MÉTODOS DE BUSCA: HASHING

SCC0601 – Introdução à Ciência da Computação II

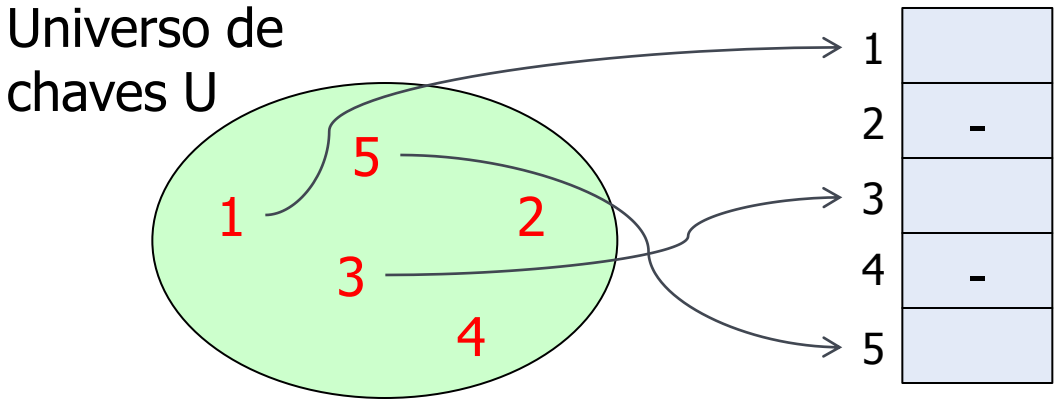
Prof. Lucas Antiqueira

INTRODUÇÃO

- Acesso sequencial = $O(n)$
- Busca binária = $O(\log_2 n)$
- Árvores binárias de busca = $O(\log_2 n)$
  (melhor caso)

INTRODUÇÃO

- Acesso em tempo constante
  - Por exemplo, endereçamento direto em um arranjo
    - Cada chave *k* é mapeada na posição *k* do arranjo
    - Função de mapeamento $f(k) = k$



INTRODUÇÃO

- Endereçamento direto
  - Vantagens
    - Acesso direto e, portanto, rápido
      - Via indexação do arranjo
  - Desvantagens
    - Uso ineficiente do espaço de armazenamento
      - Declara-se um arranjo do tamanho da maior chave?
      - E se as chaves não forem contínuas? Por exemplo, {1, 52 e 100}
      - Pode sobrar espaço? Pode faltar?

INTRODUÇÃO

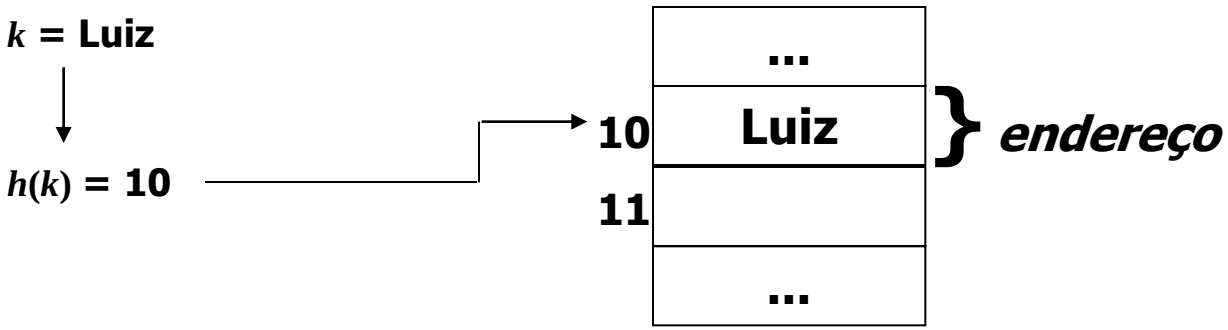
- *Hashing*
  - Acesso direto, mas endereçamento indireto
    - Função de dispersão $h(k) \neq k$, em geral
    - Resolve uso ineficiente do espaço de armazenamento
  - O(1), no melhor caso, independente do tamanho do arranjo

HASHING

- Também conhecido como tabela de *espalhamento* ou de *dispersão*
- *Hashing* é uma técnica que utiliza uma função *h* para transformar uma chave *k* em um endereço
  - O endereço é usado para gravar e buscar registros
- Idéia: particionar um conjunto de elementos (possivelmente infinito) em um número finito de classes
  - *m* classes → endereçamento de 0 a *m*-1

HASHING

- A função *h* é chamada função *hash*
- *h*(*k*) retorna o valor *hash* de *k*
  - Usado como endereço para armazenar a informação cuja chave é *k*
- *k* tem endereço *h*(*k*)

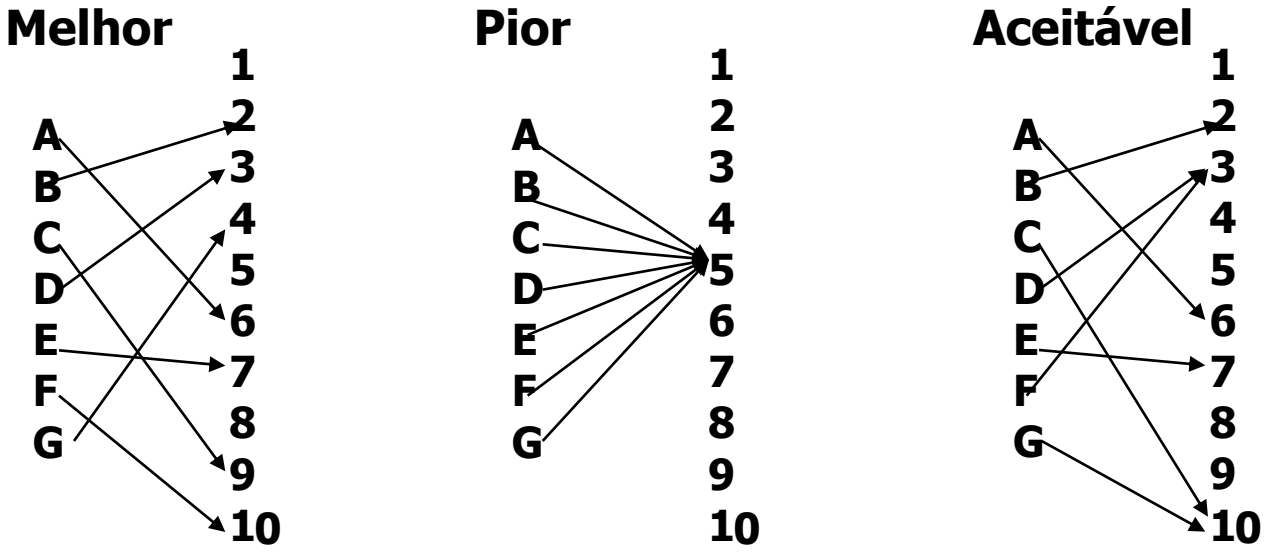


HASHING

- A função hash é utilizada para inserir ou buscar um elemento
  - Deve ser determinística, ou seja, resultar sempre no mesmo valor para uma determinada chave

HASHING

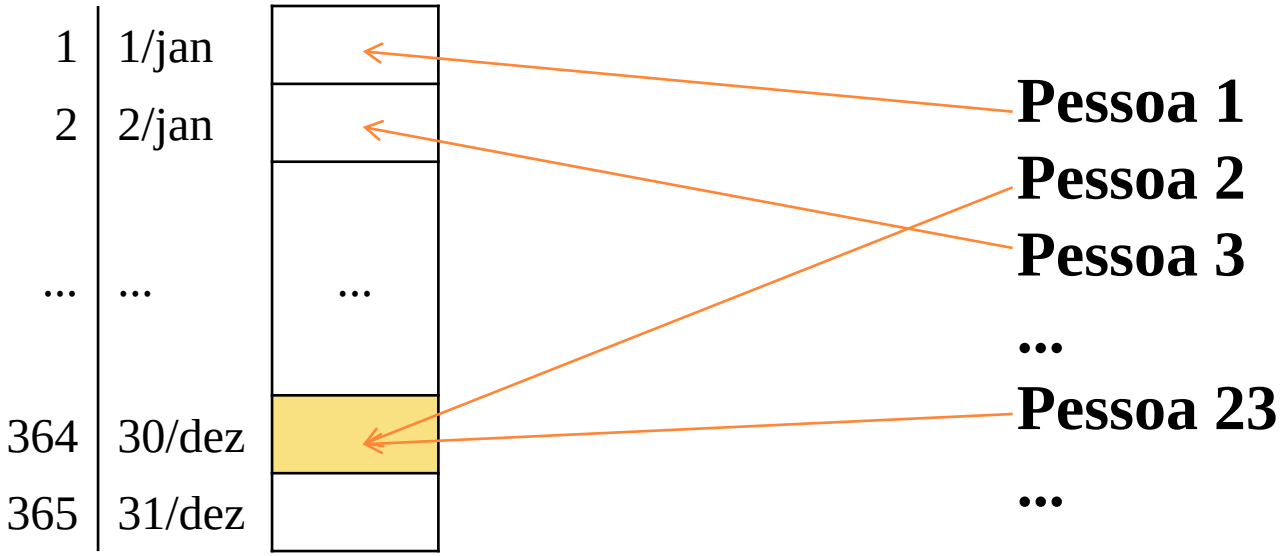
- Colisão: ocorre quando a função *hash* produz o mesmo endereço para chaves diferentes
  - As chaves com mesmo endereço são ditas **sinônimos**



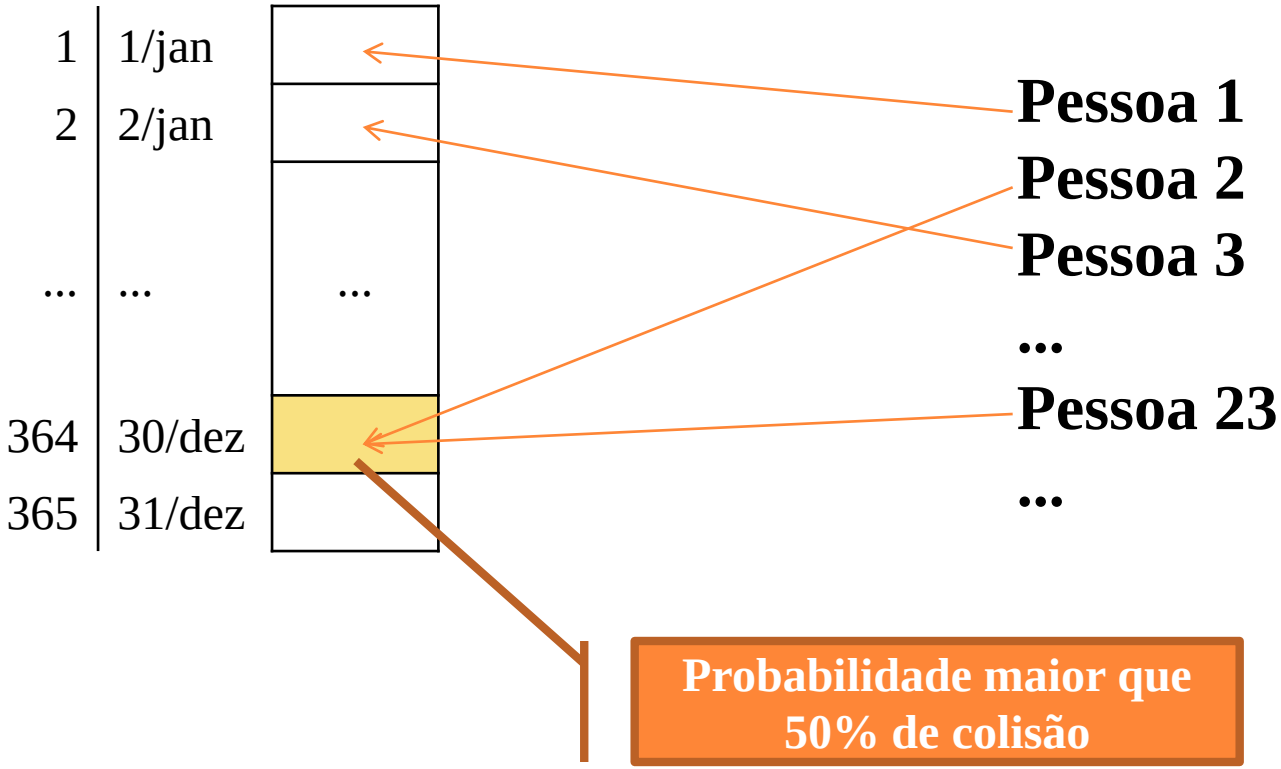
HASHING

- *Paradoxo do aniversário:*
  - Em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que 2 pessoas comemorem aniversário no mesmo dia.

Função *hashing* nesse caso é:
h(fulano) = data de aniversário de fulano



Função *hashing* nesse caso é:
h(fulano) = data de aniversário de “fulano”



HASHING

- Colisões irão fatalmente ocorrer

HASHING

- Colisões irão fatalmente ocorrer
- Probabilidade de ocorrer colisão ao se inserir n itens aleatórios em uma tabela de tamanho m:

$$p = 1 - \frac{m!}{(m-n)!\,m^n}$$

HASHING

- Colisões irão fatalmente ocorrer
- Probabilidade de ocorrer colisão ao se inserir *n* itens aleatórios em uma tabela de tamanho *m*:

$$p = 1 - \frac{m!}{(m-n)!\,m^n}$$

Para m=365

n	*p*
10	0,117
22	0,476
23	0,507
30	0,697

HASHING

- Uma boa técnica de *hashing* apresenta:
  - Uma *função hash*, que
    1. Minimiza colisões
    2. Distribui uniformemente os dados
    3. É fácil de implementar
    4. É rápida de calcular
  - Uma estratégia eficiente para *tratamento de colisões*

HASHING

- Função *hash*: Técnica simples e muito utilizada que produz bons resultados
  - Para chaves inteiras, calcular o resto da divisão *k*/*m*, sendo que o resto indica a posição de armazenamento
    - *k* = valor da chave
    - *m* = tamanho do espaço de endereçamento
  - Para chaves tipo *string*, tratar cada caracter como um valor inteiro (ASCII), somá-los, e então pegar o resto da divisão por *m*

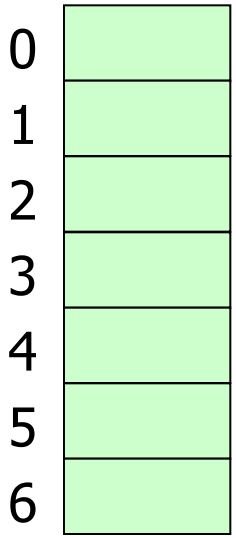
HASHING

- O tamanho do espaço de endereçamento *m* costuma apresentar bons resultados quando:
  - É um número primo não próximo a uma potência de 2

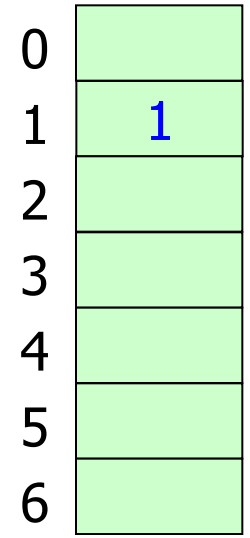
  ou

  - Não possui divisores primos menores que 20

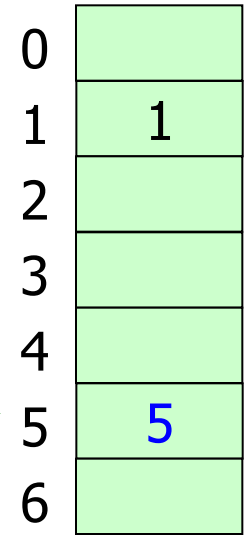
- Exemplo
  - Seja $m = 7$
    - Inserção dos números 1, 5, 10, 20, 25, 24



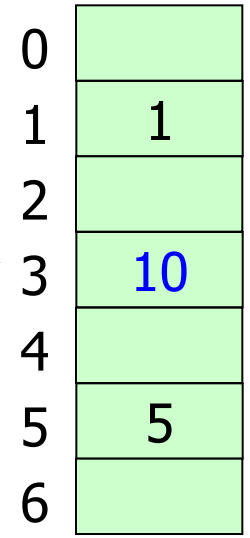
- Exemplo
  - Seja $m = 7$
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 1 % 7 = 1



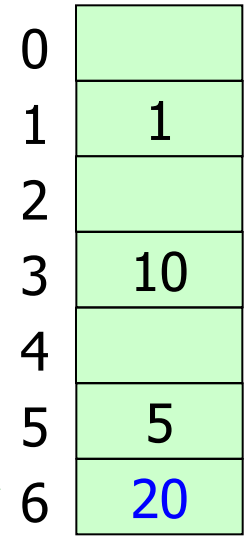
- Exemplo
  - Seja $m = 7$
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 5 % 7 = 5



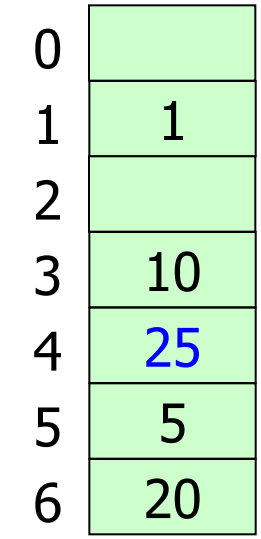
- Exemplo
  - Seja $m = 7$
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 10 % 7 = 3



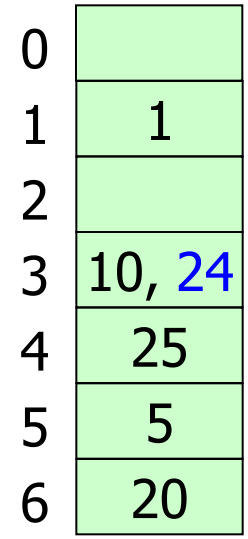
- Exemplo
  - Seja $m = 7$
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 20 % 7 = 6



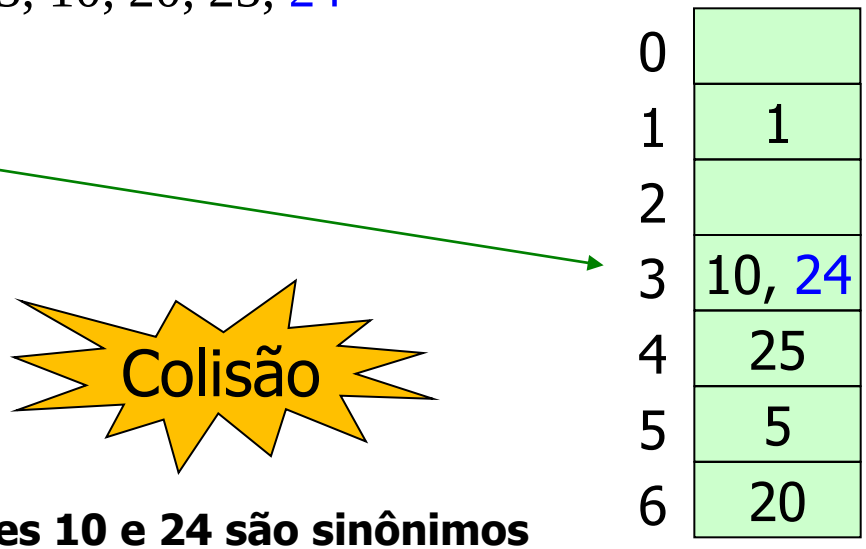
- Exemplo
  - Seja $m = 7$
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 25 % 7 = 4



- Exemplo
  - Seja $m = 7$
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 24 % 7 = 3



- Exemplo
  - Seja $m = 7$
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 24 % 7 = 3



Chaves 10 e 24 são sinônimos

- Exemplo com string
  - Seja $m = 13$
    - LOWEL = 76 79 87 69 76 (Códigos dos caracteres)

      L+O+W+E+L = 387

      76+79+87+69+76 = 387
    - h(LOWEL) = 387 % 13 = 10

HASHING

- Qual a idéia por trás da função hash que usa o resto?

HASHING

- Qual a idéia por trás da função hash que usa o resto?
  - Os elementos sempre caem no intervalo $[0, m-1]$

HASHING

- Qual a idéia por trás da função hash que usa o resto?
  - Os elementos sempre caem no intervalo [0, *m*-1]
- Outras funções hash?

HASHING

- Qual a idéia por trás da função hash que usa o resto?
  - Os elementos sempre caem no intervalo $[0, m-1]$
- Outras funções hash?
- Como tratar colisões?

OUTRAS FUNÇÕES HASH

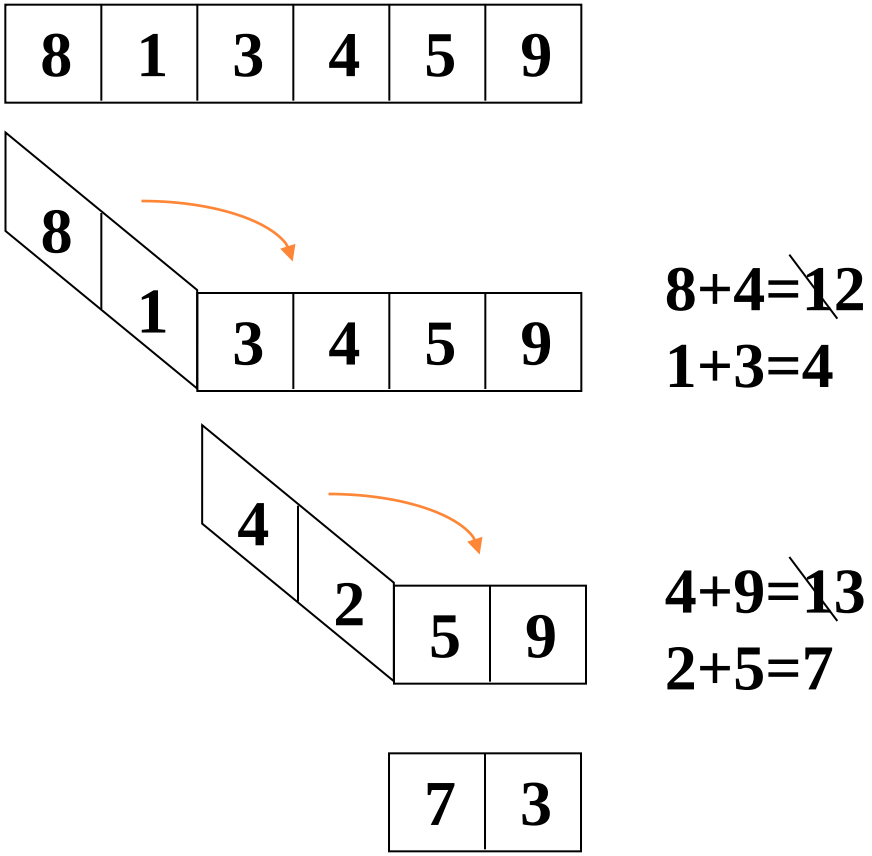
FUNÇÕES HASH

- Multiplicação
  - $h(k) = (k * A \% 1) * m$, onde A é uma constante entre 0 e 1
    - $(k*A \% 1)$ recupera a parte fracionária de $k*A$
    - Knuth sugere $A = (\sqrt{5} - 1)/2 \sim 0{,}6180$

FUNÇÕES HASH

- Método da Dobra I
  - Suponha a chave como uma seqüência de dígitos escritos num pedaço de papel.
  - O método em questão consiste basicamente em “dobrar” esse papel, de maneira os dígitos se sobreponham.
  - Estes devem então somados, sem levar em consideração o “vai um”.
  - O processo é repetido mediante a realização de novas dobras.
  - O resultado final deve conter o número de dígitos desejados para formar o endereço-base.

Método da Dobra I



FUNÇÕES HASH

- Método da Dobra II

  - Uma maneira de obter um endereço-base de *k* bits para uma chave qualquer é separar a chave em diversos campos de *k* bits e operá-los logicamente com uma operação binaria conveniente.
  - Em geral, a operação de *ou exclusivo* (XOR) produz resultados melhores do que as operações *e* ou *ou*. Isto porque o *e* de dois operandos produz um numero binário sempre menor (ou igual) do que ambos, e o *ou*, sempre maior (ou igual).

FUNÇÕES HASH

- Método da Dobra II

  - Exemplo: considere a dimensão da tabela igual a 32 (2^5). Supondo-se que cada chave ocupa 10 bits, deve-se transformar esse valor em um endereço ocupando 5 bits. Utilizando-se a operação XOR, tem-se:

    71 = 00010 00111

    endereço-base: 00010 XOR 00111 = 00101 = 5

    46 = 00001 01110

    endereço-base: 00001 XOR 01110 = 01111 = 15

FUNÇÕES HASH

- Função Meio-Quadrado
  - A chave é elevada ao quadrado e parte do resultado é usada como endereço.
  - Exemplo: $k = 3121$

    $3121^2 = 9740641$

FUNÇÕES HASH

- Função Meio-Quadrado
  - A chave é elevada ao quadrado e parte do resultado é usada como endereço.
  - Exemplo: $k = 3121$

    $3121^2 = 9740641$

    Em uma tabela com m=1000, pode-se utilizar os 3 dígitos do meio → $h(3121) = 406$

FUNÇÕES HASH

- Transformação de Raiz
  - A chave *k* é transformada em um número equivalente em outra base, ou seja, é expressada em um sistema numérico que usa outra raiz.
  - A seguir, aplica-se o método da divisão no número obtido.

FUNÇÕES HASH

- Transformação de Raiz
  - A chave *k* é transformada em um número equivalente em outra base, ou seja, é expressada em um sistema numérico que usa outra raiz.
  - A seguir, aplica-se o método da divisão no número obtido.
  - Exemplo: $k = 345$ e $m=100$

    Na base 9, *k* é 423

    $h(345) = 423 \% m = 23$

FUNÇÕES HASH

- Dispersão Dupla
  - O método da dispersão dupla calcula a sequência de tentativas de acordo com a seguinte equação:

    $h(k) = (h(k)' + a * h(k)'') \% m$
    onde a varia entre 0 e m-1

FUNÇÕES HASH

- Dispersão Dupla

  - O método da dispersão dupla calcula a sequência de tentativas de acordo com a seguinte equação:

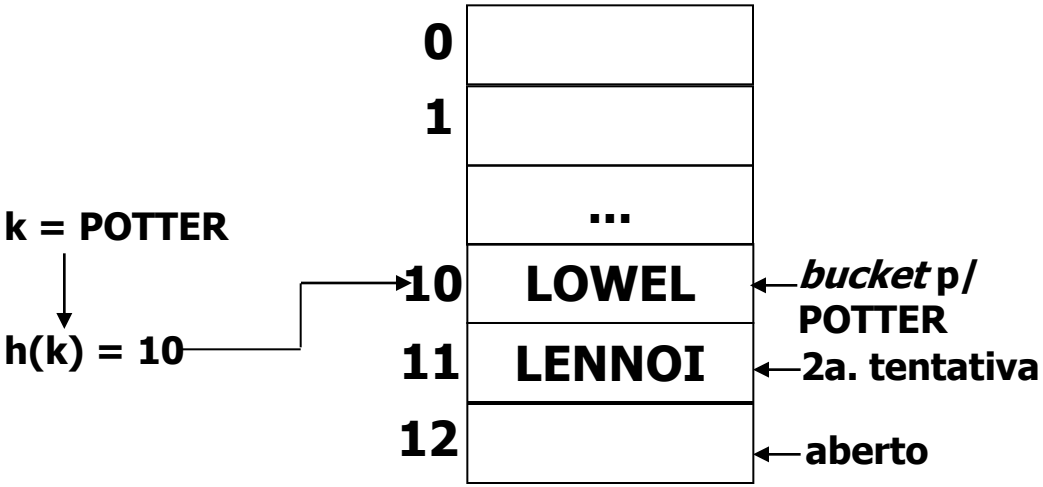
    $h(k) = (h(k)' + a * h(k)'') \% m$
    onde a varia entre 0 e m-1

  - Exemplo:
    $h(k)' = k \% m$
    $h(k)'' = 1 + k \% (m-2)$

TRATAMENTO DE COLISÕES

OVERFLOW PROGRESSIVO

- $h(k)' = (h(k) + i) \% m$, com i variando de 1 a m-1 (i é incrementado a cada tentativa)



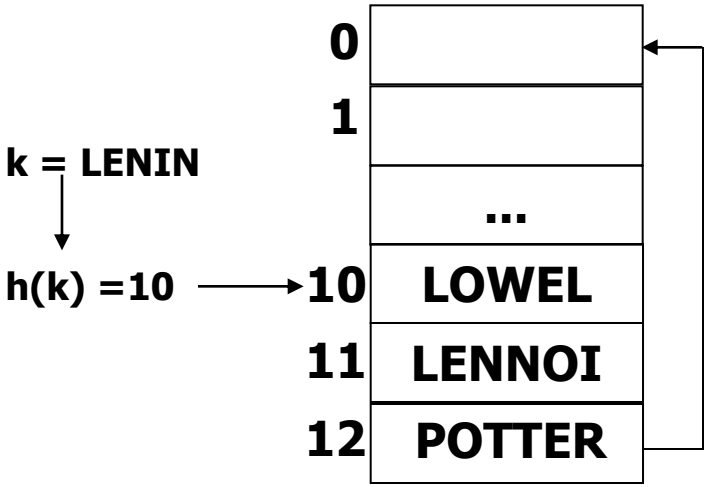
OVERFLOW PROGRESSIVO

- $h(k)' = (h(k) + i) \% m$, com i variando de 1 a m-1 (i é incrementado a cada tentativa)

0	
1	
	...
10	LOWEL
11	LENNOI
12	POTTER

OVERFLOW PROGRESSIVO

- $h(k)' = (h(k) + i) \% m$, com i variando de 1 a m-1 (i é incrementado a cada tentativa)



OVERFLOW PROGRESSIVO

- $h(k)' = (h(k) + i) \% m$, com i variando de 1 a m-1 (i é incrementado a cada tentativa)

Como saber que a informação procurada não está armazenada?

OVERFLOW PROGRESSIVO

- Exemplo de dificuldade: busca pelo nome "Smith"

h(SMITH) = 7

	...
7	**ADAMS**
8	**JONES**
9	**MORRIS**
10	**SMITH**

Pode ter que percorrer muitos campos

OVERFLOW PROGRESSIVO

- Exemplo de dificuldade: busca pelo nome "Smith"

h(SMITH) = 7

	...
7	**ADAMS**
8	**JONES**
9	
10	**SMITH**

A remoção do elemento no índice 9 pode causar uma falha na busca

OVERFLOW PROGRESSIVO

- Exemplo de dificuldade: busca pelo nome "Smith"

h(SMITH) = 7

	...
7	**ADAMS**
8	**JONES**
9	**#####**
10	**SMITH**

Solução para remoção de elementos: eliminar elemento, mas indicar que a posição foi esvaziada e que a busca deve continuar

EXERCÍCIO

- Assumindo que:
  - $m = 10$
  - $h(k) = k \% m$
  - Colisões são tratadas com sondagem linear
    $h(k)' = (h(k)+i) \% m$, com $i = 1, \ldots, m-1$

Insira os seguintes elementos em uma tabela *hash* com *overflow* progressivo

41, 10, 8, 7, 13, 52, 1, 89, 64

HASHING

- Quando deseja-se armazenar *n* elementos em uma tabela de *m* posições, tem-se o chamado fator de carga:

$$\alpha = n/m$$

OVERFLOW PROGRESSIVO

- Seja α o fator de carga de uma tabela *hash*. Conforme demonstrado por Knuth, o custo de uma pesquisa com sucesso pela sondagem linear em *overflow* progressivo é:

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

α	C
0,10	1,06
0,25	1,17
0,50	1,50
0,75	2,50
0,90	5,50
0,95	10,50

OVERFLOW PROGRESSIVO

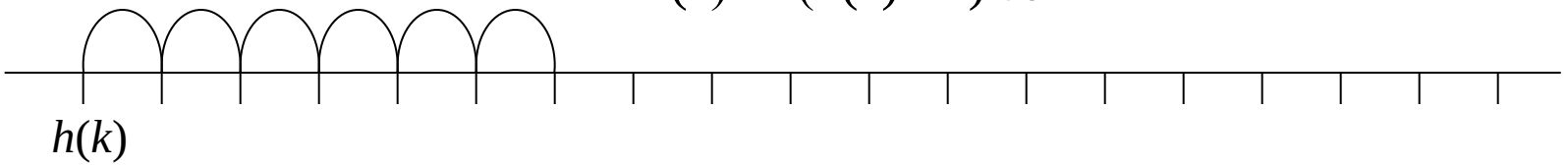
- Exemplo anterior
  - $h(k)' = (h(k) + i)\ \%\ m$, com $i=1, \ldots, m-1$
    - Chamada sondagem linear, pois todas as posições da tabela são checadas, no pior caso

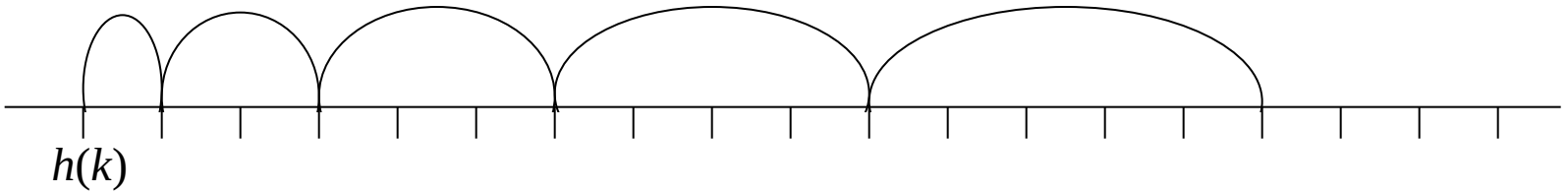
- Outro exemplo
  - $h(k)' = (h(k) + c_1 * i + c_2 * i^2)\ \%\ m$, com $i=1, \ldots, m-1$ e constantes c_1 e c_2
    - Chamada sondagem quadrática, considerada melhor do que a linear, pois evita o agrupamento de elementos

OVERFLOW PROGRESSIVO

Sondagem linear

$h(k)' = (h(k) + i) \% m$





Sondagem quadrática

$h(k)' = (h(k) + 0{,}5*i + 0{,}5*i^2) \% m$

c_1 c_2

Específicos para este exemplo

OVERFLOW PROGRESSIVO

- Vantagem
  - Simplicidade
- Desvantagens
  - Agrupamento de dados (causado por colisões)
  - Com estrutura cheia, a busca fica lenta
  - Dificulta inserções e remoções

OVERFLOW PROGRESSIVO

- Alternativamente, podemos representar a sondagem linear como uma única função dependente do número da tentativa *i*:
  - Por exemplo: $h(k, i) = (k+i) \% m$, com $i = 0, \ldots, m-1$
    - A função *h* depende agora de dois fatores: a chave *k* e a iteração *i*
    - Note que *i*=0 na primeira execução, resultando na função *hash* tradicional de divisão que já conhecíamos
    - Quando *i*=1,... , *m*-1, já estamos aplicando a técnica de sondagem linear

OVERFLOW PROGRESSIVO

Exercício: implemente sub-rotinas de inserção, busca e remoção utilizando a função *hash* anterior

Considere que a tabela *hash* é inicializada com o valor -1 em todos os endereços:

```
void inicializa(int T[], int m) {
   int i;
   for (i = 0; i < m; i++)
     T[i] = -1;
}
```

Considere também que, ao remover um elemento, seu endereço recebe o valor -2.

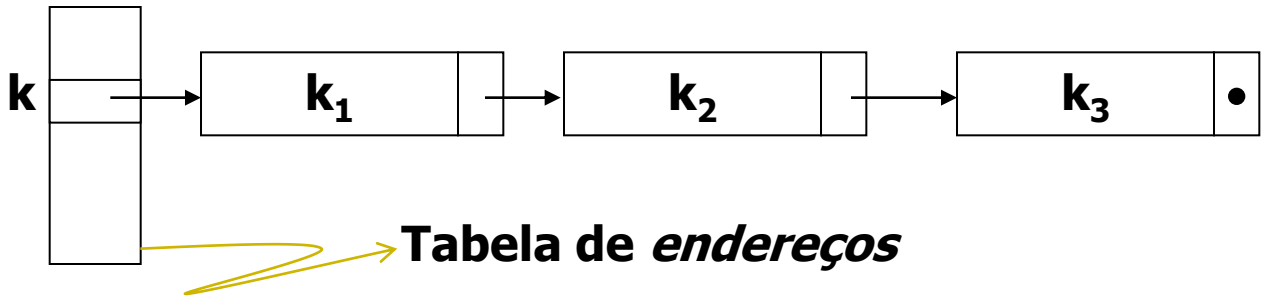
```
int inserir(int T[], int m, int k) {
  int i, j;
  for (i = 0; i < m; i++) {
       j = (k + i) % m;
       if ((T[j] == -1) || (T[j] == -2)) {
              T[j] = k;
              return j;
       }
  }
  return -1;
}
```

```
int buscar (int T[], int m, int k) {
  int i, j;
  for (i = 0; i < m; i++) {
       j = (k + i) % m;
       if (T[j] == k)
             return j;
       else if (T[j] == -1)
             return -1;
  }
  return -1;
}
```

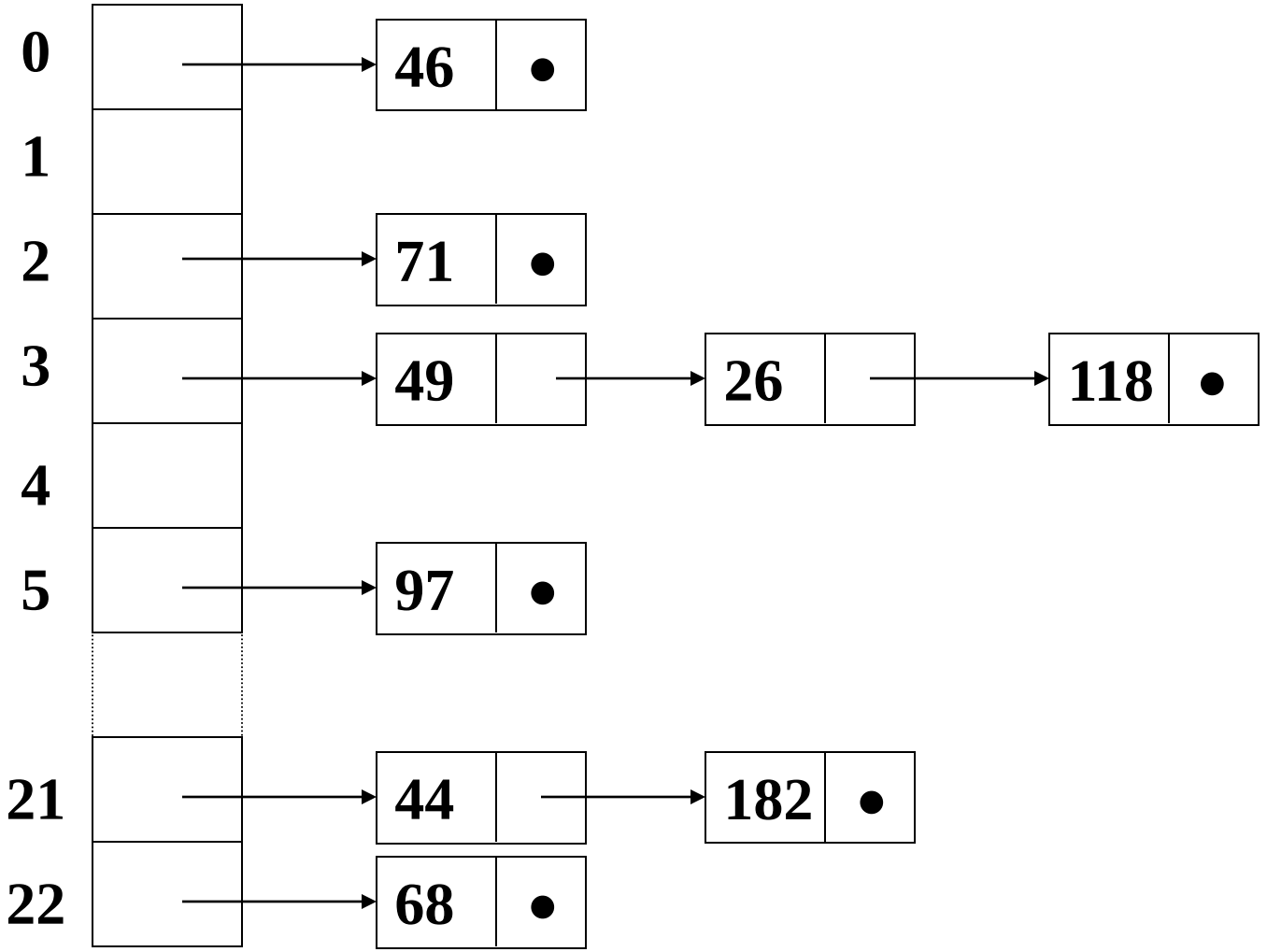
```
int remover(int T[], int m, int k) {
  int i, j;
  for (i = 0; i < m; i++) {
       j = (k + i) % m;
       if (T[j] == k) {
              T[j] = -2;
              return j;
       } else if (T[j] == -1)
              return -1;
  }
  return -1;
}
```

HASHING COM ENCADEAMENTO

- *Hashing* com encadeamento externo
  - A tabela de *endereços*, indo de 0 a *m*–1, contém apenas ponteiros para uma lista de elementos
  - Quando há colisão, o sinônimo é inserido no *endereço* como um novo nó da lista
  - Busca deve percorrer a lista

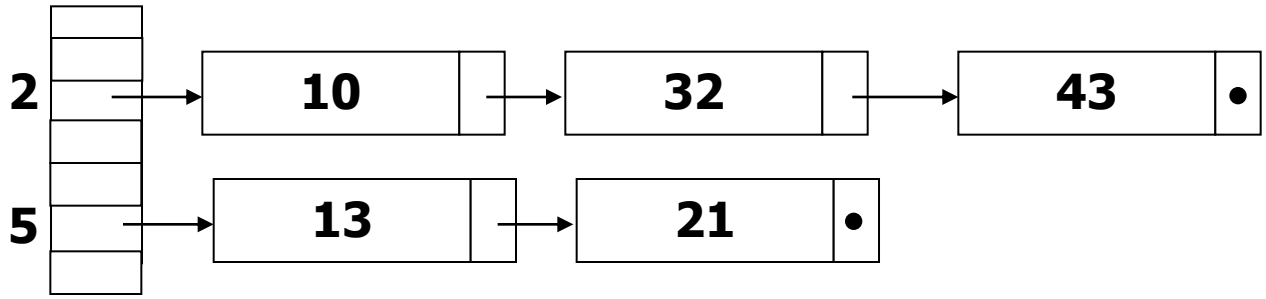


Exemplo: $h(k) = k \% 23$



HASHING COM ENCADEAMENTO

- *Hashing* com encadeamento externo
  - Se as listas estiverem ordenadas, reduz-se o tempo de busca
    - Dificuldade deste método?



HASHING COM ENCADEAMENTO

- Encadeamento externo: Vantagens
  - A tabela pode receber mais itens mesmo quando um endereço já foi ocupado
  - Permite percorrer a tabela por ordem de valor *hash*

- Encadeamento externo: Desvantagens
  - Espaço extra para as listas
  - Listas longas implicam em muito tempo gasto na busca
    - Se as listas estiverem ordenadas, reduz-se o tempo de busca
    - Custo extra com a ordenação ou inserção ordenada

HASHING COM ENCADEAMENTO

- *Hashing* com encadeamento interno
  - O encadeamento interno prevê a divisão da tabela *T* em duas zonas, uma de endereço-base, de tamanho *p*, e outra reservada aos sinônimos, de tamanho *s*.
  - Naturalmente, $p+s = m$.
  - Os valores *p* e *s* são fixos.

HASHING COM ENCADEAMENTO

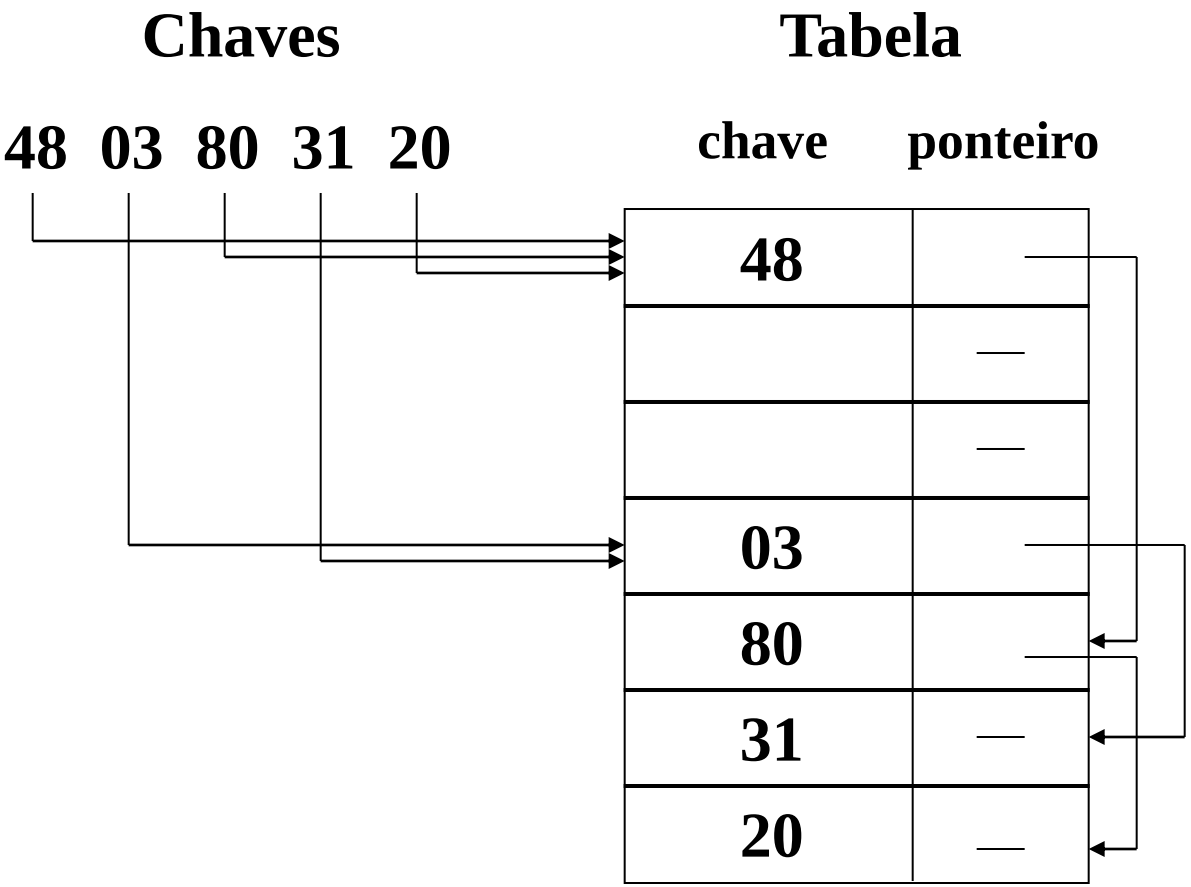
- *Hashing* com encadeamento interno
  - Dois campos tem presença obrigatória em cada nó. O primeiro é reservado ao armazenamento da chave, enquanto o segundo contem um ponteiro que indica o próximo elemento da lista de sinônimos (correspondente ao endereço-base em questão).

Exemplo: $m = 7$ (tamanho da tabela)

$p = 4$, $s = 3$

$h(x) = x \% p$

Divisão não é por *m*



HASHING COM ENCADEAMENTO

- *Hashing* com encadeamento interno

- Em principio, não se pode simplesmente remover uma chave *k* de uma lista encadeada interior, sem reorganizar a tabela, o que está fora de cogitação. Senão, pode-se perder os endereços dos elementos apontados diretamente e indiretamente pelo elemento sendo removido.

HASHING COM ENCADEAMENTO

Para contornar esse problema considera-se que cada compartimento da tabela pode estar em um dos três estados:

1. *Vazio* - pode ser utilizado para armazenar qualquer chave;
2. *Ocupado* - contem uma chave armazenada;
3. *Liberado* - está ocupado por alguma chave *k* cuja remoção foi solicitada.

- *Neste caso, o nó que contém k não deve ser removido da lista. Contudo, posteriormente, k pode ser substituído por alguma outra chave. Nessa ocasião, o compartimento torna-se, novamente, ocupado.*

Quando o tamanho do espaço de endereçamento pode aumentar, temos o

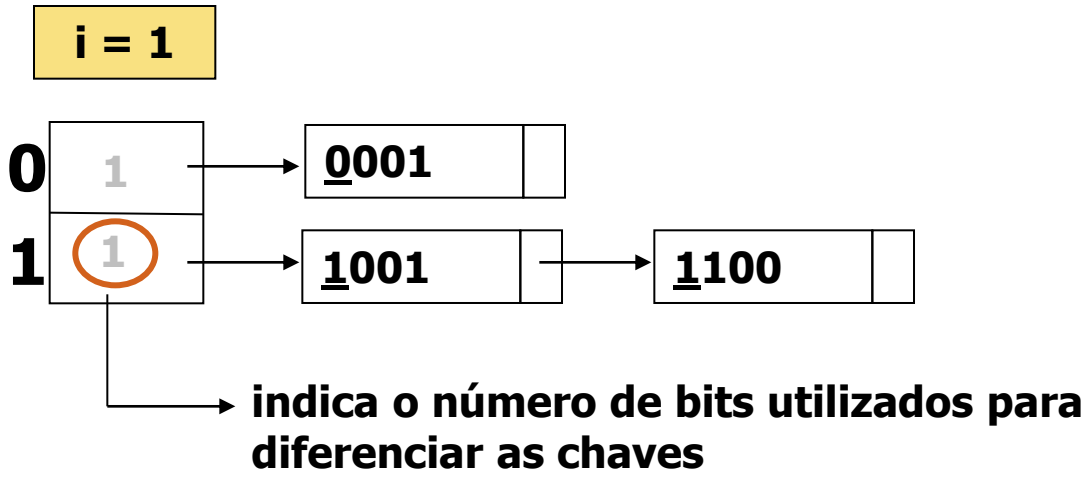
HASHING EXTENSÍVEL

HASHING EXTENSÍVEL

- Tamanho da tabela de *buckets* cresce sempre como potência de 2.
- Função *hash* computa seqüência de *m* bits para uma chave *k*. Porém, apenas os *i* bits, $i < m$, do início da seqüência são usados como endereço.
  - Se *i* é o número de bits usados, a tabela de *buckets* terá 2^i entradas.
- Tratamento de colisões: listas ordenadas.
- *N* é o número de nós permitidos por lista encadeada (externa).

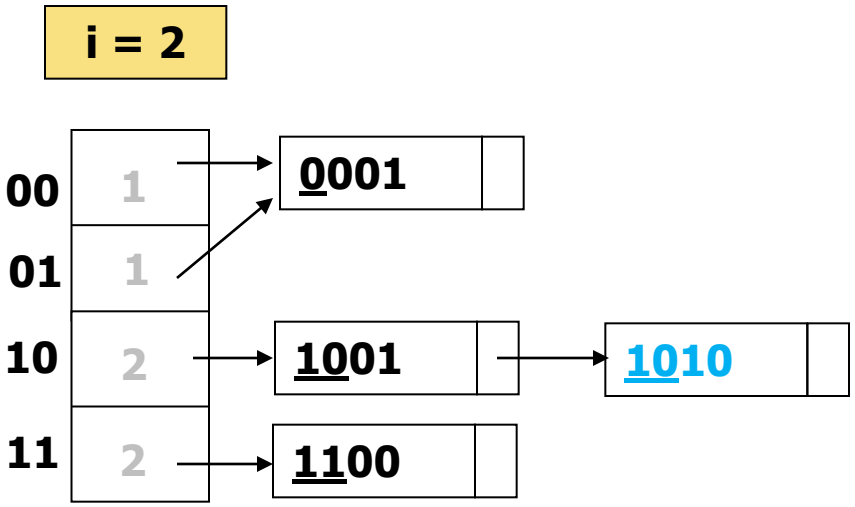
HASHING EXTENSÍVEL

- $m = 4, N = 2$



HASHING EXTENSÍVEL

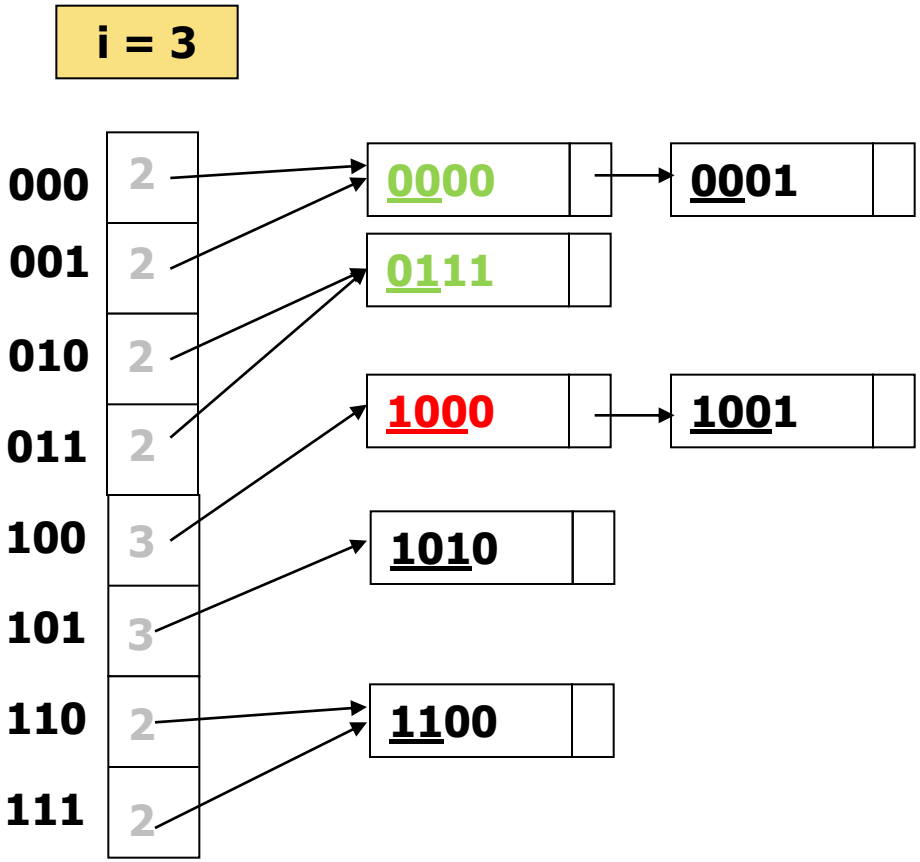
Inserindo: 1010



HASHING EXTENSÍVEL

- Exercício:
  - Insira 0000 e 0111
  - Depois insira 1000

HASHING EXTENSÍVEL



HASHING EXTENSÍVEL

- Vantagens:
  - Custo de acesso limitado por *N*.
  - A tabela pode crescer.

- Desvantagens:
  - Complexidade para gerenciar o aumento do espaço de endereços e a divisão das listas.
  - Podem existir sequências de inserções que façam a tabela crescer rapidamente tendo, contudo, um número pequeno de registros.

CONCLUSÕES

HASHING

- Pergunta
  - Quais são as principais desvantagens do *hashing*?

HASHING

- Pergunta
  - Quais são as principais desvantagens do *hashing*?
    - Os elementos da tabela não são armazenados sequencialmente
    - Não existe um método prático para percorrê-los em sequência

REVISANDO...

- Busca sequencial (com ou sem sentinela)
- Busca sequencial indexada
- Busca binária
- Busca por interpolação
- Busca em árvores binárias
- *Hashing*

MÉTODOS DE BUSCA

- Critérios para se eleger um (ou mais) método(s)?

MÉTODOS DE BUSCA

- Critérios para se eleger um (ou mais) método(s)
  - Eficiência da busca
  - Eficiência de outras operações
    - Inserção e remoção
    - Listagem e ordenação de elementos
  - Frequência das operações realizadas
  - Dificuldade de implementação
  - Consumo de memória (interna)

CRÉDITOS

Aula baseada nos materiais dos profs.
Rudinei Goularte, Thiago A. S. Pardo e Zhao Liang

E nos livros

- *M.J. Folk, B. Zoellick, G. Riccardi. File Structures. Addison-Wesley, 1998.*
- *N. Ziviani. Projeto de Algoritmos. Thomson, 2007.*
- *J.L. Szwarcfiter, L Markenzon. Estruturas de Dados e seus Algoritmos. LTC, 2010.*
- *A. Drozdek. Estrutura de Dados e Algoritmos em C++. Cengage Learning, 2002.*