

Hashing

Rafael Nunes

LABSCI-UFMG

Sumário

- Mapeamento
- Hashing
- Porque utilizar?
- Colisões
- Encadeamento Separado
- Endereçamento Aberto
 - Linear Probing
 - Double Hashing
 - Remoção
 - Expansão
- Quando não usar!

Mapeamento

Associação de cada objeto de um tipo a uma chave, permitindo a indexação.

Ex: String \rightarrow Inteiro

“Algoritmos”

253

“Hashing”

54

“Árvore”

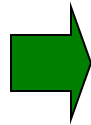
784

Útil para construir estruturas de armazenamento com custo reduzido em tempo (acesso direto).

Mapeamento

Ex: Tabuleiro de Jogo da Velha

	X	
O		O
X		



	X		O		O	X		
--	---	--	---	--	---	---	--	--



0	1	0	2	0	2	1	0	0
0	1	2	3	4	5	6	7	8

--

 = 0

X = 1

O = 2

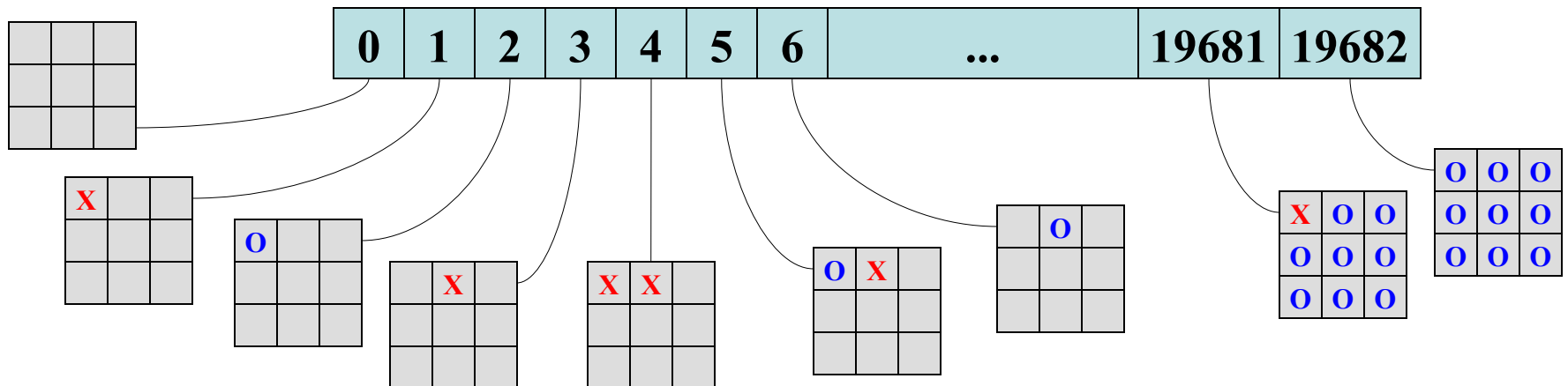
$$= 1 \times 3^1 + 2 \times 3^3 + 2 \times 3^5 + 1 \times 3^6$$

$$= 3 + 54 + 486 + 729 = 1272$$

Mapeamento

Cada tabuleiro é mapeado em um valor **único**, entre 0 e 19682. Deste modo, é possível recuperar o tabuleiro a partir de sua chave.

Através da chave, podemos indexar os objetos, por exemplo, em um array, caso o tamanho seja suficiente:



Hashing

Entretanto, se o número de chaves possíveis for muito grande, é preciso distribuir os valores possíveis entre as posições disponíveis (de 0 a $\text{length}-1$).

Esta técnica é chamada de *hashing*, e função de mapeamento chave-posição é a **função *hash***.

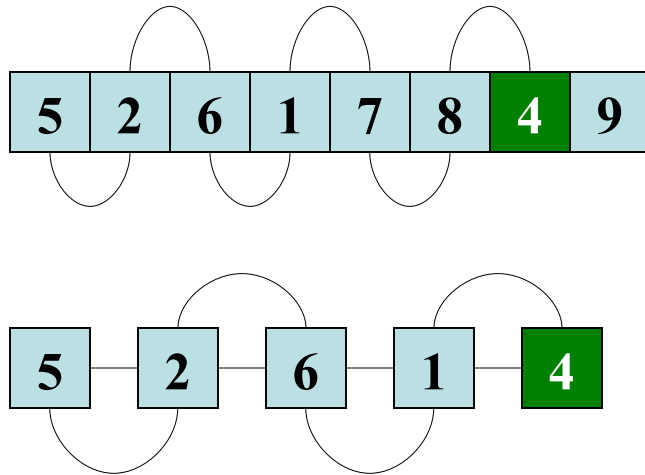
Ex: uma função *hash* simples é tirar o resto da divisão pelo tamanho da tabela

$$h(\text{chave}) = \text{chave} \% \text{array.length}$$

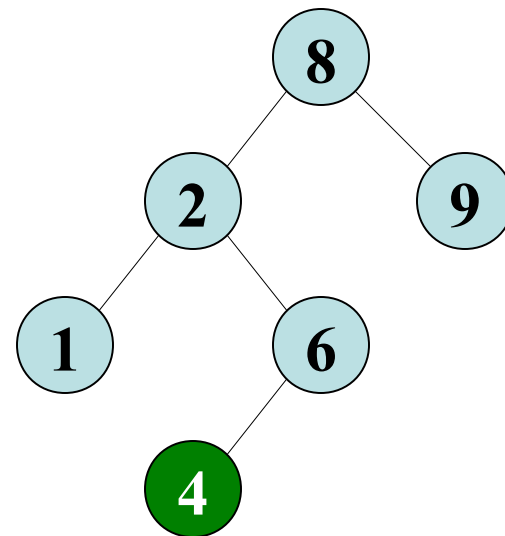
Por que usar Hashing?

Estruturas de busca sequencial levam tempo até encontrar o elemento desejado.

Ex: Arrays e listas



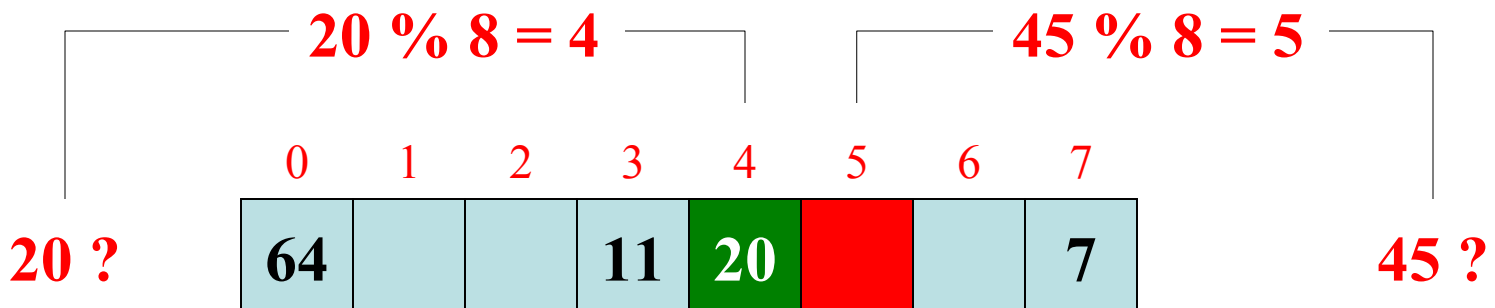
Ex: Árvores



Por que usar Hashing?

Em algumas aplicações, é necessário obter o valor com poucas comparações, logo, é preciso saber a posição em que o elemento se encontra, sem precisar varrer todas as chaves.

A estrutura com tal propriedade é chamada de **tabela hash**.



Colisões

Devido ao fato de existirem mais chaves que posições, é comum que várias chaves sejam mapeadas na mesma posição. Quando isto ocorre, dizemos que houve uma **colisão**.

Ex:	$45 \% 8 = 5$	$1256 \% 15 = 11$
	$21 \% 8 = 5$	$356 \% 15 = 11$
	$93 \% 8 = 5$	$506 \% 15 = 11$

O que fazer quando mais de um elemento for inserido na mesma posição de uma tabela *hash*?

Encadeamento Separado

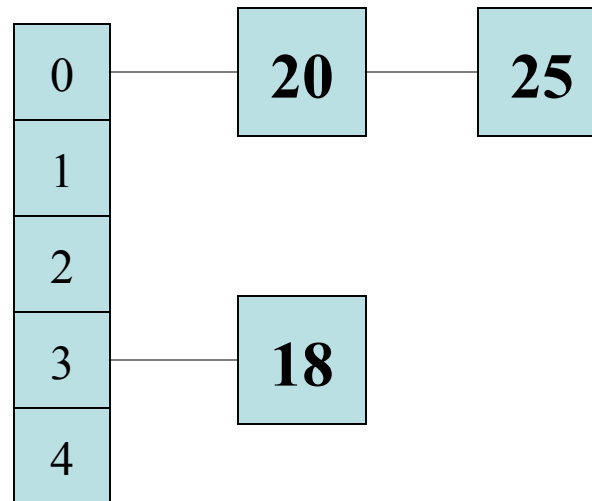
No **encadeamento separado**, a posição de inserção não muda, logo, todos devem ser inseridos na mesma posição, através de uma **lista encadeada**.

$$20 \% 5 = 0$$

$$18 \% 5 = 3$$

$$25 \% 5 = 0$$

colisão com 20



Encadeamento Separado

A tabela *hash*, neste caso, contém um array de listas:

```
class TabelaHash {  
    Lista[] listas;  
  
    public TabelaHash(int n) {  
        listas = new Lista[n];  
        for (int i = 0; i < n; i++)  
            listas[i] = new Lista();  
    }  
}
```

Encadeamento Separado

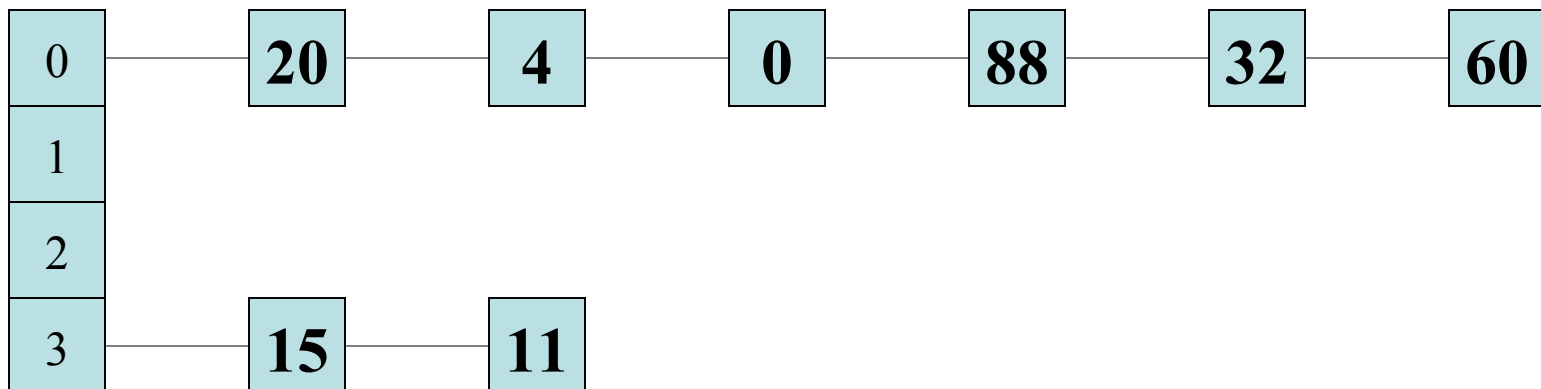
Quando uma chave for inserida, a função *hash* é aplicada, e ela é acrescentada à lista adequada:

```
int hashCode(int chave) {  
    return ...;           // função hash  
}  
  
void inserir(int chave) {  
    int i = hashCode(chave);  
    listas[i].inserir(chave);  
}
```

Encadeamento Separado

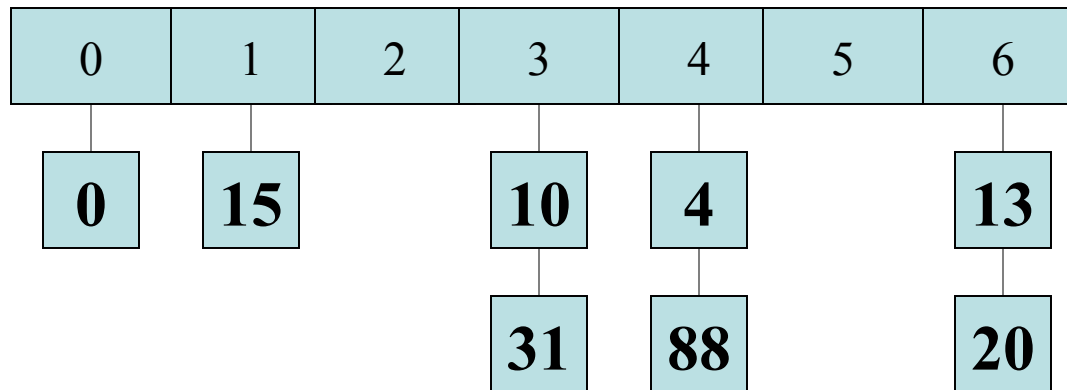
A busca é feita do mesmo modo: calcula-se o valor da função hash para a chave, e a busca é feita na lista correspondente.

Se o tamanho das listas variar muito, a busca pode se tornar ineficiente, pois a busca nas listas é seqüencial:



Encadeamento Separado

Por esta razão, a função *hash* deve distribuir as chaves entre as posições **uniformemente**:

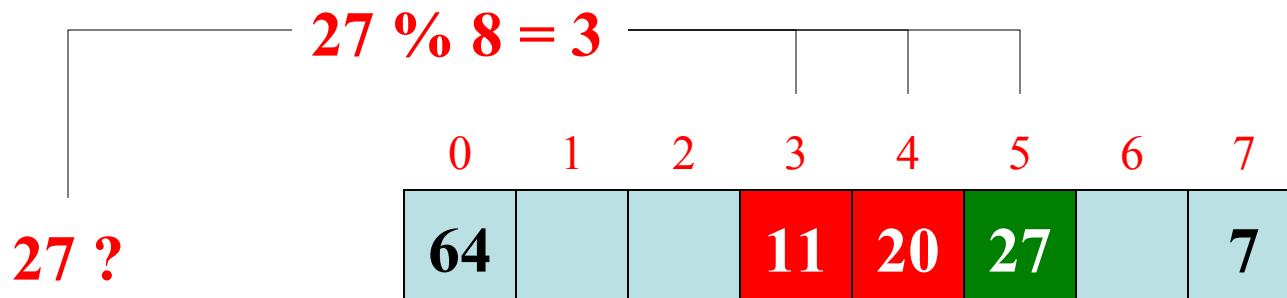


Se o tamanho da tabela for um número **primo**, há mais chances de ter uma melhor distribuição.

Endereçamento Aberto

No **endereçamento aberto**, quando uma nova chave é mapeada para uma posição já ocupada, uma nova posição é indicada para esta chave.

Com *linear probing*, a nova posição é incrementada até que uma posição vazia seja encontrada:



Endereçamento Aberto

A tabela *hash*, neste caso, contém um array de objetos, e posições vazias são indicadas por **null**. Neste caso, os objetos serão do tipo **Integer**:

```
class TabelaHash {  
    Integer[] posicoes;  
  
    public TabelaHash(int n) {  
        posicoes = new Integer[n];  
    }  
}
```


Endereçamento Aberto

Linear Probing

Na inserção, a função *hash* é calculada, e a posição incrementada, até que uma posição esteja livre:

```
void inserir(int chave) {  
    int i = hashCode(chave) ;  
  
    while (posicoes[i] != null)  
        i = (i + 1) % posicoes.length;  
  
    posicoes[i] = new Integer(chave) ;  
}
```

Endereçamento Aberto

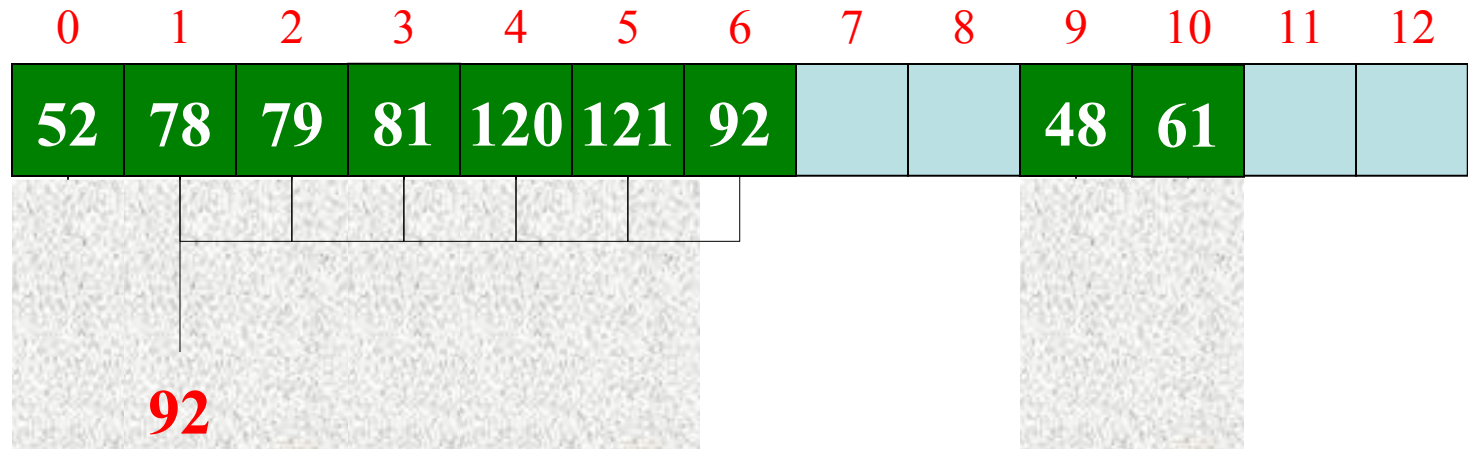
Linear Probing

Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92

Função: $\text{hash}(k) = k \% 13$

Tamanho da tabela: 13

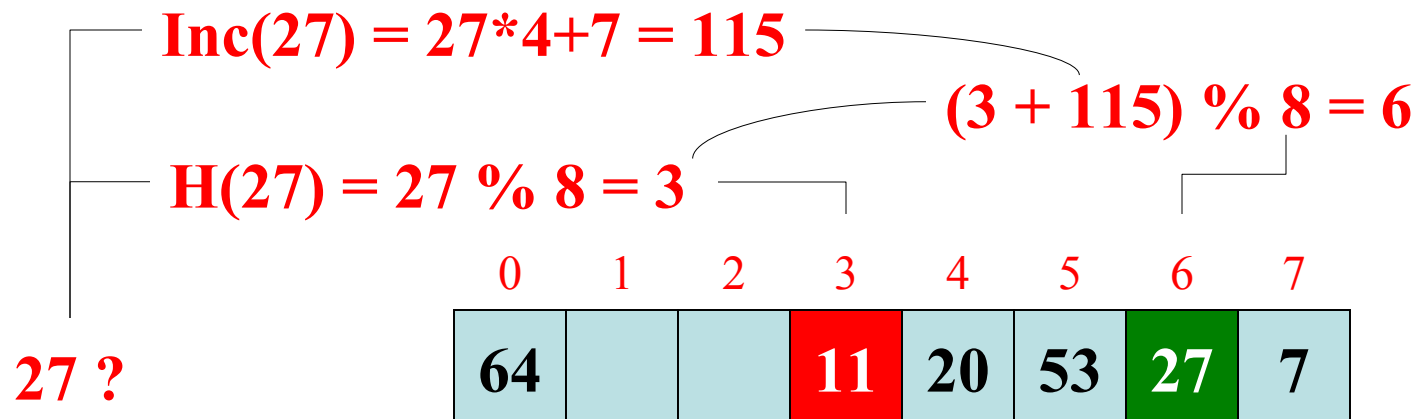
0	1	2	3	4	5	6	7	8	9	10	11	12
52	78	79	81	120	121	92			48	61		



Endereçamento Aberto

Double Hashing

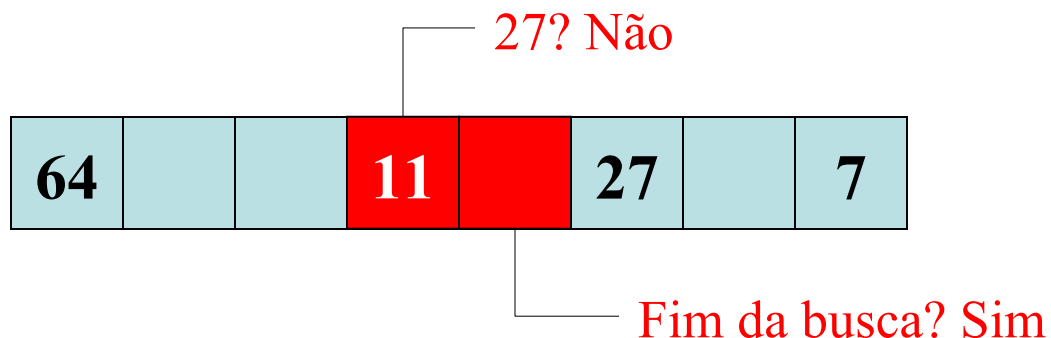
Outra política de endereçamento aberto é o chamado *double hashing*: ao invés de incrementar a posição de 1, uma função auxiliar é utilizada para calcular o incremento. Esta função também leva em conta o valor da chave.



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:



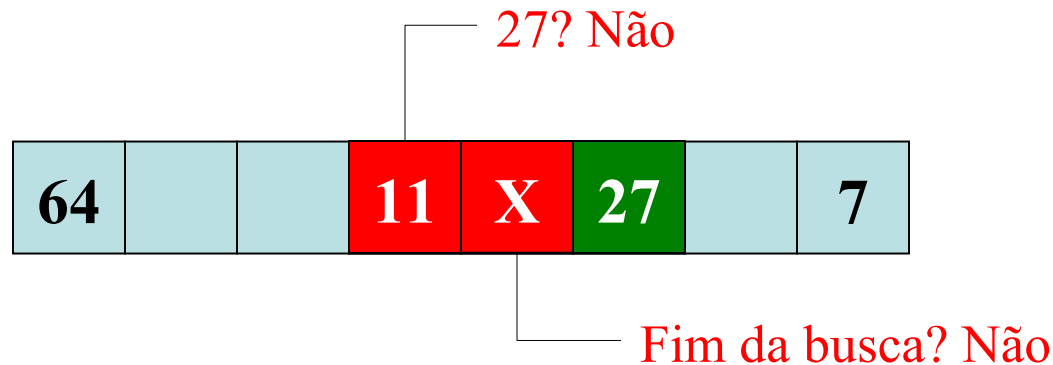
Inserção do 27

Remoção do 20

Busca pelo 27

Endereçamento Aberto : Remoção

Para contornar esta situação, mantemos um bit (ou um campo **booleano**) para indicar que um elemento foi removido daquela posição:

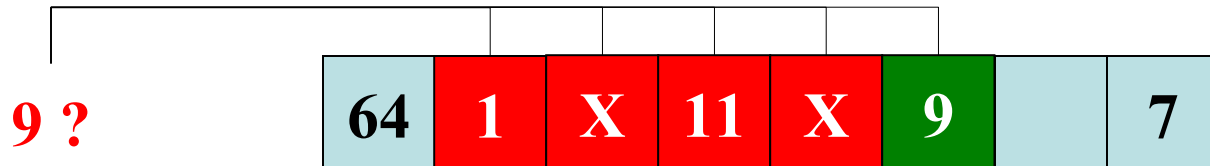


Esta posição estaria livre para uma nova **inserção**, mas não seria tratada como vazia numa **busca**.

Endereçamento Aberto : Expansão

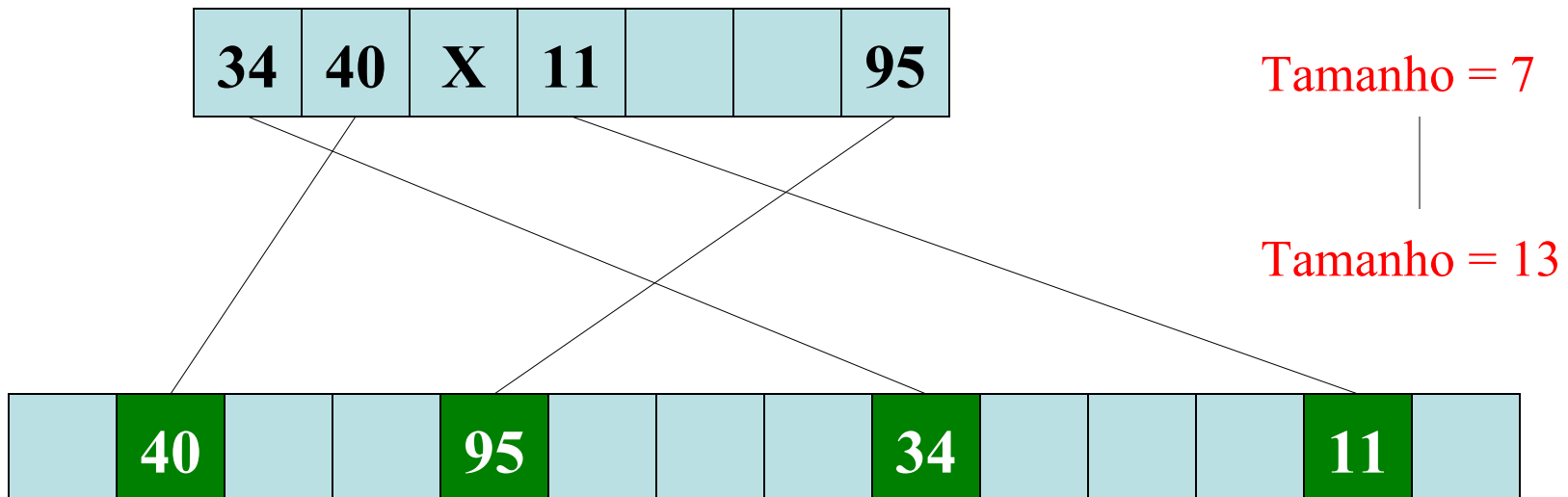
Nesta política de *hashing*, há que chamamos de **fator de carga** (*load factor*). Ele indica a porcentagem de células da tabela *hash* que estão ocupadas, incluindo as que foram removidas.

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. Como podemos ver, o tamanho atual da tabela passa a ser um parâmetro da função *hash*.



Endereçamento Aberto : Expansão

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*. Para isso é necessário escolher bem:

- a função *hash*
- o tratamento de colisões
- o tamanho da tabela

Quando não for possível definir parâmetros eficientes, pode ser melhor utilizar árvores balanceadas (como AVL), em vez de tabelas *hash*.

Referências

- **Gregory L. Heilemann: Data structures, algorithms, and object-oriented programming. McGraw-Hill, Computer Science Press, 1996.**
- **James F. Korsh & Leonard J. Garrett: Data Structures, Algorithms, and Program Style Using C. PWS-Kent, 1988.**
- **Cormen, T. T., Leiserson, C. E., and Rivest, R. L.: Introduction to Algorithms. MIT Press, 1990.**
- **Ellis Horowitz & Sartaj Sahni: Fundamentals of Data Structures. Computer Science Press, 1983.**