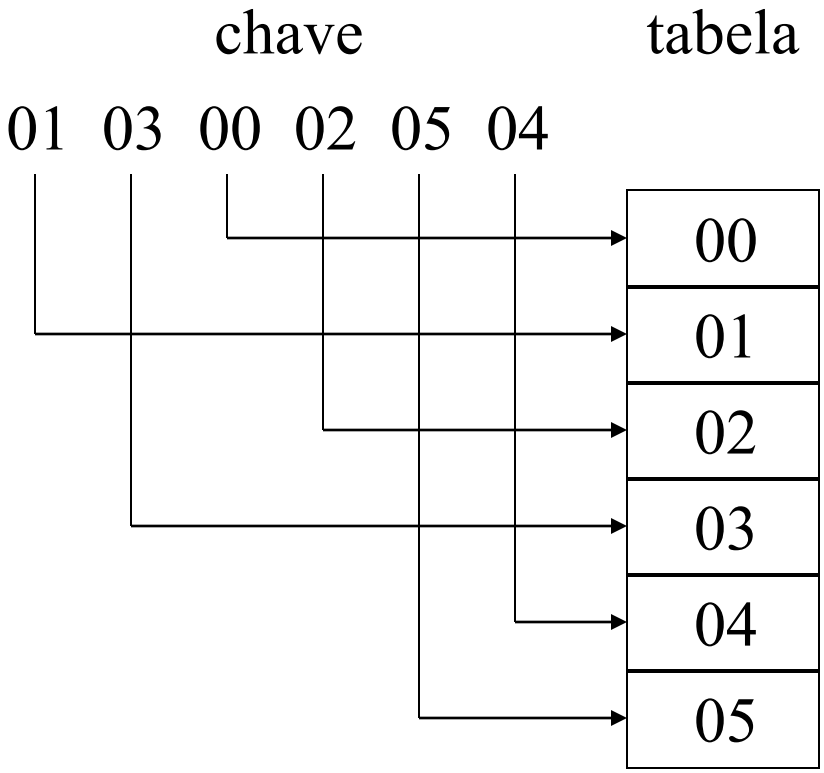
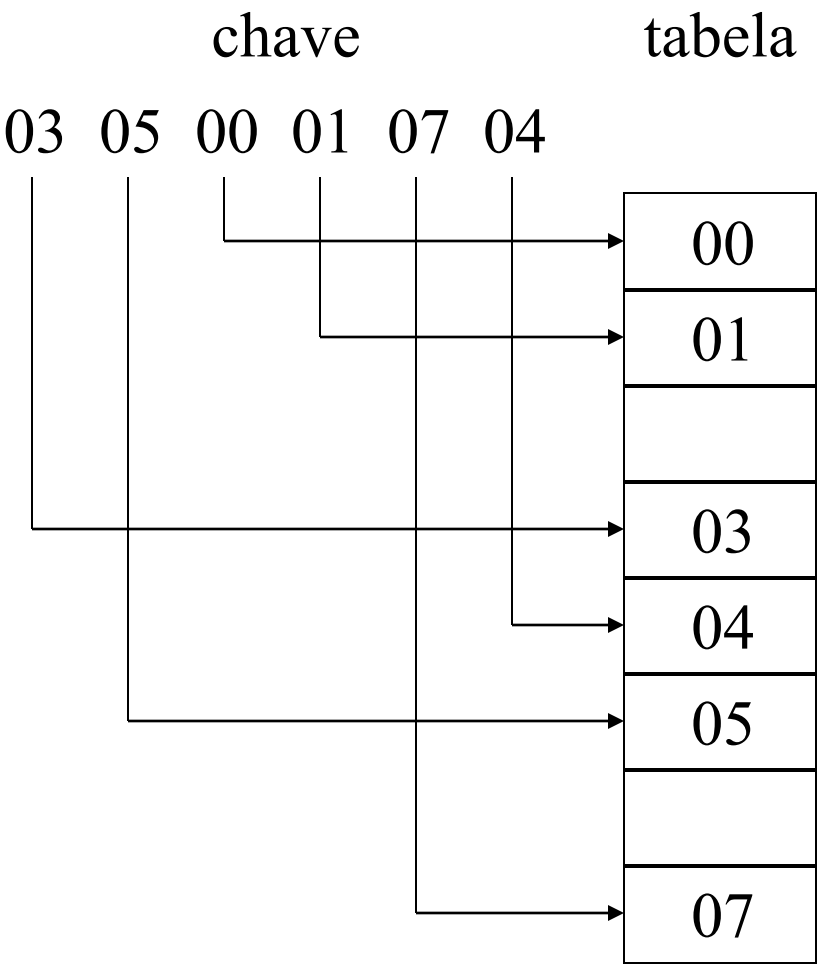
HASHING

- **Hashing** é uma técnica que busca realizar as operações de inserção, remoção e busca em tempo constante.
- **Motivação - Acesso Direto:** Suponha que existam n chaves a serem armazenadas em uma tabela T, seqüencial e de dimensão m (m compartimentos). As posições da tabela se situem no intervalo $[0, m-1]$. Se $n = m$ ou $n < m$ com $m - n$ pequeno, pode-se utilizar, diretamente, o valor de cada chave como seu índice na tabela. Isto é, cada chave x é armazenada no compartimento x.

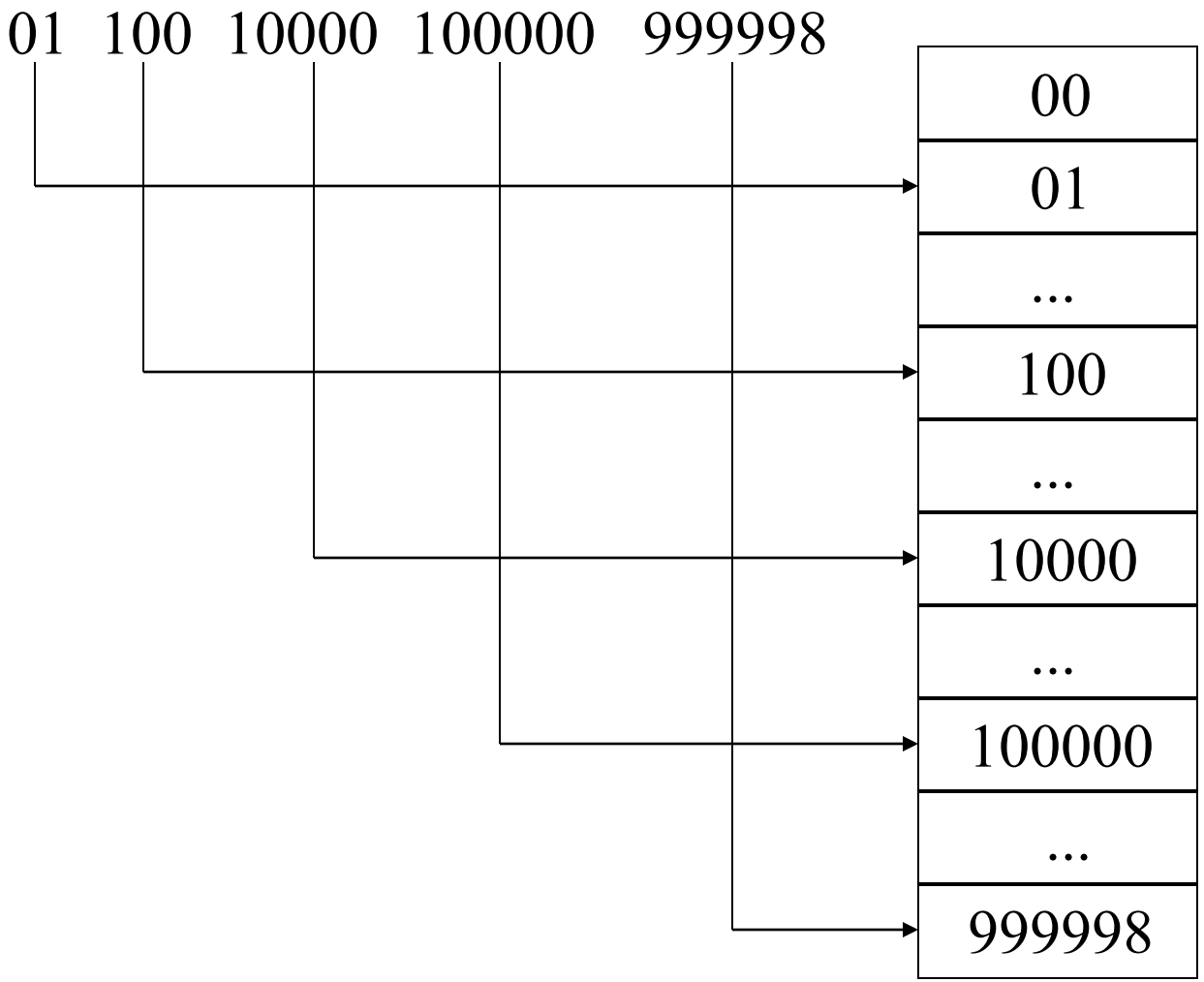
HASHING


chave
tabela
01 03 00 02 05 04
00
01
02
03
04
05


chave
tabela
03 05 00 01 07 04
00
01
03
04
05
07

HASHING

Por que não utilizar a técnica de acesso direto mesmo com espaços vazios? A resposta é simples. A quantidade de de espaços vazios pode ser proibitiva.



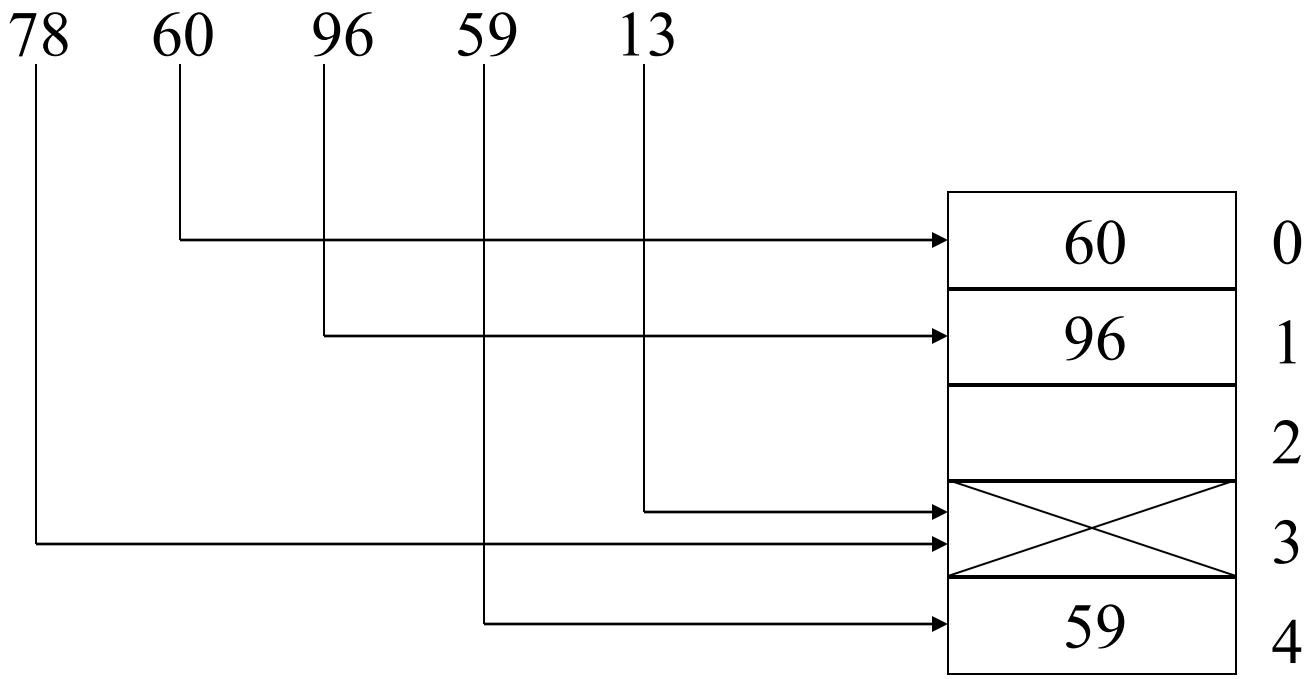
HASHING

- Como resolver essa questão? Uma idéia é transformar cada chave x em um valor no intervalo [0, m-1] através da aplicação de uma **função de dispersão (hash)**. Dada a chave x, determina-se o valor $h(x)$, denominado **endereço-base de *x***. Se o compartimento estiver desocupado, poderá ser utilizado para armazenar a chave x.
- **Colisão**. É possível a existência de duas (ou mais) chaves $x \neq y$, tal que $h(x) = h(y)$. Neste caso, o compartimento $h(x)$ já poderia então estar ocupado pela chave y. Esse fenômeno é denominado **colisão**, e as chaves x e y são **sinônimas em relação a *h***.
- Na ocorrência desse fato, utiliza-se um procedimento especial para o armazenamento de x, denominado **tratamento de colisão**.

HASHING

Exemplo de Colisão:

$h(x) = x \bmod 5$



HASHING

2.1 Funções Hash

Idealmente, uma função hash satisfazer as seguintes condições:

- **produzir um numero baixo de colisões**;
- **ser facilmente compatível** - considera o tempo consumido para calcular $h(x)$
- **ser uniforme** - Significa que, idealmente, a função h deve ser tal que todos os compartimentos possuem a mesma probabilidade de serem escolhidos, i.e., $P_k = 1/m$, para todos as chaves x e todos os endereços $k = h(x) \in [0, m-1]$.

HASHING

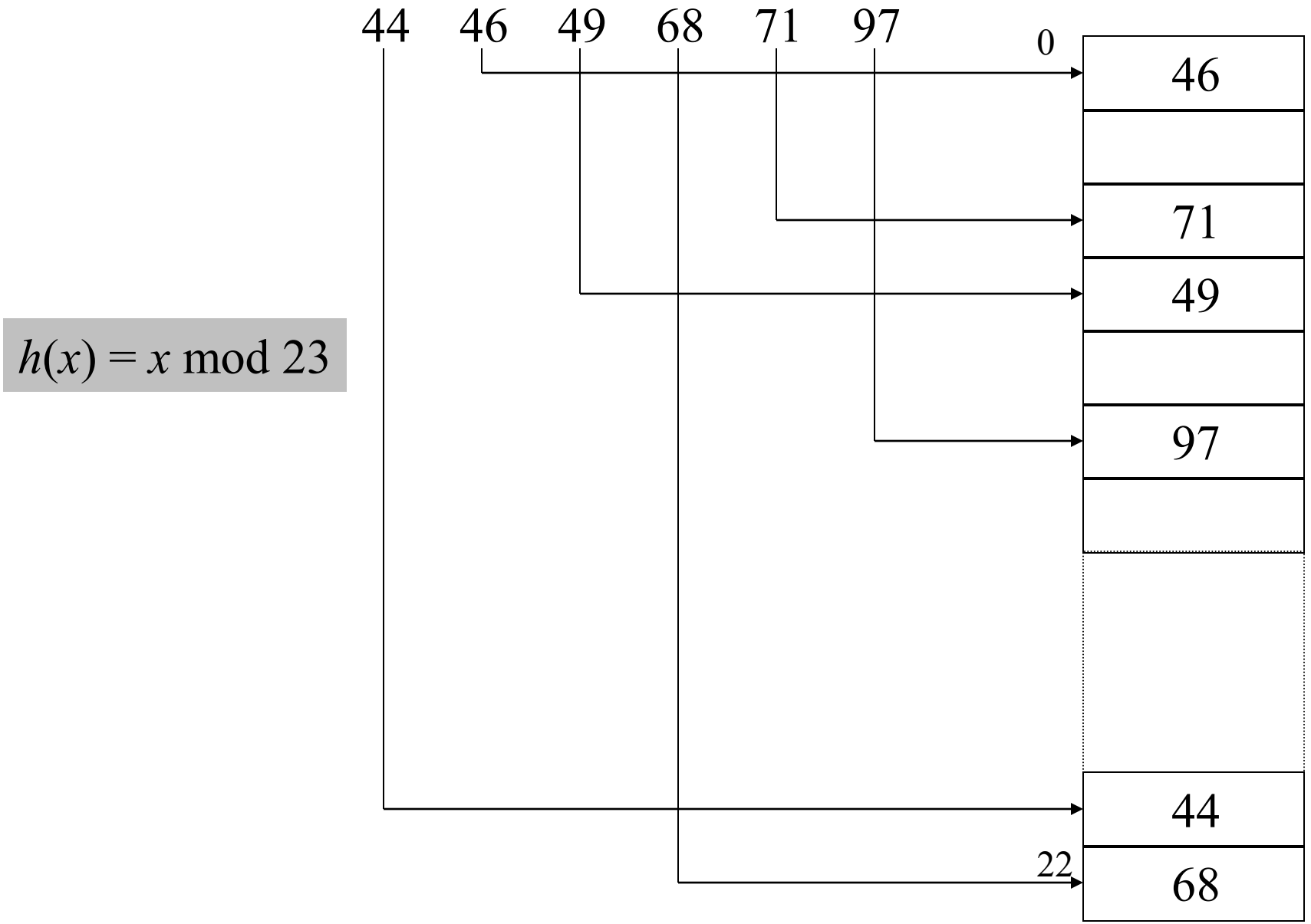
2.1.1 Método da Divisão

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

resultando em endereços no intervalo [0, m-1].

- Nesse caso, alguns valores de m são melhores que outros. Por exemplo, se m é um numero par, $h(x)$ será par quando x for par e impar quando x for impar.
- Existem alguns critérios que tem sido aplicados com bom resultados práticos, como escolher m de modo que seja um numero primo não próximo a uma potência de 2.Ou então, escolher m tal que não possua divisores primos menores que 20.

HASHING



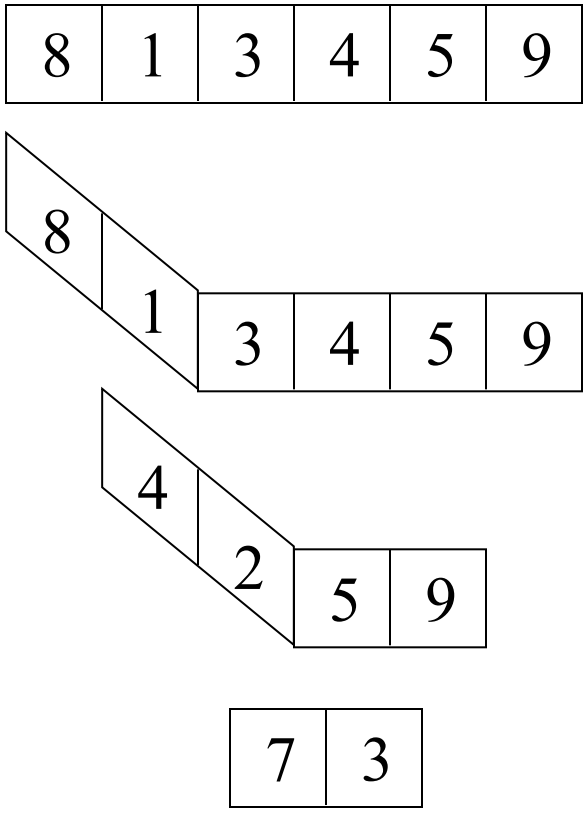
$h(x) = x \bmod 23$

HASHING

2.1.2 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 superponham. Estes devem então somados, sem levar em consideração o "vai um". Suponha que os dígitos decimais da chave sejam d_1, ..., d_k e que uma dobra seja realizada após o j-ésimo digito à esquerda. Isto implica transformar a chave em d_1', ..., d_j', d_{2j+1}, ..., d_k, onde d_i' é o digito menos significativo da soma d_i+d_{2j-i+1}, $1 \leq i < j$. O processo é repetido mediante a realização de novas dobras. O numero total de dobras e a posição j de cada uma devem ser definidos de tal forma que o resultado final contenha o numero de dígitos desejados para formar o endereço-base.

HASHING



8+4=12

1+3=4

4+9=13

2+5=7

HASHING

2.1.3 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* produz resultados melhores do que as operações *e* ou *ou*. Isto porque o *e* de dois operandos produz um numero binário sempre menor do que ambos, e o *ou* sempre maior.

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 ouex, tem-se:

71 = 00010 00111 endereço-base: 00010 ouex 00111 = 00101 = 5

46 = 00001 01110 endereço-base: 00001 ouex 01110 = 01111 = 15

HASHING

2.2 Tratamento de Colisões por Encadeamento

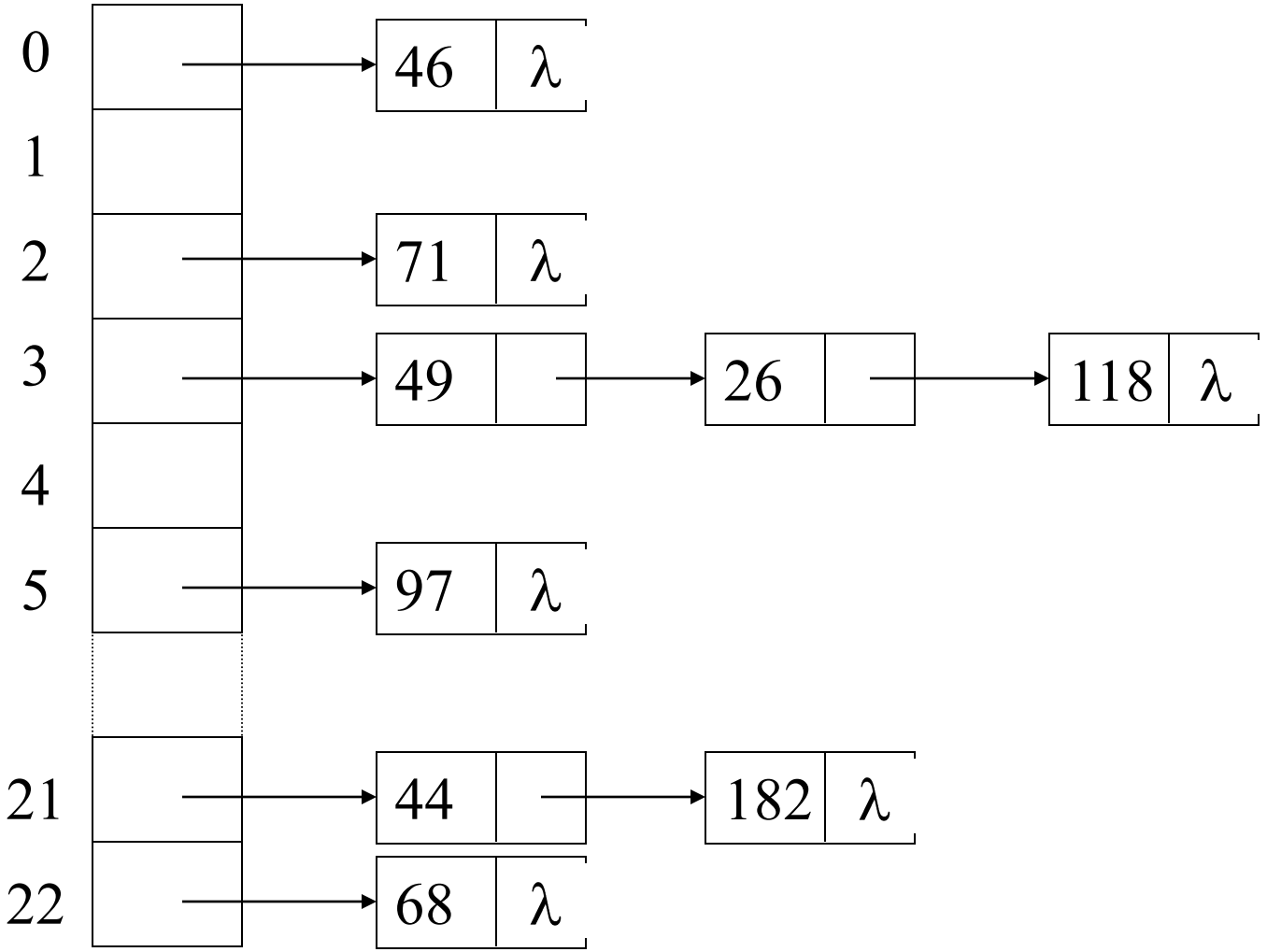
Uma idéia natural para tratar colisões consiste em armazenar as chaves sinônimas em listas encadeadas. As listas podem se encontrar no exterior da tabela ou compartilhar o mesmo espaço da tabela.

2.2.1 Encadeamento Exterior

- Manter m listas encadeadas, uma para cada possível endereço-base.
- A implementação desse método é simples aplicação dos conceitos de listas encadeadas.

HASHING

Exemplo, $h(x) = x \bmod 23$



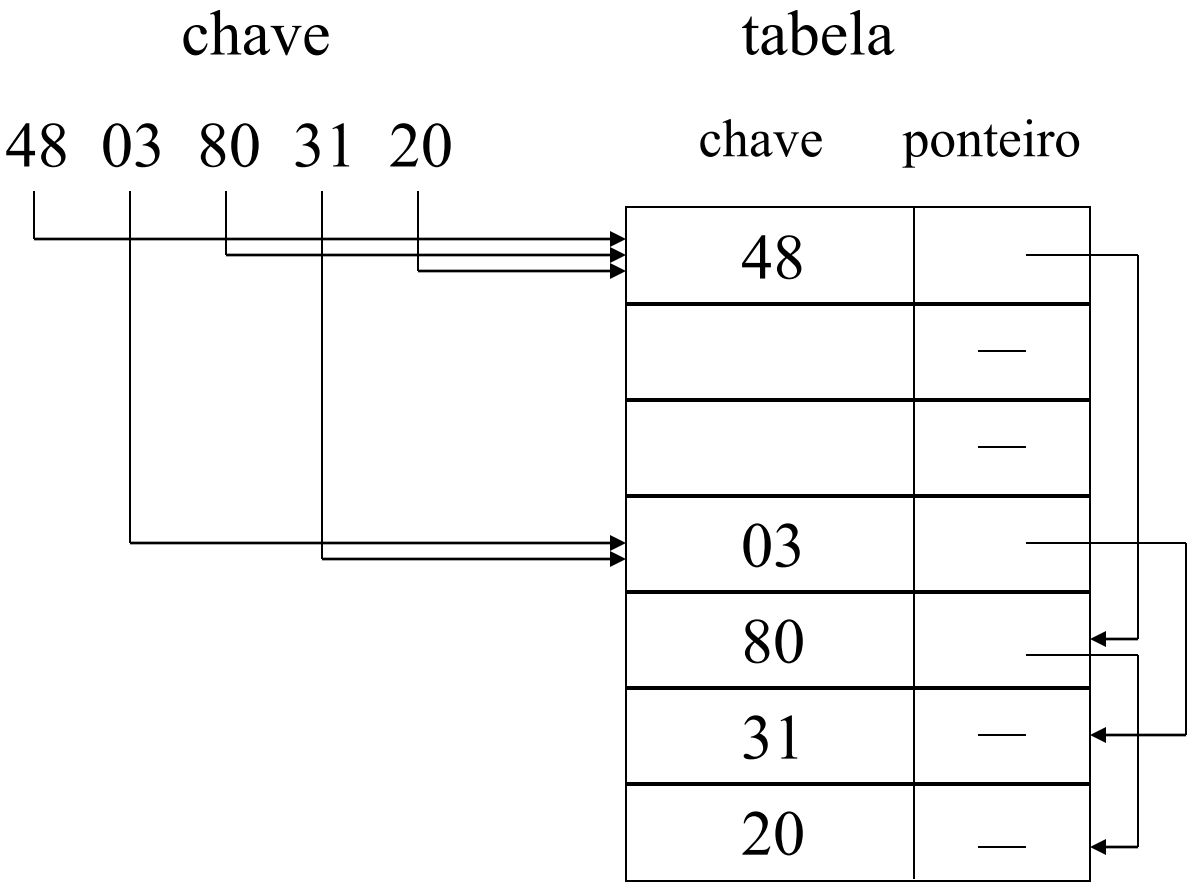
HASHING

2.2.2 Encadeamento Interior

O encadeamento interior 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. A estrutura da tabela é a mesma que no caso do encadeamento exterior. 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 correspondentes ao endereço-base em questão.

HASHING

Exemplo, $n = 5$ (numero total de chaves), $m = 7$ (tamanho da tabela)
$p = 4$, $s = 3$. $h(x) = x \bmod 4$.



HASHING

Observação:

Em principio, não se pode, simplesmente, remover uma chave *x* de uma lista encadeada interior, sem reorganizar a tabela, o que está fora de cogitação. Senão, pode perder os endereços dos elementos apontado diretamente e indiretamente do elemento sendo removido. Para contornar esse problema, considera-se que cada compartimento da tabela pode estar em um dos três estados:

vazio - pode ser utilizado para armazenar qualquer chave;

ocupado - contem uma chave armazenada;

liberado - está ocupado por alguma chave *x* cuja remoção é solicitado.

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

HASHING

2.3 Tratamento de Colisão por Endereço Aberto

- A idéia básica é armazenar as chaves sinônimas também na tabela, sem qualquer informação adicional. Quando houver alguma colisão, determina-se, também por calculo, qual o próximo compartimento a ser examinado. Se ocorre nova colisão com alguma outra chave armazenada nesse ultimo, um novo compartimento é escolhido mediante calculo, e assim por diante.
- A busca com sucesso se encerra quando um compartimento for encontrado contendo a chave procurada. O indicativo de busca sem sucesso seria a computação de um compartimento vazio, ou a exaustão da tabela.
- A remoção de chaves da tabela exige cuidados especiais, que apresenta um problema semelhante àquele do método do encadeamento interior.

HASHING

2.3.1 Tentativa Linear

Suponha que o endereço-base da chave x é $h'(x)$. Agora suponha outra chave, x', ocupando o mesmo compartimento $h'(x)$. A idéia consiste em tentar armazenar o novo nó, de chave x, no endereço-base $h'(x)+1$. Se este já está ocupado, tenta-se $h'(x)+2$, ... etc. até uma posição vazia, considerando-se a tabela circular. Então, a função hash é

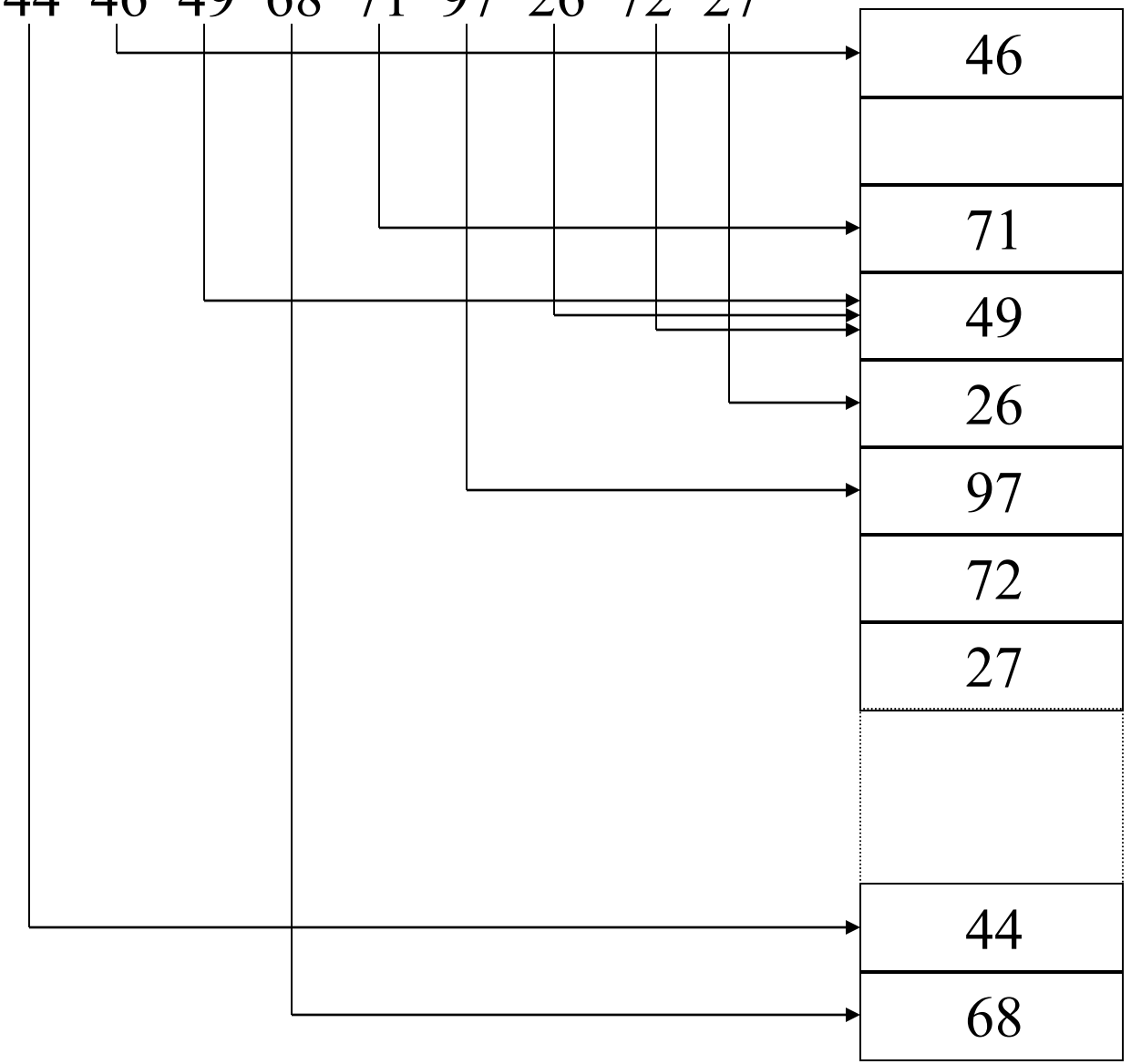
$$h(x, k) = (h'(x)+k) \bmod m,\ 0 \leq k \leq m-1.$$

HASHING

Exemplo

44 46 49 68 71 97 26 72 27

$h(x) = x \bmod 23$



HASHING

Observação

Esse método apresenta o inconveniente de produzir longos trechos consecutivos de memória ocupados, o que se denomina *agrupamento primário*. Por exemplo, suponha que exista um trecho de *j* compartimentos consecutivos ocupados e um compartimento vazio *l*, imediatamente seguinte a esses. Então, na inserção de uma nova chave *x*, a ocorrência de colisão com alguma chave ocupando qualquer desses *j* compartimentos que formam o agrupamento primário fará com que *x* seja armazenada no compartimento *l*, aumentando o tamanho do agrupamento primário para *j*+1. De fato, a formação desses trechos decorre da maneira como são tratadas as colisões. Quanto maior for o tamanho de um agrupamento primário, maior a probabilidade de aumentá-lo ainda mais, mediante a inserção de uma nova chave.

HASHING

2.3.2 Tentativa Quadrática

No método linear, as chaves tendem a se concentrar, criando agrupamentos primários, que aumentam muito o tempo de busca. A idéia agora é obter seqüências de endereços diversas para endereços-base próximos, porem diferentes. A função hash é então

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

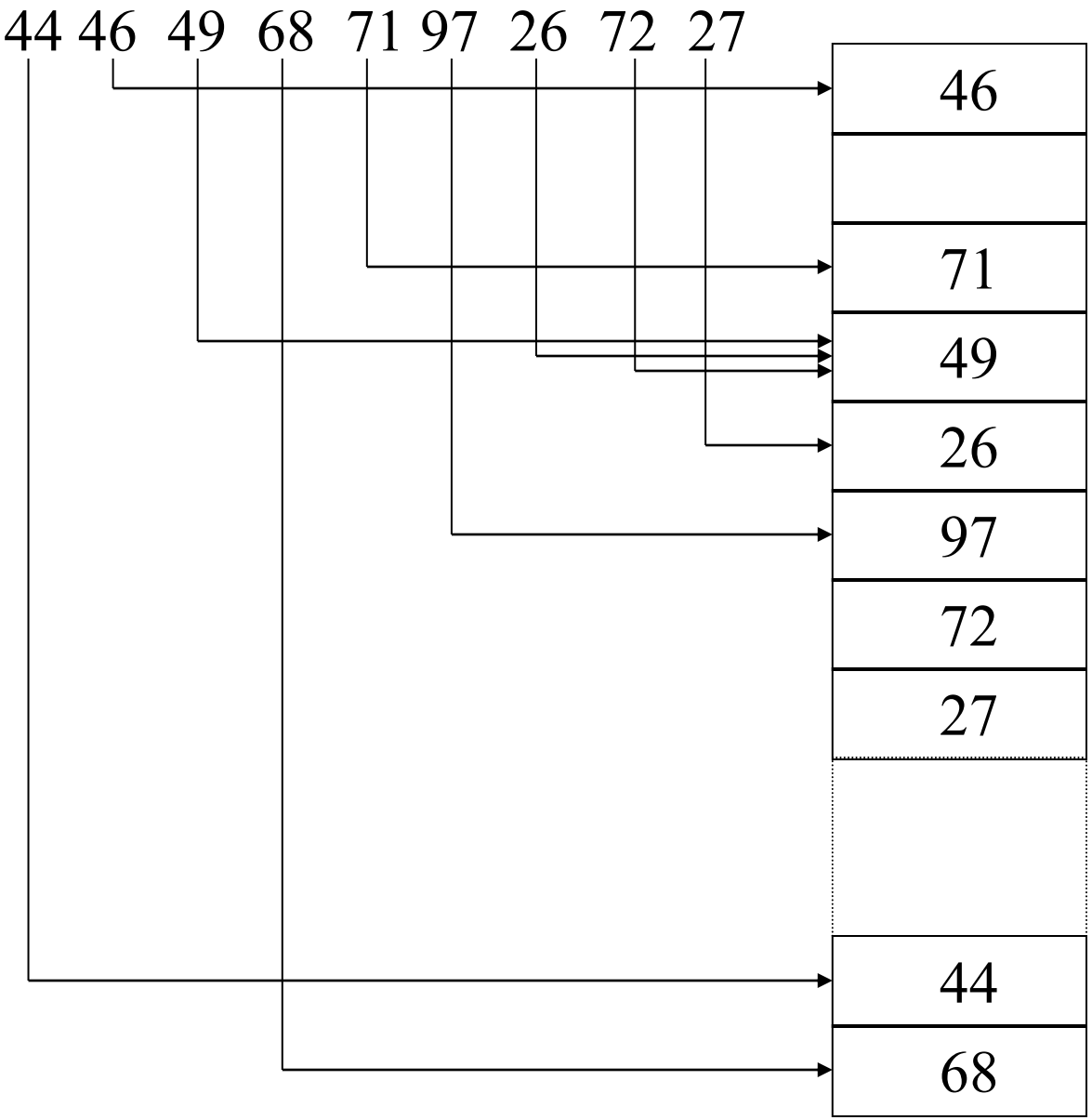
onde c_1, c_2 são constantes, $c_2 \neq 0$ e $0 \leq k \leq m\text{-}1$.

Para aplicação do método da tentativa quadrática, os valores de m, c_1 e c_2 devem ser escolhidos de tal forma que os endereços-base $h(x, k)$ correspondam a varrer toda a tabela, para $k = 0, 1, ..., m\text{-}1$. As equações recorrentes apresentadas a seguir fornecem uma maneira de calcularam, diretamente, esses endereços:

$$h(x, 0) = h'(x)$$

$$h(x, k) = (h(x, k\text{-}1) + k) \bmod m \qquad 0 < \mathrm{k} < m$$

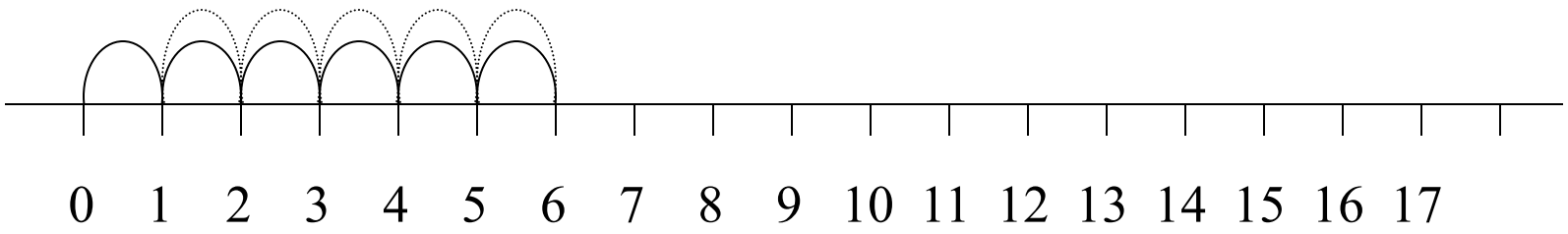
HASHING

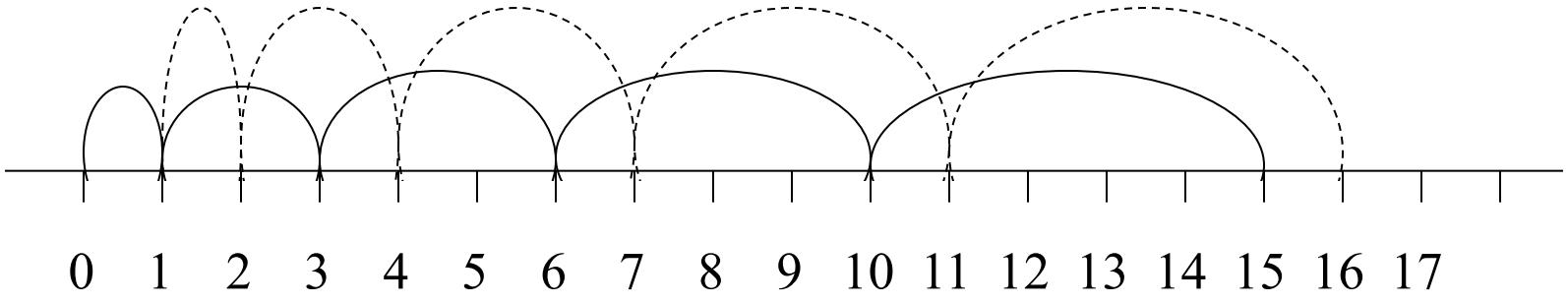


Ao se aplicar a tentativa quadrática às chaves do mesmo exemplo da seção anterior (tentativa linear), a tabela resultante é a mesma. Entretanto, as chaves 26, 72 e 27, que provocam colisões,são alocadas por tentativa linear após, respectivamente, 2, 4 e 4 tentativas, enquanto, as serem alocadas por tentativa quadrática, as tentativas são em números de 2, 3 e 3.

$h(x) = x \bmod 23$

HASHING


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17