

Dicionários e Tabelas Hash

Conceitos, complexidade e funcionamento

Prof. Marcelo de Souza

45RPE – Resolução de Problemas com Estruturas de Dados
Universidade do Estado de Santa Catarina

Dicionários

Conceitos básicos



Um **dicionário** armazena entradas compostas por uma **chave** e um **valor**.

- ▶ A chave serve para buscar o registro.
- ▶ O valor armazena o registro associado à chave.

Dicionários

Conceitos básicos



Um **dicionário** armazena entradas compostas por uma **chave** e um **valor**.

- ▶ A chave serve para buscar o registro.
- ▶ O valor armazena o registro associado à chave.

Também chamado de **mapa**, **tabela** ou **array associativo**, o dicionário organiza e acessa as entradas pelas suas **chaves**, em vez das suas posições, i.e. a chave é o índice da estrutura.

Dicionários

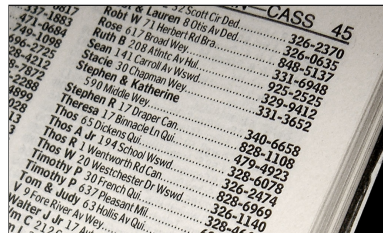
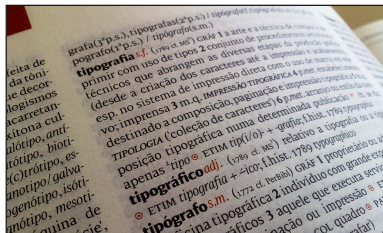
Conceitos básicos

Um **dicionário** armazena entradas compostas por uma **chave** e um **valor**.

- ▶ A chave serve para buscar o registro.
- ▶ O valor armazena o registro associado à chave.

Também chamado de **mapa**, **tabela** ou **array associativo**, o dicionário organiza e acessa as entradas pelas suas **chaves**, em vez das suas posições, i.e. a chave é o índice da estrutura.

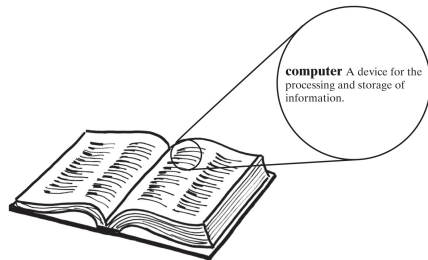
Exemplos no cotidiano: dicionários, listas telefônicas, cardápios, índices remissivos, ...





Características:

- ▶ Na maioria das implementações, **a chave é única**.
 - ▶ Ao inserir uma entrada com chave existente, o valor atual é substituído pelo novo valor.
- ▶ A chave pode ser um objeto de **qualquer classe**.
 - ▶ Necessário garantir a comparação de chaves.



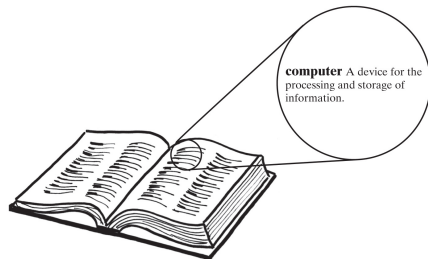


Características:

- ▶ Na maioria das implementações, **a chave é única**.
 - ▶ Ao inserir uma entrada com chave existente, o valor atual é substituído pelo novo valor.
- ▶ A chave pode ser um objeto de **qualquer classe**.
 - ▶ Necessário garantir a comparação de chaves.

Operações:

- ▶ $D[k]$: retorna o valor associado à chave k .
- ▶ $D[k] = v$: insere uma entrada com chave k e valor v .
- ▶ $D.\text{pop}(k)$: remove a entrada com chave k .



Dicionários

Análise de complexidade



Podemos implementar um dicionário usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.



Podemos implementar um dicionário usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

Operação	Arranjo		Encadeamento	
	Não ordenado	Ordenado	Não ordenado	Ordenado
Consulta	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Inserção	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Remoção	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

— a inserção em um arranjo ordenado é feita em $\mathcal{O}(\log n)$ na substituição.



Podemos implementar um dicionário usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

Operação	Arranjo		Encadeamento		Hashing
	Não ordenado	Ordenado	Não ordenado	Ordenado	
Consulta	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Inserção	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Remoção	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$

— a inserção (put) em um arranjo ordenado é feita em $\mathcal{O}(\log n)$ na substituição.

Hashing permite complexidade constante das operações de um dicionário no **caso médio**.

- ▶ É a forma como um dicionário é implementado na maioria das linguagens.
- ▶ Veja: <https://realpython.com/python-dicts>.



A eficiência das operações em um dicionário simples (baseado em um arranjo ordenado) é:

- ▶ $\mathcal{O}(\log n)$ para a **consulta**, usando a busca binária;
- ▶ $\mathcal{O}(n)$ para **inserção** e **remoção**, devido à realocação de elementos no arranjo.



A eficiência das operações em um dicionário simples (baseado em um arranjo ordenado) é:

- ▶ $\mathcal{O}(\log n)$ para a **consulta**, usando a busca binária;
- ▶ $\mathcal{O}(n)$ para **inserção** e **remoção**, devido à realocação de elementos no arranjo.

Ao usar **hashing**, podemos determinar o índice de cada entrada no dicionário, com isso:

- ▶ Não precisamos buscar uma entrada desejada;
- ▶ A posição de uma entrada é definida pela chave e não há realocação;
- ▶ Resultado: mais **eficiência**!

Hashing

Conceitos básicos

A eficiência das operações em um dicionário simples (baseado em um arranjo ordenado) é:

- ▶ $\mathcal{O}(\log n)$ para a **consulta**, usando a busca binária;
- ▶ $\mathcal{O}(n)$ para **inserção** e **remoção**, devido à realocação de elementos no arranjo.

Ao usar **hashing**, podemos determinar o índice de cada entrada no dicionário, com isso:

- ▶ Não precisamos buscar uma entrada desejada;
- ▶ A posição de uma entrada é definida pela chave e não há realocação;
- ▶ Resultado: mais **eficiência**!

Dicionários que usam *hashing* são chamados de **tabelas hash** (ou *hash tables*).



Exemplo

Base de dados de estudantes



Queremos armazenar os dados dos **estudantes** da universidade em um dicionário.

- ▶ **Chave:** matrícula (String);
- ▶ **Valor:** objeto da classe Estudante contendo matrícula, nome, fase e média geral.

Exemplo

Base de dados de estudantes



Queremos armazenar os dados dos **estudantes** da universidade em um dicionário.

- ▶ **Chave:** matrícula (String);
- ▶ **Valor:** objeto da classe Estudante contendo matrícula, nome, fase e média geral.

Um exemplo de **matrícula** é “523-1247”, onde:

- ▶ “523” é o código da universidade, comum a todos os estudantes;
- ▶ “1247” é o código de identificação, único para cada estudante.

Exemplo

Base de dados de estudantes



Queremos armazenar os dados dos **estudantes** da universidade em um dicionário.

- ▶ **Chave:** matrícula (String);
- ▶ **Valor:** objeto da classe `Estudante` contendo matrícula, nome, fase e média geral.

Um exemplo de **matrícula** é “523-1247”, onde:

- ▶ “523” é o código da universidade, comum a todos os estudantes;
- ▶ “1247” é o código de identificação, único para cada estudante.

O código de identificação varia de “0000” a “9999”.

- ▶ **Ideia:** usar o código de identificação como posição (índice) da entrada no dicionário!
- ▶ Esse é o princípio do *hashing*.



Exemplo

Estudantes – tabela *hash*

Chaves

Valores

$\langle \text{"523-1247"}, \text{"Ross"} \rangle$

$\langle \text{"523-3761"}, \text{"Monica"} \rangle$

$\langle \text{"523-8147"}, \text{"Rachel"} \rangle$

$\langle \text{"523-0158"}, \text{"Joey"} \rangle$

$\langle \text{"523-6358"}, \text{"Chandler"} \rangle$

Exemplo

Estudantes – tabela *hash*



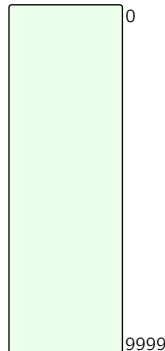
Chaves	Valores	Código <i>hash</i>
$\langle \text{"523-1247"}, \text{"Ross"} \rangle$	\longrightarrow	1247
$\langle \text{"523-3761"}, \text{"Monica"} \rangle$	\longrightarrow	3761
$\langle \text{"523-8147"}, \text{"Rachel"} \rangle$	\longrightarrow	8147
$\langle \text{"523-0158"}, \text{"Joey"} \rangle$	\longrightarrow	158
$\langle \text{"523-6358"}, \text{"Chandler"} \rangle$	\longrightarrow	6358

Exemplo

Estudantes – tabela *hash*

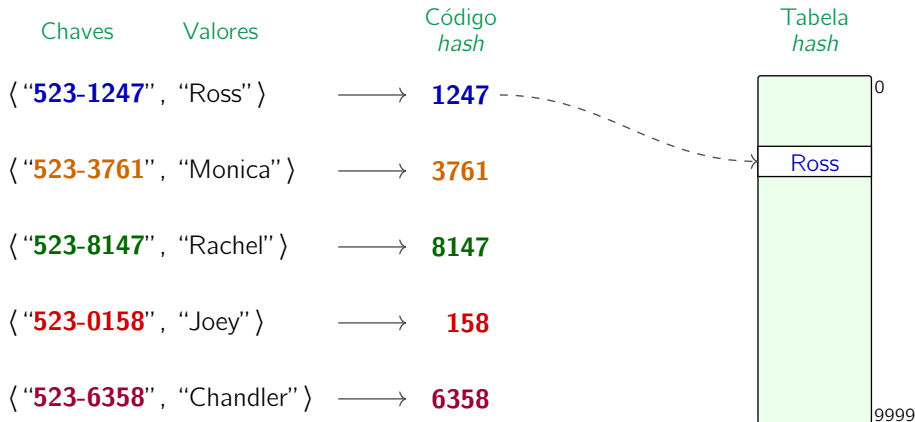
Chaves	Valores	Código <i>hash</i>
$\langle \text{"523-1247"}, \text{"Ross"} \rangle$	→	1247
$\langle \text{"523-3761"}, \text{"Monica"} \rangle$	→	3761
$\langle \text{"523-8147"}, \text{"Rachel"} \rangle$	→	8147
$\langle \text{"523-0158"}, \text{"Joey"} \rangle$	→	158
$\langle \text{"523-6358"}, \text{"Chandler"} \rangle$	→	6358

Tabela
hash



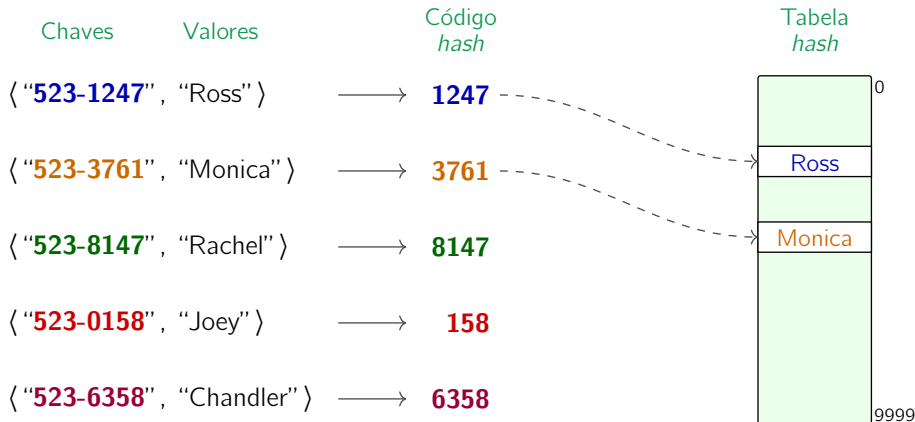
Exemplo

Estudantes – tabela *hash*



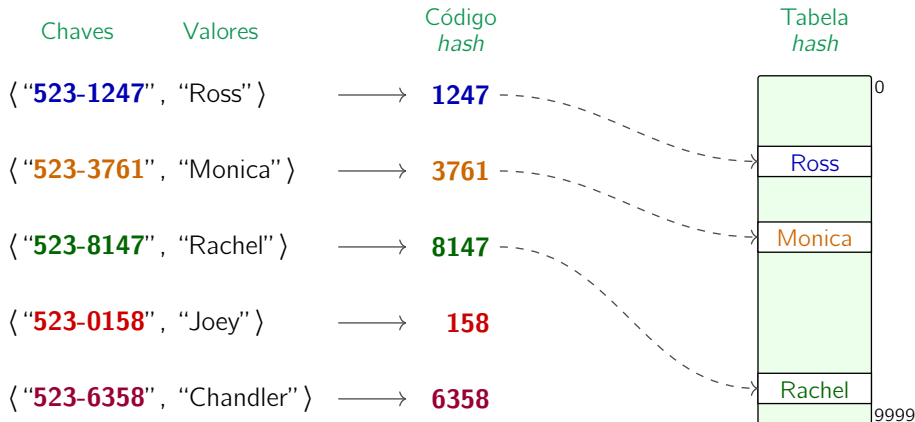
Exemplo

Estudantes – tabela *hash*



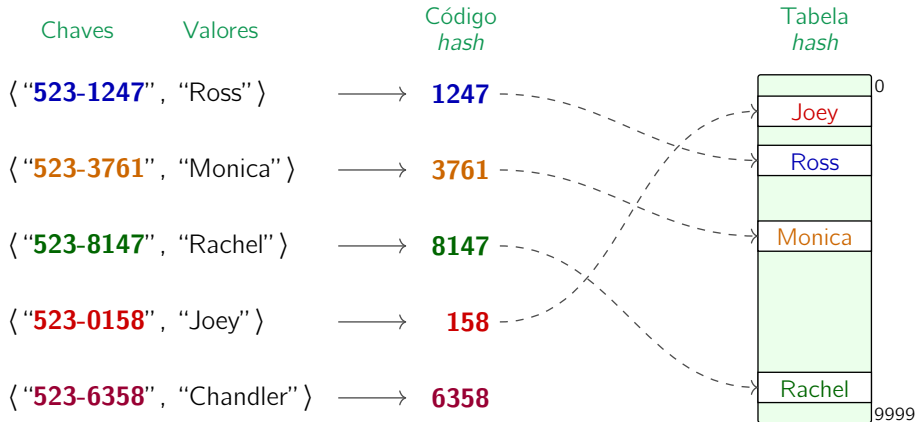
Exemplo

Estudantes – tabela *hash*



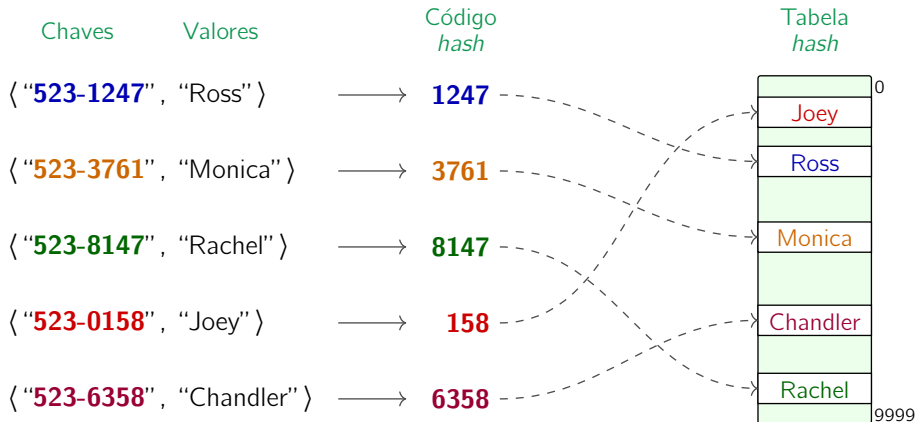
Exemplo

Estudantes – tabela *hash*



Exemplo

Estudantes – tabela *hash*



Função *hash*

Função *hash* perfeita



Função hash: dada a chave, “calcula” o índice onde a entrada está (ou será) armazenada.

► Exemplo: $h(\text{"523-1247"}) \rightarrow 1247$.

Função *hash*

Função *hash* perfeita



Função hash: dada a chave, “calcula” o índice onde a entrada está (ou será) armazenada.

► Exemplo: $h(\text{"523-1247"}) \rightarrow 1247$.

Função hash perfeita: mapeia cada chave para um índice diferente do vetor.

► Com isso, temos acesso à entrada em $\mathcal{O}(1)$, i.e. tempo constante!

Função *hash*

Função *hash* perfeita



Função *hash*: dada a chave, “calcula” o índice onde a entrada está (ou será) armazenada.

- ▶ Exemplo: $h(\text{"523-1247"}) \rightarrow 1247$.

Função *hash* perfeita: mapeia cada chave para um índice diferente do vetor.

- ▶ Com isso, temos acesso à entrada em $\mathcal{O}(1)$, i.e. tempo constante!

Na prática, nem sempre conseguimos projetar uma função *hash* perfeita.

- ▶ Não conhecemos todos os possíveis valores de chave;
- ▶ A tabela possui capacidade menor que o número de entradas possíveis.

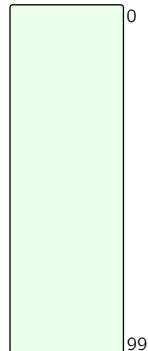


Exemplo

Estudantes – tabela *hash* com 100 entradas

Chaves	Valores	Código <i>hash</i>
$\langle \text{"523-1247"}, \text{"Ross"} \rangle$	\longrightarrow	1247
$\langle \text{"523-3761"}, \text{"Monica"} \rangle$	\longrightarrow	3761
$\langle \text{"523-8147"}, \text{"Rachel"} \rangle$	\longrightarrow	8147
$\langle \text{"523-0158"}, \text{"Joey"} \rangle$	\longrightarrow	158
$\langle \text{"523-6358"}, \text{"Chandler"} \rangle$	\longrightarrow	6358

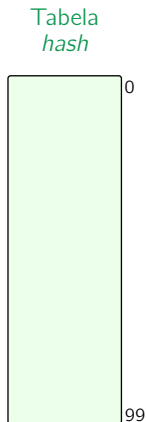
Tabela
hash



Exemplo

Estudantes – tabela *hash* com 100 entradas

Chaves	Valores		Código <i>hash</i>		Índice (% 100)
⟨ “523-1247”, “Ross” ⟩		→	1247	→	47
⟨ “523-3761”, “Monica” ⟩		→	3761	→	61
⟨ “523-8147”, “Rachel” ⟩		→	8147	→	47
⟨ “523-0158”, “Joey” ⟩		→	158	→	58
⟨ “523-6358”, “Chandler” ⟩		→	6358	→	58



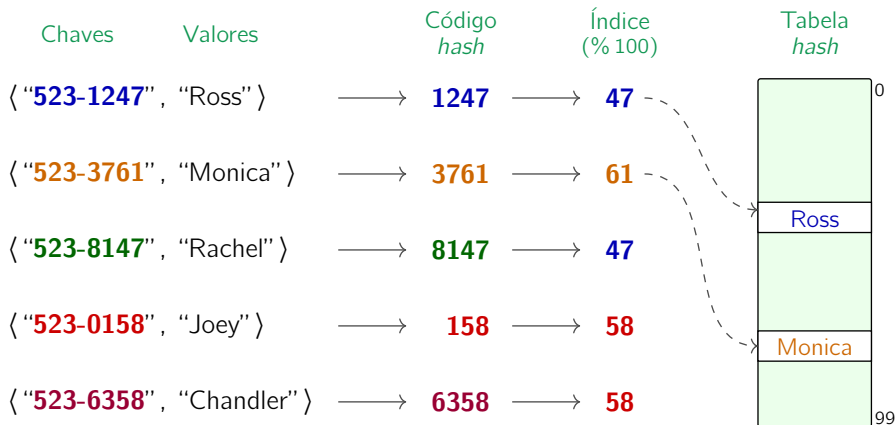
Exemplo

Estudantes – tabela *hash* com 100 entradas

Chaves	Valores	Código <i>hash</i>	Índice (% 100)	Tabela <i>hash</i>
⟨ “523-1247”, “Ross” ⟩	→	1247	→ 47	<div>0</div> <div>Ross</div> <div>99</div>
⟨ “523-3761”, “Monica” ⟩	→	3761	→ 61	
⟨ “523-8147”, “Rachel” ⟩	→	8147	→ 47	
⟨ “523-0158”, “Joey” ⟩	→	158	→ 58	
⟨ “523-6358”, “Chandler” ⟩	→	6358	→ 58	

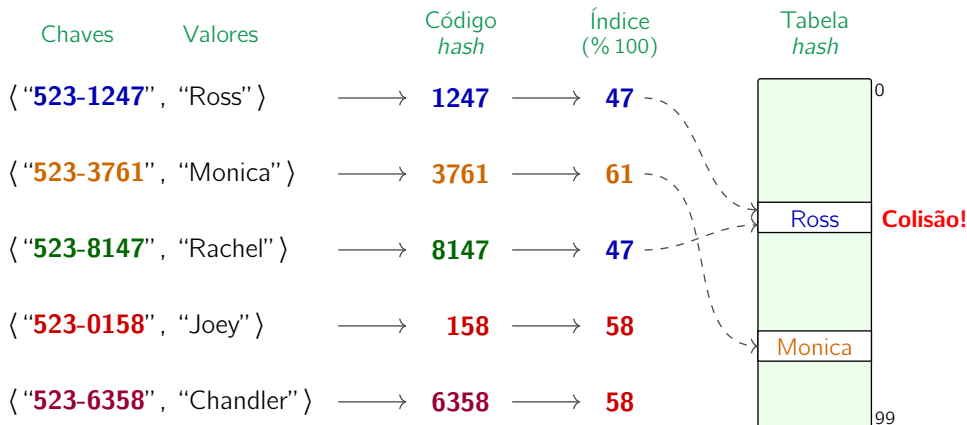
Exemplo

Estudantes – tabela *hash* com 100 entradas



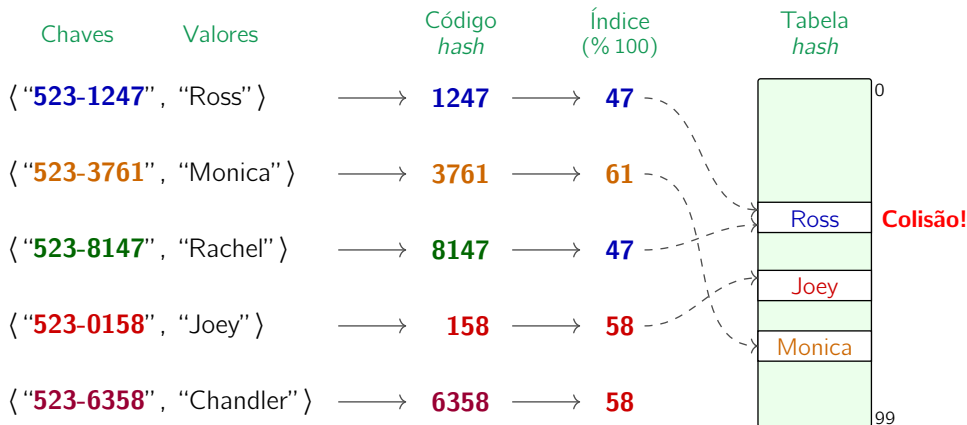
Exemplo

Estudantes – tabela *hash* com 100 entradas



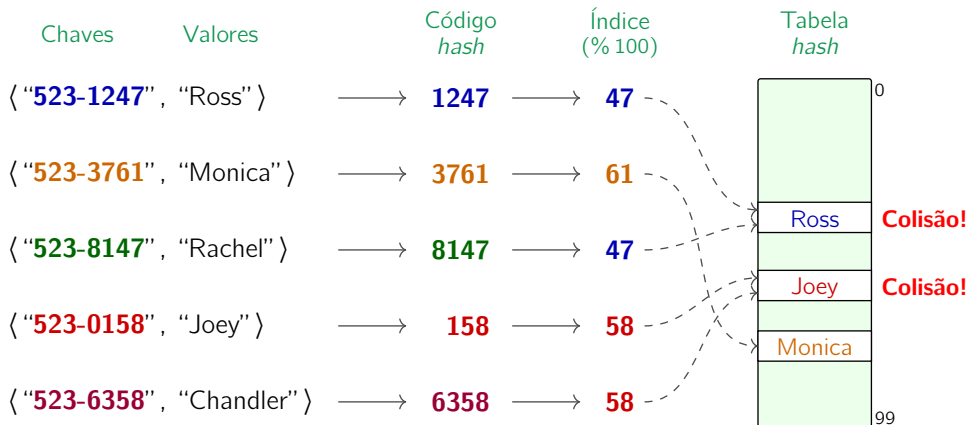
Exemplo

Estudantes – tabela *hash* com 100 entradas



Exemplo

Estudantes – tabela *hash* com 100 entradas



Função *hash*

Função *hash* não perfeita



Em resumo, uma função *hash* típica possui duas etapas:

1. Converter a chave para um valor inteiro, chamado **código hash** (ou *hash code*).
2. Comprimir o código *hash* para o intervalo de índices da tabela *hash* (operação **módulo**).

Função *hash*

Função *hash* não perfeita

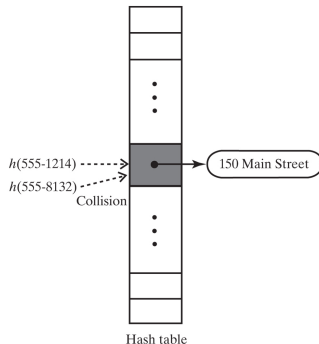


Em resumo, uma função *hash* típica possui duas etapas:

1. Converter a chave para um valor inteiro, chamado **código hash** (ou *hash code*).
2. Comprimir o código *hash* para o intervalo de índices da tabela *hash* (operação **módulo**).

Uma boa função *hash* deve apresentar duas características:

- ▶ Ser **eficiente** para computar.
- ▶ Minimizar a ocorrência de **colisões**.
 - ▶ i.e., distribuir as chaves de maneira uniforme.





Colisão: a função *hash* calcula um índice da tabela *hash* já ocupado por outro objeto.



Colisão: a função *hash* calcula um índice da tabela *hash* já ocupado por outro objeto.

Neste caso, precisamos **resolver a colisão**. Para isso, temos duas estratégias:

1. Encontrar outra posição (livre) na estrutura. —→ **endereçamento aberto** (*probing*)
2. Armazenar mais de uma entrada no mesmo índice. —→ **encadeamento** (*chaining*)



Colisão: a função *hash* calcula um índice da tabela *hash* já ocupado por outro objeto.

Neste caso, precisamos **resolver a colisão**. Para isso, temos duas estratégias:

1. Encontrar outra posição (livre) na estrutura. → **endereçamento aberto** (*probing*)
2. Armazenar mais de uma entrada no mesmo índice. → **encadeamento** (*chaining*)

Resolução de colisões com **encadeamento**:

1. A tabela *hash* armazena uma coleção de entradas (*bucket*) em cada posição/índice.
2. Essa coleção pode ser implementada por outro arranjo ou por uma lista encadeada.

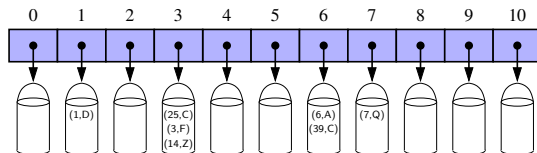
Colisão: a função *hash* calcula um índice da tabela *hash* já ocupado por outro objeto.

Neste caso, precisamos **resolver a colisão**. Para isso, temos duas estratégias:

1. Encontrar outra posição (livre) na estrutura. → **endereçamento aberto** (*probing*)
2. Armazenar mais de uma entrada no mesmo índice. → **encadeamento** (*chaining*)

Resolução de colisões com **encadeamento**:

1. A tabela *hash* armazena uma coleção de entradas (*bucket*) em cada posição/índice.
2. Essa coleção pode ser implementada por outro arranjo ou por uma lista encadeada.



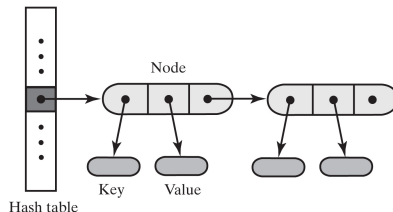
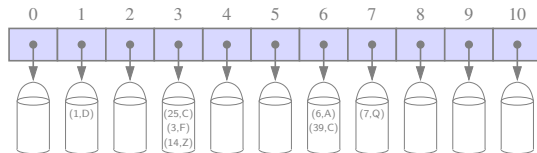
Colisão: a função *hash* calcula um índice da tabela *hash* já ocupado por outro objeto.

Neste caso, precisamos **resolver a colisão**. Para isso, temos duas estratégias:

1. Encontrar outra posição (livre) na estrutura. → **endereçamento aberto** (*probing*)
2. Armazenar mais de uma entrada no mesmo índice. → **encadeamento** (*chaining*)

Resolução de colisões com **encadeamento**:

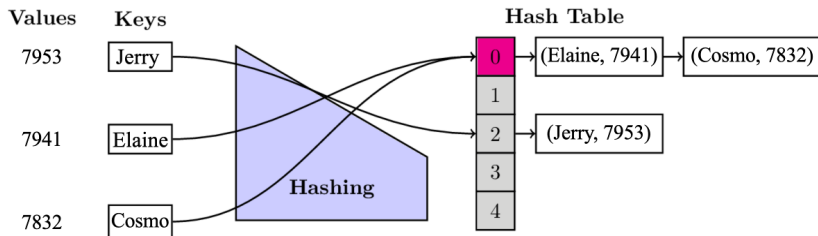
1. A tabela *hash* armazena uma coleção de entradas (*bucket*) em cada posição/índice.
2. Essa coleção pode ser implementada por outro arranjo ou por uma lista encadeada.



Tabelas *hash*

Definição


Uma **tabela hash** é composta por um **arranjo**, onde cada posição armazena uma coleção de entradas (chave e valor). Essa coleção é usualmente implementada por uma **lista encadeada**. A **função hash** é responsável por mapear chaves para posições do arranjo.





Exemplo

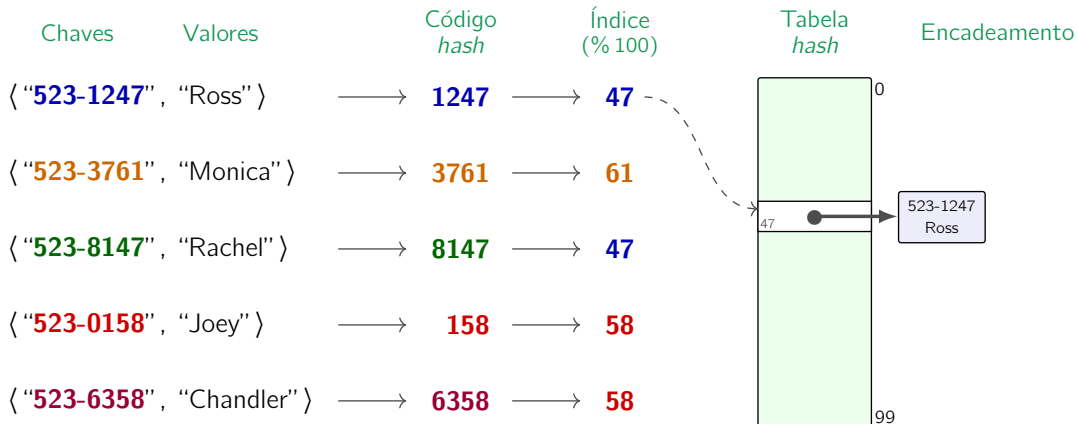
Estudantes – tabela *hash* com 100 entradas e resolução de colisões

Chaves	Valores	Código <i>hash</i>	Índice (% 100)	Tabela <i>hash</i>
⟨ “523-1247”, “Ross” ⟩	→	1247	→ 47	 <div>0</div> <div>99</div>
⟨ “523-3761”, “Monica” ⟩	→	3761	→ 61	
⟨ “523-8147”, “Rachel” ⟩	→	8147	→ 47	
⟨ “523-0158”, “Joey” ⟩	→	158	→ 58	
⟨ “523-6358”, “Chandler” ⟩	→	6358	→ 58	



Exemplo

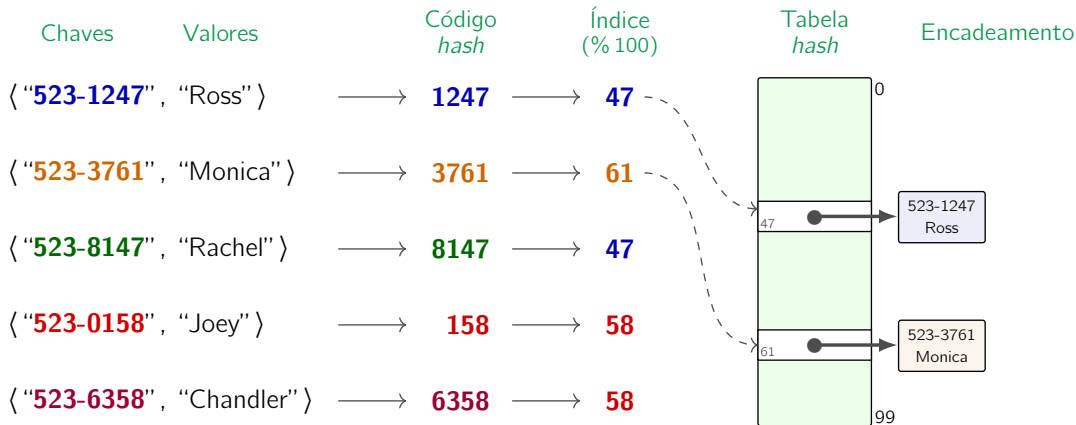
Estudantes – tabela *hash* com 100 entradas e resolução de colisões





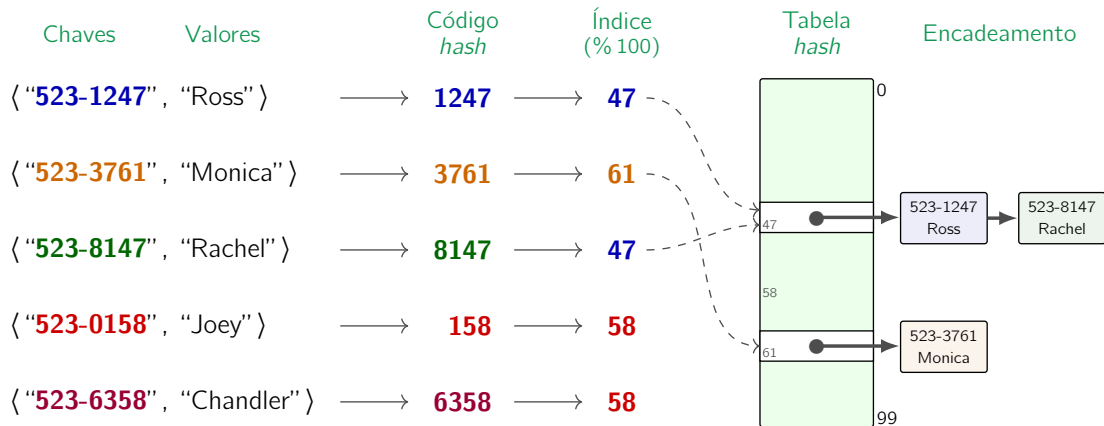
Exemplo

Estudantes – tabela *hash* com 100 entradas e resolução de colisões



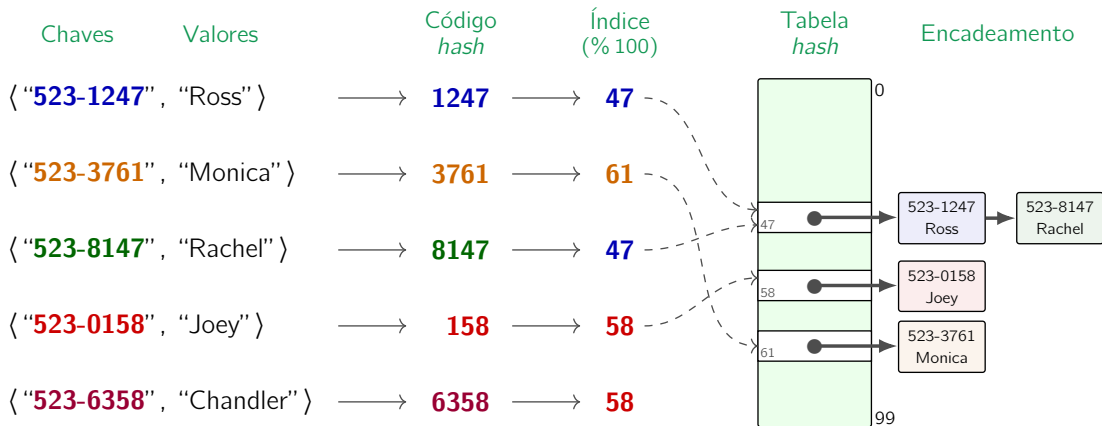
Exemplo

Estudantes – tabela *hash* com 100 entradas e resolução de colisões



Exemplo

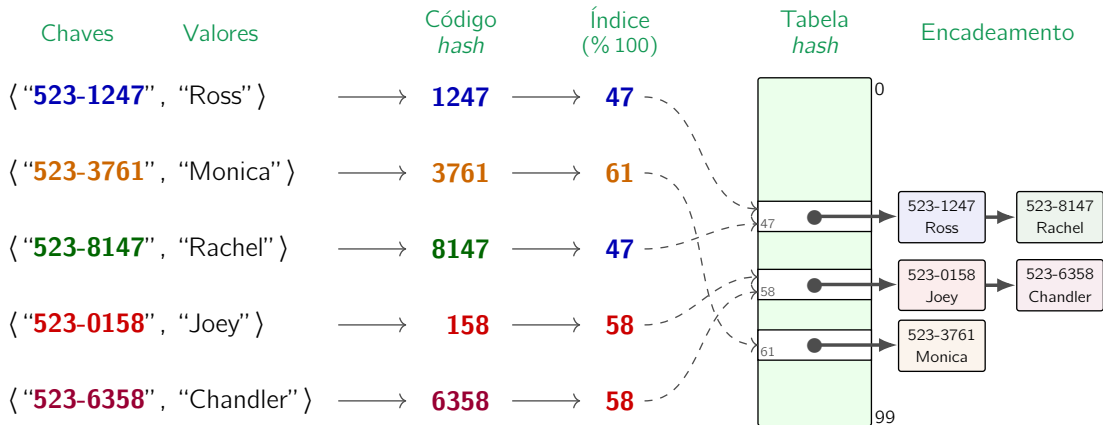
Estudantes – tabela *hash* com 100 entradas e resolução de colisões





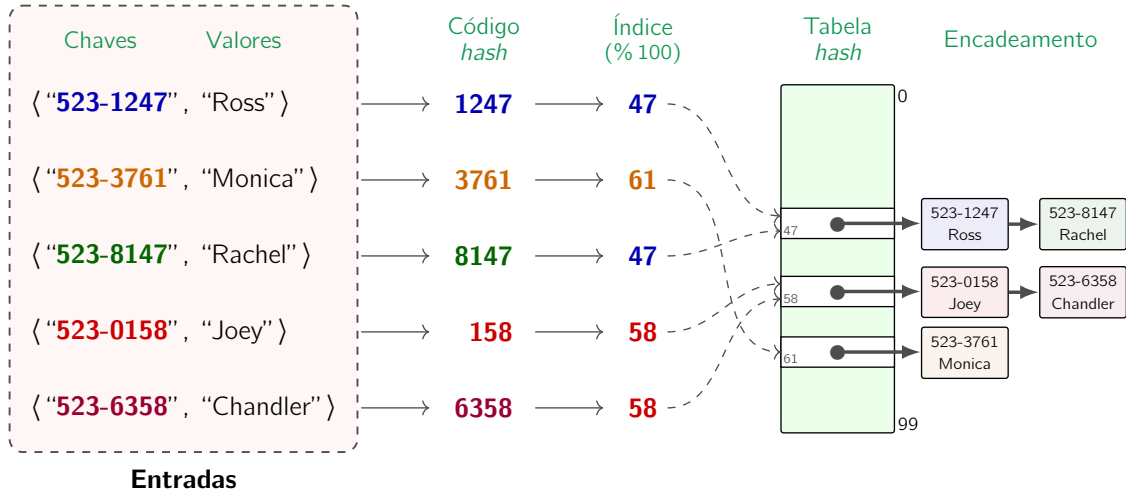
Exemplo

Estudantes – tabela *hash* com 100 entradas e resolução de colisões



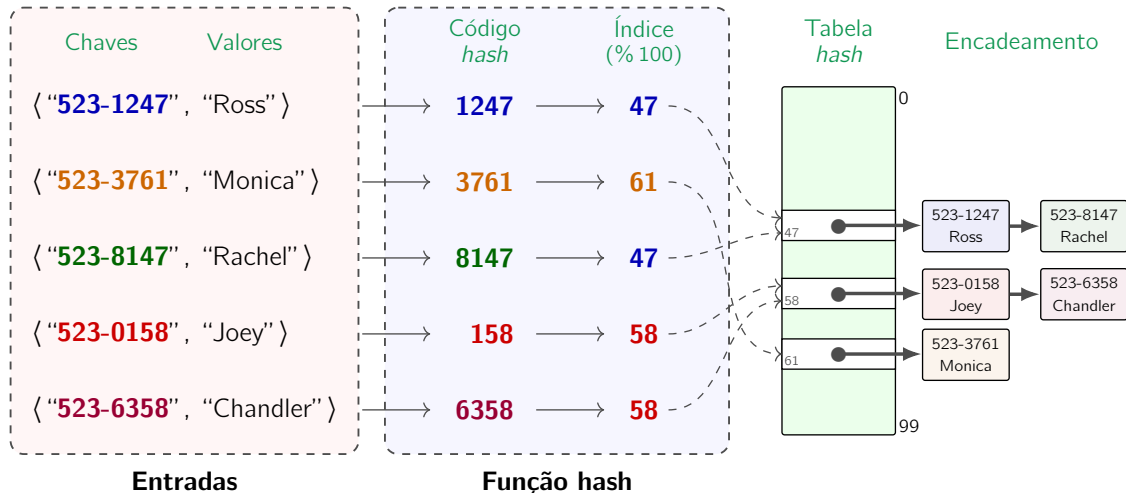
Exemplo

Estudantes – tabela *hash* com 100 entradas e resolução de colisões



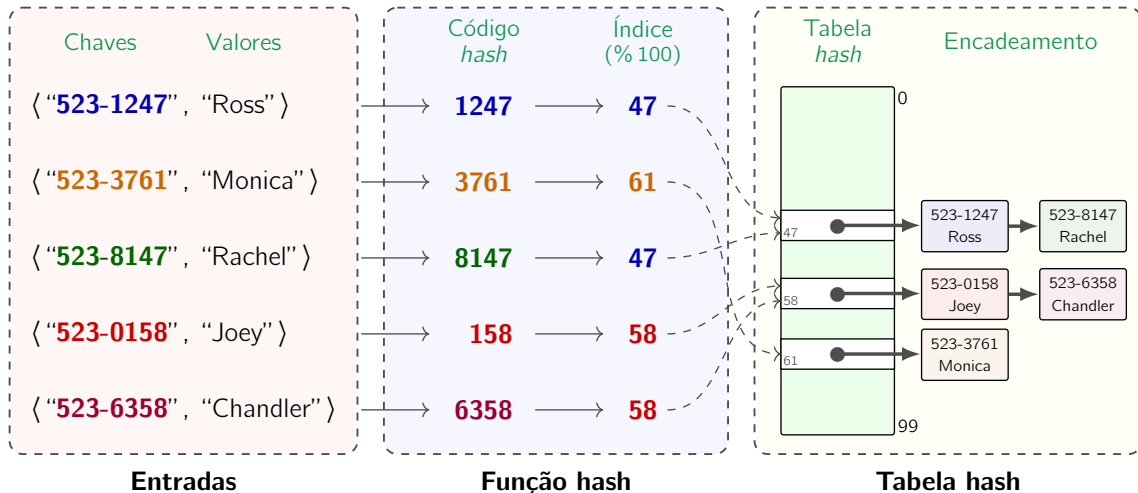
Exemplo

Estudantes – tabela *hash* com 100 entradas e resolução de colisões



Exemplo

Estudantes – tabela *hash* com 100 entradas e resolução de colisões





O Java fornece a conversão de qualquer tipo em um valor inteiro pelo método `hashCode`.

- ▶ Todos os tipos herdam esse método de `Object`.



O Java fornece a conversão de qualquer tipo em um valor inteiro pelo método `hashCode`.

- ▶ Todos os tipos herdam esse método de `Object`.

Para tipos primitivos (inteiro, string), o Java fornece implementações específicas do `hashCode`.

- ▶ Pode ser usado diretamente pelas funções *hash*.



O Java fornece a conversão de qualquer tipo em um valor inteiro pelo método `hashCode`.

- ▶ Todos os tipos herdam esse método de `Object`.

Para tipos primitivos (inteiro, `string`), o Java fornece implementações específicas do `hashCode`.

- ▶ Pode ser usado diretamente pelas funções *hash*.

Para outros tipos (classes), o Java usa o endereço de memória do objeto para o `hashCode`.

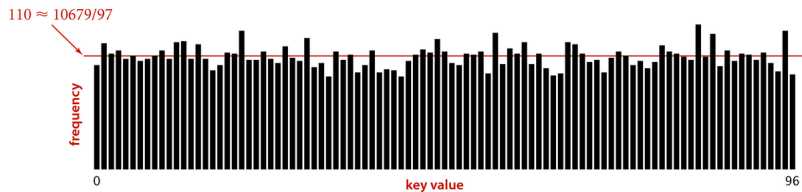
- ▶ Neste caso, objetos com os mesmos valores, mas armazenados em locais diferentes recebem códigos *hash* distintos, o que não é desejado.
- ▶ Recomenda-se sobrescrever `hashCode` e implementar a geração do código *hash*.

Exemplo

Distribuição da função *hash* para strings



Valores *hash* (i.e. índices) calculados a partir do hashCode padrão para o conjunto de palavras (excluídas as repetidas) do livro “*A Tale of Two Cities*”, para um arranjo com 97 posições.



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, $M = 97$)



A complexidade das operações está diretamente relacionada à qualidade da função *hash*.

- ▶ Uma **função hash perfeita** mapeia cada chave para um índice distinto (**melhor caso**). Logo, o acesso à estrutura (para consulta, inserção e remoção) ocorre em $\mathcal{O}(1)$.
- ▶ Uma **função hash não perfeita** pode mapear todas as chaves para um mesmo índice, no **pior caso**, gerando máxima colisão. Neste caso, as operações custarão $\mathcal{O}(n)$.



A complexidade das operações está diretamente relacionada à qualidade da função *hash*.

- ▶ Uma **função hash perfeita** mapeia cada chave para um índice distinto (**melhor caso**). Logo, o acesso à estrutura (para consulta, inserção e remoção) ocorre em $\mathcal{O}(1)$.
- ▶ Uma **função hash não perfeita** pode mapear todas as chaves para um mesmo índice, no **pior caso**, gerando máxima colisão. Neste caso, as operações custarão $\mathcal{O}(n)$.

Felizmente, implementações de tabelas *hash* usam técnicas para melhorar o desempenho, como:

- ▶ Controlar o fator de carga da estrutura, aumentando dinamicamente sua capacidade;
- ▶ Usar valores de capacidade que minimizam a ocorrência de colisões.



A complexidade das operações está diretamente relacionada à qualidade da função *hash*.

- ▶ Uma **função hash perfeita** mapeia cada chave para um índice distinto (**melhor caso**). Logo, o acesso à estrutura (para consulta, inserção e remoção) ocorre em $\mathcal{O}(1)$.
- ▶ Uma **função hash não perfeita** pode mapear todas as chaves para um mesmo índice, no **pior caso**, gerando máxima colisão. Neste caso, as operações custarão $\mathcal{O}(n)$.

Felizmente, implementações de tabelas *hash* usam técnicas para melhorar o desempenho, como:

- ▶ Controlar o fator de carga da estrutura, aumentando dinamicamente sua capacidade;
- ▶ Usar valores de capacidade que minimizam a ocorrência de colisões.

Com isso, as operações de consulta, inserção e remoção de uma tabela *hash* possuem complexidade de tempo $\mathcal{O}(1)$ no **caso médio**!

45RPE – Resolução de Problemas com Estruturas de Dados
Prof. Marcelo de Souza