

Estruturas de Dicionário

Eduardo Furlan Miranda

2024-02-01

Adaptado do material do Prof. Guilherme Tomaschewski Netto

Tabelas de acesso direto

Adaptado do livro:

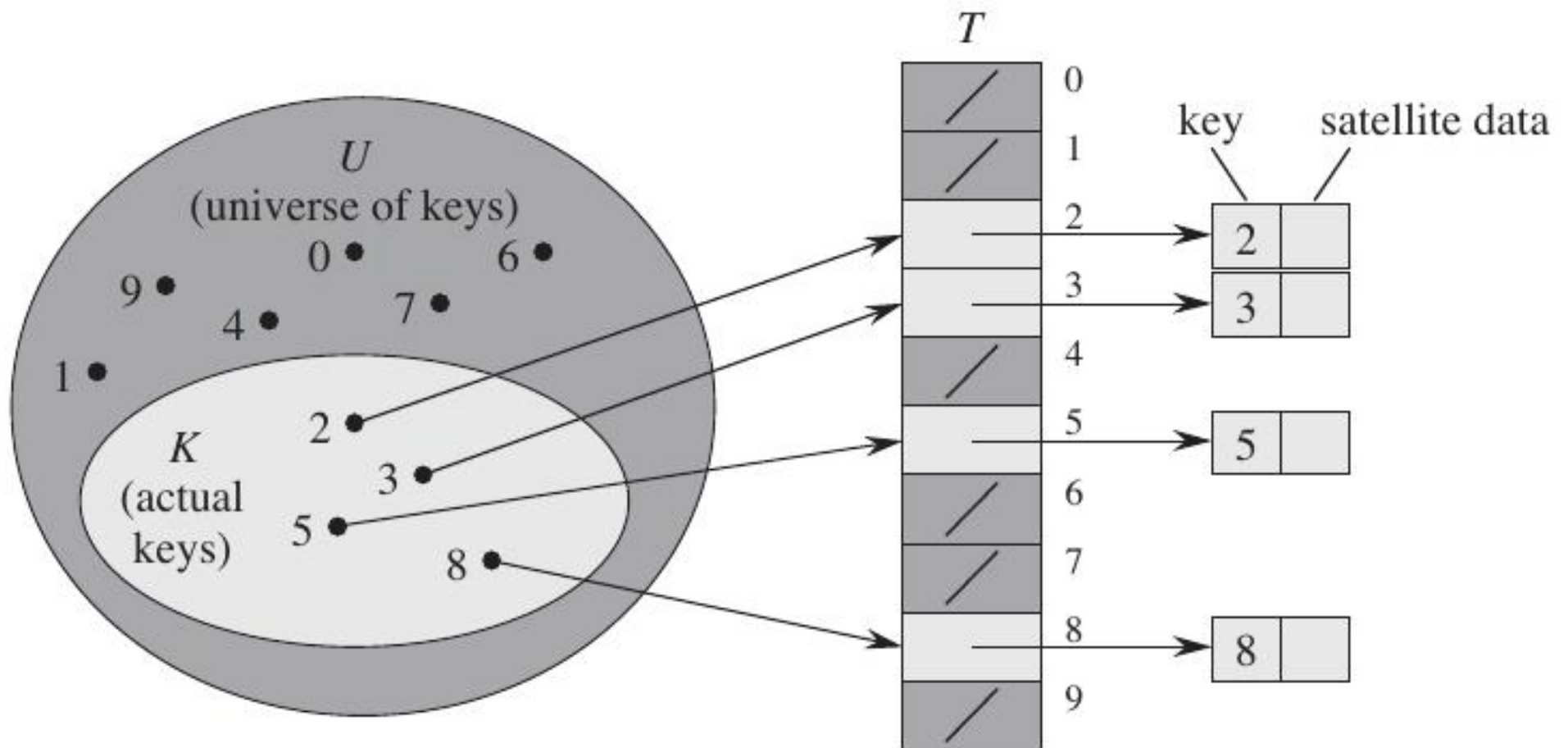
CORMEN, T. H. Algoritmos - Teoria e Prática. [S. I.]: GEN LTC, 2012. ISBN 978-85-352-3699-6.

Tabelas de acesso direto

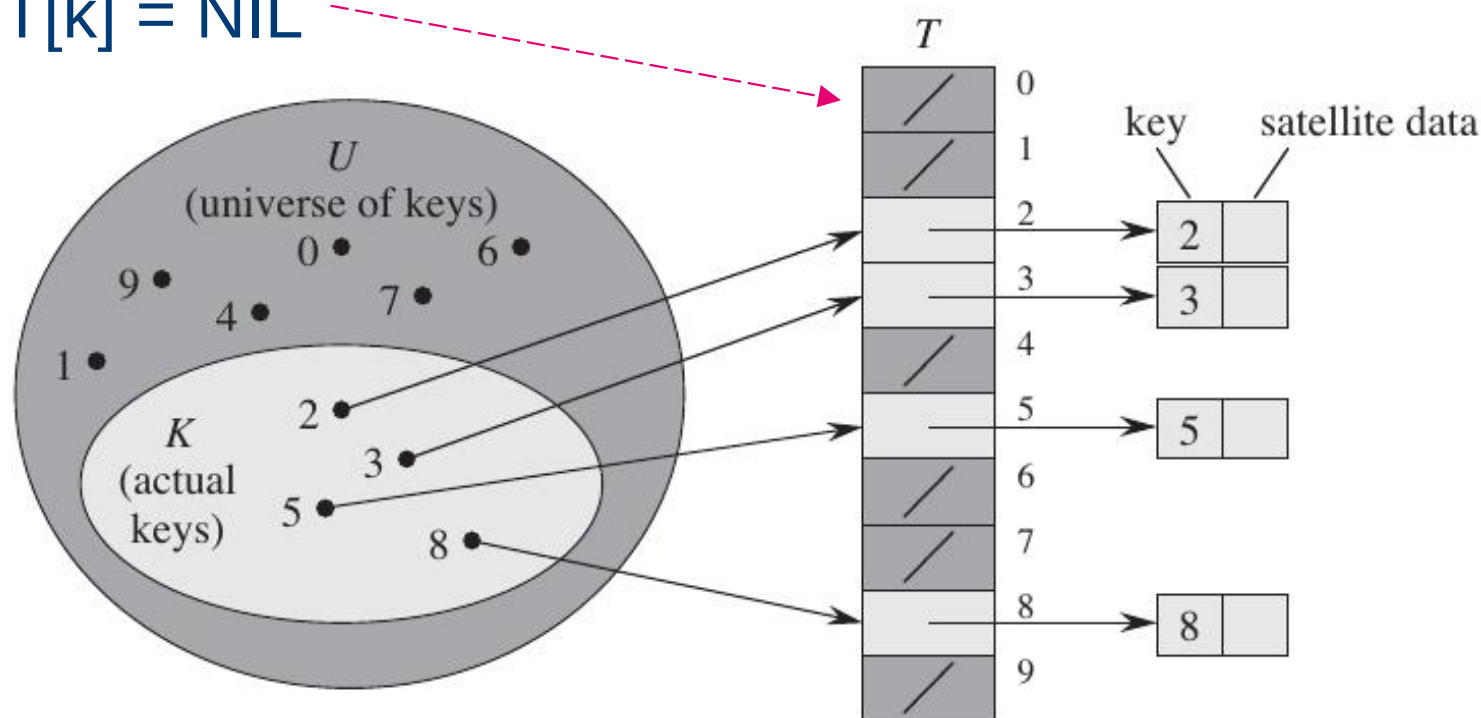
(Direct-address tables)

- O método simples de endereçamento direto funciona bem quando o universo U de chaves é relativamente pequeno
- Assumiremos que uma aplicação precisa de um conjunto dinâmico em que cada elemento tenha uma chave extraída do Universo $U = \{0; 1; \dots ; m - 1\}$, sendo m não muito grande
- Assumiremos que dois elementos não podem ter a mesma chave

- Para representar o conjunto dinâmico, usamos um array, ou tabela de endereço direto, denotado por $T[0 \dots m - 1]$



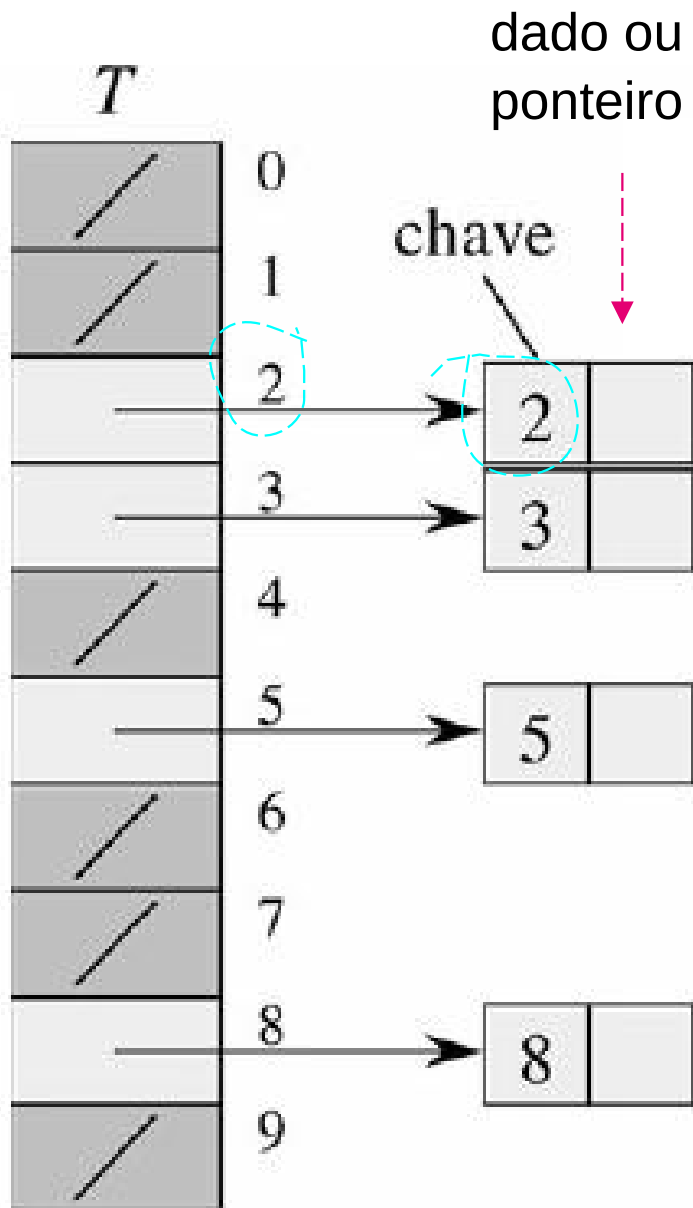
- Cada posição (ou slot) corresponde a uma chave no universo U
- O slot k aponta para um elemento do conjunto com a chave k
- Se o conjunto não contém nenhum elemento com chave k , então $T[k] = \text{NIL}$



- Os dados satélite (*satellite data*) podem ser ponteiros ou o próprio dado em si
- Uma forma de implementar T seria uma matriz com 2 colunas
 - A primeira armazena a chave
 - A segunda armazena os dados satélites (ponteiros ou dados)
 - O primeiro índice da matriz poderia corresponder à chave
- A ideia principal é que o acesso é direto, sem uma tradução ou manipulação da chave para encontrar o slot

Operações de dicionário

- As operações do dicionário são triviais de implementar
 - Cada uma dessas operações leva somente o tempo $O(1)$
- DIRECT-ADDRESS-SEARCH(T, k)
 - return $T[k] = x$
- DIRECT-ADDRESS-INSERT(T, x)
 - $T[x.key] = x$
- DIRECT-ADDRESS-DELETE(T, x)
 - $T[x.key] = \text{NIL}$



- A chave pode acabar repetindo o índice
- Em algumas aplicações, para economizar espaço, a chave pode ser omitida
 - E o índice usado no lugar, mantendo o dado ou ponteiro
- Neste caso precisamos determinar uma forma especial para indicar uma lacuna (“NIL”).

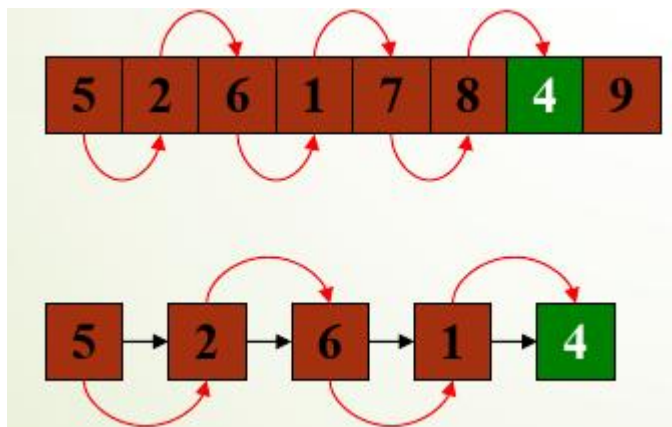
Hashing

Adaptado do material do Prof. Guilherme Tomaschewski Netto

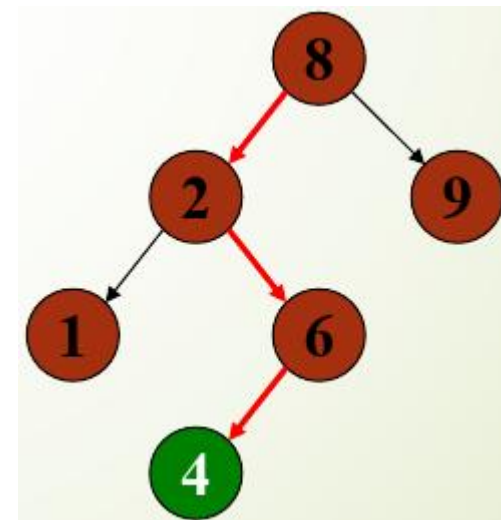
TOMASCHEWSKI NETTO, Prof. Guilherme. Hashing. 2019.
<http://netto.ufpel.edu.br/lib/exe/fetch.php?media=aed2:hashing.pdf>

Hashing

- Quando tenho uma tabela com uma chave e vários valores por linha, quero encontrar, inserir e apagar registros usando as suas chaves de forma rápida
- As estruturas de busca sequencial e binária podem levar mais tempo para encontrar o elemento, do que usando hash



Arrays e listas



Árvores

Motivação

- Suponha que você pudesse criar um array com acesso direto a qualquer item
- Para aplicações do tipo dicionário, onde gostaríamos de consultar constantemente elementos de tabela, isso seria ideal
- Exemplos de aplicação
 - Tabela de símbolos em compiladores, que podem crescer muito em tamanho e o acesso é frequente
 - Eu tenho um array indexado comum e os elementos usados são esparsos. Preciso de uma maneira de reduzir o tamanho ocupado
 - Eu tenho uma tabela enorme, o valor que procuro está no último registro, e não quero varrer a tabela toda até encontrar

Motivação

- Em algumas aplicações, é necessário obter o valor usando poucas comparações
- Neste caso, é possível identificar a localização do elemento sem usar todas as chaves
- A tabela hash procura resolver estas questões



Hashing

- O objetivo do hashing é mapear uma grande quantidade de chaves em um espaço pequeno de inteiros
- Para isso usamos uma função chamada *hash function*
- O hash code (um inteiro), que é criado pela função de hash, é usado para localizar um determinado item que estamos buscando

Funções hashing

- Transforma as chaves de pesquisa em endereços para a tabela
- Permite obter o valor armazenado no endereço da chave
- Deve ser fácil de computar
- A distribuição das chaves deve ser equiprovável
 - Tentando evitar colisões

Funções hashing

- Seja M o tamanho da tabela
 - A função de hashing mapeia as chaves de entrada em inteiros dentro do intervalo $[1 \dots M]$
- Formalmente
 - A função de hashing $h(k_j) \rightarrow [1, M]$ recebe uma chave $k \in \{k_0, \dots, k_m\}$ e retorna um número i , que é o índice do subconjunto $m_i \in [1, M]$ onde o elemento que possui essa chave vai ser manipulado

Função de dispersão (hash, espalhamento)¹⁶

- $H(\text{chave}) \rightarrow \text{endereço}$
 - $H(3204) = 504$
- Depende do número de endereços e da natureza da chave
- Registros do índice deve ser de tamanho fixo
- Quantidade fixa de endereços

Tabela de dispersão

$H(1) = 2$
 $H(2) = 8$
 $H(3) = 6$
 $H(4) = 5$
 $H(5) \neq 0$

	Código	Endereço
0	5	432
1		
2	1	0
3		
4		
5	4	
6	3	
7		
8	2	108
9		

End	Cod	Título	Autor	Preço
0	1	Java web services	Kalin	34.50
108	2	Java	Deitel	150.80
216	3	Web services em php	Jane	80.75
324	4	Programacao Java	Decio	55.7
432	5	Desenvolv. Web	Quian	45.0

Função de dispersão - exemplo

- Elevar a chave ao quadrado e pegar um grupo de dígitos do meio:

$$A = h(452) = 452^2 = 205209 \rightarrow A = 52$$

(dois dígitos supondo um arquivo com 100 endereços)

Função de dispersão - exemplo

- Multiplicar o valor ASCII de 2 letras e usar o resto da divisão do número de endereços (1000)

Chave	Cálculo	Endereço
Joaquim	$74 \times 79 = 5846$	846
Carla	$67 \times 65 = 4355$	355
Guilherme	$71 \times 85 = 6035$	35

Funções hashing

- Existem várias funções Hashing, ex.:
 - Resto da Divisão
 - Meio do Quadrado
 - Método da Dobra
 - Método da Multiplicação
 - Hashing Universal

Resto da Divisão


- Forma mais simples e mais utilizada
 - A chave é interpretada como um valor numérico que é dividido por um valor
- O endereço de um elemento na tabela é dado simplesmente pelo resto da divisão da sua chave por M ($F_h(x) = x \bmod M$), onde M é o tamanho da tabela e x é um inteiro correspondendo à chave

$$0 \leq F(x) < M$$

Resto da divisão

- Ex: $M=1001$ e a sequência de chaves: 1030, 839, 10054 e 2030

resto da divisão
de 1030 por 1001



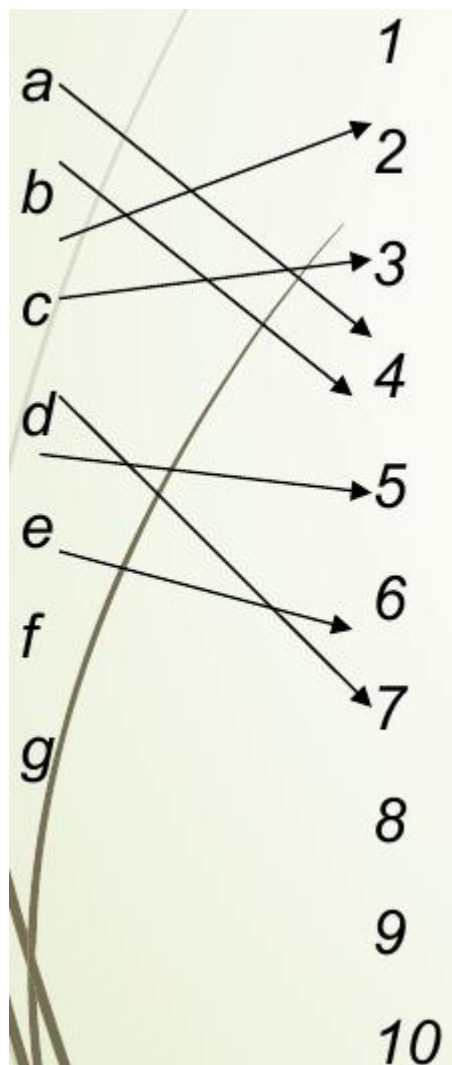
<u>Chave</u>	<u>Endereço</u>
<u>1030</u>	<u>29</u>
<u>10054</u>	<u>53</u>
<u>839</u>	<u>838</u>
<u>2030</u>	<u>29</u>

Resto da divisão - desvantagens

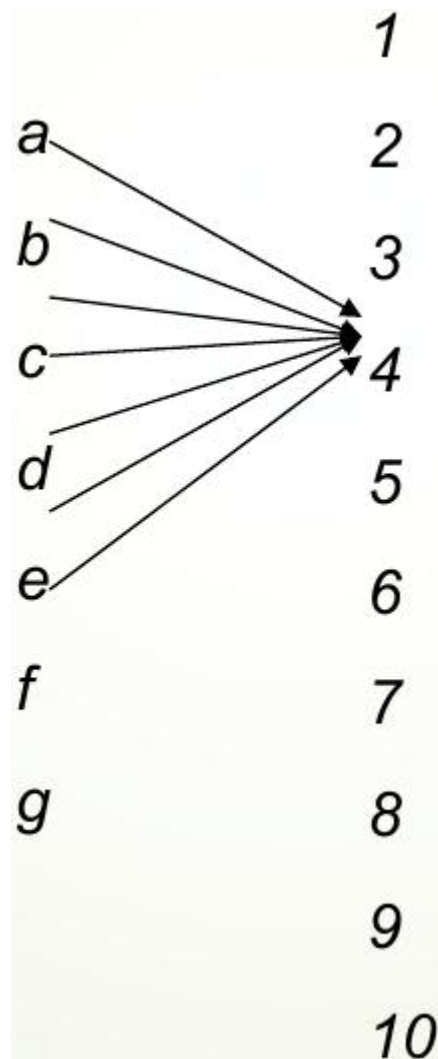
- Função extremamente dependente do valor de M escolhido
 - M deve ser um número primo
 - Valores recomendáveis de M devem ser > 20

Funções hashing

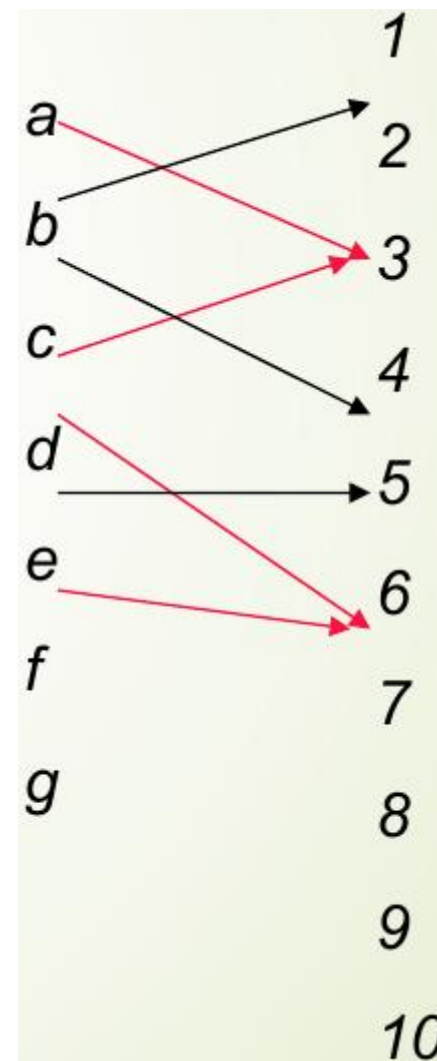
ideal (uniforme)



ruim



aceitável



Colisões

- Seja qual for a função, na prática existem “sinônimos”
 - Chaves distintas que resultam em um mesmo valor de hashing
- Quando duas ou mais chaves sinônimas são mapeadas para a mesma posição da tabela, diz-se que ocorre uma colisão

Colisões

- Qualquer que seja a função de transformação, existe a possibilidade de colisões, que devem ser resolvidas, mesmo que se obtenha uma distribuição de registros de forma uniforme
- Tais colisões devem ser corrigidas de alguma forma
- O ideal seria uma função HASH tal que, dada uma chave $1 \leq I \leq 26$, a probabilidade da função retornar a chave x , seja $\text{PROB}(F_h(x) = I) = 1/26$, ou seja, não tenha colisões, mas tal função é difícil, se não impossível

Resto da divisão - colisão

- No exemplo dado, $M=1001$ e a sequência de chaves: 1030, 839, 10054 e 2031

<u>Chave</u>	<u>Endereço</u>
<u>1030</u>	<u>29</u>
<u>10054</u>	<u>53</u>
<u>839</u>	<u>838</u>
<u>2030</u>	<u>29</u>

O valor de $h(k)$ é o mesmo

→ colisão

Tratamento de colisões

- Alguns dos algoritmos de Tratamento de Colisões são
 - Endereçamento fechado
 - Endereçamento aberto
 - Hashing linear
 - Hashing duplo (ou re-hash)

Endereçamento fechado

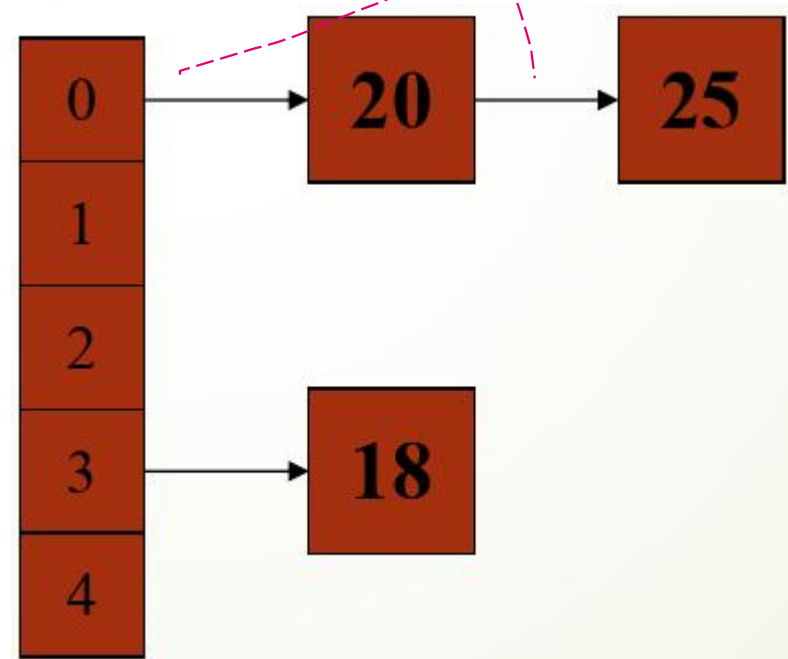
- Também chamado de Overflow Progressivo Encadeado
 - Algoritmo: usar uma **lista encadeada** para cada endereço da tabela
 - Vantagens
 - Só sinônimos são acessados em uma busca
 - Processo simples
 - Desvantagens
 - É necessário um campo extra para os ponteiros de ligação
 - Tratamento especial das chaves
 - As que estão com endereço base e as que estão encadeadas

Endereçamento fechado

- No endereçamento fechado, a posição de inserção não muda
- Todos devem ser inseridos na mesma posição
 - Através de uma **lista ligada** em cada uma

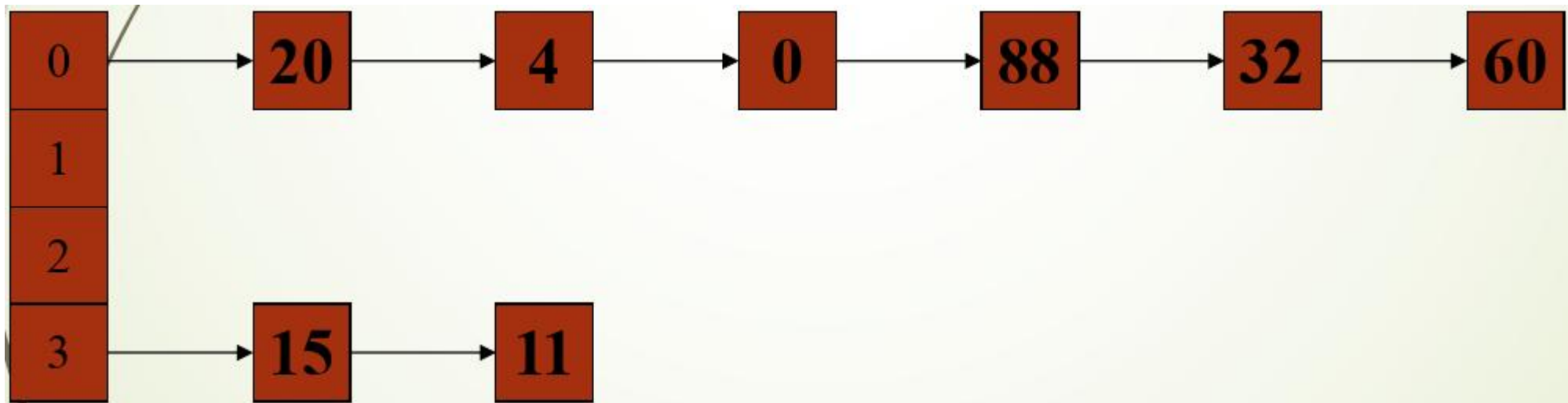
$$\begin{array}{l} 20 \bmod 5 = 0 \\ 18 \bmod 5 = 3 \\ 25 \bmod 5 = 0 \end{array}$$

colisão



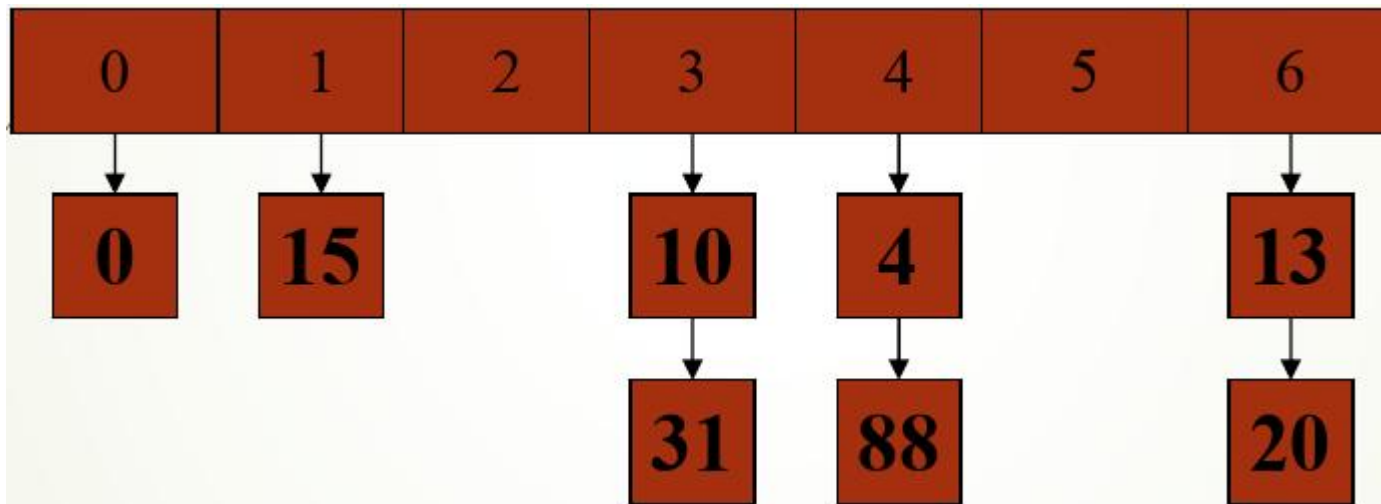
Endereçamento fechado

- A busca é feita do mesmo modo: calcula-se o valor da função hash para a chave, e a busca é feita na lista ligada correspondente
- Se o tamanho das listas variar muito, a busca pode se tornar ineficiente, pois a busca nas listas se torna sequencial



Endereçamento fechado

- É obrigação da função HASH distribuir as chaves entre as posições de maneira uniforme



Endereçamento aberto

Hashing linear

- Também conhecido como Overflow Progressivo
- Consiste em procurar a próxima posição vazia depois do endereço-base da chave
- Vantagem
 - Simplicidade

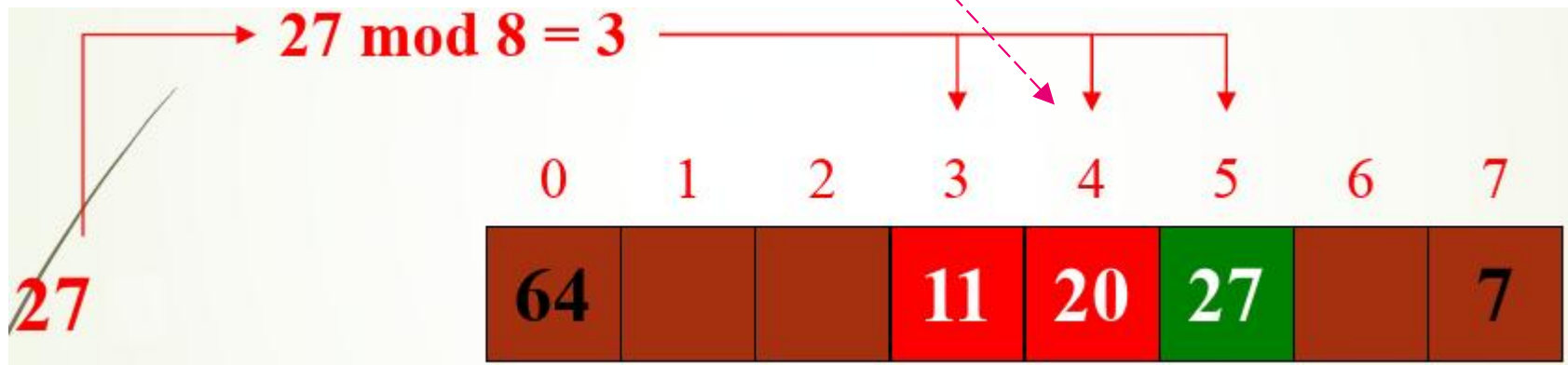
Hashing linear

- Desvantagem

- Se ocorrerem muitas colisões, pode ocorrer um clustering (agrupamento) de chaves em uma certa área
- Isso pode fazer com que sejam necessários muitos acessos para recuperar um certo registro
- O problema vai ser agravado se a densidade de ocupação para o arquivo for alta
 - Se a característica dos dados acaba gerando sempre o mesmo agrupamento

Hashing linear

tenta a próxima
já está ocupado também



3 já está ocupado

5 está vazio

Hashing linear

Tamanho da tabela: 13

Valores	52	78	79	48	61	81	120	79	121	92
Função $\text{hash}(k) = k \bmod 13$	0	0	1	9	9	3	3	1	4	1

já está ocupado

tenta o próximo



Hashing duplo (ou re-hash)

- Ao invés de incrementar a posição de 1, uma função hash auxiliar é utilizada para calcular o incremento
 - Esta função também leva em conta o valor da chave
- Vantagem
 - Tende a espalhar melhor as chaves pelos endereços
- Desvantagem
 - Os endereços podem estar muito distantes um do outro
 - O princípio da localidade e memória cache é violado

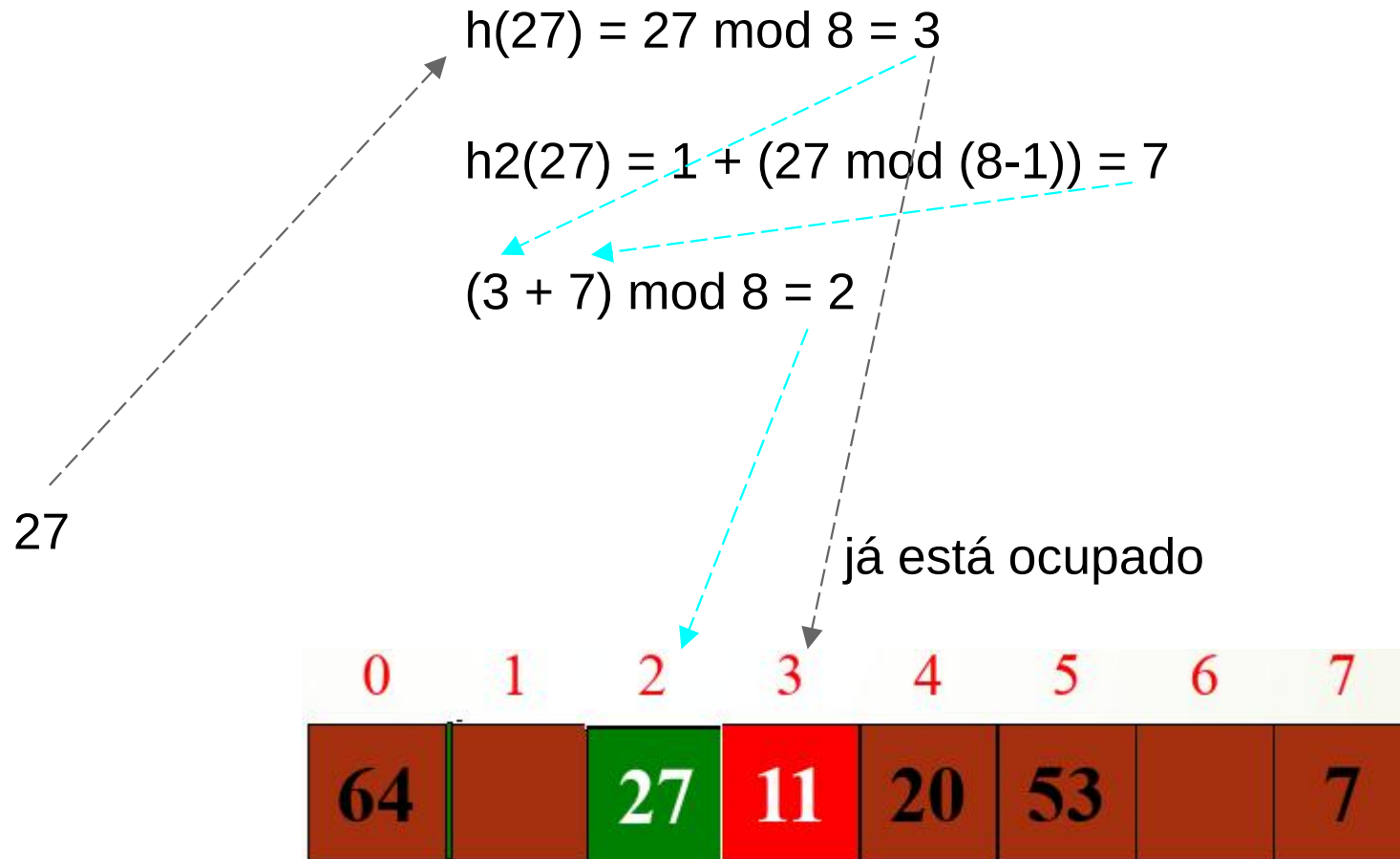
Hashing duplo

- Se o endereço estiver ocupado, aplique uma segunda função hash para obter um número c
- c é adicionado ao endereço gerado pela 1ª função hash para produzir um endereço de overflow
- Se este novo endereço estiver ocupado
 - Continue somando c ao endereço de overflow, até que uma posição vazia seja encontrada

Hashing duplo

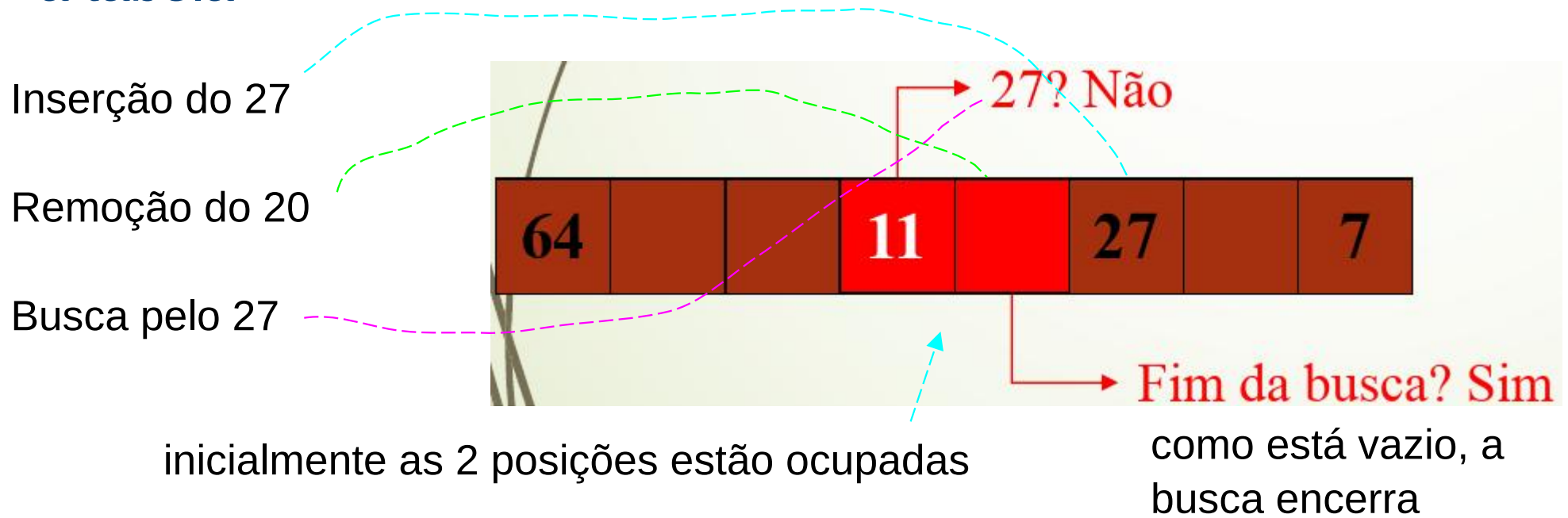
- Para o primeiro cálculo
 - $h(k) = k \bmod N$
- Caso haja colisão, inicialmente calculamos $h_2(K)$, que pode ser definida como
 - $h_2(k) = 1 + (k \bmod (N-1))$
- Em seguida calculamos a função re-hashing como sendo
 - $rh(i,k) = (i + h_2(k)) \bmod N$

Hashing duplo



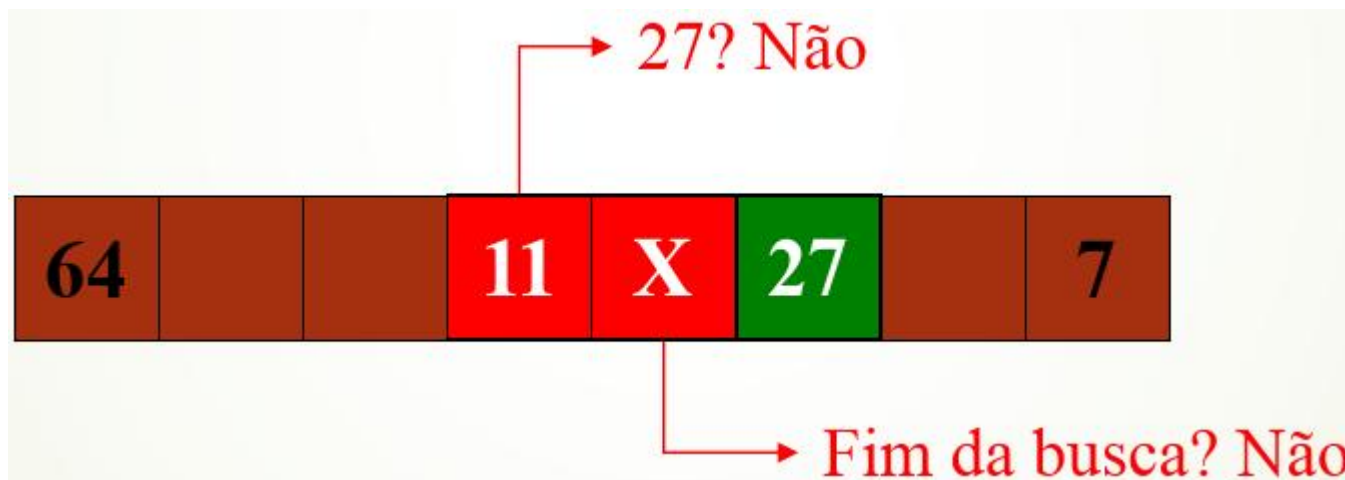
Endereçamento aberto - remoção

- Para fazer uma **busca** com endereçamento aberto, basta aplicar a função hash, e a função de incremento até que o elemento ou uma posição vazia sejam encontrados
- Porém, quando um elemento é **removido**, a posição vazia pode ser encontrada antes, mesmo que o elemento pertença a tabela



Endereçamento aberto - remoção

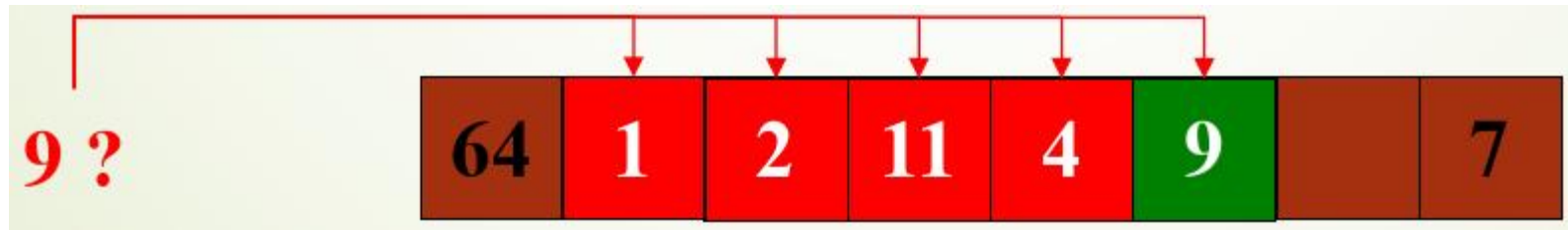
- Para contornar esta situação, mantemos um bit (ou um campo booleano) para indicar que um elemento foi removido daquela posição
- Esta posição estaria livre para uma nova inserção, mas não seria tratada como vazia numa busca



Tabelas hash dinâmicas

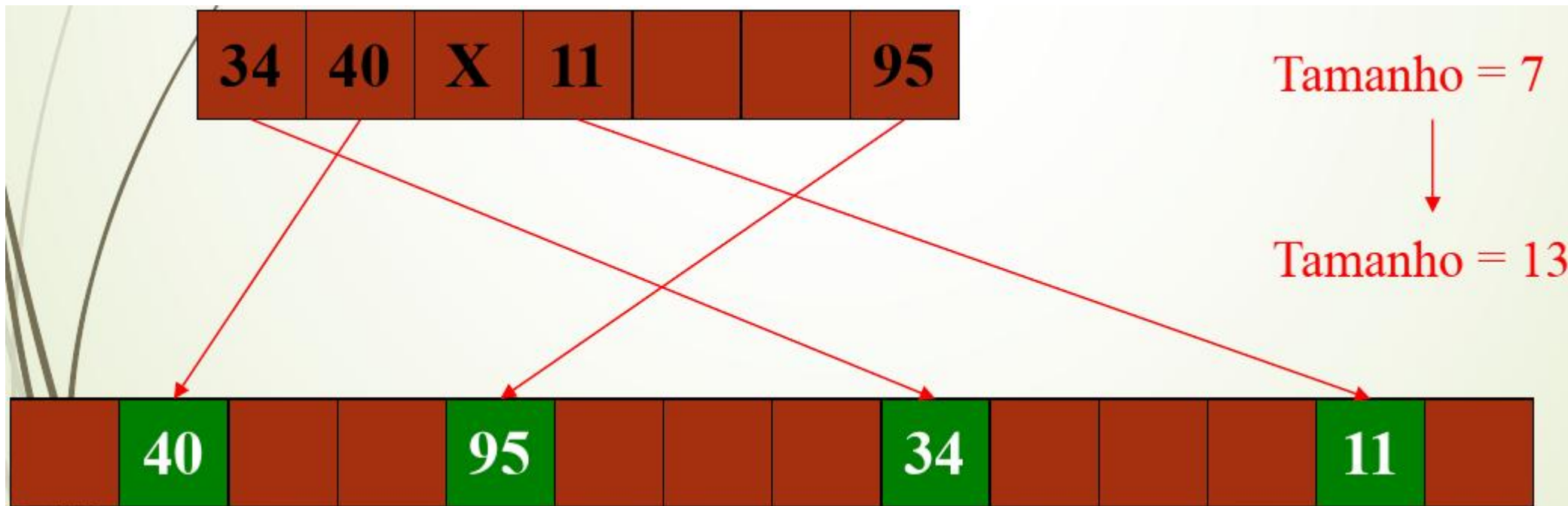
Endereçamento aberto - expansão

- Na política de hashing, há o que chamamos de fator de carga (load factor)
 - Indica a porcentagem de células da tabela hash que estão ocupadas, incluindo as que foram removidas
- Quando este fator fica muito alto (ex: excede 50%), as operações na tabela passam a demorar mais, pois o número de colisões aumenta



Endereçamento aberto – expansão

- Quando isto ocorre, é necessário expandir o array que constitui a tabela, e reorganizar os elementos na nova tabela
- O tamanho atual da tabela passa a ser um parâmetro da função hash



Endereçamento aberto - expansão

- O problema é: Quando expandir a tabela?
- O momento de expandir a tabela pode variar
 - Quando não for possível inserir um elemento
 - Quando metade da tabela estiver ocupada
 - Quando o load factor atingir um valor escolhido
- A terceira opção é a mais comum, pois é um meio termo entre as outras duas

Quando não usar hashing?

- Muitas colisões diminuem muito o tempo de acesso e modificação de uma tabela hash
- É necessário escolher bem
 - A função hash
 - O algoritmo de tratamento de colisões
 - O tamanho da tabela
- Quando não for possível definir parâmetros eficientes, pode ser melhor, p. ex., utilizar árvores balanceadas

Referências

TOMASCHEWSKI NETTO, G. Hashing. [S. l.]: Universidade Federal de Pelotas, UFPEL, 2019. Disponível em: <http://netto.ufpel.edu.br/lib/exe/fetch.php?media=aed2:hashing.pdf>