

Prof. Hélder Pereira Borges

helder@ifma.edu.br

ALGORITMOS E ESTRUTURAS DE DADOS II

Tabelas Hash



IFMA

Departamento de Computação

Introdução

- A quantidade de dados disponíveis **cresce exponencialmente** a cada dia, dificultando um armazenamento e busca eficaz de todos eles
 - Na programação do dia a dia, essa quantidade pode ainda não ser tão grande, mesmo assim precisa ser armazenada, acessada e processada de maneira fácil e eficiente
- Uma estrutura de dados muito comum usada para esse fim é a estrutura de dados **Array**, presente em quase todas LP
- Embora Arrays sejam muito utilizados, dada a crescente quantidade de dados, se faz necessário mecanismos + eficientes

- Em uma estrutura de dados a **eficiência** das operações fundamentais é crítica

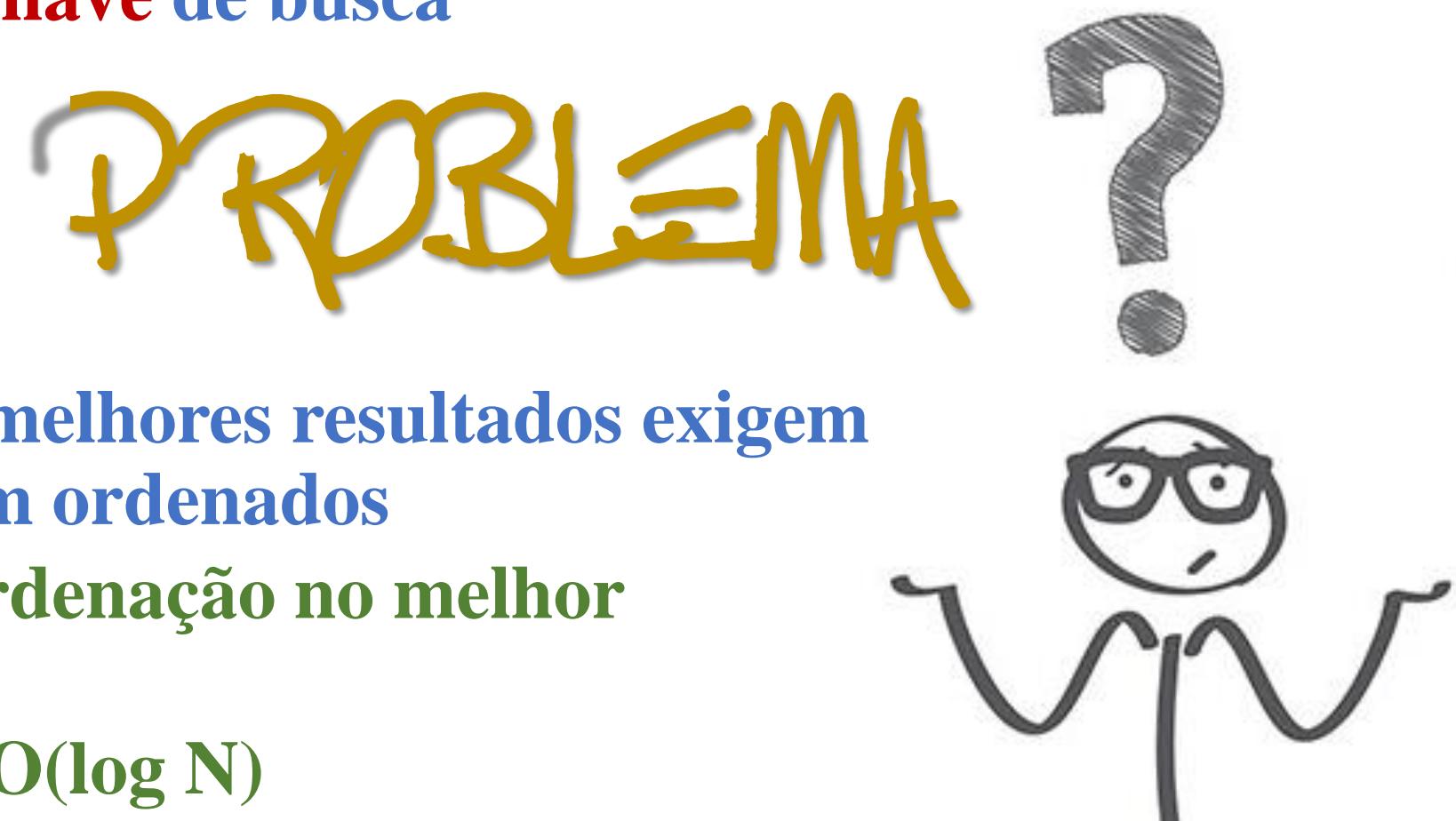
INSERÇÃO, BUSCA E REMOÇÃO

- Um **Array**, apesar de ser uma estrutura elementar, é uma excelente opção, eficiente para diversos cenários
 - Fornece acesso, inserção e remoção em tempo $O(1)$
 - Isto se o índice em questão for conhecido
 - O custo da operação é dado pela operação primitiva

Introdução

... estudar e praticar ... the best way for everything

- Usualmente, o processo de recuperação de informações ocorre através da utilização de **métodos de busca**, geralmente baseados em vasculhar o conjunto de dados, comparando cada **chave** dos registros com uma **chave** de busca



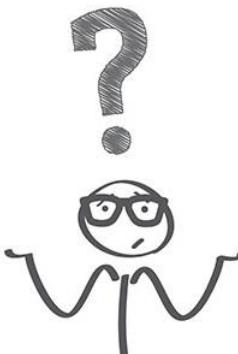
Uma busca ideal deveria
realizar um **acesso direto**
ao **registro**, sem as
comparações,
representando um **tempo**
constante $O(1)$

Introdução

- Um **vetor** é uma estrutura de dados que permite armazenar e recuperar as informações através de **índices**

- Custo do acesso: $O(1)$

- Porém, normalmente, vetores **não dispõem** de mecanismos para **identificar/calcular** o índice onde uma determinada informação está armazenada



- Na prática, o custo da busca em um vetor **não é $O(1)$**

- **O acesso pode ser, mas a busca não**



- O uso de uma **Tabela Hash** tem o objetivo de proporcionar uma busca na coleção com o **custo $O(1)$**



Exemplo

- Armazenar Objeto Aluno (matrícula e nome)
 - Matrícula é um inteiro entre 0 e 4999
- Um array seria uma estrutura adequada?
 - Armazenar o aluno na posição correspondente ao de sua matrícula
 - A inserção, a busca e a remoção desse objeto pode ser realizada em tempo $O(1)$
 - Existe uma correspondência direta entre o identificador único do aluno (a chave) e o seu índice no array

O array usado assim é denominado de
Tabela de Acesso Direto

Exemplo - Problemas

- Matrícula com 8 dígitos (aaaa-ss-pp)
 - Domínio teórico: [00000000, 99999999]
 - Seria preciso criar um Array com 10^8 posições
 - Qual a realidade do número de matrículas???
- E se a chave for um CPF (11 dígitos)
 - Além do desperdício de memória, é possível criar arrays de 10^{11} ??
- E se a chave não for representada por um inteiro
 - Ex: ABC1234

100.000.000

10^8

10^{11}

10.000.000.000

Tabela Hash

- Codinomes
 - Tabela de Espalhamento ou Tabela de Dispersão
 - O objetivo é que os registros sejam armazenados na estrutura a partir de um **endereçamento** gerado através de uma **transformação aritmética** considerando uma chave de pesquisa
 - A finalidade é alcançar uma **eficiência $O(1)$** nas operações de busca, inserção e remoção
 - O desempenho das operações de **inserção e remoção** não devem ser afetado pelas variações na quantidade de registros armazenados em uma coleção
- minimizar a complexidade de tempo em operações básicas

Tabela Hash

FUNÇÃO HASHING

- Utiliza uma função para **distribuir / espalhar** os elementos da coleção na estrutura de dados
 - Adiciona os elementos **dispersos** na estrutura, ficando estes de uma forma desordenada
- Em resumo, uma Tabela Hash é uma estrutura de dados que permite acesso aos elementos da coleção em tempo aproximado a $O(1)$, dada a utilização de uma **Função Hashing**



Tabela Hash

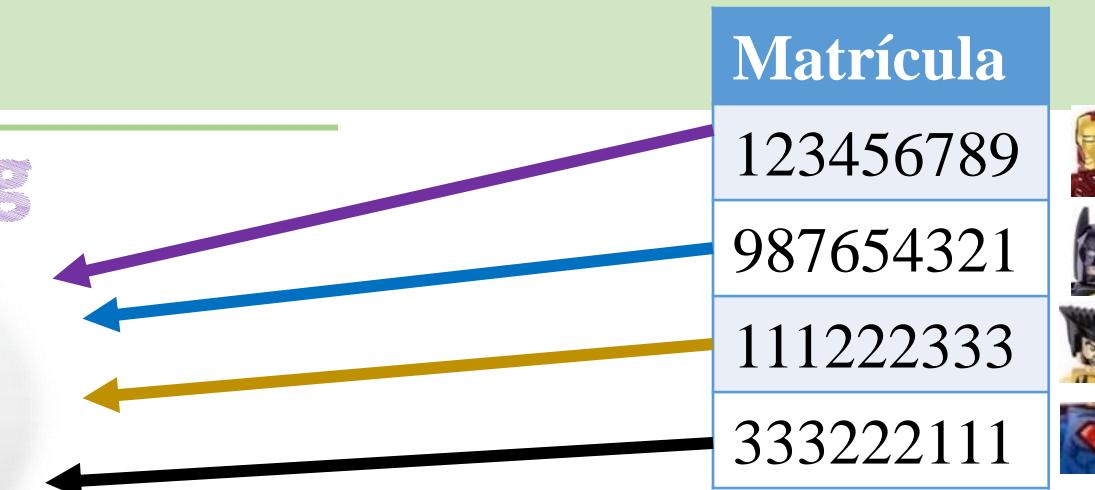
... estudar e praticar ... the best way for everything



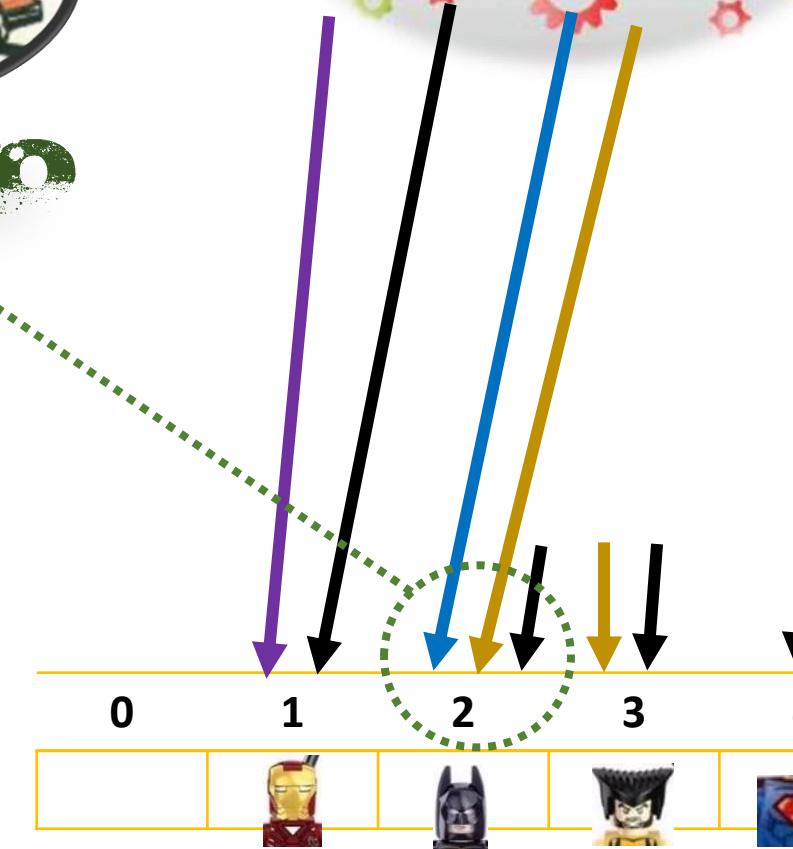
colisão

Estrutura
de Dados

Função Hashing



?? e agora ??
procurar uma vaga livre?



bucket

Tabela Hash

... estudar e praticar ... the best way for everything

- O espalhamento auxilia no processo de busca porque uma Tabela Hash trabalha com o conceito de associação entre **chaves e valores**
 - Chave
 - Parte da informação que identifica um elemento, tanto no momento de ser inserido, quanto em sua procura
 - Valor
 - Corresponde ao índice (**posição na estrutura**) onde o elemento se encontra ou deve ser armazenado, sendo calculado a partir da chave

Na média, o custo desta operação deve ser $O(1)$

Tabela Hash - Vantagens

- Ideal para implementar estruturas de dados de valor-chave
- Operações de inserção, exclusão e pesquisa são eficientes
- Redução do uso de memória, pois aloca um espaço fixo para armazenar elementos
- Escalabilidade, ainda que uma coleção fique muito grande, é mantido o tempo de acesso constante
- Armazenamento seguro e verificação de integridade, sendo de grande valia para segurança e criptografia
- Implementação relativamente simples

Tabela Hash - Desvantagens

- **Desvantagens**
 - Custo elevado caso seja necessário ordenar os elementos pela chave
 - Pior caso $O(n)$, onde **n** é o tamanho da coleção
 - Existência de colisões
 - Dois elementos querendo ocupar a mesma posição na estrutura de dados

- **Indexação de banco de dados**
 - Hashing é usado para indexar e recuperar dados de forma eficiente em bancos de dados e outros sistemas de armazenamento de dados
- **Armazenamento de senha**
 - Hashing é usado para armazenar senhas com segurança, aplicando uma função de hash à senha e armazenando o resultado com hash, em vez da senha de texto
- **Compressão de dados**
 - Hashing é usado em algoritmos de compressão de dados, como o algoritmo de codificação de Huffman, para codificar dados com eficiência

- **Algoritmos de pesquisa**
 - Hashing é usado para implementar algoritmos de pesquisa, como filtros bloom para pesquisas e consultas rápidas
- **Criptografia**
 - Hashing é usado em criptografia para gerar assinaturas digitais, códigos de autenticação de mensagem (MACs) e funções de derivação de chave
 - **MD5 e a família Secure Hash Algorithm (SHA)**
- **Balanceamento de carga**
 - Hashing é usado em algoritmos de balanceamento de carga, como hash consistente, para distribuir solicitações a servidores em uma rede

- **Blockchain**
 - Hashing é usado na tecnologia blockchain, como o protocolo **PoW** (prova de trabalho), para garantir a integridade e o consenso do blockchain
- **Processamento de imagem**
 - Hashing é usado em aplicativos de processamento de imagem, como hash perceptivo, para detectar e evitar duplicações e modificações de imagem
- **Comparação de arquivos**
 - Hashing é usado em algoritmos de comparação de arquivos, como as funções de hash MD5 e SHA-1, para comparar e verificar a integridade dos arquivos

- **Detecção de fraude**
 - Hashing é usado em aplicações de detecção de fraude e segurança cibernética, como detecção de intrusão e software antivírus, para detectar e prevenir atividades maliciosas
- **Compiladores**
 - Implementação da tabela de símbolos

Função Hashing / Função de Mapeamento

- Utilizada para calcular a posição do objeto na estrutura
 - Calcula utilizando uma chave como parâmetro principal
 - A chave é definida a partir de um atributo do objeto
 - Uma boa função hash deve satisfazer a hipótese de **hash uniforme**, pelo menos aproximadamente, garantindo uma distribuição uniforme entre os índices disponíveis
 - Cada chave tem igual probabilidade de ser mapeada para qualquer uma das posições na estrutura
 - Extremamente relevante para o desempenho da Tabela Hash



Função de Hashing (id aluno)

posição na
estrutura

Função Hashing / Função de Mapeamento

- Requisitos
 - Preferencialmente **simples** e de **baixo custo computacional** para não comprometer o desempenho
 - Garantir que valores distintos produzam índices diferentes, tanto quanto possível
 - Pode ser dependente da natureza e domínio da chave utilizada
 - Importante conhecer o tipo de dado da chave para definir a melhor função

- O endereço de armazenamento ou busca será o **resto da divisão da chave pelo tamanho da estrutura**
 - A chave (k) deve ser um número inteiro
 - $h(k) = k \ mod \ M + 1$ para $m_i \in [1, M]$
 - $h(k) = k \ mod \ M$ para $m_i \in [0, M-1]$
 - O resultado desta operação estará sempre no intervalo $[0, M-1]$ ou $[1, M]$
 - Exemplo considerando um vetor com $M = 500 \gg [0, 499]$

O uso de números primos para definir o tamanho da estrutura **melhora a distribuição** dos elementos porque dificulta a formação de padrões nas posições

Chave	Índice
12768	268
17	17
968	468
1974	474

- **Vantagens**
 - Fácil de computar
 - Rápida
- **Desvantagens**
 - Função depende do valor de M
 - M deve ser um número primo
 - Alta probabilidade de colisões

Dica: mod em java = %

- Solução baseada em duas etapas

- Multiplicar a chave k por uma constante A , onde $A \in [0, 1]$ e calcular o resto da divisão por 1
 - A literatura sugere a utilização de $A = 0.6180339887$
- Extrair o piso, (arredondamento para baixo), da multiplicação da fração por um valor h
 - A escolha de h não é crítica
 - Para facilitar a implementação, o valor de h deve ser uma potência de 2

$$h(k) = \lfloor h * (k * A \bmod 1) \rfloor$$

$h = 64$

chave	Índice
1000	2
123456	0
9999	46
10	11

- Uma Função Hash, usualmente é **determinística**, desta forma, poderia ser **manipulada** para degradar o desempenho do sistema
 - Ex: definir um conjunto de chaves em que todas colidam
- A ideia consiste na utilização de **parâmetros** distintos a cada execução, assim, apesar da mesma entrada, os endereços na estrutura serão distintos

$$h_u(k) = (|a*k + b| \bmod p) \bmod m$$

- Onde:
 - **m** é o tamanho da estrutura e **k** é a chave
 - **p** é um número primo maior que **m**
 - **a** e **b** são constantes aleatórias; intervalo [0, ..., p-1] para **a** e intervalo [1, ..., p-1] para **b**

- Garantia de não haver colisões entre as chaves
 - Chaves distintas sempre geram índices diferentes
 - No pior caso, tanto para inserir quanto para buscar, a operação será realizada em tempo constante $O(1)$
 - Utilizado apenas em sistemas críticos, nos quais a colisão não é tolerável devido as implicâncias
 - Aplicações específicas, como por exemplo, dicionário de palavras reservadas de uma linguagem de programação
 - A restrição é que se faz necessário **conhecer previamente TODAS** as chaves que serão utilizadas

Hash Imperfeito

- É a estrutura mais comumente encontrada
- Dado duas chaves distintas, o resultado da função poderá ser o mesmo, indicando o mesmo endereço na estrutura de dados
- A colisão não necessariamente é um erro
 - É indesejada porque degrada o desempenho do sistema
- Uma distribuição estritamente uniforme é complexa e computacionalmente custosa, degradando o desempenho
- Usualmente, diversos tipos de tabelas Hash utilizam estruturas auxiliares para tratar o evento da colisão

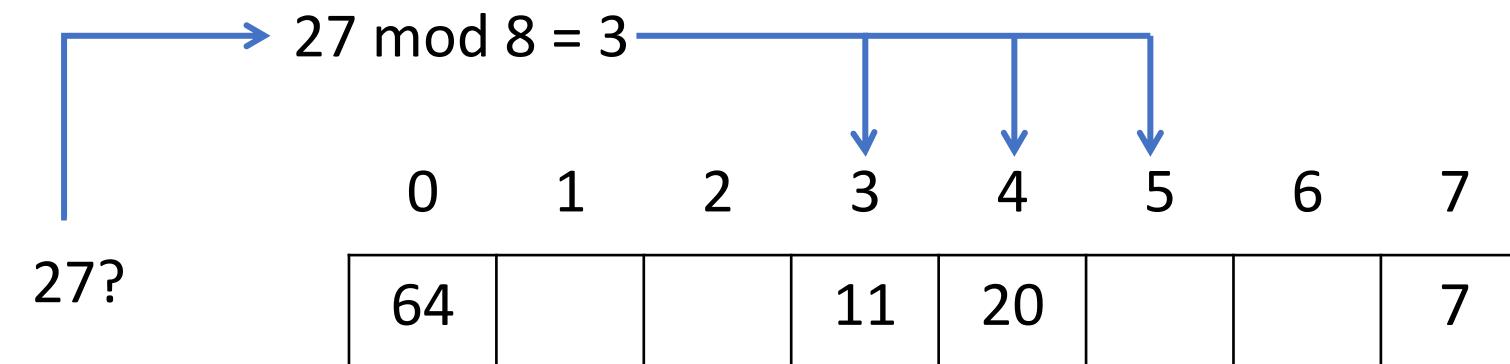
Tratamento de Colisões

- Na prática, o uso de uma Tabela Hash pressupõe dois aspectos
 - Uma função Hash
 - Precisa prover um espalhamento o mais uniforme possível
 - A definição adequada desta função e do tamanho da estrutura, naturalmente pode diminuir as colisões
 - Uma abordagem para tratar colisões
 - Frequentemente se tem mais chaves do que índices de armazenamento, logo irão ocorrer colisões que também são uniformemente distribuídas
 - Colisões são teoricamente inevitáveis

Tratamento de Colisões: Endereçamento Aberto

• Sondagem Linear

- Procurar a próxima posição vazia, depois do endereço calculado a partir da chave
- Tenta espalhar os elementos de forma sequencial
- Vantagem: simplicidade
- Desvantagem: a inserção e busca passam a ter desempenho $O(n)$



Tratamento de Colisões: Endereçamento Aberto

- Sondagem Linear
 - Vantagem
 - Simplicidade
 - Desvantagem
 - Suscetível a um agrupamento primário
 - São construídas longas sequências de posições ocupadas
 - Degrada o desempenho, a inserção e busca passam a ter desempenho $O(n)$

Tratamento de Colisões: Endereçamento Aberto

- **Sondagem Quadrática**

- Distribui os elementos a partir de uma equação do 2º grau

$$posicao = ph + (c_1 * i) + (c_2 * i^2)$$

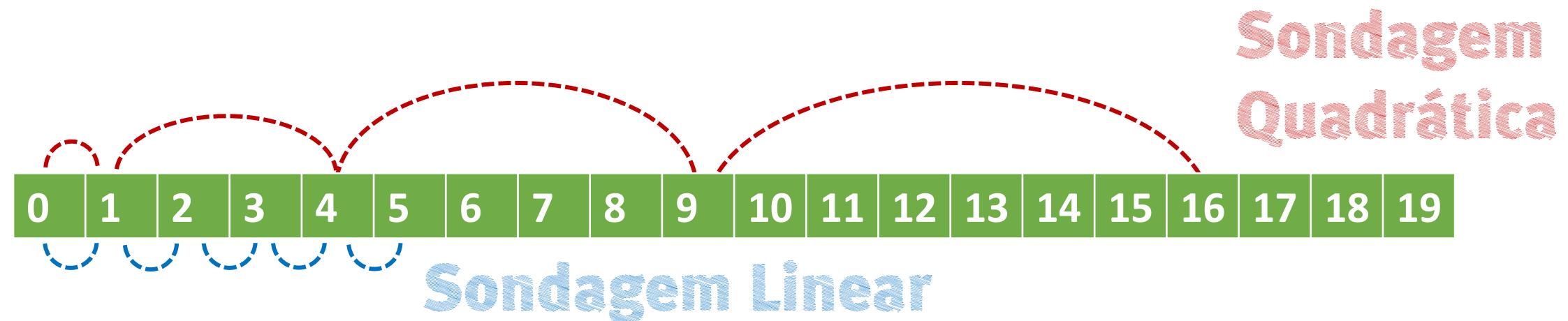
ph = posição inicial, obtida pela função de hashing;
i é tentativa atual; c_1 e c_2 são os coeficientes da equação

- 1º elemento, ($i=0$), fica na posição da função
- 2º elemento, ($i=1$), $ph + (c_1 * 1) + (c_2 * 1^2)$
- 3º elemento, ($i=2$), $ph + (c_1 * 2) + (c_2 * 2^2)$
-

Tratamento de Colisões: Endereçamento Aberto

- **Sondagem Quadrática**

- Resolve o agrupamento primário, porém gera um agrupamento secundário
 - As chaves que geram a mesma posição inicial, produzem as mesmas posições na sondagem quadrática
 - Com agrupamentos secundários a degradação no desempenho é menor que no primário



Tratamento de Colisões: Endereçamento Aberto

- **Sondagem Espalhamento Duplo / Duplo Hash**
 - Distribui os elementos usando **duas** funções hash distintas
 - H1: usada para calcular a **posição inicial**
 - H2: usada para calcular os **deslocamentos** em relação a posição inicial no caso de colisão

$$posicao = H1 + i * H2$$

i é a tentativa corrente

- Resultado de H2 nunca pode ser zero
 - 1º elemento, (i=0) >> fica na posição dada por H1
 - 2º elemento, (i=1) >> $H1 + 1 * H2$
 - 3º elemento, (i=2) >> $H1 + 2 * H2$
 -

Tratamento de Colisões: Endereçamento Fechado

- Utiliza uma lista encadeada para cada endereço da tabela
- Vantagem
 - Não é preciso recalcular endereços
 - Sinônimos são encontrados em uma única busca
 - A inserção continua sendo realizada em tempo constante para listas não ordenadas
- Desvantagem
 - A busca consumirá tempo proporcional ao tamanho da lista
 - Tratamento em listas pode consumir tempo $O(n)$

Tratamento de Colisões: Endereçamento Fechado

- Exemplificando
 - Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

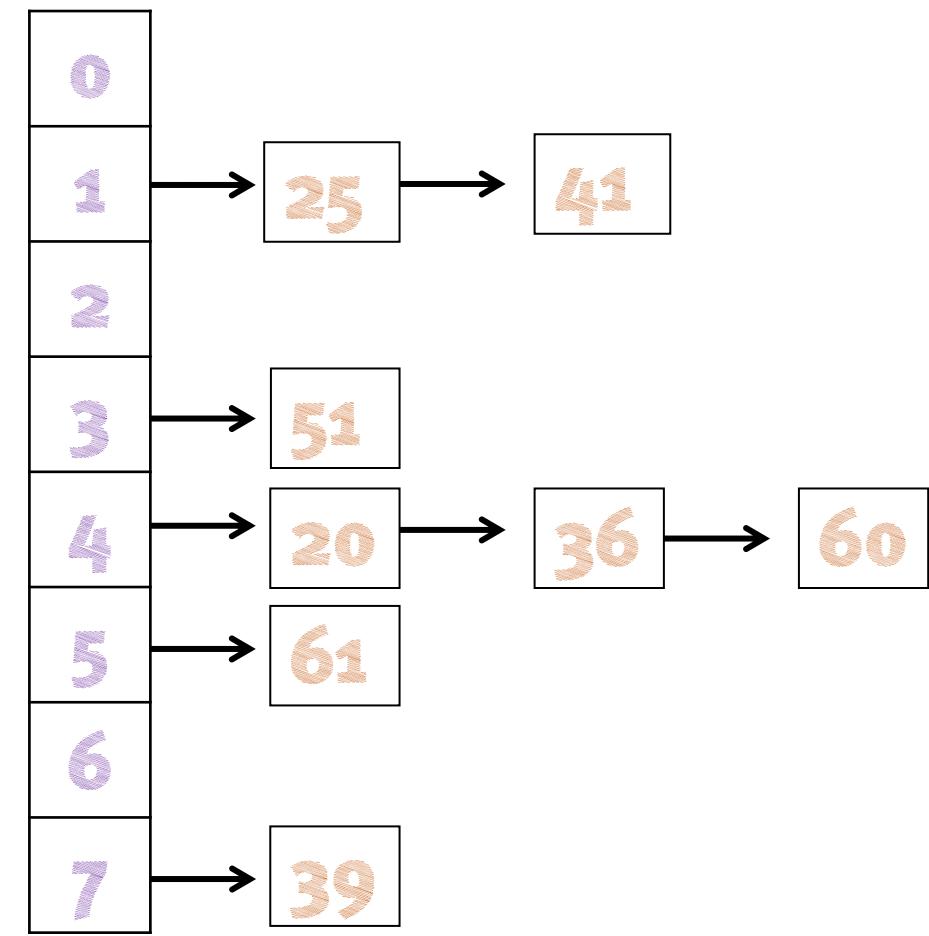
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Fator de Carga / Load Factor

- Estatística que verifica a proporção de chaves dentro do array

$$fc = c / n$$

onde c = qtd chaves inseridas e n = tamanho array

- O valor de fc varia entre 0.0 e 1.0
 - Quanto maior o fc, maior probabilidade de colisão e degradação do tempo de execução
 - No encadeamento, haverão listas cada vez maiores
 - No endereçamento aberto, o máximo do fator é 1.0
 - Em ambas soluções haverá degradação do desempenho

Fator de Carga / Load Factor

- Resumindo, com o crescimento da qtd de dados, é inevitável que o array fique menor que o necessário
- Resize
 - ação para quando o número de elementos armazenados se aproxima do máximo permitido
- Rehash
 - para utilizar a nova estrutura é preciso reallocar cada elemento inserido anteriormente, irão para novas posições

fc pequeno » resize frequente

fc grande » aumento de colisões

Java utiliza 0,75 como fator de carga

Prof. Hélder Pereira Borges

helder@ifma.edu.br

ALGORITMOS E ESTRUTURAS DE DADOS II

Tabelas Hash



IFMA

Departamento de Computação

Prof. Hélder Pereira Borges

helder@ifma.edu.br

ALGORITMOS E ESTRUTURAS DE DADOS II

Tabelas Hash



IFMA

Departamento de Computação

Introdução

- A quantidade de dados disponíveis **cresce exponencialmente** a cada dia, dificultando um armazenamento e busca eficaz de todos eles
 - Na programação do dia a dia, essa quantidade pode ainda não ser tão grande, mesmo assim precisa ser armazenada, acessada e processada de maneira fácil e eficiente
- Uma estrutura de dados muito comum usada para esse fim é a estrutura de dados **Array**, presente em quase todas LP
- Embora Arrays sejam muito utilizados, dada a crescente quantidade de dados, se faz necessário mecanismos + eficientes

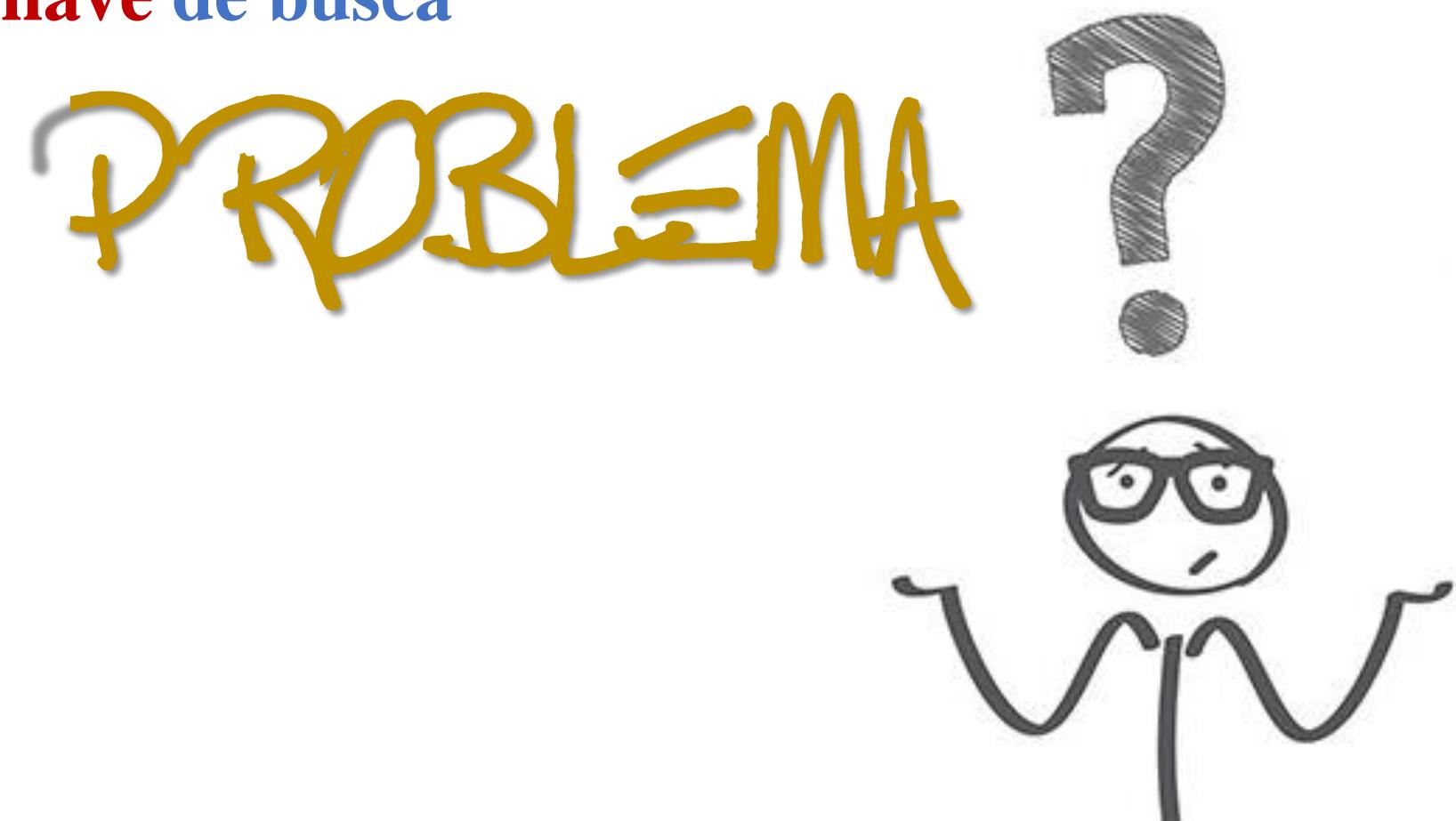
- Em uma estrutura de dados a **eficiência** das operações fundamentais é crítica

INSERÇÃO, BUSCA E REMOÇÃO

- Um **Array**, apesar de ser uma estrutura elementar, é uma excelente opção, eficiente para diversos cenários
 - Fornece acesso, inserção e remoção em tempo $O(1)$
 - Isto se o índice em questão for conhecido
 - O custo da operação é dado pela operação primitiva

Introdução

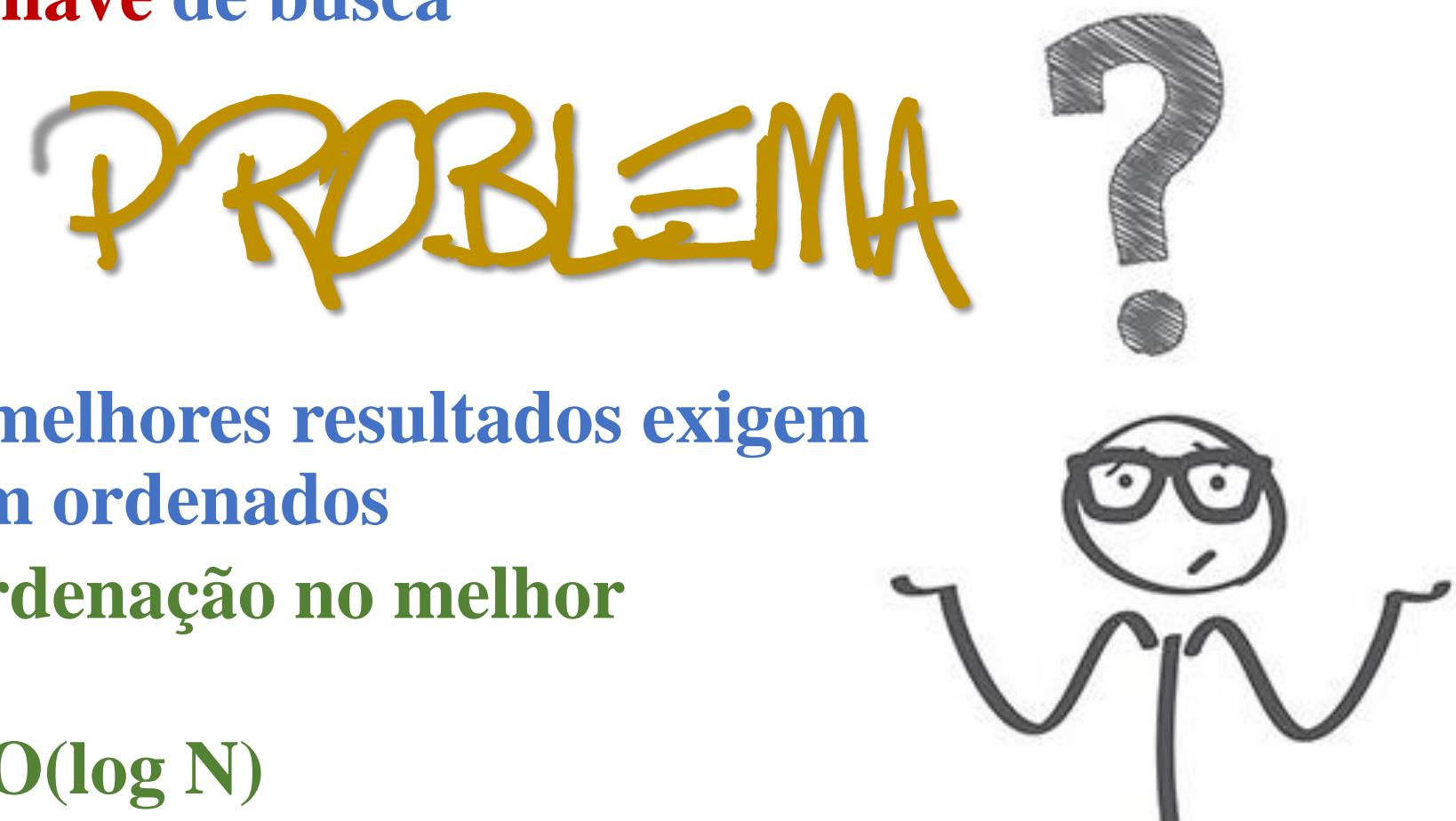
- Usualmente, o processo de recuperação de informações ocorre através da utilização de **métodos de busca**, geralmente baseados em vasculhar o conjunto de dados, comparando cada **chave** dos registros com uma **chave** de busca



Introdução

... estudar e praticar ... the best way for everything

- Usualmente, o processo de recuperação de informações ocorre através da utilização de **métodos de busca**, geralmente baseados em vasculhar o conjunto de dados, comparando cada **chave** dos registros com uma **chave** de busca



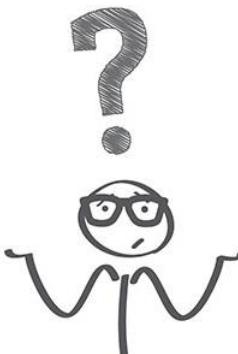
Uma busca ideal deveria
realizar um **acesso direto**
ao registro, sem as
comparações,
representando um tempo
constante **O(1)**

Introdução

- Um **vetor** é uma estrutura de dados que permite armazenar e recuperar as informações através de **índices**

- Custo do acesso: $O(1)$

- Porém, normalmente, vetores **não dispõem** de mecanismos para **identificar/calcular** o índice onde uma determinada informação está armazenada



- Na prática, o custo da busca em um vetor **não é $O(1)$**

- **O acesso pode ser, mas a busca não**



- O uso de uma **Tabela Hash** tem o objetivo de proporcionar uma busca na coleção com o **custo $O(1)$**



Exemplo

- Armazenar Objeto Aluno (matrícula e nome)
 - Matrícula é um inteiro entre 0 e 4999
- Um array seria uma estrutura adequada?
 - Armazenar o aluno na posição correspondente ao de sua matrícula
 - A inserção, a busca e a remoção desse objeto pode ser realizada em tempo $O(1)$
 - Existe uma correspondência direta entre o identificador único do aluno (a chave) e o seu índice no array

O array usado assim é denominado de
Tabela de Acesso Direto

Exemplo - Problemas

- Matrícula com 8 dígitos (aaaa-ss-pp)
 - Domínio teórico: [00000000, 99999999]
 - Seria preciso criar um Array com 10^8 posições
 - Qual a realidade do número de matrículas???
- E se a chave for um CPF (11 dígitos)
 - Além do desperdício de memória, é possível criar arrays de 10^{11} ??
- E se a chave não for representada por um inteiro
 - Ex: ABC1234

Exemplo - Problemas

- Matrícula com 8 dígitos (aaaa-ss-pp)
 - Domínio teórico: [00000000, 99999999]
 - Seria preciso criar um Array com 10^8 posições
 - Qual a realidade do número de matrículas???
- E se a chave for um CPF (11 dígitos)
 - Além do desperdício de memória, é possível criar arrays de 10^{11} ??
- E se a chave não for representada por um inteiro
 - Ex: ABC1234

100.000.000

Exemplo - Problemas

- Matrícula com 8 dígitos (aaaa-ss-pp)
 - Domínio teórico: [00000000, 99999999]
 - Seria preciso criar um Array com 10^8 posições
 - Qual a realidade do número de matrículas???
- E se a chave for um CPF (11 dígitos)
 - Além do desperdício de memória, é possível criar arrays de 10^{11} ??
- E se a chave não for representada por um inteiro
 - Ex: ABC1234

100.000.000

10^8

10^{11}

10.000.000.000

Tabela Hash

- Codinomes
 - Tabela de Espalhamento ou Tabela de Dispersão
 - O objetivo é que os registros sejam armazenados na estrutura a partir de um **endereçamento** gerado através de uma **transformação aritmética** considerando uma chave de pesquisa
 - A finalidade é alcançar uma **eficiência $O(1)$** nas operações de busca, inserção e remoção
 - O desempenho das operações de **inserção e remoção** não devem ser afetado pelas variações na quantidade de registros armazenados em uma coleção
- minimizar a complexidade de tempo em operações básicas

Tabela Hash

FUNÇÃO HASHING

- Utiliza uma função para **distribuir / espalhar** os elementos da coleção na estrutura de dados
 - Adiciona os elementos **dispersos** na estrutura, ficando estes de uma forma desordenada
- Em resumo, uma Tabela Hash é uma estrutura de dados que permite acesso aos elementos da coleção em tempo aproximado a $O(1)$, dada a utilização de uma **Função Hashing**



Tabela Hash

... estudar e praticar ... the best way for everything

Estrutura de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111

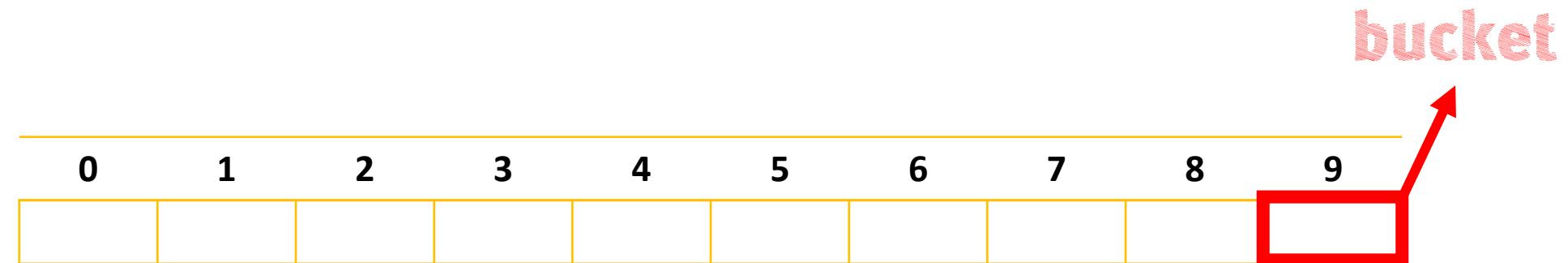


Tabela Hash

... estudar e praticar ... the best way for everything

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111

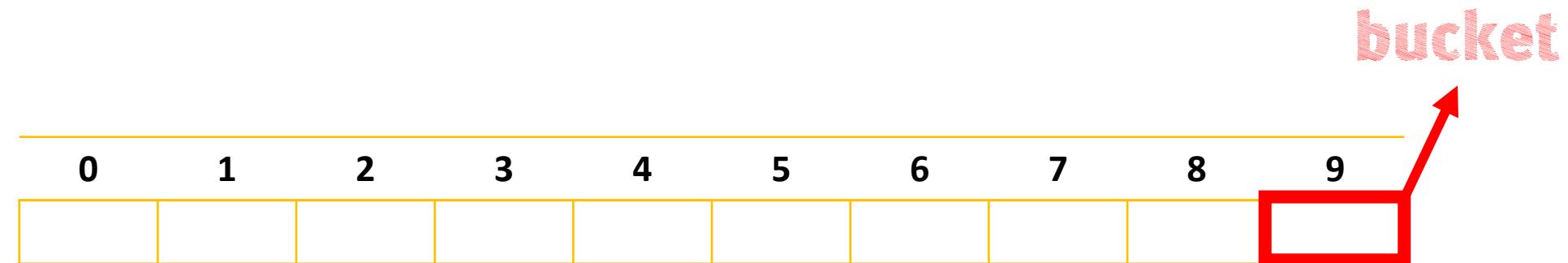


Tabela Hash

... estudar e praticar ... the best way for everything

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111



Tabela Hash

... estudar e praticar ... the best way for everything

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111



Tabela Hash

... estudare praticar ... (the best way) for everything

Estrutura de Dados



Matrícula

123456789
987654321
111222333
333222111

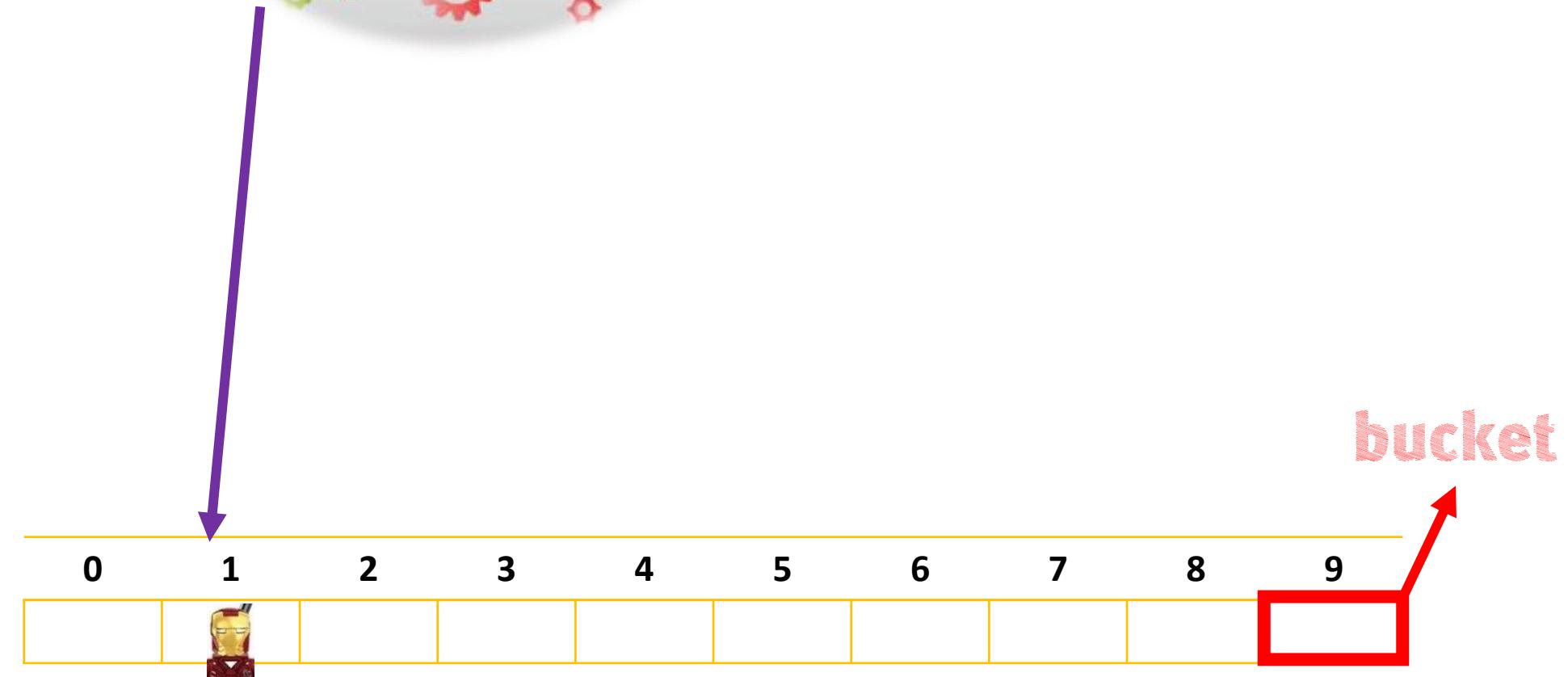


Tabela Hash

... estudar e praticar ... the best way for everything

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111

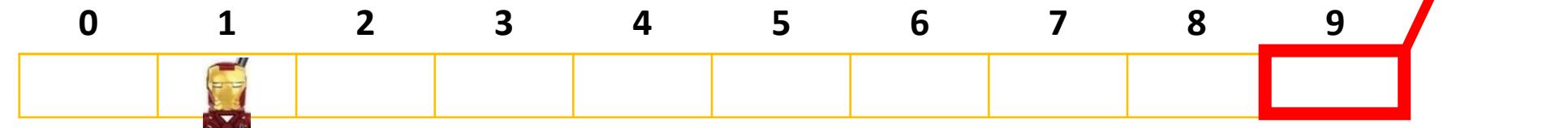


Tabela Hash

... estudar e praticar ... the best way for everything

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111

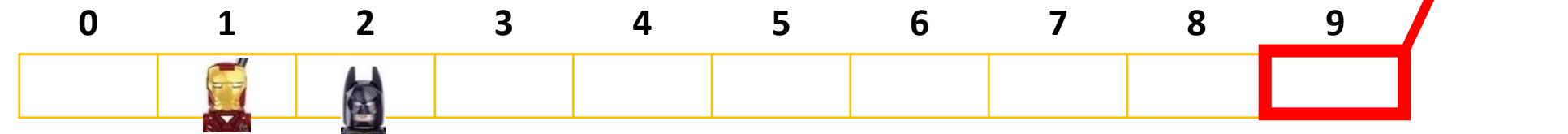


Tabela Hash

... estudar e praticar ... the best way for everything

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111

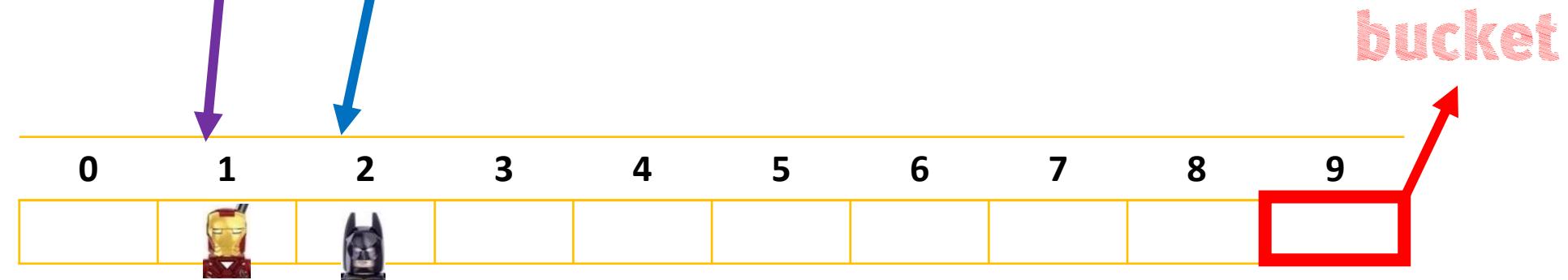
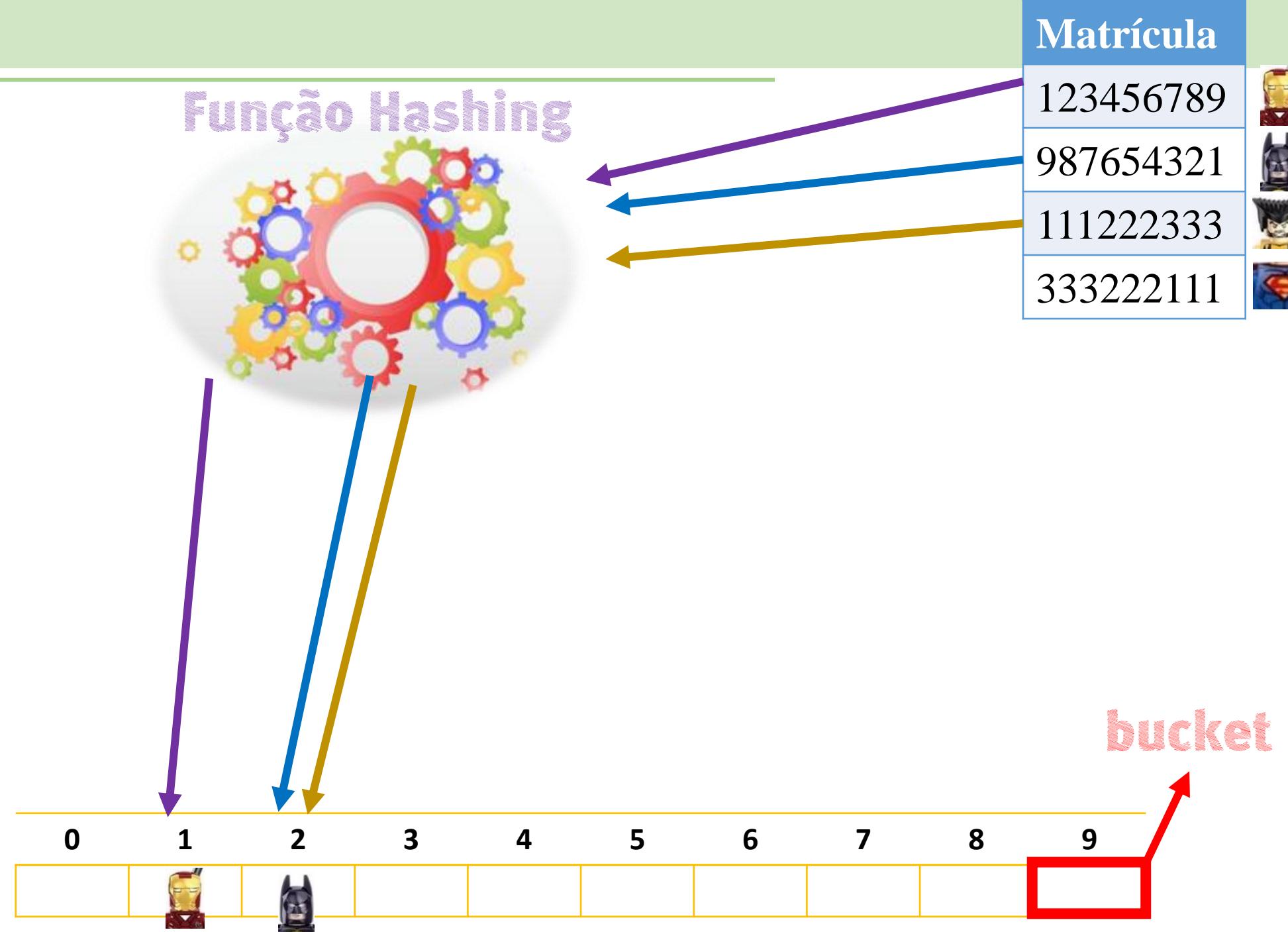


Tabela Hash

... estudar e praticar ... the best way for everything

Estrutura
de Dados

Função Hashing



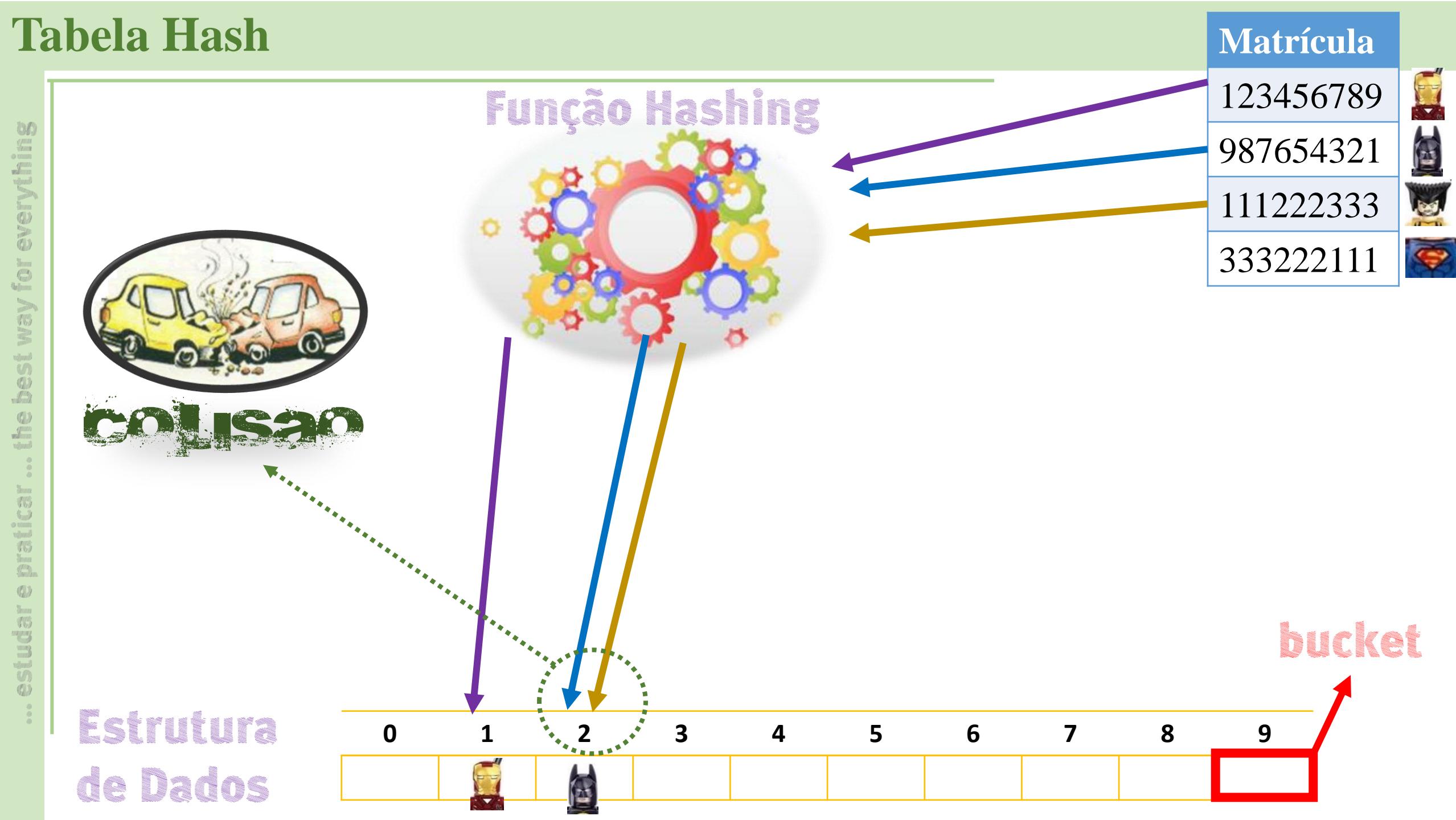


Tabela Hash

... estudar e praticar ... the best way for everything



colisão

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111



?? e agora ??

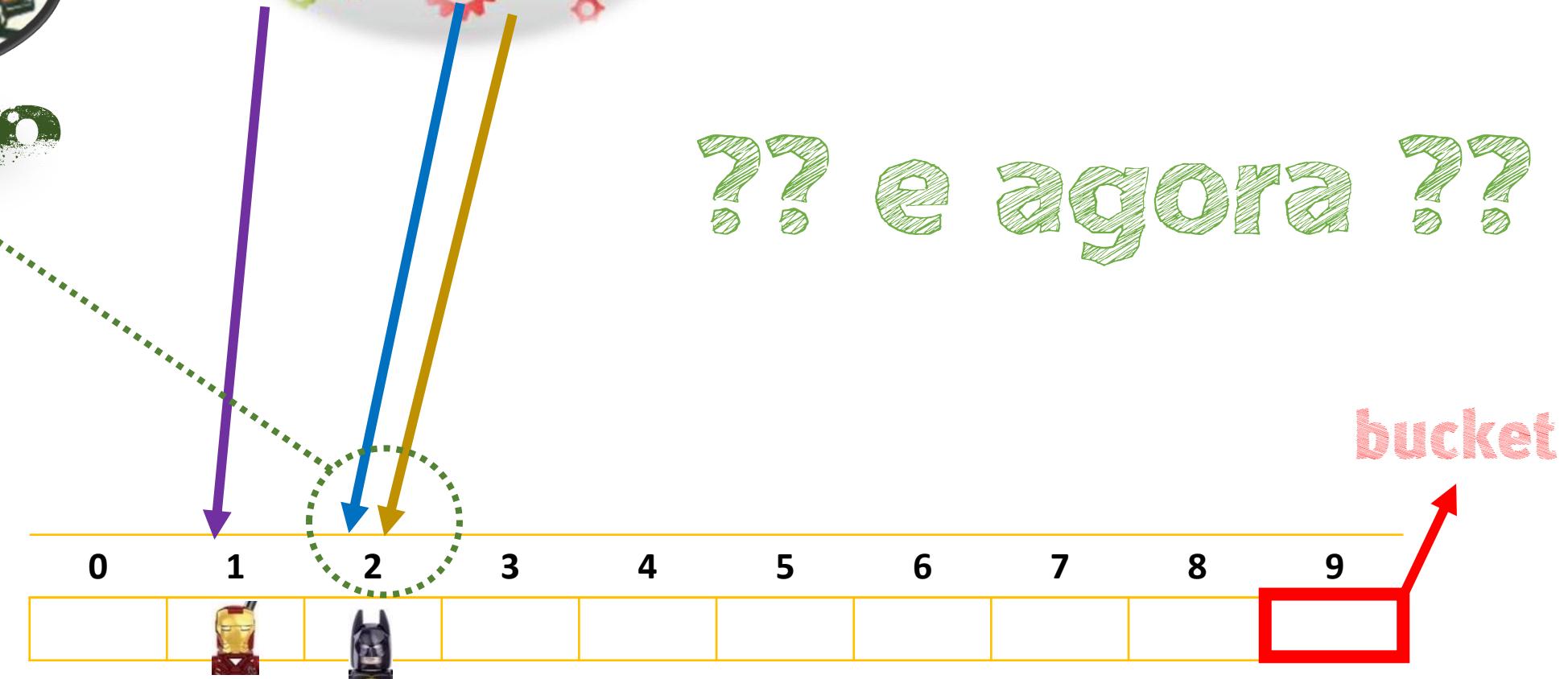


Tabela Hash

... estudar e praticar ... the best way for everything



colisão

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111



?? e agora ??
procurar uma vaga livre?

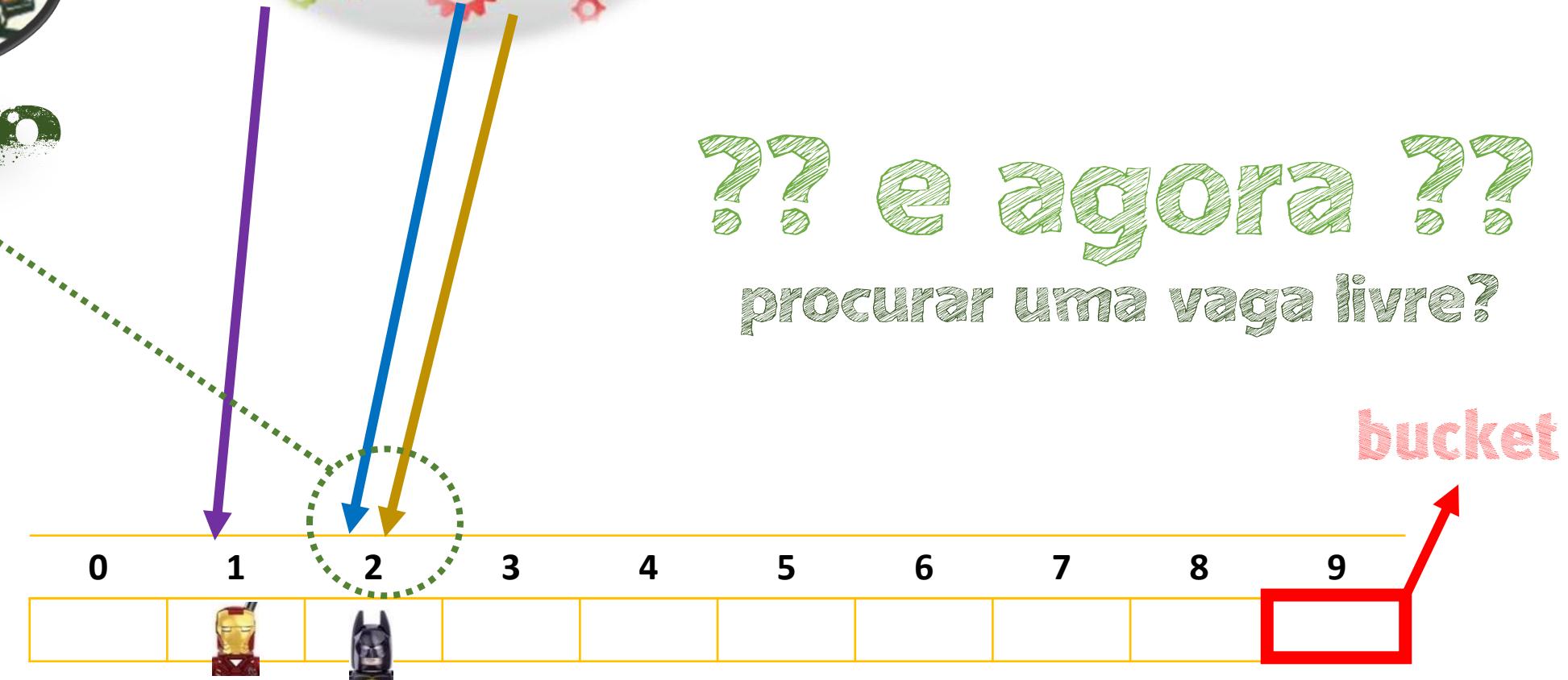


Tabela Hash

... estudar e praticar ... the best way for everything



colisão

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111



?? e agora ??
procurar uma vaga livre?

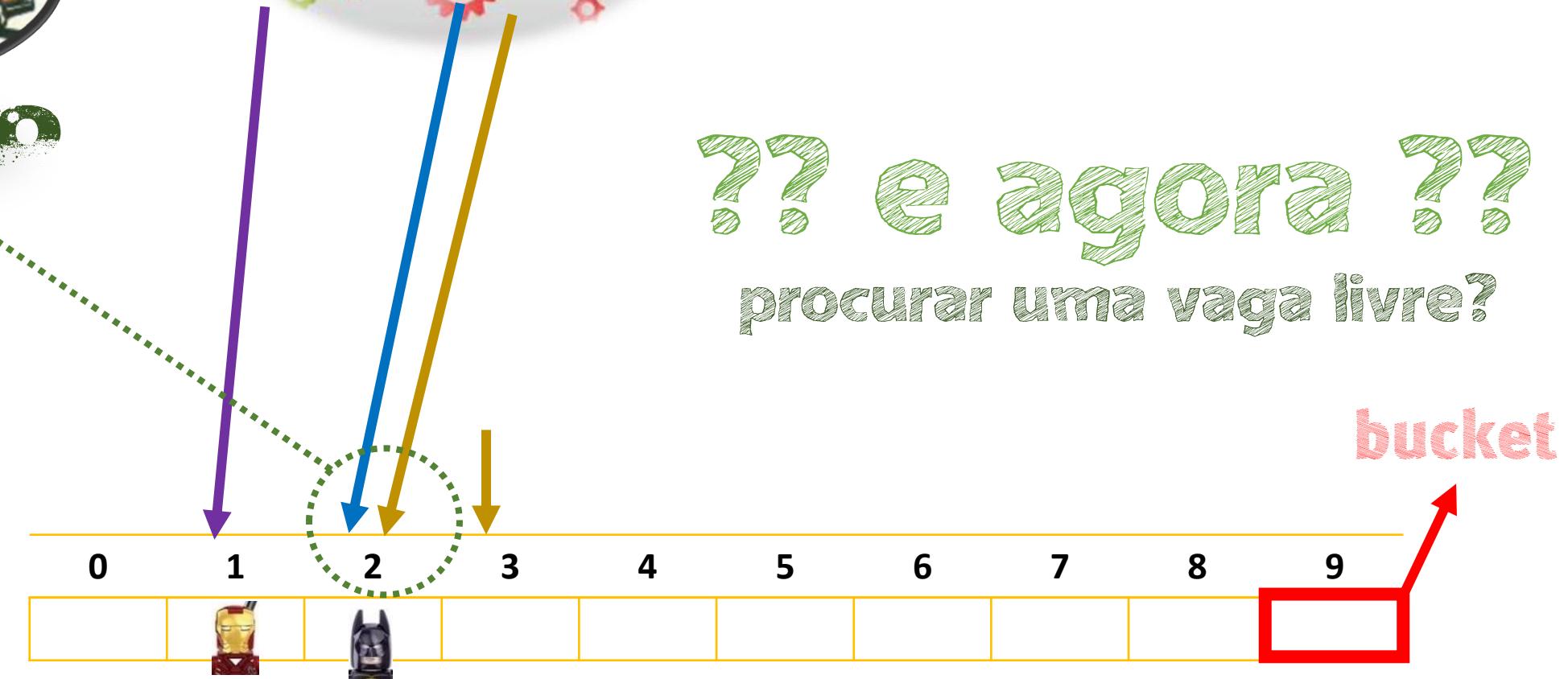


Tabela Hash

... estudar e praticar ... the best way for everything



colisão

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111



?? e agora ??
procurar uma vaga livre?

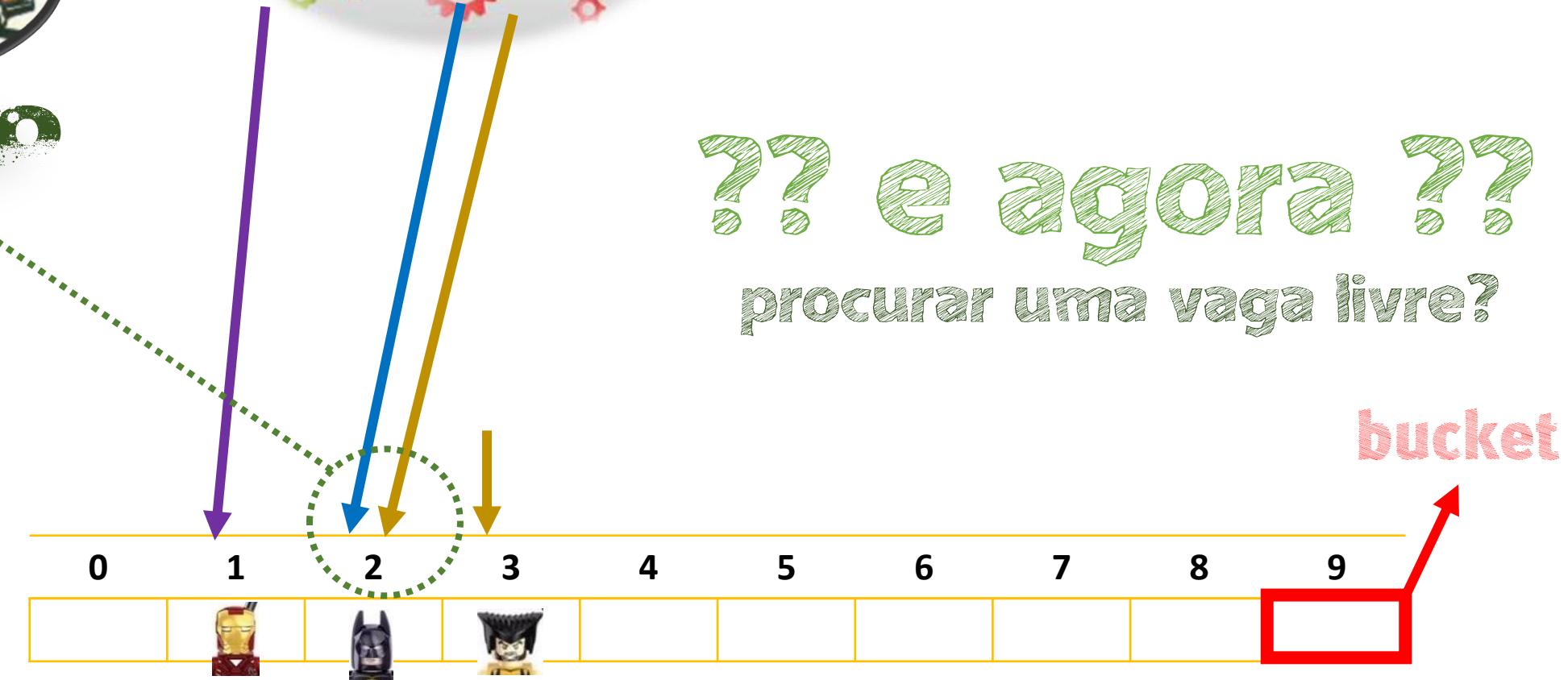


Tabela Hash

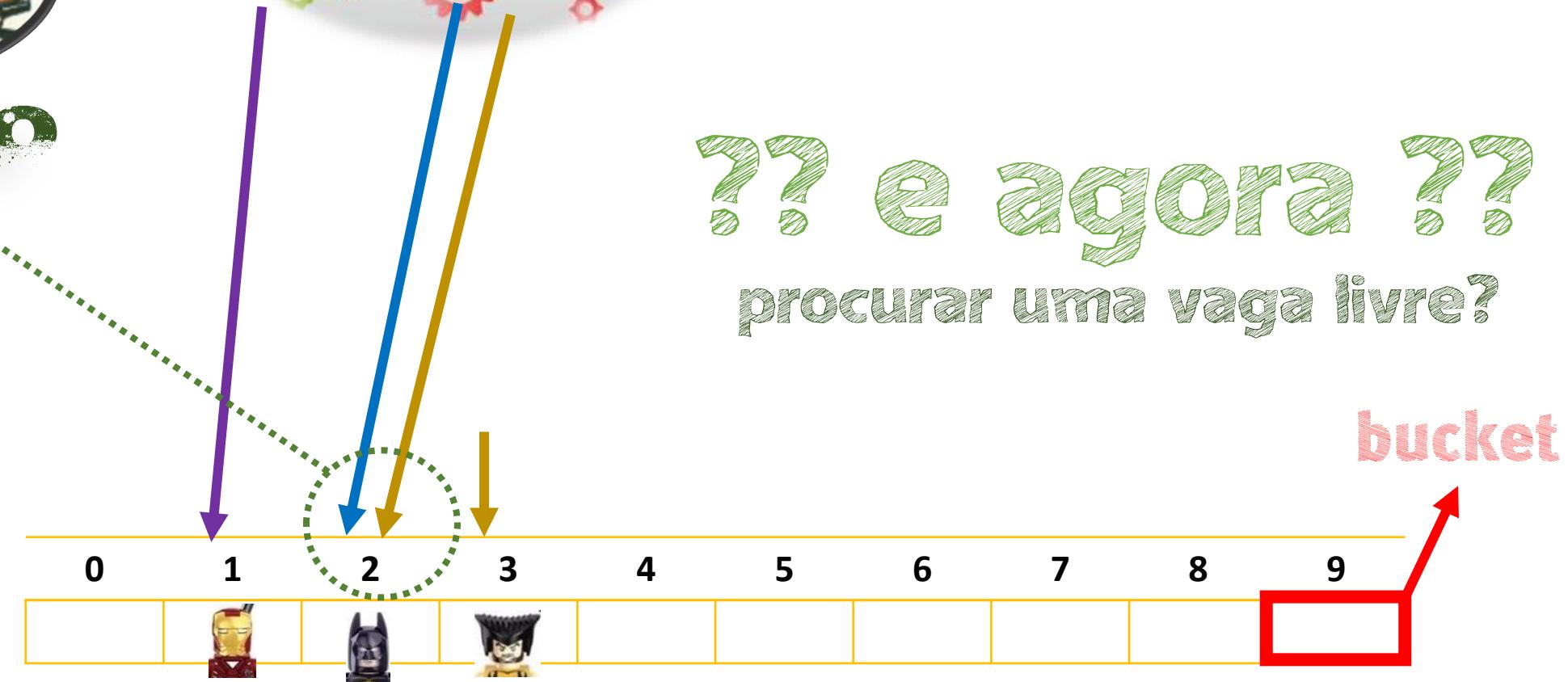
... es una práctica ... la mejor manera para todo



cadilisao

Estrutura de Dados

Função Hashing



Matrícula

123456789
987654321
111222333
333222111



Tabela Hash

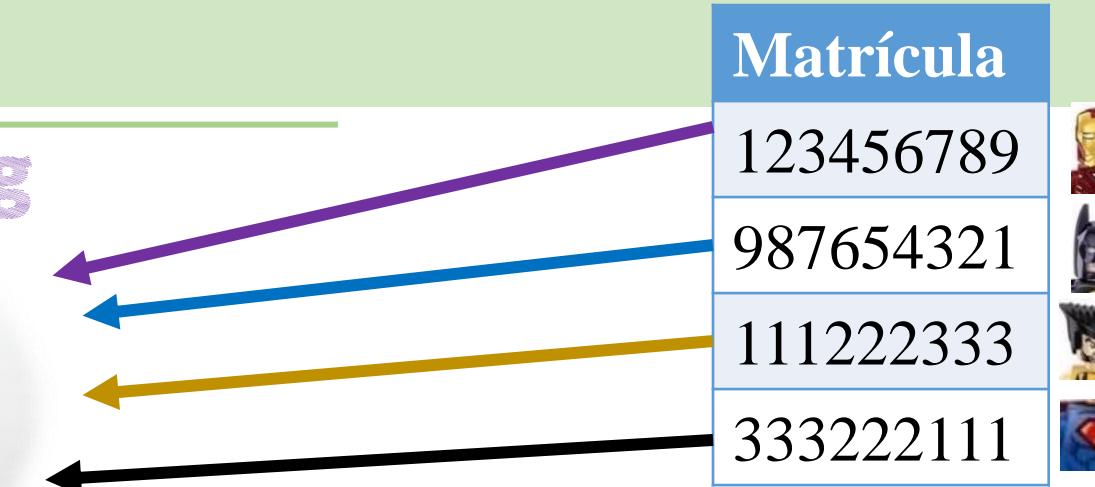
... estudar e praticar ... the best way for everything



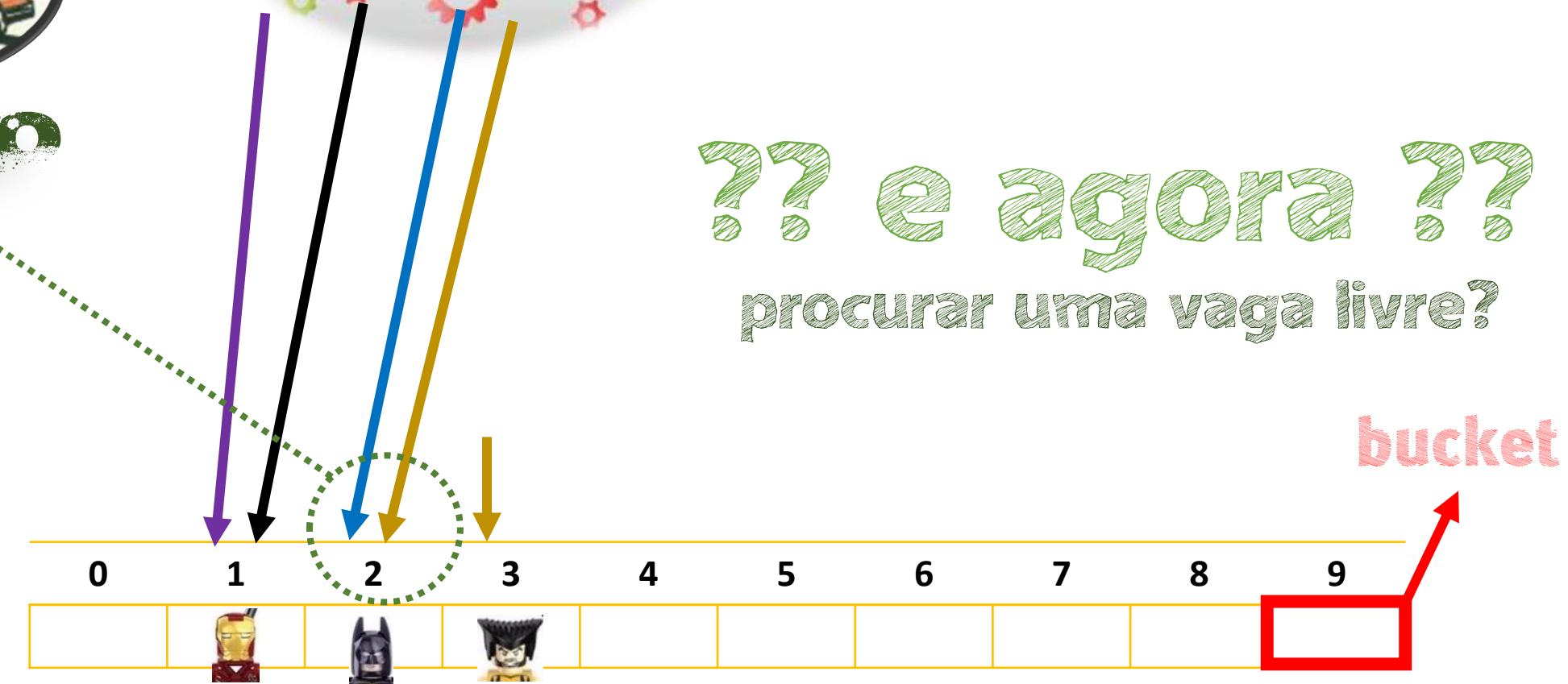
colisão

Estrutura
de Dados

Função Hashing



?? e agora ??
procurar uma vaga livre?



bucket

Tabela Hash

... estudar e praticar ... the best way for everything



colisão

Estrutura
de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111



?? e agora ??
procurar uma vaga livre?

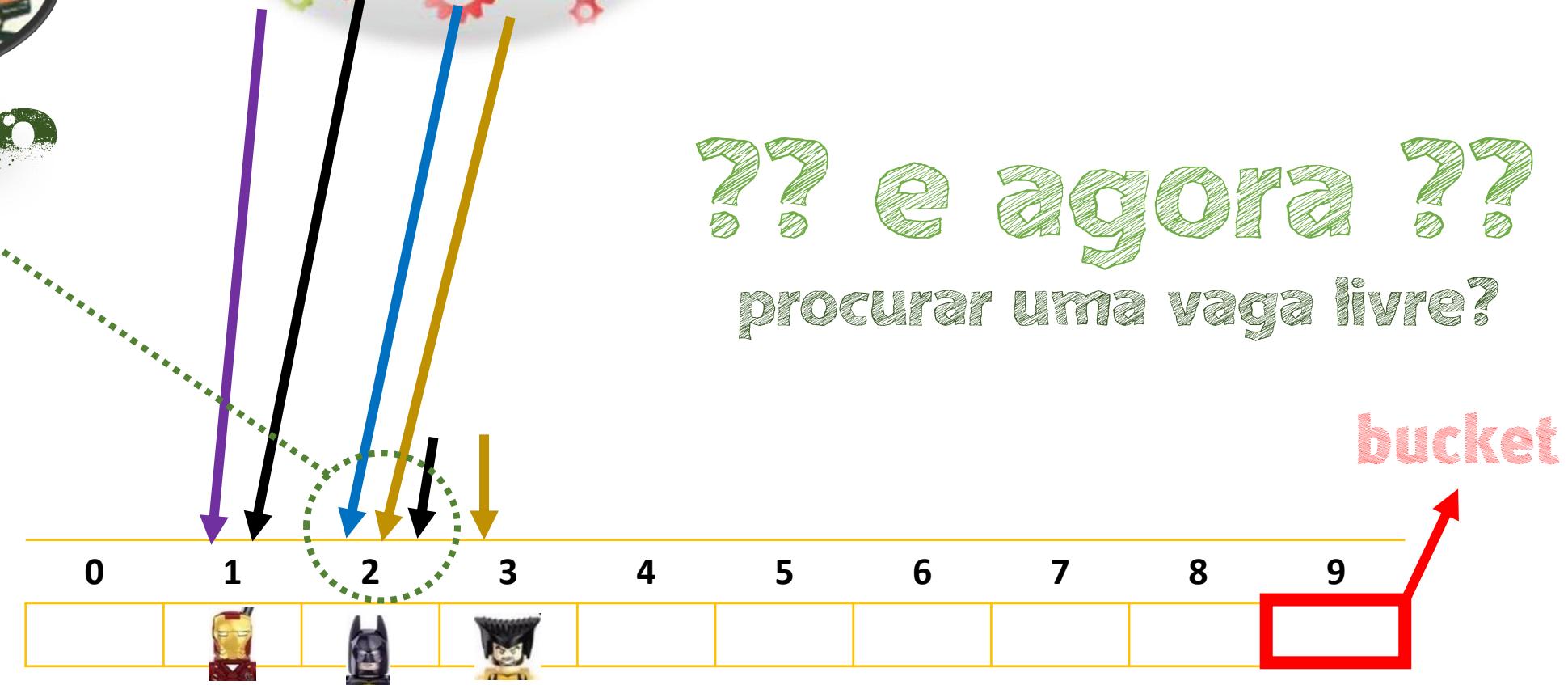


Tabela Hash

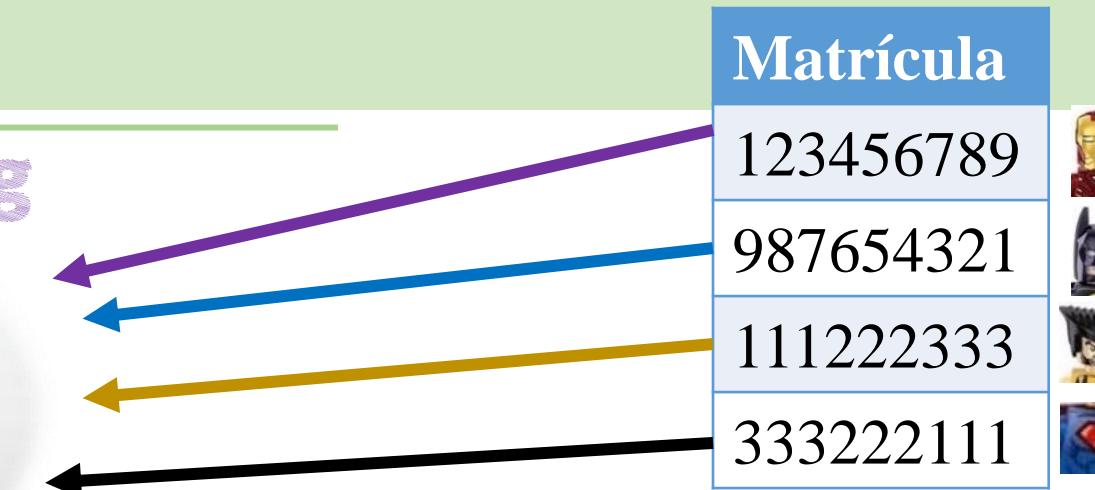
... estudar e praticar ... the best way for everything



colisão

Estrutura
de Dados

Função Hashing



?? e agora ??
procurar uma vaga livre?

bucket

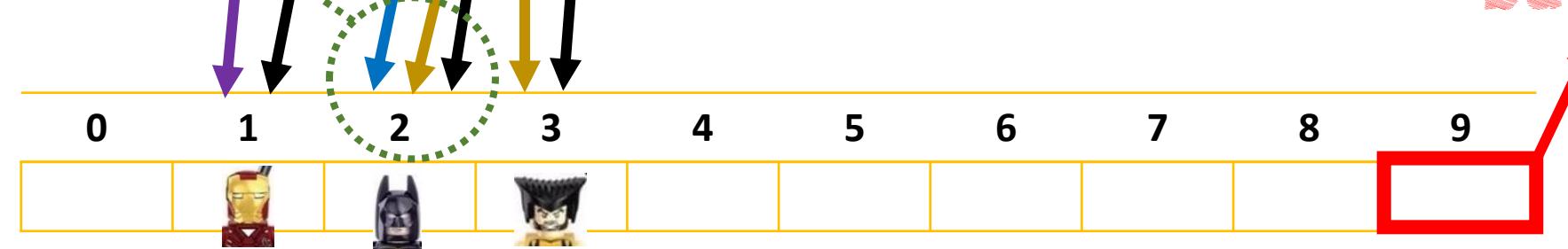


Tabela Hash

... estudar e praticar ... the best way for everything



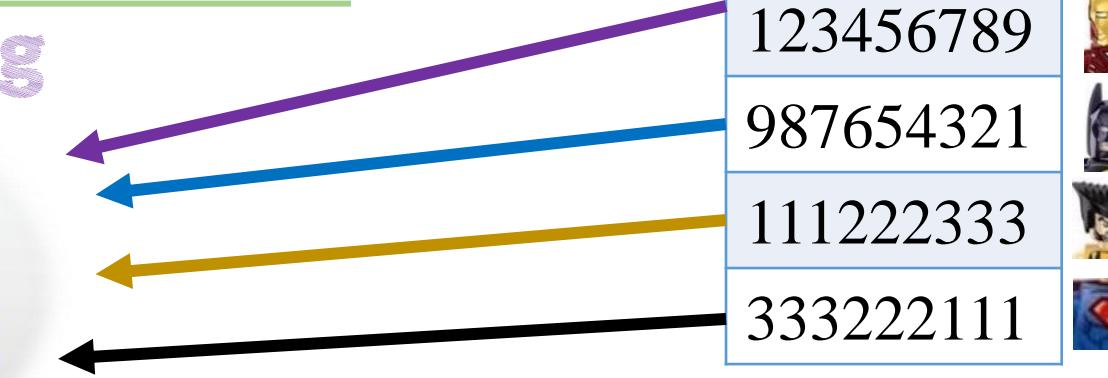
colisão

Estrutura
de Dados

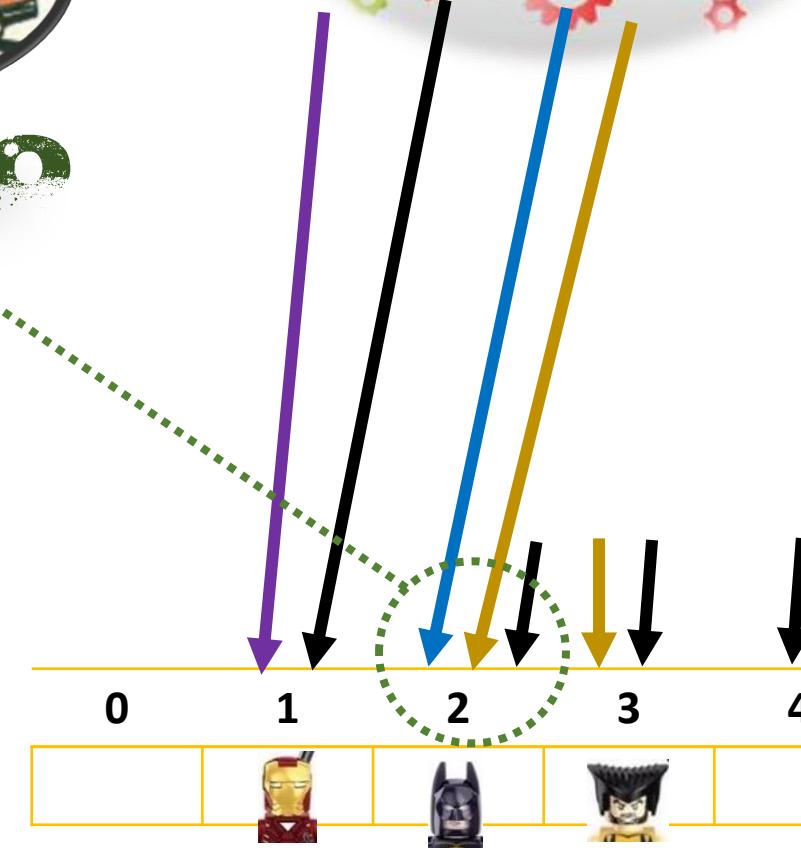
Função Hashing



Matrícula	Icone
123456789	Iron Man
987654321	Batman
111222333	Wolverine
333222111	Superman



?? e agora ??
procurar uma vaga livre?



bucket

Tabela Hash

... es una práctica ... la mejor manera para todo



Collision

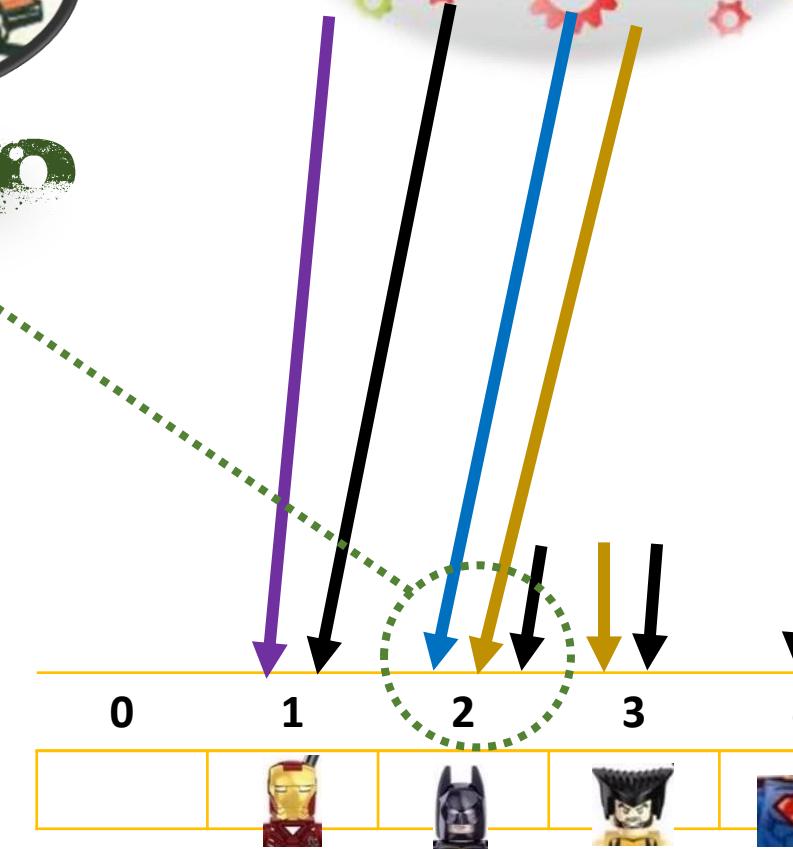
Estrutura de Dados

Função Hashing



Matrícula
123456789
987654321
111222333
333222111

?? e agora ?? procurar uma vaga livre?



bucket

Tabela Hash

... estudar e praticar ... the best way for everything

- O espalhamento auxilia no processo de busca porque uma Tabela Hash trabalha com o conceito de associação entre **chaves e valores**
 - Chave
 - Parte da informação que identifica um elemento, tanto no momento de ser inserido, quanto em sua procura
 - Valor
 - Corresponde ao índice (**posição na estrutura**) onde o elemento se encontra ou deve ser armazenado, sendo calculado a partir da chave

Na média, o custo desta operação deve ser $O(1)$

Tabela Hash - Vantagens

- Ideal para implementar estruturas de dados de valor-chave
- Operações de inserção, exclusão e pesquisa são eficientes
- Redução do uso de memória, pois aloca um espaço fixo para armazenar elementos
- Escalabilidade, ainda que uma coleção fique muito grande, é mantido o tempo de acesso constante
- Armazenamento seguro e verificação de integridade, sendo de grande valia para segurança e criptografia
- Implementação relativamente simples

Tabela Hash - Desvantagens

- **Desvantagens**
 - Custo elevado caso seja necessário ordenar os elementos pela chave
 - Pior caso $O(n)$, onde **n** é o tamanho da coleção
 - Existência de colisões
 - Dois elementos querendo ocupar a mesma posição na estrutura de dados

- **Indexação de banco de dados**
 - Hashing é usado para indexar e recuperar dados de forma eficiente em bancos de dados e outros sistemas de armazenamento de dados
- **Armazenamento de senha**
 - Hashing é usado para armazenar senhas com segurança, aplicando uma função de hash à senha e armazenando o resultado com hash, em vez da senha de texto
- **Compressão de dados**
 - Hashing é usado em algoritmos de compressão de dados, como o algoritmo de codificação de Huffman, para codificar dados com eficiência

- **Algoritmos de pesquisa**
 - Hashing é usado para implementar algoritmos de pesquisa, como filtros bloom para pesquisas e consultas rápidas
- **Criptografia**
 - Hashing é usado em criptografia para gerar assinaturas digitais, códigos de autenticação de mensagem (MACs) e funções de derivação de chave
 - **MD5 e a família Secure Hash Algorithm (SHA)**
- **Balanceamento de carga**
 - Hashing é usado em algoritmos de balanceamento de carga, como hash consistente, para distribuir solicitações a servidores em uma rede

- **Blockchain**
 - Hashing é usado na tecnologia blockchain, como o protocolo **PoW** (prova de trabalho), para garantir a integridade e o consenso do blockchain
- **Processamento de imagem**
 - Hashing é usado em aplicativos de processamento de imagem, como hash perceptivo, para detectar e evitar duplicações e modificações de imagem
- **Comparação de arquivos**
 - Hashing é usado em algoritmos de comparação de arquivos, como as funções de hash MD5 e SHA-1, para comparar e verificar a integridade dos arquivos

- **Detecção de fraude**
 - Hashing é usado em aplicações de detecção de fraude e segurança cibernética, como detecção de intrusão e software antivírus, para detectar e prevenir atividades maliciosas
- **Compiladores**
 - Implementação da tabela de símbolos

Função Hashing / Função de Mapeamento

- Utilizada para calcular a posição do objeto na estrutura
 - Calcula utilizando uma chave como parâmetro principal
 - A chave é definida a partir de um atributo do objeto
 - Uma boa função hash deve satisfazer a hipótese de **hash uniforme**, pelo menos aproximadamente, garantindo uma distribuição uniforme entre os índices disponíveis
 - Cada chave tem igual probabilidade de ser mapeada para qualquer uma das posições na estrutura
 - Extremamente relevante para o desempenho da Tabela Hash



Função Hashing / Função de Mapeamento

- Requisitos
 - Preferencialmente **simples** e de **baixo custo computacional** para não comprometer o desempenho
 - Garantir que valores distintos produzam índices diferentes, tanto quanto possível
 - Pode ser dependente da natureza e domínio da chave utilizada
 - Importante conhecer o tipo de dado da chave para definir a melhor função

- O endereço de armazenamento ou busca será o **resto da divisão da chave pelo tamanho da estrutura**
 - A chave (k) deve ser um número inteiro
 - $h(k) = k \ mod \ M + 1$ para $m_i \in [1, M]$
 - $h(k) = k \ mod \ M$ para $m_i \in [0, M-1]$
 - O resultado desta operação estará sempre no intervalo $[0, M-1]$ ou $[1, M]$
 - Exemplo considerando um vetor com $M = 500 \gg [0, 499]$

O uso de números primos para definir o tamanho da estrutura melhora a distribuição dos elementos porque dificulta a formação de padrões nas posições

- O endereço de armazenamento ou busca será o **resto da divisão da chave pelo tamanho da estrutura**
 - A chave (k) deve ser um número inteiro
 - $h(k) = k \ mod \ M + 1$ para $m_i \in [1, M]$
 - $h(k) = k \ mod \ M$ para $m_i \in [0, M-1]$
 - O resultado desta operação estará sempre no intervalo $[0, M-1]$ ou $[1, M]$
 - Exemplo considerando um vetor com $M = 500 \gg [0, 499]$

O uso de números primos para definir o tamanho da estrutura melhora a distribuição dos elementos porque dificulta a formação de padrões nas posições

Chave	Índice
12768	
17	
968	
1974	

- O endereço de armazenamento ou busca será o **resto da divisão da chave pelo tamanho da estrutura**
 - A chave (k) deve ser um número inteiro
 - $h(k) = k \ mod \ M + 1$ para $m_i \in [1, M]$
 - $h(k) = k \ mod \ M$ para $m_i \in [0, M-1]$
 - O resultado desta operação estará sempre no intervalo $[0, M-1]$ ou $[1, M]$
 - Exemplo considerando um vetor com $M = 500 \gg [0, 499]$

O uso de números primos para definir o tamanho da estrutura melhora a distribuição dos elementos porque dificulta a formação de padrões nas posições

Chave	Índice
12768	268
17	
968	
1974	

- O endereço de armazenamento ou busca será o **resto da divisão da chave pelo tamanho da estrutura**
 - A chave (k) deve ser um número inteiro
 - $h(k) = k \ mod \ M + 1$ para $m_i \in [1, M]$
 - $h(k) = k \ mod \ M$ para $m_i \in [0, M-1]$
 - O resultado desta operação estará sempre no intervalo $[0, M-1]$ ou $[1, M]$
 - Exemplo considerando um vetor com $M = 500 \gg [0, 499]$

O uso de números primos para definir o tamanho da estrutura melhora a distribuição dos elementos porque dificulta a formação de padrões nas posições

Chave	Índice
12768	268
17	17
968	
1974	

- O endereço de armazenamento ou busca será o **resto da divisão da chave pelo tamanho da estrutura**
 - A chave (k) deve ser um número inteiro
 - $h(k) = k \ mod \ M + 1$ para $m_i \in [1, M]$
 - $h(k) = k \ mod \ M$ para $m_i \in [0, M-1]$
 - O resultado desta operação estará sempre no intervalo $[0, M-1]$ ou $[1, M]$
 - Exemplo considerando um vetor com $M = 500 \gg [0, 499]$

O uso de números primos para definir o tamanho da estrutura melhora a distribuição dos elementos porque dificulta a formação de padrões nas posições

Chave	Índice
12768	268
17	17
968	468
1974	

- O endereço de armazenamento ou busca será o **resto da divisão da chave pelo tamanho da estrutura**
 - A chave (k) deve ser um número inteiro
 - $h(k) = k \ mod \ M + 1$ para $m_i \in [1, M]$
 - $h(k) = k \ mod \ M$ para $m_i \in [0, M-1]$
 - O resultado desta operação estará sempre no intervalo $[0, M-1]$ ou $[1, M]$
 - Exemplo considerando um vetor com $M = 500 \gg [0, 499]$

O uso de números primos para definir o tamanho da estrutura melhora a distribuição dos elementos porque dificulta a formação de padrões nas posições

Chave	Índice
12768	268
17	17
968	468
1974	474

- **Vantagens**
 - Fácil de computar
 - Rápida
- **Desvantagens**
 - Função depende do valor de M
 - M deve ser um número primo
 - Alta probabilidade de colisões

Dica: mod em java = %

- Solução baseada em duas etapas

- Multiplicar a chave k por uma constante A , onde $A \in [0, 1]$ e calcular o resto da divisão por 1
 - A literatura sugere a utilização de $A = 0.6180339887$
- Extrair o piso, (arredondamento para baixo), da multiplicação da fração por um valor h
 - A escolha de h não é crítica
 - Para facilitar a implementação, o valor de h deve ser uma potência de 2

$$h(k) = \lfloor h * (k * A \bmod 1) \rfloor$$

$h = 64$

chave	Índice
1000	
123456	
9999	
10	

- Solução baseada em duas etapas

- Multiplicar a chave k por uma constante A , onde $A \in [0, 1]$ e calcular o resto da divisão por 1
 - A literatura sugere a utilização de $A = 0.6180339887$
- Extrair o piso, (arredondamento para baixo), da multiplicação da fração por um valor h
 - A escolha de h não é crítica
 - Para facilitar a implementação, o valor de h deve ser uma potência de 2

$$h(k) = \lfloor h * (k * A \bmod 1) \rfloor$$

$h = 64$

chave	Índice
1000	2
123456	
9999	
10	

- Solução baseada em duas etapas

- Multiplicar a chave k por uma constante A , onde $A \in [0, 1]$ e calcular o resto da divisão por 1
 - A literatura sugere a utilização de $A = 0.6180339887$
- Extrair o piso, (arredondamento para baixo), da multiplicação da fração por um valor h
 - A escolha de h não é crítica
 - Para facilitar a implementação, o valor de h deve ser uma potência de 2

$$h(k) = \lfloor h * (k * A \bmod 1) \rfloor$$

$h = 64$

chave	Índice
1000	2
123456	0
9999	
10	

- Solução baseada em duas etapas

- Multiplicar a chave k por uma constante A , onde $A \in [0, 1]$ e calcular o resto da divisão por 1
 - A literatura sugere a utilização de $A = 0.6180339887$
- Extrair o piso, (arredondamento para baixo), da multiplicação da fração por um valor h
 - A escolha de h não é crítica
 - Para facilitar a implementação, o valor de h deve ser uma potência de 2

$$h(k) = \lfloor h * (k * A \bmod 1) \rfloor$$

$h = 64$

chave	Índice
1000	2
123456	0
9999	46
10	

- Solução baseada em duas etapas

- Multiplicar a chave k por uma constante A , onde $A \in [0, 1]$ e calcular o resto da divisão por 1
 - A literatura sugere a utilização de $A = 0.6180339887$
- Extrair o piso, (arredondamento para baixo), da multiplicação da fração por um valor h
 - A escolha de h não é crítica
 - Para facilitar a implementação, o valor de h deve ser uma potência de 2

$$h(k) = \lfloor h * (k * A \bmod 1) \rfloor$$

$h = 64$

chave	Índice
1000	2
123456	0
9999	46
10	11

- Uma Função Hash, usualmente é **determinística**, desta forma, poderia ser **manipulada** para degradar o desempenho do sistema
 - Ex: definir um conjunto de chaves em que todas colidam
- A ideia consiste na utilização de **parâmetros** distintos a cada execução, assim, apesar da mesma entrada, os endereços na estrutura serão distintos

$$h_u(k) = (|a*k + b| \bmod p) \bmod m$$

- Onde:
 - **m** é o tamanho da estrutura e **k** é a chave
 - **p** é um número primo maior que **m**
 - **a** e **b** são constantes aleatórias; intervalo [0, ..., p-1] para **a** e intervalo [1, ..., p-1] para **b**

- Garantia de não haver colisões entre as chaves
 - Chaves distintas sempre geram índices diferentes
 - No pior caso, tanto para inserir quanto para buscar, a operação será realizada em tempo constante $O(1)$
 - Utilizado apenas em sistemas críticos, nos quais a colisão não é tolerável devido as implicâncias
 - Aplicações específicas, como por exemplo, dicionário de palavras reservadas de uma linguagem de programação
 - A restrição é que se faz necessário **conhecer previamente TODAS** as chaves que serão utilizadas

Hash Imperfeito

- É a estrutura mais comumente encontrada
- Dado duas chaves distintas, o resultado da função poderá ser o mesmo, indicando o mesmo endereço na estrutura de dados
- A colisão não necessariamente é um erro
 - É indesejada porque degrada o desempenho do sistema
- Uma distribuição estritamente uniforme é complexa e computacionalmente custosa, degradando o desempenho
- Usualmente, diversos tipos de tabelas Hash utilizam estruturas auxiliares para tratar o evento da colisão

Tratamento de Colisões

- Na prática, o uso de uma Tabela Hash pressupõe dois aspectos
 - Uma função Hash
 - Precisa prover um espalhamento o mais uniforme possível
 - A definição adequada desta função e do tamanho da estrutura, naturalmente pode diminuir as colisões
 - Uma abordagem para tratar colisões
 - Frequentemente se tem mais chaves do que índices de armazenamento, logo irão ocorrer colisões que também são uniformemente distribuídas
 - Colisões são teoricamente inevitáveis

Tratamento de Colisões: Endereçamento Aberto

• Sondagem Linear

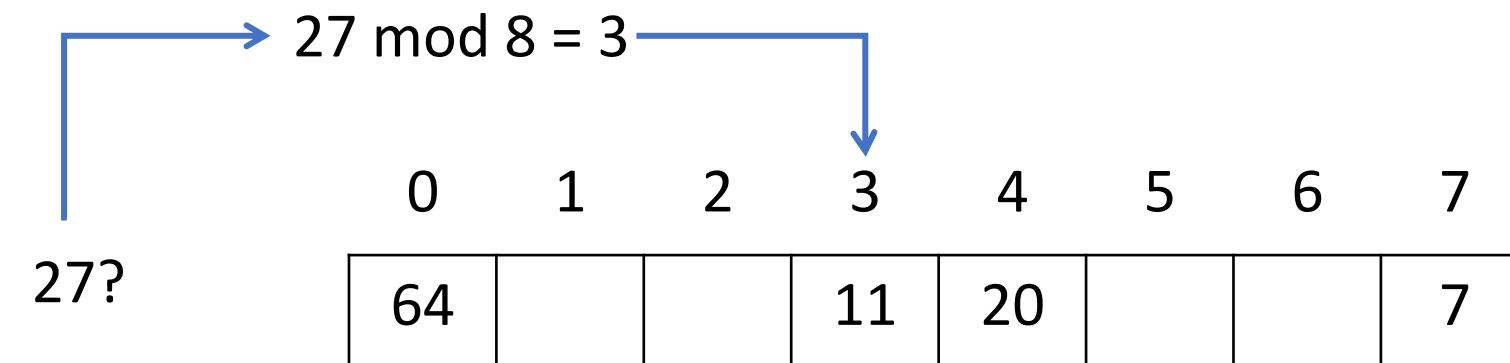
- Procurar a próxima posição vazia, depois do endereço calculado a partir da chave
- Tenta espalhar os elementos de forma sequencial
- Vantagem: simplicidade
- Desvantagem: a inserção e busca passam a ter desempenho $O(n)$

0	1	2	3	4	5	6	7
64			11	20			7

Tratamento de Colisões: Endereçamento Aberto

• Sondagem Linear

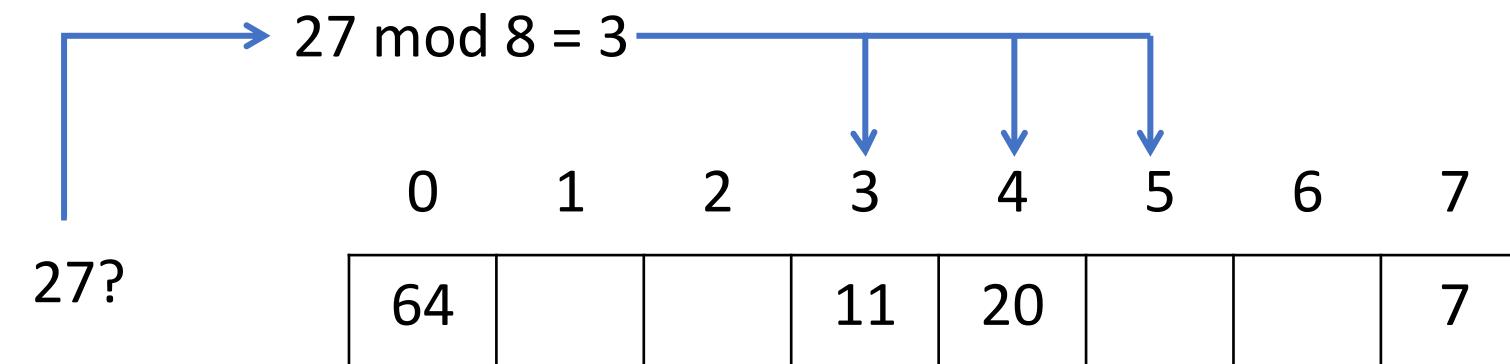
- Procurar a próxima posição vazia, depois do endereço calculado a partir da chave
- Tenta espalhar os elementos de forma sequencial
- Vantagem: simplicidade
- Desvantagem: a inserção e busca passam a ter desempenho $O(n)$



Tratamento de Colisões: Endereçamento Aberto

• Sondagem Linear

- Procurar a próxima posição vazia, depois do endereço calculado a partir da chave
- Tenta espalhar os elementos de forma sequencial
- Vantagem: simplicidade
- Desvantagem: a inserção e busca passam a ter desempenho $O(n)$



Tratamento de Colisões: Endereçamento Aberto

- Sondagem Linear
 - Vantagem
 - Simplicidade
 - Desvantagem
 - Suscetível a um agrupamento primário
 - São construídas longas sequências de posições ocupadas
 - Degrada o desempenho, a inserção e busca passam a ter desempenho $O(n)$

Tratamento de Colisões: Endereçamento Aberto

- **Sondagem Quadrática**

- Distribui os elementos a partir de uma equação do 2º grau

$$posicao = ph + (c_1 * i) + (c_2 * i^2)$$

ph = posição inicial, obtida pela função de hashing;
i é tentativa atual; c_1 e c_2 são os coeficientes da equação

- 1º elemento, ($i=0$), fica na posição da função
- 2º elemento, ($i=1$), $ph + (c_1 * 1) + (c_2 * 1^2)$
- 3º elemento, ($i=2$), $ph + (c_1 * 2) + (c_2 * 2^2)$
-

Tratamento de Colisões: Endereçamento Aberto

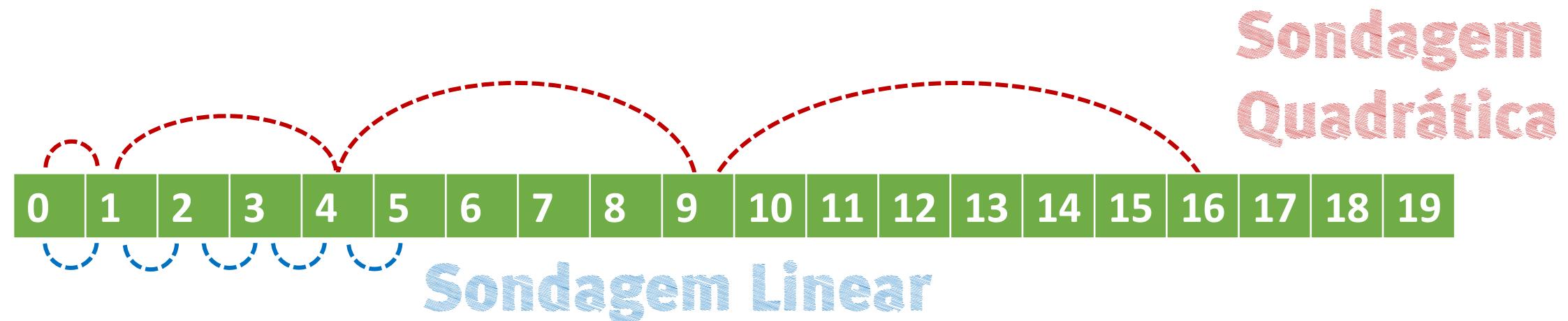
- **Sondagem Quadrática**

- Resolve o agrupamento primário, porém gera um agrupamento secundário
 - As chaves que geram a mesma posição inicial, produzem as mesmas posições na sondagem quadrática
 - Com agrupamentos secundários a degradação no desempenho é menor que no primário

Tratamento de Colisões: Endereçamento Aberto

- **Sondagem Quadrática**

- Resolve o agrupamento primário, porém gera um agrupamento secundário
 - As chaves que geram a mesma posição inicial, produzem as mesmas posições na sondagem quadrática
 - Com agrupamentos secundários a degradação no desempenho é menor que no primário



Tratamento de Colisões: Endereçamento Aberto

- **Sondagem Espalhamento Duplo / Duplo Hash**
 - Distribui os elementos usando **duas** funções hash distintas
 - H1: usada para calcular a **posição inicial**
 - H2: usada para calcular os **deslocamentos** em relação a posição inicial no caso de colisão

$$posicao = H1 + i * H2$$

i é a tentativa corrente

- Resultado de H2 nunca pode ser zero
 - 1º elemento, (i=0) >> fica na posição dada por H1
 - 2º elemento, (i=1) >> $H1 + 1 * H2$
 - 3º elemento, (i=2) >> $H1 + 2 * H2$
 -

Tratamento de Colisões: Endereçamento Fechado

- Utiliza uma lista encadeada para cada endereço da tabela
- Vantagem
 - Não é preciso recalcular endereços
 - Sinônimos são encontrados em uma única busca
 - A inserção continua sendo realizada em tempo constante para listas não ordenadas
- Desvantagem
 - A busca consumirá tempo proporcional ao tamanho da lista
 - Tratamento em listas pode consumir tempo $O(n)$

Tratamento de Colisões: Endereçamento Fechado

- Exemplificando
 - Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$

Tratamento de Colisões: Endereçamento Fechado

- **Exemplificando**

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$

0
1
2
3
4
5
6
7

Tratamento de Colisões: Endereçamento Fechado

- **Exemplificando**

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



0
1
2
3
4
5
6
7

Tratamento de Colisões: Endereçamento Fechado

- **Exemplificando**

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



0
1
2
3
4
5
6
7

Tratamento de Colisões: Endereçamento Fechado

- **Exemplificando**

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

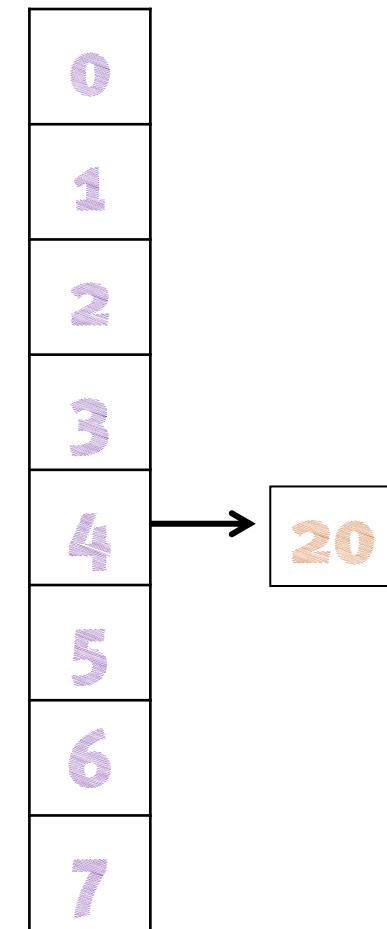
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

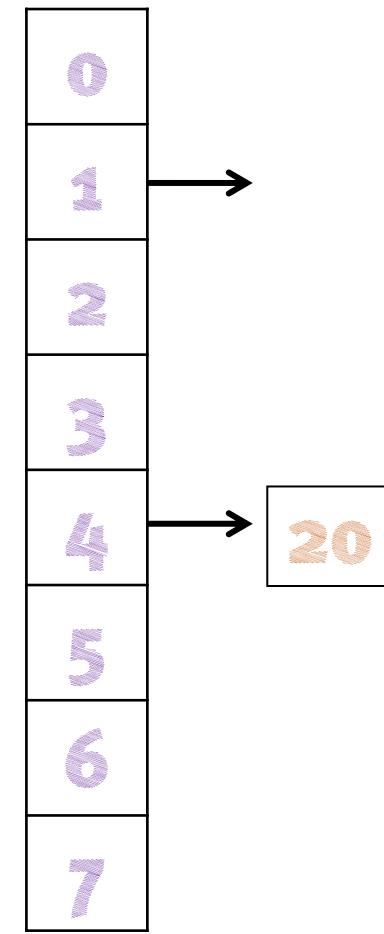
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

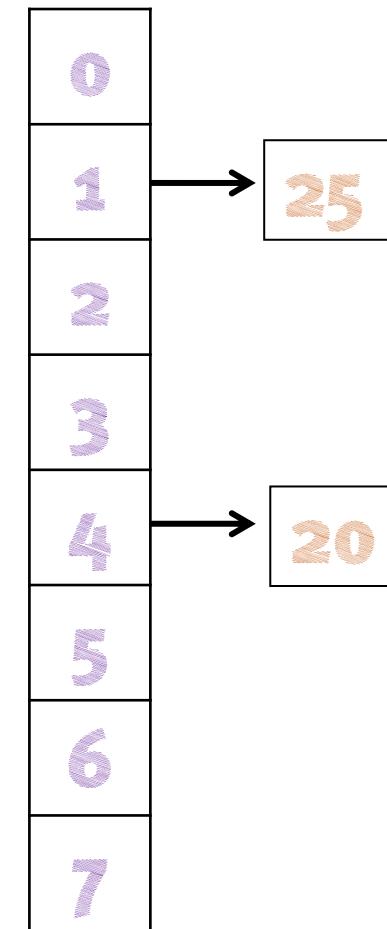
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

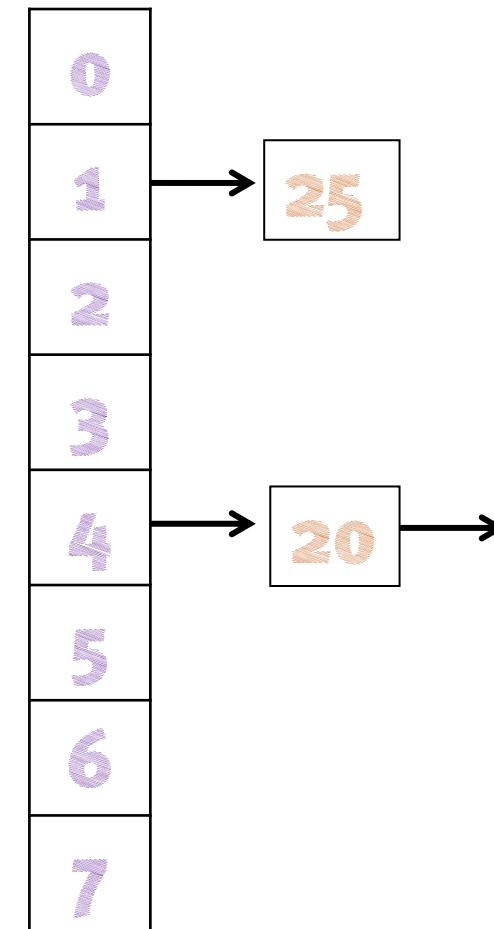
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

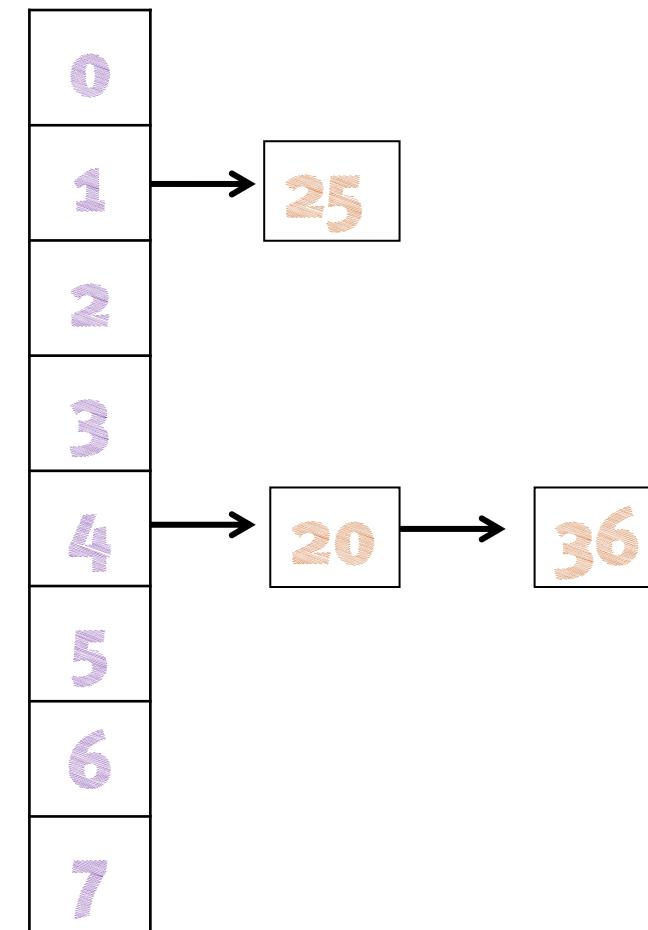
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

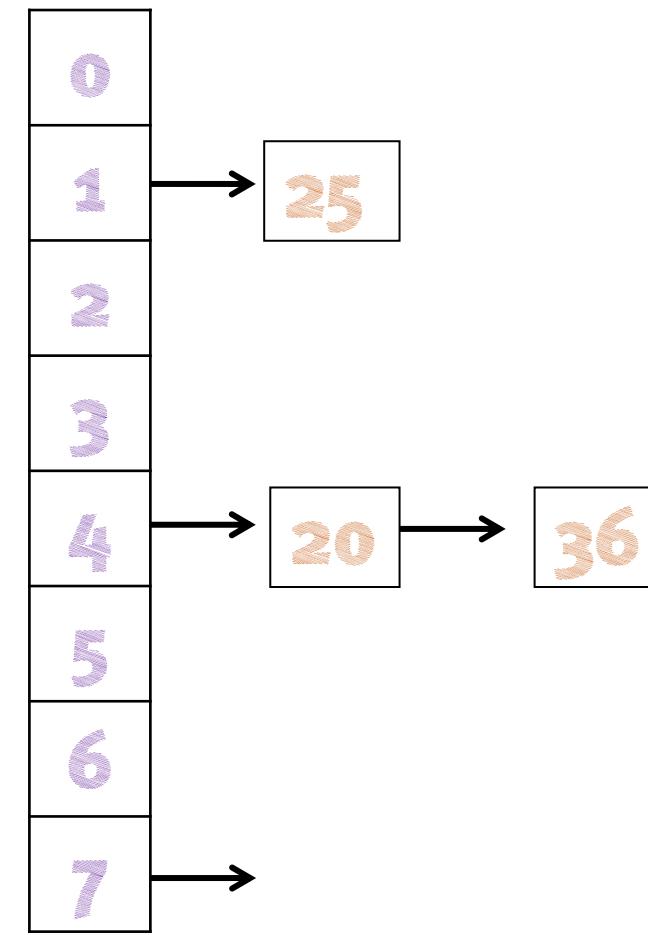
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

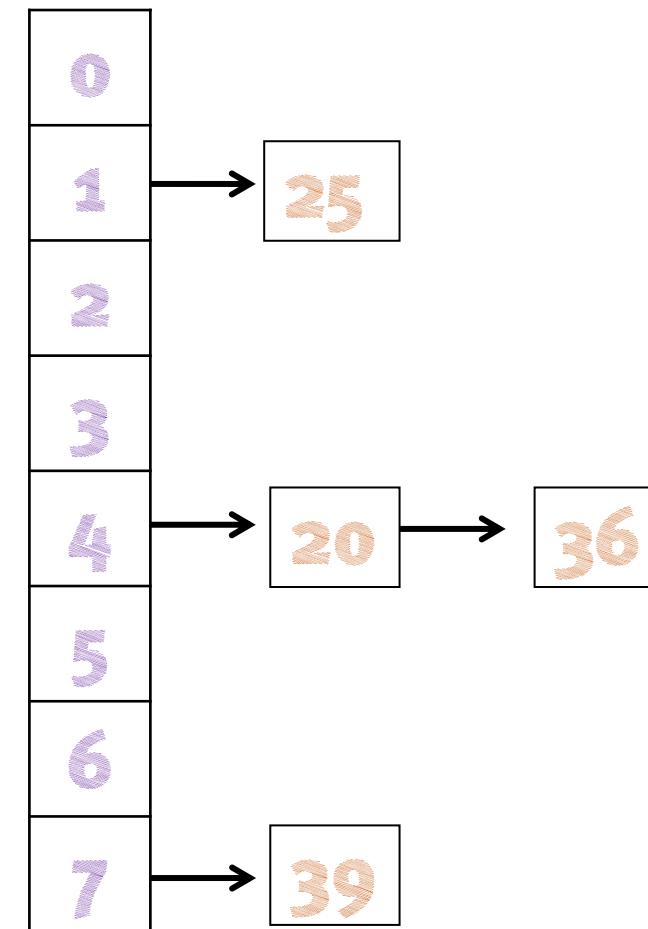
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

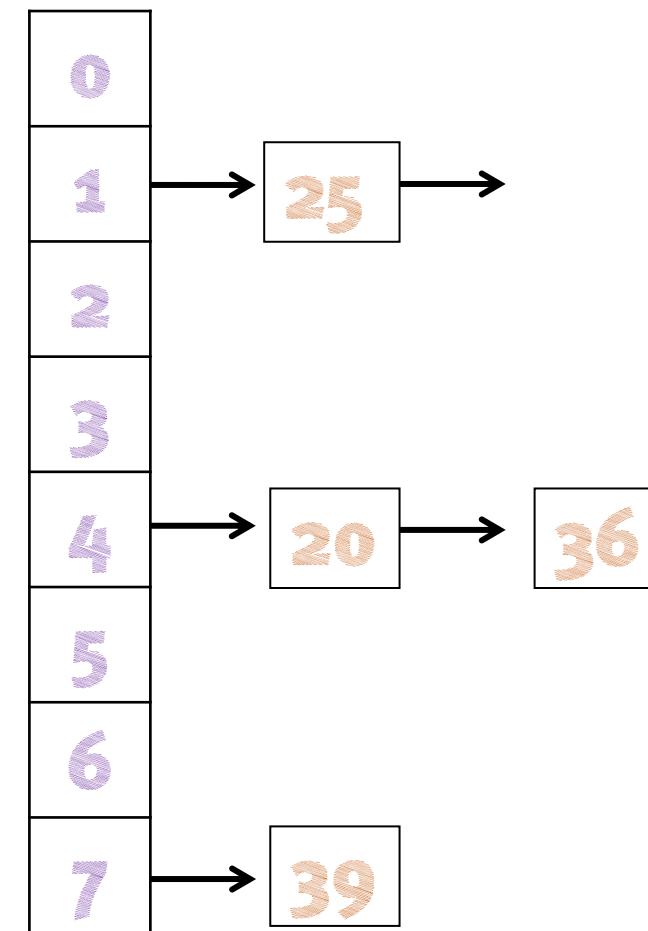
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

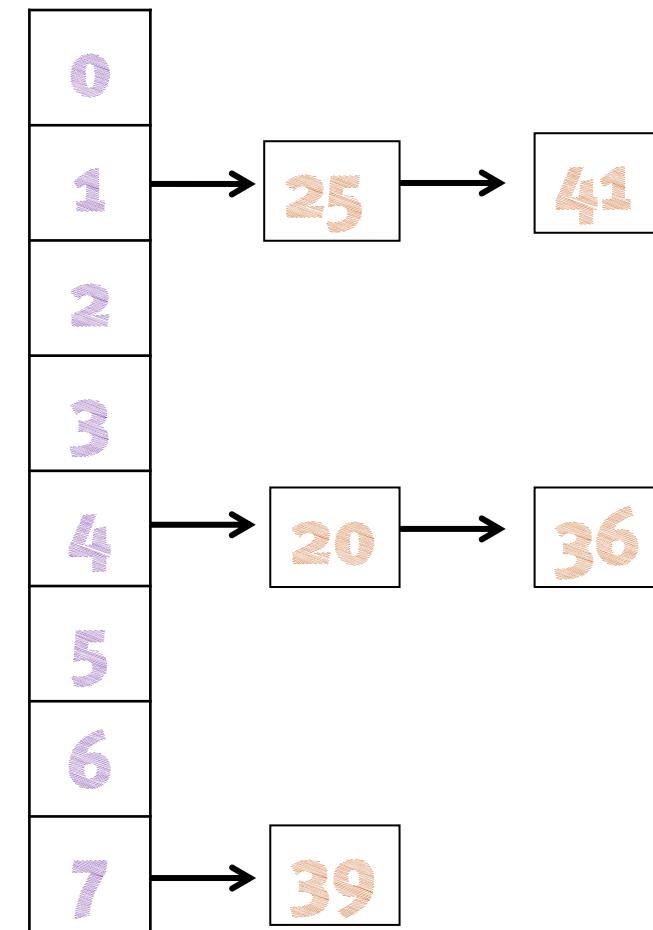
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

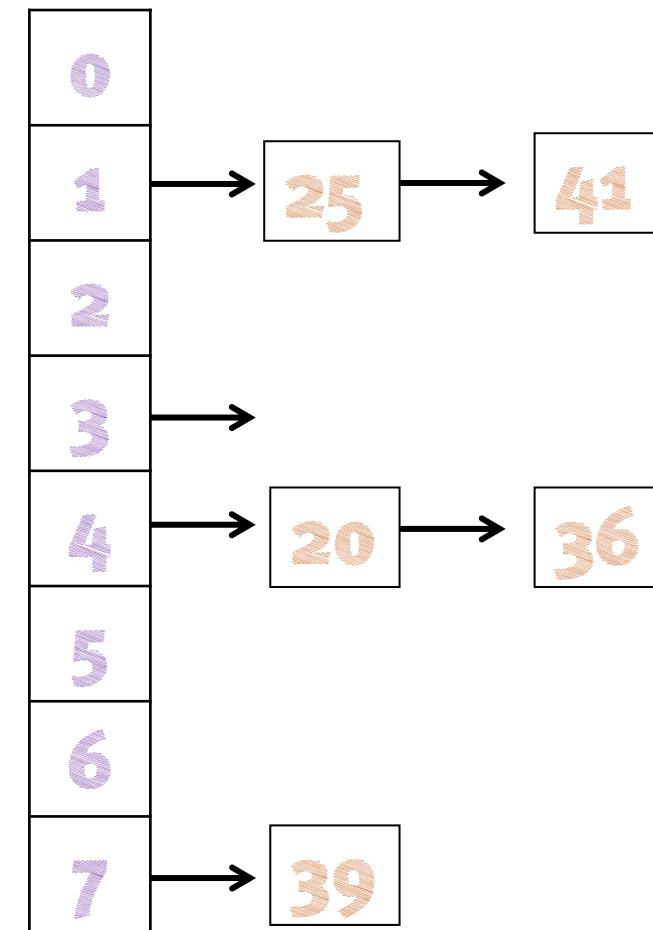
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

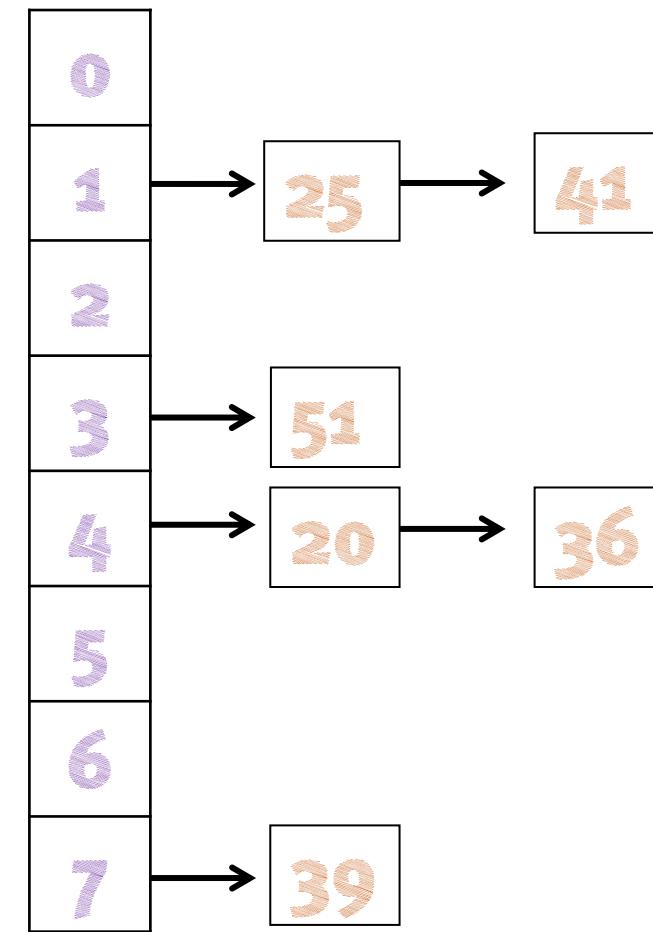
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando

- Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

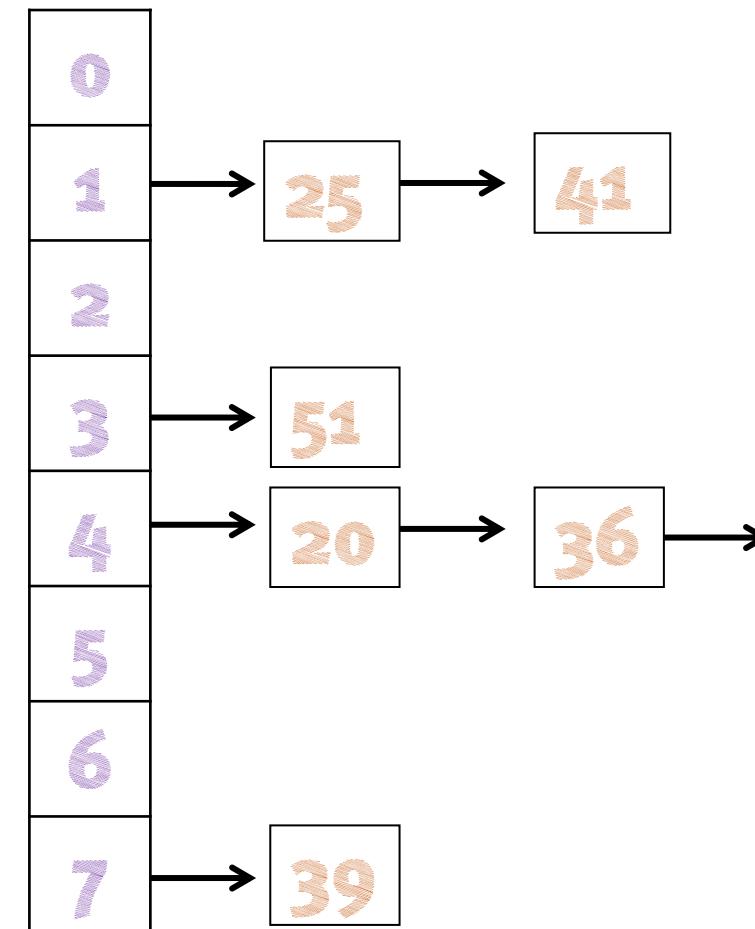
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando
 - Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

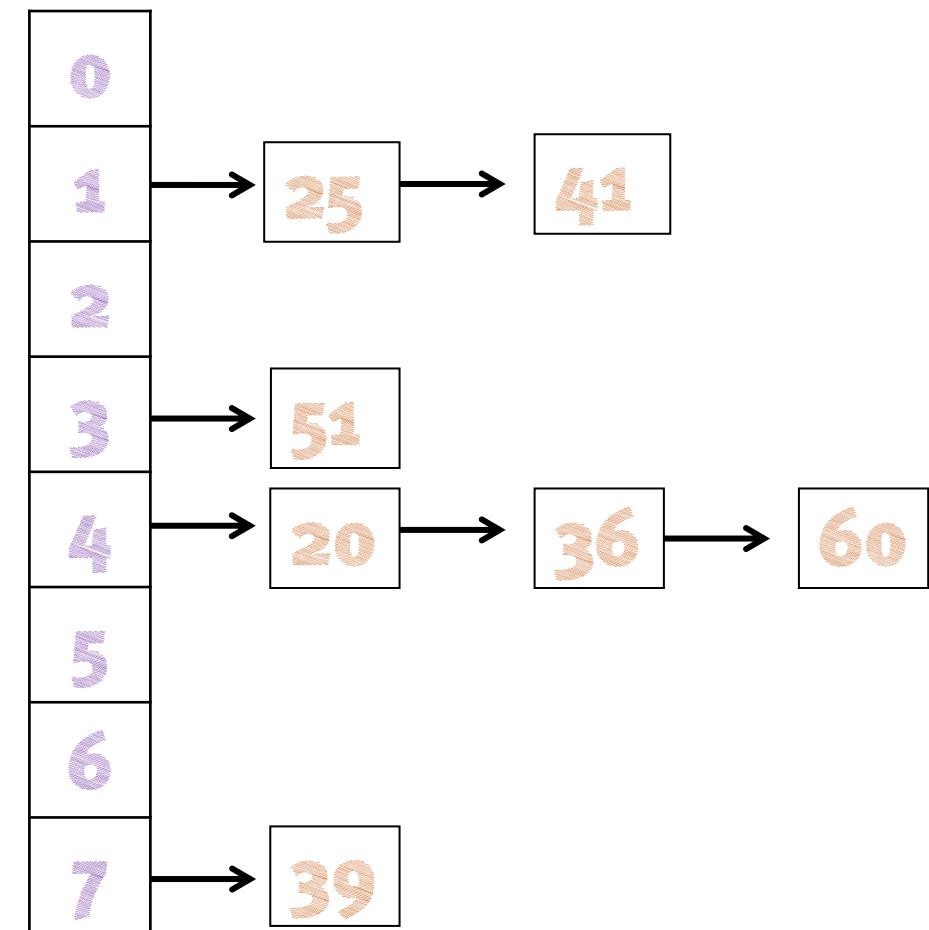
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando
 - Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

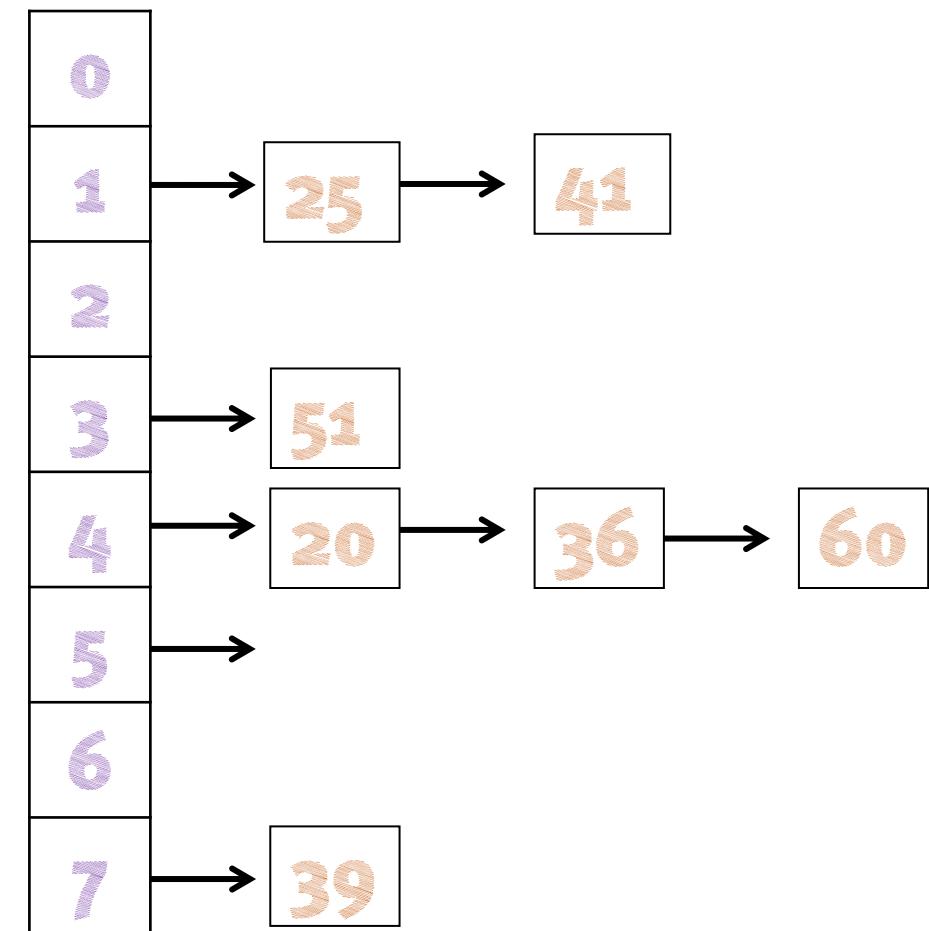
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Tratamento de Colisões: Endereçamento Fechado

- Exemplificando
 - Para inserir os elementos 20, 25, 36, 39, 41, 51, 60 e 61 na Tabela usando o método tradicional

$$20 \bmod 8 = 4$$

$$25 \bmod 8 = 1$$

$$36 \bmod 8 = 4$$

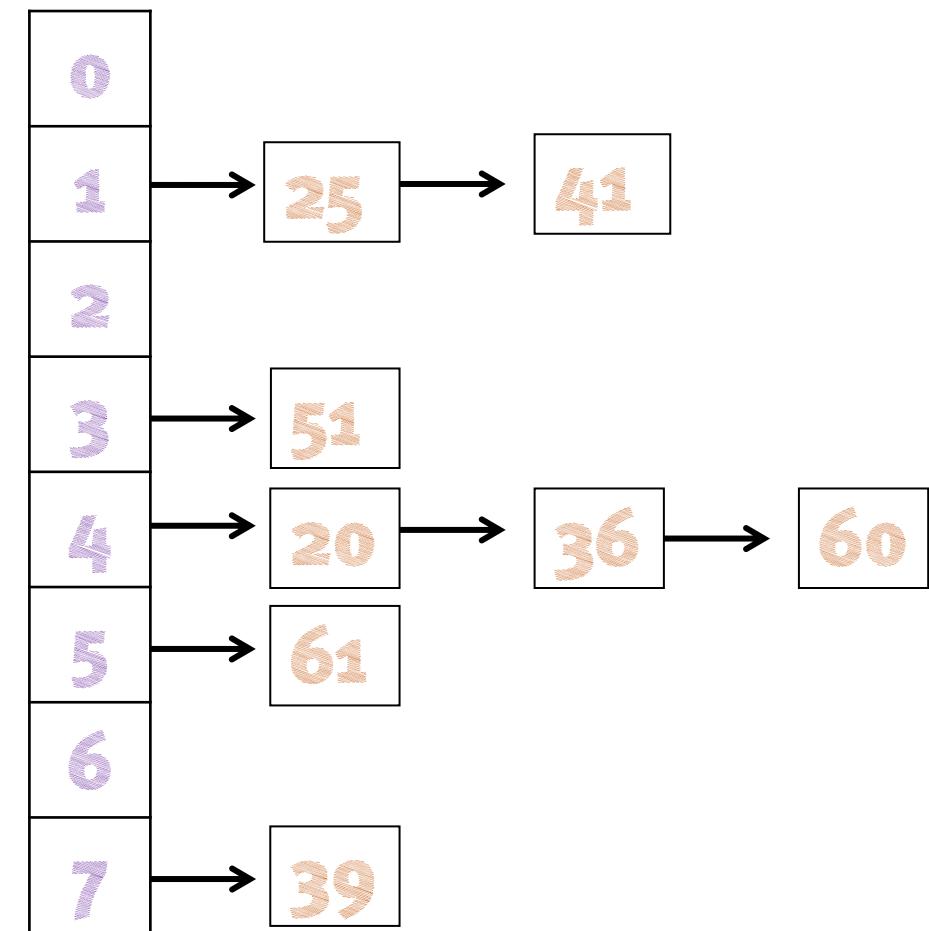
$$39 \bmod 8 = 7$$

$$41 \bmod 8 = 1$$

$$51 \bmod 8 = 3$$

$$60 \bmod 8 = 4$$

$$61 \bmod 8 = 5$$



Fator de Carga / Load Factor

- Estatística que verifica a proporção de chaves dentro do array

$$fc = c / n$$

onde c = qtd chaves inseridas e n = tamanho array

- O valor de fc varia entre 0.0 e 1.0
 - Quanto maior o fc, maior probabilidade de colisão e degradação do tempo de execução
 - No encadeamento, haverão listas cada vez maiores
 - No endereçamento aberto, o máximo do fator é 1.0
 - Em ambas soluções haverá degradação do desempenho

Fator de Carga / Load Factor

- Resumindo, com o crescimento da qtd de dados, é inevitável que o array fique menor que o necessário
- Resize
 - ação para quando o número de elementos armazenados se aproxima do máximo permitido
- Rehash
 - para utilizar a nova estrutura é preciso reallocar cada elemento inserido anteriormente, irão para novas posições

fc pequeno » resize frequente

fc grande » aumento de colisões

Java utiliza 0,75 como fator de carga

Prof. Hélder Pereira Borges

helder@ifma.edu.br

ALGORITMOS E ESTRUTURAS DE DADOS II

Tabelas Hash



IFMA

Departamento de Computação