1

T[end].estado = ocupado
T[end].chave = 28



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

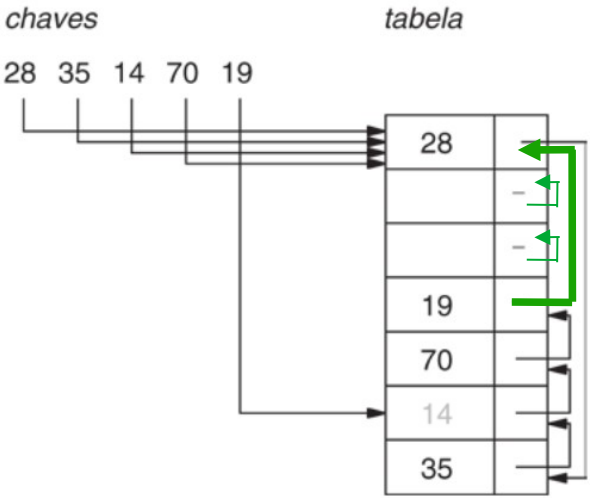
Busca (T, x, end, a): **a** indica presença/ausência da chave; **end** auxiliará a inserção

/* se a = 1 então chave encontrada no slot end ; se a = 2 a chave não está na tabela e end tem 2 possibilidades:
1) end = -1 => lista h(x) está vazia ou completa; 2) end = j ≥ 0 => slot j é da lista h(x) e está desocupado */

```
a ← 0; end ← h(x); j ← -1;   /* a = 0 significa que ainda não sei */
enquanto a = 0
   se T[end].estado = não ocupado
        j ←end
   se T[end].chave = x e T[end].estado = ocupado
        a ← 1   /* chave encontrada */
   senão
        end ← T[end].pont
        se end = h(x)    /* voltei para o início da lista circular */
             a←2; end ← j      /* chave não encontrada */
```

Ex: busca(20, end, a):
a = 0
end = 6 (h(x) = 0)
j = -1

T[end].estado = ocupado
T[end].chave = 35



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

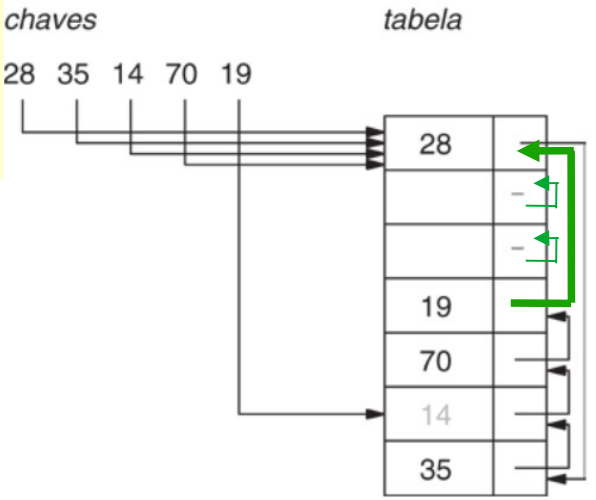
Busca (T, x, end, a): **a** indica presença/ausência da chave; **end** auxiliará a inserção

/* se a = 1 então chave encontrada no slot end ; se a = 2 a chave não está na tabela e end tem 2 possibilidades:
1) end = -1 => lista h(x) está vazia ou completa; 2) end = j ≥ 0 => slot j é da lista h(x) e está desocupado */

```
a ← 0; end ← h(x); j ← -1;   /* a = 0 significa que ainda não sei */
enquanto a = 0
   se T[end].estado = não ocupado
        j ← end
   se T[end].chave = x e T[end].estado = ocupado
        a ← 1   /* chave encontrada */
   senão
        end ← T[end].pont
        se end = h(x)    /* voltei para o início da lista circular */
             a ← 2; end ← j      /* chave não encontrada */
```

Ex: busca(20, end, a):
a = 0
end = 5 (h(x) = 0)
j = 5

T[end].estado = não ocupado
T[end].chave = 14



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

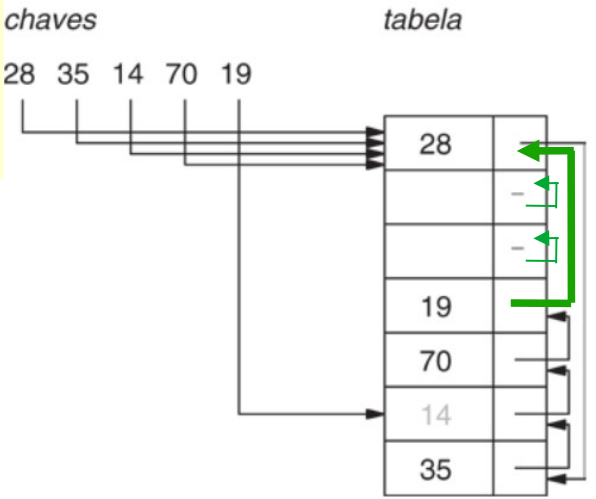
Busca (T, x, end, a): **a** indica presença/ausência da chave; **end** auxiliará a inserção

/* se a = 1 então chave encontrada no slot end ; se a = 2 a chave não está na tabela e end tem 2 possibilidades:
1) end = -1 => lista h(x) está vazia ou completa; 2) end = j ≥ 0 => slot j é da lista h(x) e está desocupado */

```
a ← 0; end ← h(x); j ← -1;   /* a = 0 significa que ainda não sei */
enquanto a = 0
   se T[end].estado = não ocupado
        j ← end
   se T[end].chave = x e T[end].estado = ocupado
        a ← 1   /* chave encontrada */
   senão
        end ← T[end].pont
        se end = h(x)    /* voltei para o início da lista circular */
             a ← 2; end ← j      /* chave não encontrada */
```

Ex: busca(20, end, a):
a = 0
end = 4 (h(x) = 0)
j = 5

T[end].estado = ocupado
T[end].chave = 70



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

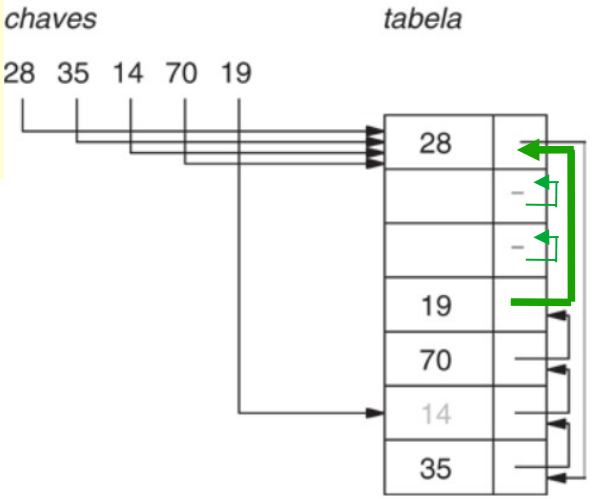
Busca (T, x, end, a): **a** indica presença/ausência da chave; **end** auxiliará a inserção

/* se a = 1 então chave encontrada no slot end ; se a = 2 a chave não está na tabela e end tem 2 possibilidades:
1) end = -1 => lista h(x) está vazia ou completa; 2) end = j ≥ 0 => slot j é da lista h(x) e está desocupado */

```
a ← 0; end ← h(x); j ← -1;   /* a = 0 significa que ainda não sei */
enquanto a = 0
   se T[end].estado = não ocupado
        j ← end
   se T[end].chave = x e T[end].estado = ocupado
        a ← 1   /* chave encontrada */
   senão
        end ← T[end].pont
        se end = h(x)    /* voltei para o início da lista circular */
              a ← 2; end ← j      /* chave não encontrada */
```

Ex: busca(20, end, a):
a = 0
end = 3 (h(x) = 0)
j = 5

T[end].estado = ocupado
T[end].chave = 19



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

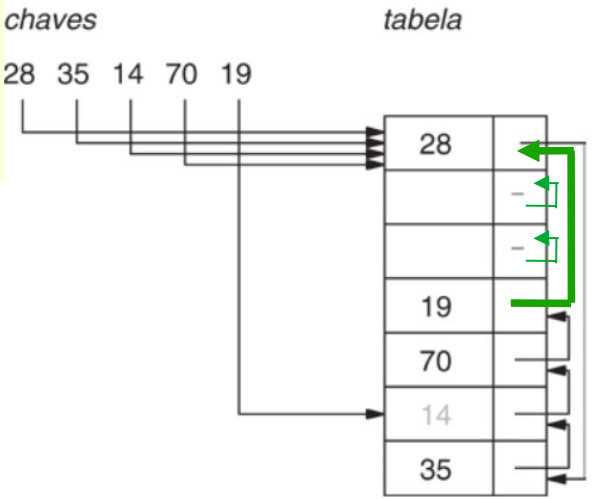
Busca (T, x, end, a): **a** indica presença/ausência da chave; **end** auxiliará a inserção

/* se a = 1 então chave encontrada no slot end ; se a = 2 a chave não está na tabela e end tem 2 possibilidades:
1) end = -1 => lista h(x) está vazia ou completa; 2) end = j ≥ 0 => slot j é da lista h(x) e está desocupado */

```
a ← 0; end ← h(x); j ← -1;   /* a = 0 significa que ainda não sei */
enquanto a = 0
   se T[end].estado = não ocupado
        j ← end
   se T[end].chave = x e T[end].estado = ocupado
        a ← 1   /* chave encontrada */
   senão
        end ← T[end].pont
        se end = h(x)    /* voltei para o início da lista circular */
              a ← 2; end ← j      /* chave não encontrada */
```

Ex: busca(20, end, a):
a = 0
end = 0 (h(x) = 0)
j = 5

T[end].estado = ocupado
T[end].chave = 28



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

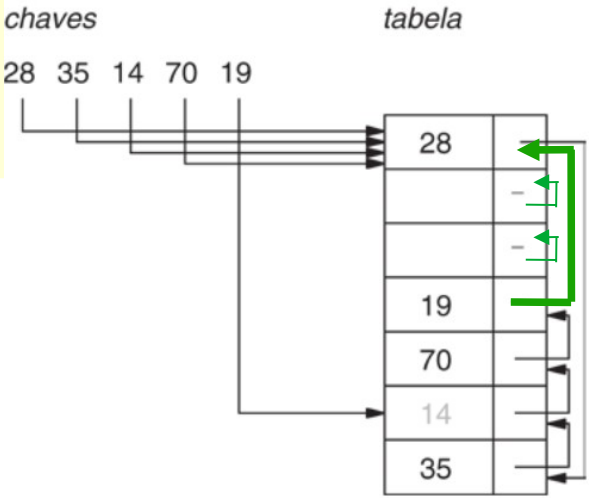
Busca (T, x, end, a): **a** indica presença/ausência da chave; **end** auxiliará a inserção

/* se a = 1 então chave encontrada no slot end ; se a = 2 a chave não está na tabela e end tem 2 possibilidades:
1) end = -1 => lista h(x) está vazia ou completa; 2) end = j ≥ 0 => slot j é da lista h(x) e está desocupado */

```
a ← 0; end ← h(x); j ← -1;   /* a = 0 significa que ainda não sei */
enquanto a = 0
   se T[end].estado = não ocupado
        j ← end
   se T[end].chave = x e T[end].estado = ocupado
        a ← 1   /* chave encontrada */
   senão
        end ← T[end].pont
        se end = h(x)    /* voltei para o início da lista circular */
              a ← 2; end ← j      /* chave não encontrada */
```

Ex: busca(20, end, a):
a = 2
end = 5 (h(x) = 0)
j = 5

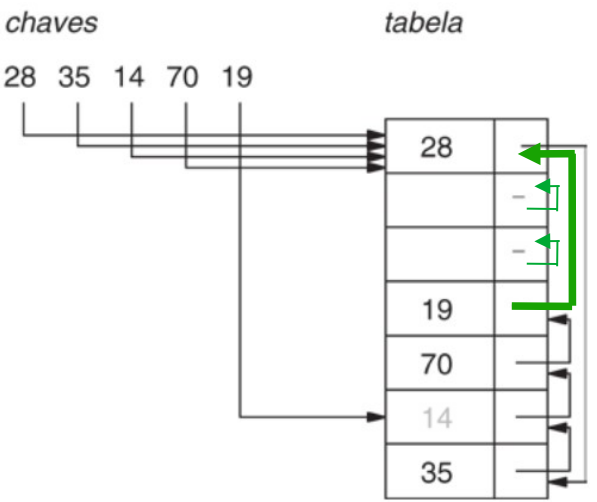
T[end].estado = ocupado
T[end].chave = 28



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Insere(T, x): **a** indica presença/ausência da chave; **end** auxiliará a inserção. Vou achar posição **j** para inserir

```
Busca(T, x, end, a)
Se a ≠ 1  /* ou seja, x não está na tabela */
  Se end ≠ -1   /* já sei onde vou inserir, no meio de uma lista */
     j ← end
  Senão  /* se end = -1 não sei ainda se a lista está vazia ou completa (sem slots desocupados) */
     i ← 1;  j ← h(x); /* i vai controlar até quando procurar um slot  (até no máximo olhar todos os m slots) */
     Enquanto i ≤ m
        se T[j].estado = ocupado
           j ← (j + 1) mod m  /* procuro na próxima posição; quando chegar no fim da tabela volto para o início */
           i ← i + 1
        Senão i ← m + 2 /* para sair do loop, equivalente a um break ou last */
     Se i = m + 1
          PARE / OVERFLOW – TABELA CHEIA
      Temp ← T[j].pont    /* aqui j = h(x) (se a lista vazia) ou j = uma posição vazia qualquer (de qualquer lista) */
      T[j].pont ← T[h(x)].pont   /* vou inserir entre o primeiro e segundo elemento da lista */
      T[h(x)].pont ← temp
  T[j].chave ← x
  T[j].estado ← ocupado
Senão PARE / CHAVE JÁ EXISTENTE
```



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Insere(T, x): **a** indica presença/ausência da chave; **end** auxiliará a inserção. Vou achar posição **j** para inserir

Busca(T, x, end, a)

Se a ≠ 1 */* ou seja, x não está na tabela */*

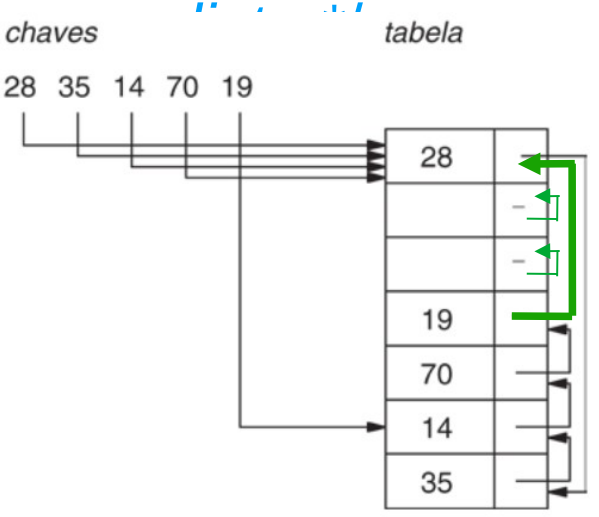
Se end ≠ -1 */* já sei onde vou inserir, no meio d

j ← end

Senão */* se end = -1 não s esta
(sem slots desocupados)

i ← 1; j ← h(x); */* i vai controlar até quando p
(até no máximo olhar todos os m slots) */*

Ex: Insere (T, 105)
m = 7
x = 105, h(x) = 0
end = -1 (nesse caso lista cheia)
a = 2
i = 1
j = 0
T[j].estado = ocupado



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Insere(T, x): **a** indica presença/ausência da chave; **end** auxiliará a inserção. Vou achar posição **j** para inserir

Busca(T, x, end, a)

Se a ≠ 1 */* ou seja, x não está na tabela */*

Se end ≠ -1 */* já sei onde vou inserir, no meio d

j ← end

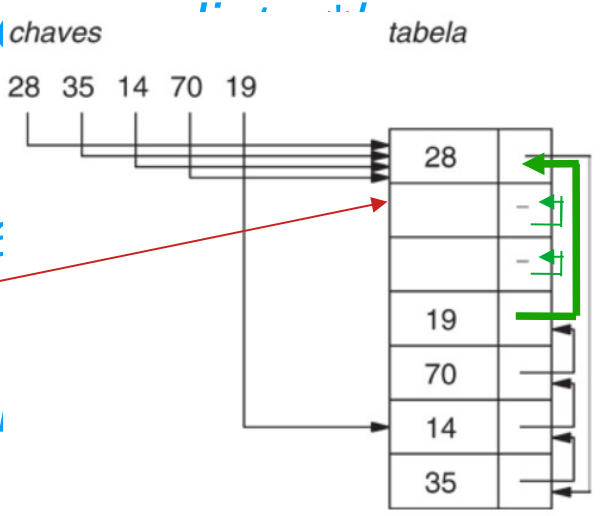
Senão */* se end = -1 não

(sem slots desocupados)

i ← 1; j ← h(x); */* i vai controlar até quando p

*(até no máximo olhar todos os m slots) */*

Ex: Insere (T, 105)
m = 7
x = 105, h(x) = 0
end = -1 (nesse caso lista cheia)
a = 2
i = 2
j = 1
T[j].estado = não ocupado



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Insere(T, x): **a** indica presença/ausência da chave; **end** auxiliará a inserção. Vou achar posição **j** para inserir

Busca(T, x, end, a)

Se a ≠ 1 */* ou seja, x não está na tabela */*

Se end ≠ -1 */* já sei onde vou inserir, no meio d

j ← end

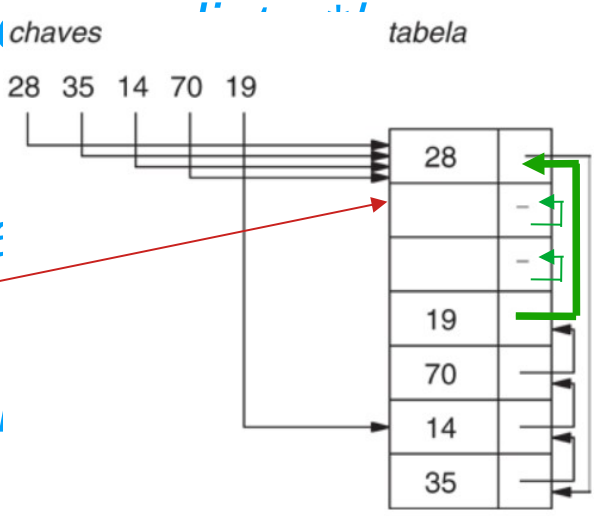
Senão */* se end = -1 não s esta

(sem slots desocupados)

i ← 1; j ← h(x); */* i vai c do p

*(até no máximo olhar todos os m slots) */*

Ex: Insere (T, 105)
m = 7
x = 105, h(x) = 0
end = -1 (nesse caso lista cheia)
a = 2
i = 9
j = 1
T[j].estado = não ocupado
T[j].pont = 1
Temp = 1



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Insere(T, x): **a** indica presença/ausência da chave; **end** auxiliará a inserção. Vou achar posição **j** para inserir

Busca(T, x, end, a)

Se a ≠ 1 */* ou seja, x não está na tabela */*

Se end ≠ -1 */* já sei onde vou inserir, no meio d

j ← end

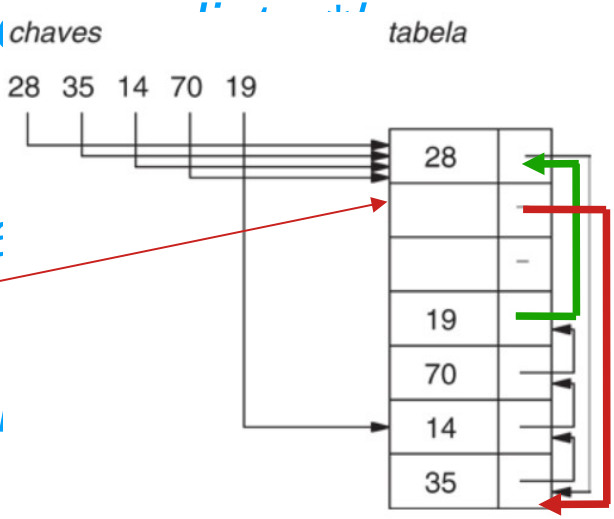
Senão */* se end = -1 não

(sem slots desocupados)

i ← 1; j ← h(x); */* i vai

*(até no máximo olhar todos os m slots) */*

Ex: Insere (T, 105)
m = 7
x = 105, h(x) = 0
end = -1 (nesse caso lista cheia)
a = 2
i = 9
j = 1
T[j].estado = não ocupado
T[j].pont = 6
Temp = 1



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Insere(T, x): **a** indica presença/ausência da chave; **end** auxiliará a inserção. Vou achar posição **j** para inserir

Busca(T, x, end, a)

Se a ≠ 1 */* ou seja, x não está na tabela */*

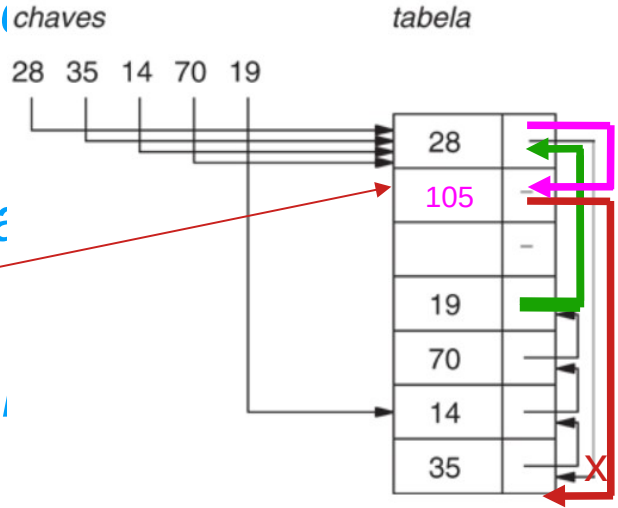
Se end ≠ -1 */* já sei onde vou inserir, no meio d

j ← end

Senão */* se end = -1 não s esta
(sem slots desocupados)

i ← 1; j ← h(x); */* i vai d do p
(até no máximo olhar todo

Ex: Insere (T, 105)
m = 7
x = 105, h(x) = 0
end = -1 (nesse caso lista cheia)
a = 2
i = 9
j = 1
T[j].estado = não ocupado
T[j].pont = 6
Temp = 1
T[h(x)=0].pont ← 1
T[j].chave = 105



Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Remove(T, x, end, a):

Busca(T, x, end, a) /* **a** indica presença/ausência da chave; **end** indica em que slot está a chave*/

Se a = 1 /* chave existente no slot **end** */

T[end].estado ← não ocupado

Senão

CHAVE NÃO EXISTENTE

Tratamento de colisões - 1) Endereçamento fechado
1.2) Encadeamento interior – opção 2

Complexidades:

Busca / Remove / Insere: O(n) no pior caso

Tratamento de colisões

Estratégias:

A) Hashing estático (tamanho da tabela é constante)

1) Encadeamento ou endereçamento fechado – colisões vão para uma lista ligada

1.1) Encadeamento exterior (fora da tabela)

1.2) Encadeamento interior (dentro da tabela)

2) Endereçamento aberto (chaves dentro da tabela, sem ponteiros)

2.1) Tentativa/Sondagem linear

2.2) Tentativa/Sondagem quadrática

2.3) Dispersão dupla / Hash duplo

B) Hashing dinâmico (tabela pode expandir/encolher)

3) Hashing extensível (estrutura de dados adicional)

4) Hashing linear

Referências

Conceitos gerais de Hashing:

SZWARCFITER, J. L.; MARKENZON, L. Estruturas de Dados e Seus Algoritmos. Ed. LTC, 3ª ed, 2013. Capítulo 10 (figuras do livro)

Slides dos Profs. M. Chaim, Delano Beder e L. Digiampietri