Tabelas hash

Mapas eficientes

Prof. Marcelo de Souza

45EST – Algoritmos e Estruturas de Dados
Universidade do Estado de Santa Catarina

UDESC

Material de apoio

Leitura principal:

- Capítulo 10 de Goodrich et al. (2014)[1] – Mapas, tabelas hash e skip lists.
- Capítulo 22 de Carrano e Henry (2018)[2] – *Introducing hashing*.

Leitura complementar:

- Capítulo 8 de Szwarcfiter e Markenzon (2009)[3] – Tabelas de dispersão.

[1] Michael T Goodrich et al. (2014). *Data structures and algorithms in Java*. 6ª ed. John Wiley & Sons.

[2] Frank M. Carrano e Timothy M. Henry (2018). *Data Structures and Abstractions with Java (5th Edition)*. 5th. Pearson.

[3] Jayme Luiz Szwarcfiter e Lilian Markenzon (2009). *Estruturas de Dados e seus Algoritmos*. Vol. 2. Livros Técnicos e Científicos.

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.

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**!

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

Base de dados de estudantes

Uma tabela *hash* para os estudantes

Função *hash*

Função *hash* perfeita

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

- Exemplo: $\mathtt{h}(\text{“523-1247”}) \rightarrow 1247$.

Função *hash*

Função *hash* perfeita

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

- Exemplo: $\mathfrak{h}(\text{“523-1247”}) \to 1247$.

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: $\mathfrak{h}(\text{“523-1247”}) \rightarrow 1247$.

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

Base de dados de estudantes

Uma tabela *hash* com capacidade de 100 entradas para os estudantes

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*.

Função *hash*

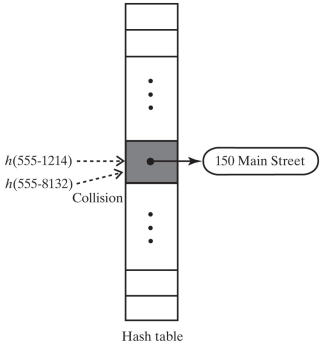
Função *hash* não perfeita

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*.

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.



Colisões

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

Colisões

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. $\longrightarrow$ **endereçamento aberto** (*probing*)
2. Armazenar mais de uma entrada no mesmo índice. $\longrightarrow$ **encadeamento** (*chaining*)

Colisões

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. $\longrightarrow$ **endereçamento aberto** (*probing*)
2. Armazenar mais de uma entrada no mesmo índice. $\longrightarrow$ **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.

Colisões

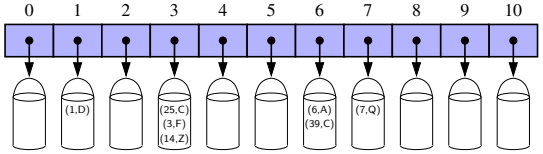
Colisão: a função *hash* calcula um índice da tabela *hash* já ocupado por outro objeto.

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.



Colisões

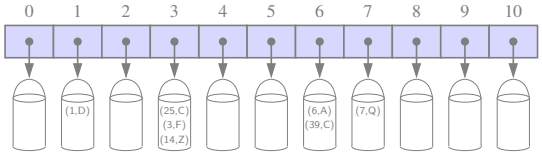
Colisão: a função *hash* calcula um índice da tabela *hash* já ocupado por outro objeto.

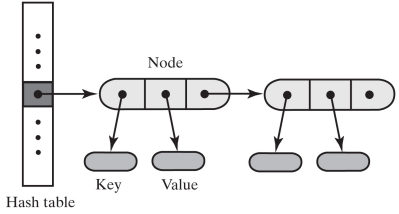
Neste caso, precisamos **resolver a colisão**. Para isso, temos duas estratégias:

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.

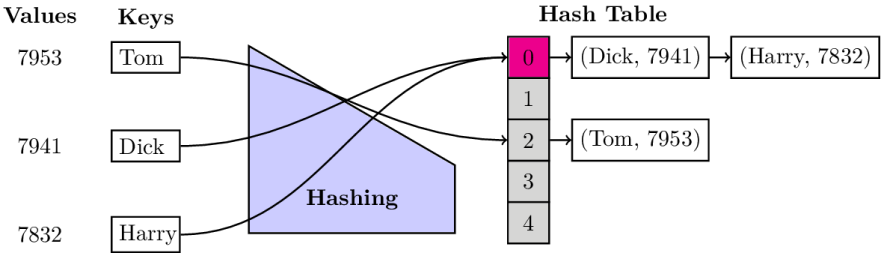




Tabelas *hash*

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

Base de dados de estudantes

Uma tabela *hash* com capacidade de 100 entradas e resolução de colisões para os estudantes

Funções *hash* em Java

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

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

Funções *hash* em Java

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*.

Funções *hash* em Java

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*.

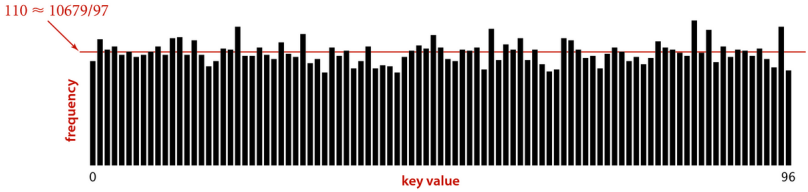
Para outros tipos (classes), o Java usa o endereço de memória do objeto para o `hashCode`.

- 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, $M = 97$)

Análise de complexidade

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, a operações custarão $\mathcal{O}(n)$.

Análise de complexidade

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, a 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.

Análise de complexidade

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, a 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**!

45EST – Algoritmos e Estruturas de Dados
Prof. Marcelo de Souza