

Loiane Groner

Estruturas de dados e algoritmos com JavaScript

2^a Edição



Escreva um código JavaScript complexo e eficaz usando
a mais recente ECMAScript

novatec

Packt

Estruturas de dados e algoritmos em JavaScript

Escreva um código JavaScript complexo e eficaz
usando a mais recente ECMAScript

2^a Edição

Loiane Groner

Packt

Novatec

Copyright © Packt Publishing 2018. First published in the English language under the title ‘Learning JavaScript Data Structures and Algorithms - Third Edition – (9781788623872)’

Copyright © Packt Publishing 2018. Publicação original em inglês intitulada ‘Learning JavaScript Data Structures and Algorithms - Third Edition – (9781788623872)’

Esta tradução é publicada e vendida com a permissão da Packt Publishing.

© Novatec Editora Ltda. [2018].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia A. Kinoshita

Revisão gramatical: Tássia Carvalho

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-728-2

Histórico de edições impressas:

Fevereiro/2019 Segunda edição

Abril/2018 Primeira reimpressão

Março/2017 Primeira edição (ISBN: 978-85-7522-553-0)

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

E-mail: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

*Aos meus pais, por seu amor e assistência, e por me guiarem durante todos
esses anos.*

*Ao meu marido, pelo apoio e por ser o meu amado companheiro na nossa
jornada de vida.*

– Loiane Groner

Sumário

Colaboradores

Prefácio

Capítulo 1 ■ JavaScript – uma visão geral rápida

Estrutura de dados e algoritmos em JavaScript

Configurando o ambiente

Configuração mínima para trabalhar com JavaScript

Usando servidores web

http-server do Node.js

Básico sobre o JavaScript

Variáveis

Escopo das variáveis

Operadores

Verdadeiro e falso

Funções dos operadores de igualdade (== e ===)

Estruturas de controle

InSTRUÇÕES condicionais

Laços

Funções

Programação orientada a objetos em Javascript

Depuração e ferramentas

Depuração com o VSCode

Resumo

Capítulo 2 ■ Visão geral sobre ECMAScript e TypeScript

ECMAScript ou JavaScript?

ES6, ES2015, ES7, ES2016, ES8, ES2017 e ES.Next

Tabela de compatibilidade

Usando o Babel.js

[Funcionalidades das versões ECMAScript 2015+](#)

[let e const no lugar de var](#)

[Escopo de variáveis com let e const](#)

[Templates literais](#)

[Funções de seta](#)

[Valores default para parâmetros de funções](#)

[Declarando os operadores de espalhamento e rest](#)

[Propriedades melhoradas de objetos](#)

[Programação orientada a objetos com classes](#)

[Herança](#)

[Trabalhando com getters e setters](#)

[Operador de exponencial](#)

[Módulos](#)

[Executando módulos ES2015 no navegador e com o Node.js](#)

[Usando importações nativas da ES2015 no Node.js](#)

[Executando módulos ES2015 no navegador](#)

[Compatibilidade de versões anteriores a ES2015+](#)

[Introdução ao TypeScript](#)

[Inferência de tipo](#)

[Interfaces](#)

[Genéricos](#)

[Outras funcionalidades do TypeScript](#)

[Verificações do TypeScript em tempo de compilação em arquivos](#)

[JavaScript](#)

[Resumo](#)

Capítulo 3 ■ Arrays

[Por que devemos usar arrays?](#)

[Criando e inicializando arrays](#)

[Acessando elementos e fazendo uma iteração em um array](#)

[Acrescentando elementos](#)

[Inserindo um elemento no final do array](#)

[Usando o método push](#)

[Inserindo um elemento na primeira posição](#)

[Usando o método unshift](#)

[Removendo elementos](#)

[Removendo um elemento do final do array](#)

[Removendo um elemento da primeira posição](#)

[Usando o método shift](#)

[Adicionando e removendo elementos de uma posição específica](#)

[Arrays bidimensionais e multidimensionais](#)

[Iterando pelos elementos de arrays bidimensionais](#)

[Arrays multidimensionais](#)

[Referências para métodos de array em JavaScript](#)

[Juntando vários arrays](#)

[Funções de iteração](#)

[Iterando com o método every](#)

[Iterando com o método some](#)

[Iterando com forEach](#)

[Usando map e filter](#)

[Usando o método reduce](#)

[ECMAScript 6 e as novas funcionalidades de array](#)

[Iterando com o laço for...of](#)

[Usando o objeto @@iterator](#)

[Métodos entries, keys e values de array](#)

[Usando o método from](#)

[Usando o método Array.of](#)

[Usando o método fill](#)

[Usando o método copyWithin](#)

[Ordenando elementos](#)

[Ordenação personalizada](#)

[Ordenando strings](#)

[Pesquisa](#)

[ECMAScript 2015 – os métodos find e findIndex](#)

[ECMAScript 2016 – usando o método includes](#)

[Convertendo um array em uma string](#)

[Classe TypedArray](#)

[Arrays em TypeScript](#)

[Resumo](#)

[Capítulo 4 ■ Pilhas](#)

[Criação de uma biblioteca de estruturas de dados e algoritmos JavaScript](#)
[Estrutura de dados de pilha](#)
[Criando uma classe Stack baseada em array](#)
[Push de elementos na pilha](#)
[Pop de elementos da pilha](#)
[Dando uma espiada no elemento que está no topo da pilha](#)
[Verificando se a pilha está vazia](#)
[Limpando os elementos da pilha](#)
[Usando a classe Stack](#)
[Criando uma classe JavaScript Stack baseada em objeto](#)
[Push de elementos na pilha](#)
[Verificando se a pilha está vazia e o seu tamanho](#)
[Pop de elementos da pilha](#)
[Dando uma espiada no topo e limpando a pilha](#)
[Criando o método toString](#)
[Protegendo os elementos internos da estrutura de dados](#)
[Convenção de nomenclatura com underscore](#)
[Classes ES2015 com símbolos no escopo](#)
[Classes ES2015 com WeakMap](#)
[Proposta para campos de classe na ECMAScript](#)
[Resolvendo problemas usando pilhas](#)
[Convertendo números decimais para binários](#)
[Algoritmo conversor de base](#)
[Resumo](#)

Capítulo 5 ■ Filas e deque

[Estrutura de dados de fila](#)
[Criando a classe Queue](#)
[Inserção de elementos na fila](#)
[Remoção de elementos da fila](#)
[Dando uma espiada no elemento que está na frente da fila](#)
[Verificando se a pilha está vazia e o seu tamanho](#)
[Limpando a fila](#)
[Criando o método toString](#)
[Usando a classe Queue](#)

[Estrutura de dados de deque](#)
[Criando a classe Deque](#)
[Adicionando elementos na frente do deque](#)
[Usando a classe Deque](#)
[Resolvendo problemas usando filas e deque](#)
[Fila circular – Batata Quente](#)
[Verificador de palíndromo](#)
[Filas de tarefas em JavaScript](#)
[Resumo](#)

[Capítulo 6 ■ Listas ligadas](#)

[Estrutura de dados da lista ligada](#)
[Criando a classe LinkedList](#)
[Inserindo elementos no final da lista ligada](#)
[Removendo elementos de uma posição específica da lista ligada](#)
[Percorrendo a lista com um laço até alcançar a posição desejada](#)
[Refatorando o método remove](#)
[Inserindo um elemento em qualquer posição](#)
[Método indexOf: devolvendo a posição de um elemento](#)
[Removendo um elemento da lista ligada](#)
[Métodos isEmpty, size e getHead](#)
[Método toString](#)
[Listas duplamente ligadas](#)
[Inserindo um novo elemento em qualquer posição](#)
[Removendo elementos de qualquer posição](#)
[Listas ligadas circulares](#)
[Inserindo um novo elemento em qualquer posição](#)
[Removendo elementos de qualquer posição](#)
[Listas ligadas ordenadas](#)
[Inserindo elementos na ordem](#)
[Criando a classe StackLinkedList](#)
[Resumo](#)

[Capítulo 7 ■ Conjuntos](#)

[Estruturando um conjunto de dados](#)

[Criando uma classe Set](#)
[Método has\(element\)](#)
[Método add](#)
[Métodos delete e clear](#)
[Método size](#)
[Método values](#)
[Usando a classe Set](#)
[Operações em conjuntos](#)
[União de conjuntos](#)
[Intersecção de conjuntos](#)
[Aperfeiçoando o método intersection](#)
[Diferença entre conjuntos](#)
[Subconjunto](#)
[ECMAScript 2015 – a classe Set](#)
[Operações com a classe Set da ES2015](#)
[Simulando a operação de união](#)
[Simulando a operação de intersecção](#)
[Simulando a operação de diferença](#)
[Usando o operador de espalhamento](#)
[Multiconjuntos ou bags](#)
[Resumo](#)

Capítulo 8 ■ Dicionários e hashes

[Estrutura de dados de dicionário](#)
[Criando a classe Dictionary](#)
[Verificando se uma chave está presente no dicionário](#)
[Definindo uma chave e um valor no dicionário, e a classe ValuePair](#)
[Removendo um valor do dicionário](#)
[Obtendo um valor do dicionário](#)
[Métodos keys, values e valuePairs](#)
[Iterando pelos ValuePairs do dicionário com forEach](#)
[Métodos clear, size, isEmpty e toString](#)
[Usando a classe Dictionary](#)
[Tabela hash](#)
[Criando uma classe HashTable](#)

[Criando uma função de hash](#)
[Inserindo uma chave e um valor na tabela hash](#)
[Obtendo um valor da tabela hash](#)
[Removendo um valor da tabela hash](#)
[Usando a classe HashTable](#)
[Tabela hash versus conjunto hash](#)
[Tratando colisões nas tabelas hash](#)
[Encadeamento separado](#)
[Método put](#)
[Método get](#)
[Método remove](#)
[Sondagem linear](#)
[Método put](#)
[Método get](#)
[Método remove](#)
[Criando funções melhores de hash](#)
[Classe Map da ES2015](#)
[Classes WeakMap e WeakSet da ES2015](#)
[Resumo](#)

Capítulo 9 ■ Recursão

[Entendendo a recursão](#)
[Calculando o fatorial de um número](#)
[Fatorial iterativo](#)
[Fatorial recursivo](#)
[Pilha de chamadas](#)
[Limitação do tamanho da pilha de chamadas em JavaScript](#)
[Sequência de Fibonacci](#)
[Fibonacci iterativo](#)
[Fibonacci recursivo](#)
[Fibonacci com memoização](#)
[Por que usar recursão? É mais rápido?](#)
[Resumo](#)

Capítulo 10 ■ Árvores

[Estrutura de dados de árvore](#)

[Terminologia de árvores](#)

[Árvore binária e árvore binária de busca](#)

[Criando as classes Node e BinarySearchTree](#)

[Inserindo uma chave na BST](#)

[Percorrendo uma árvore](#)

[Percorso em-ordem](#)

[Percorso pré-ordem](#)

[Percorso pós-ordem](#)

[Pesquisando valores em uma árvore](#)

[Pesquisando valores mínimos e máximos](#)

[Pesquisando um valor específico](#)

[Removendo um nó](#)

[Removendo uma folha](#)

[Removendo um nó com um filho à esquerda ou à direita](#)

[Removendo um nó com dois filhos](#)

[Árvores autobalanceadas](#)

[Árvore de Adelson-Velskii e Landi \(árvore AVL\)](#)

[Altura de um nó e o fator de balanceamento](#)

[Operações de balanceamento – rotações na árvore AVL](#)

[Rotação Esquerda-Esquerda: rotação simples à direita](#)

[Rotação Direita-Direita: rotação simples à esquerda](#)

[Esquerda-Direita: rotação dupla à direita](#)

[Direita-Esquerda: rotação dupla à esquerda](#)

[Inserindo um nó na árvore AVL](#)

[Removendo um nó da árvore AVL](#)

[Árvore rubro-negra](#)

[Inserindo um nó na árvore rubro-negra](#)

[Verificando as propriedades da árvore rubro-negra após a inserção](#)

[Rotações na árvore rubro-negra](#)

[Resumo](#)

[Capítulo 11 ■ Heap binário e heap sort](#)

[Estrutura de dados do heap binário](#)

[Criando a classe MinHeap](#)

[Representação da árvore binária com um array](#)
[Inserindo um valor no heap](#)
[Operação de sift up](#)
[Encontrando os valores mínimo e máximo no heap](#)
[Extraindo os valores mínimo e máximo do heap](#)
[Operação de sift down](#)
[Criando a classe MaxHeap](#)
[Algoritmo de heap sort](#)
[Resumo](#)

Capítulo 12 ■ Grafos

[Terminologia dos grafos](#)
[Grafos direcionados e não direcionados](#)
[Representando um grafo](#)
[A matriz de adjacências](#)
[Lista de adjacências](#)
[Matriz de incidências](#)
[Criando a classe Graph](#)
[Percorrendo grafos](#)
[Busca em largura \(BFS\)](#)
[Encontrando os caminhos mais curtos usando BFS](#)
[Estudos adicionais sobre algoritmos de caminhos mais curtos](#)
[Busca em profundidade \(DFS\)](#)
[Explorando o algoritmo DFS](#)
[Ordenação topológica usando DFS](#)
[Algoritmos de caminho mais curto](#)
[Algoritmo de Dijkstra](#)
[Algoritmo de Floyd-Warshall](#)
[Árvore de extensão mínima \(MST\)](#)
[Algoritmo de Prim](#)
[Algoritmo de Kruskal](#)
[Resumo](#)

Capítulo 13 ■ Algoritmos de ordenação e de busca

[Algoritmos de ordenação](#)

[Bubble sort](#)
[Bubble sort melhorado](#)
[Selection sort](#)
[Insertion sort](#)
[Merge sort](#)
[Quick sort](#)
[Processo de partição](#)
[Quick sort em ação](#)
[Counting sort](#)
[Bucket sort](#)
[Radix sort](#)
[Algoritmos de busca](#)
[Busca sequencial](#)
[Busca binária](#)
[Busca por interpolação](#)
[Algoritmos de embaralhamento](#)
[Algoritmo de embaralhamento de Fisher-Yates](#)
[Resumo](#)

Capítulo 14 ■ Designs de algoritmos e técnicas

[Dividir e conquistar](#)
[Busca binária](#)
[Programação dinâmica](#)
[Problema do número mínimo de moedas para troco](#)
[Problema da mochila](#)
[Maior subsequência comum](#)
[Multiplicação de cadeia de matrizes](#)
[Algoritmos gulosos](#)
[Problema do número mínimo de moedas para troco](#)
[Problema fracionário da mochila](#)
[Algoritmos de backtracking](#)
[Rato em um labirinto](#)
[Solucionador de sudoku](#)
[Introdução à programação funcional](#)
[Programação funcional versus programação imperativa](#)

[ES2015+ e a programação funcional](#)

[Caixa de ferramentas funcional de JavaScript – map, filter e reduce](#)

[Bibliotecas e estruturas de dados funcionais de JavaScript](#)

[Resumo](#)

Capítulo 15 ■ Complexidade de algoritmos

[Notação big-O](#)

[Compreendendo a notação big-O](#)

[O\(1\)](#)

[O\(n\)](#)

[O\(n²\)](#)

[Comparando as complexidades](#)

[Estruturas de dados](#)

[Grafos](#)

[Algoritmos de ordenação](#)

[Algoritmos de busca](#)

[Introdução à teoria de NP-completo](#)

[Problemas impossíveis e algoritmos heurísticos](#)

[Divertindo-se com algoritmos](#)

[Resumo](#)

Colaboradores

Sobre a autora

Loiane Groner tem mais de dez anos de experiência no desenvolvimento de aplicações corporativas. Atualmente, trabalha como analista de negócios e desenvolvedora de Java/HTML5/JavaScript em uma instituição financeira norte-americana.

É apaixonada por tecnologia, publica artigos em seu blog e ministra palestras em conferências sobre Java, ExtJS, Cordova, Ionic, TypeScript e Angular.

É Google Developer Expert em Web Technologies (Tecnologias Web) e Angular, e Microsoft Most Valuable Professional em Visual Studio e Development Technologies (Tecnologias de Desenvolvimento). É também autora de outros livros da Packt.

Gostaria de agradecer aos meus pais a educação, a orientação e os conselhos que me deram por todos esses anos, e ao meu marido por ser paciente e me apoiar, incentivando-me para que eu continue fazendo o que amo.

Também gostaria de agradecer aos leitores deste e de outros livros que escrevi o apoio e o feedback. Muito obrigada!

Sobre os revisores

Todd Zebert é desenvolvedor web full stack e trabalha atualmente na Miles.

Já foi revisor técnico de vários livros e vídeos, é palestrante frequente em conferências sobre JavaScript, Drupal e tecnologias relacionadas, além de ter um blog de tecnologia na Medium.

Tem experiência prévia diversificada em tecnologia, incluindo infraestrutura, engenharia de rede, gerenciamento de projetos e liderança em TI. Começou a trabalhar com desenvolvimento web no navegador Mosaic original.

É empreendedor e faz parte da comunidade de startups de Los Angeles.

Acredita em trabalhos voluntários, código aberto, maker/STEM/STEAM e em retribuição à comunidade.

Kashyap Mukkamala é um arquiteto de software entusiasmado, trabalha na Egen Solutions Inc. e é autor do livro *Hands-On Data Structures and Algorithms with JavaScript*. Quando não está resolvendo problemas de empresas Fortune 500 na Egen, Kashyap concentra-se em construir a web do futuro e, com isso, está ajudando a comunidade a crescer e a aprender.

Prefácio

JavaScript, uma das linguagens de programação mais populares atualmente, é conhecida como a linguagem da internet porque os navegadores a entendem de modo nativo, sem a instalação de qualquer plugin. A linguagem JavaScript evoluiu muito, a ponto de não ser mais apenas uma linguagem de frontend; nos dias de hoje, ela está igualmente presente no servidor (NodeJS), no banco de dados (MongoDB) e em dispositivos móveis, além de ser usada em dispositivos embarcados e na **Iot** (Internet of Things, ou Internet das Coisas).

Conhecer as estruturas de dados é muito importante para qualquer profissional da área de tecnologia. Trabalhar como desenvolvedor significa ser capaz de resolver problemas com a ajuda das linguagens de programação, e as estruturas de dados são uma parte indispensável das soluções que precisamos criar para resolver esses problemas. Escolher uma estrutura de dados incorreta também pode impactar o desempenho do programa que escrevemos. Por isso, é importante conhecer as diferentes estruturas de dados e saber aplicá-las de forma apropriada.

Os algoritmos são o estado da arte em ciência da computação. Há muitas maneiras de resolver o mesmo problema, e algumas abordagens são melhores que outras.

É por isso que conhecer os algoritmos mais famosos também é muito importante.

Este livro foi escrito para iniciantes que queiram conhecer estruturas de dados e algoritmos, mas também para aqueles que já tenham familiaridade com eles, mas desejem conhecê-los usando JavaScript.

Boa programação!

A quem este livro se destina

Se você é estudante de ciência da computação ou está iniciando a sua carreira na área de tecnologia e quer explorar os melhores recursos de JavaScript, este livro foi escrito para você. Se já tem familiaridade com programação, mas quer aperfeiçoar suas habilidades com algoritmos e

estruturas de dados, este livro também foi feito para você.

Basta ter conhecimento básico de JavaScript e de lógica de programação para começar a se divertir com os algoritmos.

0 que este livro inclui

O Capítulo 1, *JavaScript – uma visão geral rápida*, apresenta o básico sobre JavaScript, aquilo que devemos conhecer antes de ver as estruturas de dados e os algoritmos. Além disso, aborda a configuração do ambiente de desenvolvimento de que precisaremos neste livro.

O Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*, aborda algumas funcionalidades novas de JavaScript introduzidas a partir de 2015, além de incluir as funcionalidades básicas do TypeScript, um superconjunto do JavaScript.

O Capítulo 3, *Arrays*, explica como usar a estrutura de dados mais básica e mais utilizada, os arrays. Esse capítulo mostra como declarar, inicializar, adicionar e remover elementos de um array. Também aborda o uso dos métodos do Array nativo de JavaScript.

O Capítulo 4, *Pilhas*, apresenta a estrutura de dados de pilha, mostrando como criar, adicionar e remover elementos dela. Também mostra como usar pilhas para resolver alguns problemas relacionados à ciência da computação.

O Capítulo 5, *Filas e deque*s, apresenta a estrutura de dados de fila, mostrando como criar, adicionar e remover elementos dela. Inclui a estrutura de dados deque (fila de duas pontas), um tipo especial de fila. Também mostra como usar filas para resolver alguns problemas relacionados à ciência da computação e explica as principais diferenças entre filas e pilhas.

O Capítulo 6, *Listas ligadas*, explica como criar a estrutura de dados de lista ligada a partir do zero, usando objetos e o conceito de “ponteiro”. Além de descrever como declarar, criar, adicionar e remover elementos, o capítulo também inclui os vários tipos de listas ligadas, como as listas duplamente ligadas e as listas ligadas circulares.

O Capítulo 7, *Conjuntos*, apresenta a estrutura de dados de conjunto e como podemos usá-la para armazenar elementos não repetidos. Também explica os diferentes tipos de operações de conjunto e como implementá-

los e usá-los.

O Capítulo 8, *Dicionários e hashes*, explica as estruturas de dados de dicionários e hashes e as diferenças entre eles. Esse capítulo descreve como declarar, criar e usar as duas estruturas de dados. Além disso, explica como tratar colisões de hash e descreve técnicas para criar funções melhores de hash.

O Capítulo 9, *Recursão*, apresenta o conceito de recursão e mostra as diferenças entre algoritmos declarativos e recursivos.

O Capítulo 10, *Árvores*, descreve a estrutura de dados de árvore e a sua terminologia, concentrando-se nos dados da Árvore Binária de Busca (Binary Search Tree) – em seus métodos para pesquisar, percorrer, adicionar e remover nós. Também apresenta as árvores autobalanceadas, como a AVL e a Rubro-Negra (Red-Black).

O Capítulo 11, *Heap binário e Heap sort*, aborda as estruturas de dados de min heap e max heap, apresenta como usar o heap como uma fila de prioridades e discute o famoso algoritmo heap sort.

O Capítulo 12, *Grafos*, explica o mundo maravilhoso dos grafos e suas aplicações em problemas do mundo real. Apresenta a terminologia mais comum associada aos grafos, as diferentes maneiras de representá-los e como percorrê-los usando os algoritmos BFS (Breadth-First-Search, ou Busca em Largura) e DFS (Depth-First-Search, ou Busca em Profundidade) e suas aplicações.

O Capítulo 13, *Algoritmos de ordenação e de busca*, explora os algoritmos mais usados de ordenação, como o bubble sort (ordenação por flutuação – e a sua versão melhorada), o selection sort (ordenação por seleção), o insertion sort (ordenação por inserção), o merge sort (ordenação por mistura) e o quick sort (ordenação rápida). Também inclui o counting sort (ordenação por contagem) e o radix sort (ordenação de raízes), dois algoritmos de ordenação distribuídos. Além disso, discute como funcionam os algoritmos de busca, como os de busca sequencial e de busca binária, e como embaralhar arrays.

O Capítulo 14, *Designs de algoritmos e técnicas*, apresenta algumas técnicas de algoritmos, além de descrever alguns dos algoritmos mais famosos. Também inclui uma introdução à programação funcional em JavaScript.

O Capítulo 15, *Complexidade de algoritmos*, apresenta a notação big-O (O

grande) e seus conceitos, além de incluir uma “colinha” com a complexidade dos algoritmos implementados neste livro. Inclui também uma introdução aos problemas NP-completos e às soluções heurísticas. Por fim, o capítulo explica como levar o seu conhecimento sobre algoritmos para o próximo patamar.

Para aproveitar ao máximo este livro

Embora este livro inclua uma rápida introdução ao JavaScript em seu primeiro capítulo, você precisará ter um conhecimento básico dessa linguagem e de lógica de programação.

Para testar os exemplos de código apresentados, você precisará de um editor de código (como o Atom ou o Visual Studio Code) para ler o código, além de um navegador (Chrome, Firefox ou Edge).

Também poderá testar os exemplos online acessando <https://javascript-ds-algorithms-book.firebaseio.com/>. Lembre-se também de abrir as ferramentas do desenvolvedor no navegador para que veja o que está sendo gerado como saída no console.

Convenções usadas

Há uma série de convenções textuais usadas neste livro.

CódigoNoTexto: indica palavras referentes a código no texto, nomes de tabelas de banco de dados, nomes de pastas, nomes e extensões de arquivos, nomes de path, URLs dummy, entrada de usuário e handles do Twitter. Eis um exemplo: “Monte o arquivo baixado `WebStorm-10*.dmg` contendo a imagem de disco como outro disco em seu sistema.”

Um bloco de código é apresentado assim:

```
class Stack {  
  constructor() {  
    this.items = []; // {1}  
  }  
}
```

Se quisermos chamar a sua atenção para uma parte específica de um bloco de código, as linhas ou itens relevantes estarão em negrito:

```
const stack = new Stack();  
console.log(stack.isEmpty()); // exibe true
```

Qualquer entrada ou saída na linha de comando será apresentada assim:

npm install http-server -g

Negrito: indica um termo novo, uma palavra importante ou palavras que você vê na tela. Por exemplo, palavras em menus ou em caixas de diálogo aparecem no texto dessa forma. Eis um exemplo: “Selecione **System info** (Info sobre o sistema) no painel **Administration** (Administração).”

Dicas e truques aparecerão desta forma.

Avisos ou notas importantes serão indicados assim.

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora pelo email: novatec@novatec.com.br.

Temos uma página web para este livro, na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição traduzida para o português

<https://www.novatec.com.br/livros/estruturas-de-dados-algoritmos-em-javascript-2ed/>

- Página da edição original em inglês

<https://www.packtpub.com/web-development/learning-javascript-data-structures-and-algorithms-third-edition>

- Código-fonte dos exemplos no GitHub

<https://github.com/PacktPublishing/Learning-JavaScript-Data-Structures-and-Algorithms-Third-Edition>.

Para obter mais informações sobre os livros da Novatec, acesse nosso site em: <http://www.novatec.com.br>.

CAPÍTULO 1

JavaScript – uma visão geral rápida

A linguagem JavaScript é extremamente eficaz, sendo uma das mais populares do mundo e uma das mais proeminentes na internet. Por exemplo, o GitHub (o maior host de códigos do mundo, disponível em <https://github.com>) hospeda mais de 400 mil repositórios de JavaScript (a maior parte dos projetos disponíveis está em JavaScript; consulte <http://githut.info>). O número de projetos em JavaScript e no GitHub aumenta a cada ano.

A linguagem JavaScript não é usada apenas no frontend. Ela também pode ser utilizada no backend, sendo o Node.js a tecnologia responsável por isso. O número de **npms** (Node Package Modules, ou Módulos do Pacote Node), disponíveis em <https://www.npmjs.org>, também tem crescido exponencialmente. Além disso, o JavaScript pode ser usado em desenvolvimento para dispositivos móveis, e é um dos frameworks mais populares no Apache Cordova (<https://cordova.apache.org>), um framework híbrido para dispositivos móveis, o qual permite que os desenvolvedores programem usando HTML, CSS e JavaScript, possibilitando construir um aplicativo e gerar um arquivo APK para Android e IPA para iOS (Apple). E, claro, não vamos nos esquecer das aplicações desktop. Podemos escrever aplicações desktop compatíveis com Linux, Mac OS e Windows usando um framework JavaScript chamado Electron (<https://electron.atom.io>). O JavaScript também é usado em dispositivos embarcados e dispositivos para **IoT** (Internet of Things, ou Internet das Coisas). Como você pode ver, o JavaScript está em todos os lugares!

Assim, se você é ou vai ser um desenvolvedor web, é mandatório ter essa linguagem em seu currículo.

Neste capítulo, conheceremos a sintaxe e algumas funcionalidades básicas necessárias para usar Javascript, a fim de começarmos a desenvolver a nossa própria estrutura de dados e os nossos algoritmos. Discutiremos os seguintes assuntos:

- configuração do ambiente e o básico sobre JavaScript;
- estruturas de controle e funções;
- programação orientada a objetos em Javascript;
- depuração e ferramentas.

Estrutura de dados e algoritmos em JavaScript

Neste livro, conheceremos as estruturas de dados e os algoritmos mais usados. Por que devemos usar JavaScript para conhecê-los? Já respondemos a essa pergunta. Além de muito popular, a linguagem JavaScript é apropriada para conhecer as estruturas de dados por ser uma linguagem funcional. Além do mais, essa pode ser uma maneira muito divertida de aprender algo novo, pois é muito diferente (e mais fácil) em comparação a conhecer as estruturas de dados com uma linguagem padrão como **C**, **Java** ou **Python**. E quem disse que estruturas de dados e algoritmos foram feitos apenas para linguagens como C e Java? Talvez você precise fazer implementações em algumas dessas linguagens enquanto desenvolve o frontend também.

Conhecer as estruturas de dados e os algoritmos é muito importante. O primeiro motivo é o fato de as estruturas de dados e os algoritmos serem capazes de resolver os problemas mais comuns de modo eficiente. Isso fará diferença na qualidade do código-fonte que você escreverá no futuro (inclusive no desempenho; se você escolher a estrutura de dados ou o algoritmo incorreto, conforme o cenário, poderá ter alguns problemas de desempenho). Em segundo lugar, os algoritmos são estudados nas faculdades, junto com os conceitos introdutórios de ciência da computação. Por fim, se você planeja conseguir um emprego em uma das maiores empresas de **TI** (Information Technology, ou Tecnologia da Informação) – como Google, Amazon, Microsoft, eBay e outras –, as estruturas de dados e os algoritmos serão assuntos das perguntas nas entrevistas.

Vamos começar!

Configurando o ambiente

Um dos prós da linguagem JavaScript, quando comparada com outras linguagens, é que você não precisa instalar nem configurar um ambiente

complicado para começar a usá-la. Todo computador já tem o ambiente necessário, mesmo que um usuário jamais chegue a escrever uma única linha de código-fonte. Tudo que precisamos é de um navegador!

Para executar os exemplos deste livro, é recomendável ter um navegador moderno instalado, como o Google Chrome ou o Firefox (você pode usar aquele de que mais gosta), um editor de sua preferência (como o Visual Studio Code) e um servidor web (XAMPP ou qualquer outro de sua preferência, mas esse passo é opcional). O Chrome, o Firefox, o VS Code e o XAMPP estão disponíveis para Windows, Linux e Mac OS.

Configuração mínima para trabalhar com JavaScript

O ambiente mais simples que você pode usar para desenvolvimento com JavaScript é um navegador. Os navegadores modernos (Chrome, Firefox, Safari e Edge) têm uma funcionalidade chamada **Developer Tools** (Ferramentas do desenvolvedor). Para acessar o DevTools no Chrome (Figura 1.1), clique no menu no canto superior direito, **More Tools | Developer Tools (Mais ferramentas | Ferramentas do desenvolvedor)**:

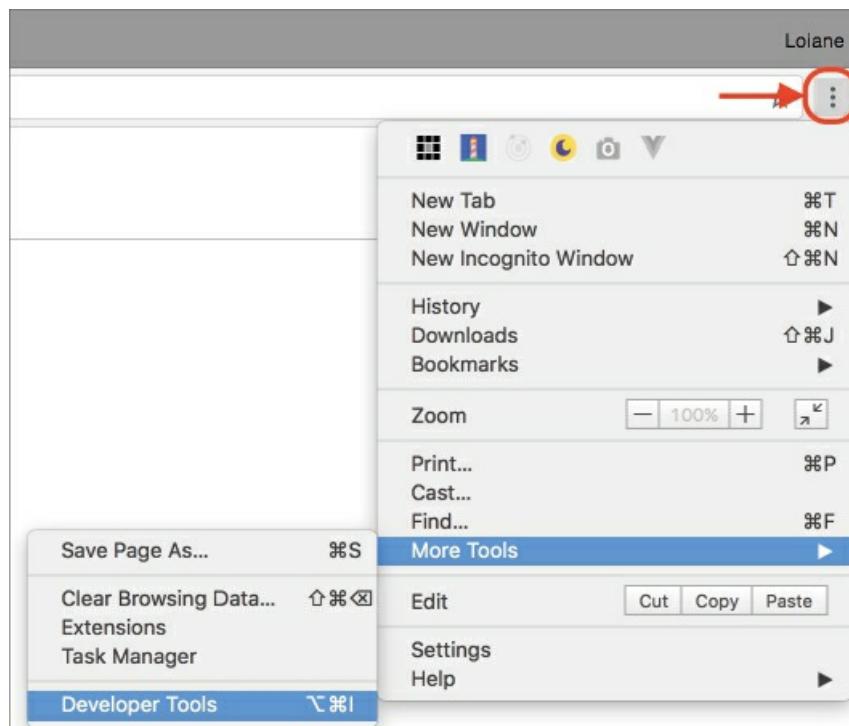


Figura 1.1

Ao abrir o DevTools, você verá a aba **Console** e poderá escrever todo o seu

código JavaScript na área de linha de comando, conforme mostra a Figura 1.2 (para executar o código-fonte, é necessário teclar *Enter*):

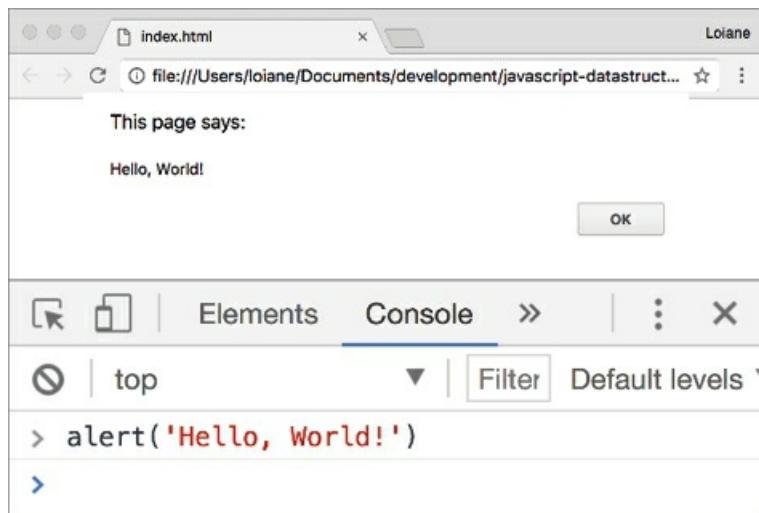


Figura 1.2

Usando servidores web

A segunda opção de ambiente que você pode querer instalar em seu computador também é simples, mas exige a instalação de um servidor web. Se um arquivo HTML contiver somente código JavaScript simples, que não exija nenhuma requisição para um servidor (chamadas **Ajax**), ele poderá ser executado no navegador clicando com o botão direito do mouse no arquivo HTML e selecionando a opção **Open with** (Abrir com). O código que desenvolveremos neste livro é simples e poderá ser executado usando essa abordagem. No entanto, é sempre bom ter um servidor web instalado.

Há muitas opções gratuitas e de código aberto disponíveis. Se você tem familiaridade com **PHP**, o XAMPP (<https://www.apachefriends.org>) é uma boa opção, e está disponível para Linux, Windows e Mac OS.

Como o nosso foco estará no JavaScript do lado do servidor e no navegador, há também um servidor web mais simples que você poderá instalar no Chrome. É a extensão **Web Server for Chrome**, que pode ser baixada a partir de <https://goo.gl/pxqLmU>. Depois de instalá-la, é possível acessá-la por meio do URL `chrome://apps:` no Chrome (Figura 1.3).

Depois de abrir a extensão **Web Server** (Figura 1.4), você poderá escolher a pasta que quer servir no navegador, com a opção **CHOOSE**

FOLDER (Selecionar pasta). É possível criar uma pasta na qual execute o código-fonte que implementaremos neste livro, ou fazer o download do código-fonte deste livro, extraí-lo em um diretório de sua preferência e acessá-lo por meio do URL informado (o default é <http://127.0.0.1:8887>):

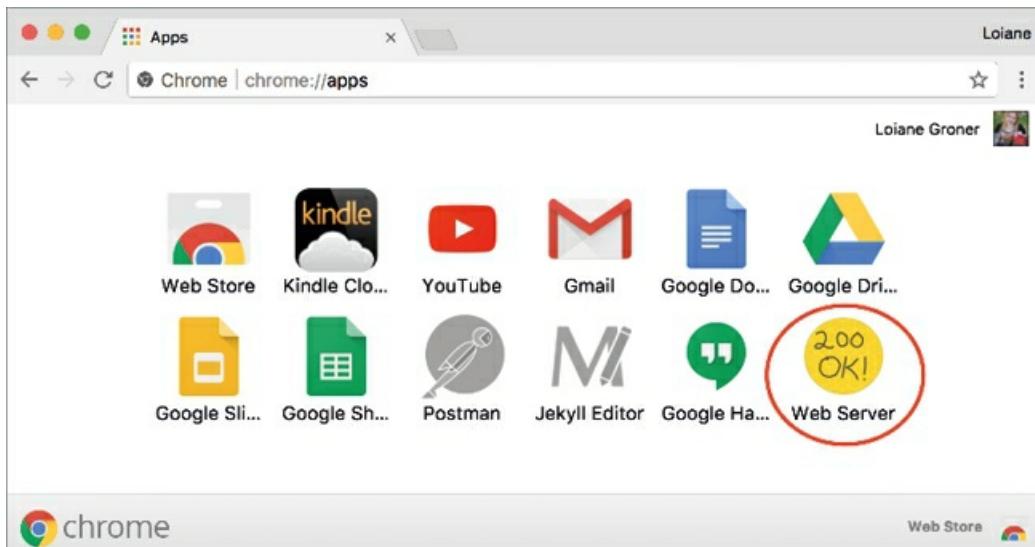


Figura 1.3

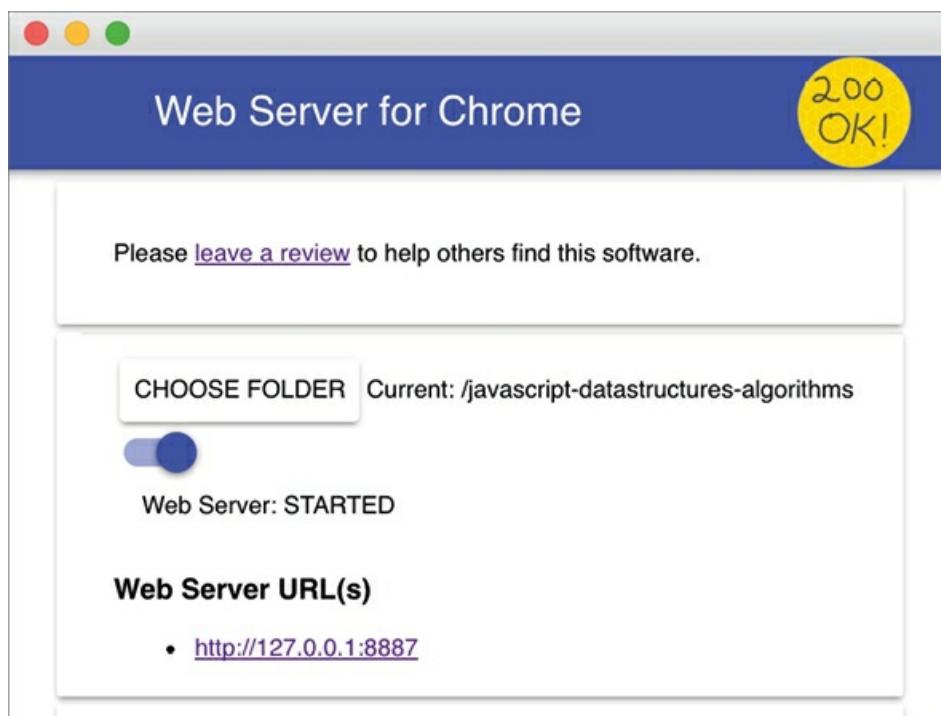


Figura 1.4

Todos os exemplos deste livro podem ser executados acessando

<http://127.0.0.1:8887/examples>. Você encontrará um `index.html` com uma lista de todos os exemplos, conforme mostra a Figura 1.5.

Ao executar os exemplos, lembre-se sempre de ter o Developer Tools ativado e a aba Console aberta para ver a saída. A extensão Web Server for Chrome também foi desenvolvida com JavaScript. A fim de ter uma experiência melhor, é recomendável usar essa extensão para executar os exemplos deste livro ou instalar o `http-server` do Node.js, que será visto na próxima seção.

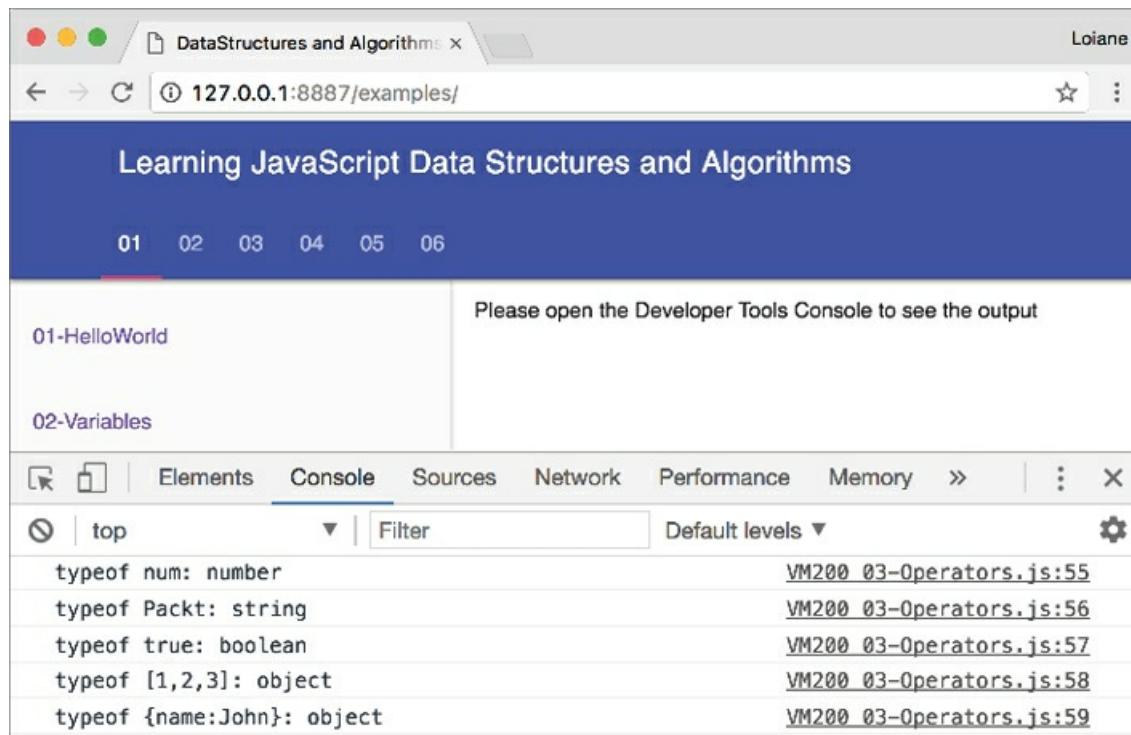


Figura 1.5

http-server do Node.js

A terceira opção é ter um ambiente 100% JavaScript! Para esse ambiente, precisamos do Node.js instalado. Acesse <http://nodejs.org>, faça o download e instale o Node.js. Depois de instalá-lo, abra a aplicação de Terminal (se estiver usando Windows, abra o Prompt de Comandos com o Node.js, instalado com ele) e execute o comando a seguir:

```
npm install http-server -g
```

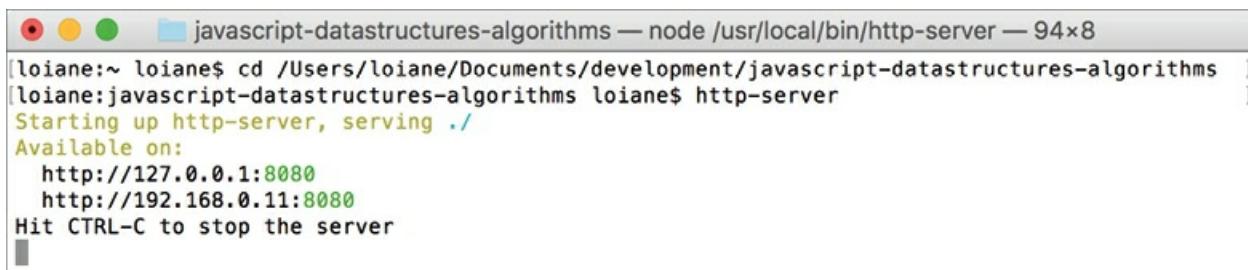
Certifique-se de ter digitado o comando, em vez de copiá-lo e colá-lo. Se

você copiar o comando, alguns erros poderão ocorrer. É possível também executar o comando como administrador. Em sistemas Linux e Mac, use o seguinte comando:

```
sudo npm install http-server -g
```

O comando instalará o **http-server**, um servidor JavaScript. Para iniciar um servidor e executar os exemplos deste livro na aplicação de Terminal, mude o diretório para a pasta que contém o código-fonte do livro e digite **http-server**, como mostra a Figura 1.6.

Para executar os exemplos, abra o navegador e acesse o localhost na porta especificada pelo comando **http-server**.



The screenshot shows a terminal window titled "javascript-datastructures-algorithms — node /usr/local/bin/http-server — 94x8". The user has navigated to the directory "/Users/loiane/Documents/development/javascript-datastructures-algorithms". They run the command "http-server" and see the output: "Starting up http-server, serving ./". It then lists the available addresses: "Available on:" followed by "http://127.0.0.1:8080" and "http://192.168.0.11:8080". A final message says "Hit CTRL-C to stop the server".

Figura 1.6

Passos detalhados para fazer o download do pacote de código e executar os exemplos foram mencionados no prefácio deste livro. Por favor, consulte esse texto. O pacote de código do livro também está disponível no GitHub em <https://github.com/loiane/javascript-datastructures-algorithms>.

Básico sobre o JavaScript

Antes de mergulhar de cabeça nas diversas estruturas de dados e algoritmos, vamos apresentar uma visão geral rápida da linguagem JavaScript. Esta seção apresentará o básico sobre JavaScript, necessário para implementar os algoritmos que criaremos nos capítulos subsequentes.

Para começar, vamos observar duas maneiras diferentes de usar o código JavaScript em uma página HTML. O primeiro exemplo é demonstrado pelo código a seguir. Precisamos criar um arquivo HTML (**01-HelloWorld.html**) e escrever o código aí. Nesse exemplo, declaramos a tag **script** no arquivo HTML e, dentro dessa tag, temos o código JavaScript:

```
<!DOCTYPE html>
```

```
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <script>
    alert('Hello, World!');
  </script>
</body>
</html>
```

Experimente usar a extensão Web Server for Chrome ou o `http-server` para executar o código anterior e veja a sua saída no navegador.

Para o segundo exemplo, precisamos criar um arquivo JavaScript (podemos salvá-lo como `01-Helloworld.js`) e, nesse arquivo, inserir o código a seguir:

```
alert('Hello, World!');
```

Então, o nosso arquivo HTML terá um aspecto semelhante a este:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <script src="01-Helloworld.js"></script>
</body>
</html>
```

O segundo exemplo mostra como incluir um arquivo JavaScript em um arquivo HTML.

Se executarmos qualquer um desses dois exemplos, a saída será a mesma. No entanto, o segundo exemplo é o mais usado pelos desenvolvedores JavaScript.

Você poderá encontrar instruções `include` de JavaScript ou código JavaScript na tag `head` em alguns exemplos na internet. Seguindo a melhor prática, incluiremos qualquer código JavaScript no final da tag `body`. Desse modo, o navegador fará o parse do HTML, e ele será exibido antes de os scripts serem carregados. Com isso, a página terá um melhor desempenho.

Variáveis

As variáveis armazenam dados que podem ser definidos, atualizados e recuperados sempre que necessário. Os valores atribuídos a uma variável têm um tipo. Em JavaScript, os tipos disponíveis são: **number** (número), **string**, **boolean** (booleano), **function** (função) e **object** (objeto). Também temos **undefined** (indefinido) e **null** (nulo), junto com arrays, datas e expressões regulares.

Embora a linguagem JavaScript tenha diferentes tipos de variáveis disponíveis, ela não é **fortemente tipada** como C/C++, C# e Java. Em linguagens fortemente tipadas, devemos definir o tipo da variável em sua declaração (em Java, por exemplo, para declarar uma variável inteira, usamos `int num = 1;`). Em JavaScript, basta usar a palavra reservada `var`; não é necessário declarar o tipo da variável. Por esse motivo, a linguagem JavaScript não é fortemente tipada. Entretanto, há discussões e uma especificação em versão preliminar para uma tipagem estática opcional (<https://github.com/dslomov/typed-objects-es7>), que poderá se tornar parte da especificação de JavaScript (**ECMAScript**) no futuro. Também podemos usar o **TypeScript** se quisermos definir tipos para nossas variáveis ao trabalhar com JavaScript. Conheceremos melhor a ECMAScript e o TypeScript mais adiante neste capítulo.

A seguir, apresentamos um exemplo de uso de variáveis em JavaScript:

```
var num = 1; // {1}
num = 3; // {2}
var price = 1.5; // {3}
var myName = 'Packt'; // {4}
var trueValue = true; // {5}
var nullVar = null; // {6}
var und; // {7}
```

- Na linha `{1}`, temos um exemplo de como declarar uma variável em JavaScript (estamos declarando um número). Embora não seja necessário fazer a declaração com a palavra reservada `var`, especificá-la sempre que estivermos declarando uma nova variável é uma boa prática.
- Na linha `{2}`, atualizamos uma variável existente. A linguagem JavaScript não é fortemente tipada. Isso significa que podemos declarar uma variável, inicializá-la com um número e depois atualizá-la com uma string ou com qualquer outro tipo de dado. Atribuir um valor a uma

variável cujo tipo seja diferente de seu tipo original também não é uma boa prática.

- Na linha {3}, declaramos igualmente um número, porém, dessa vez, é um número decimal de ponto flutuante.
- Na linha {4}, declaramos uma string; na linha {5}, declaramos um booleano. Na linha {6}, declaramos um valor `null` e, na linha {7}, declaramos uma variável `undefined`.
- Um valor `null` quer dizer sem valor, e `undefined` significa uma variável que foi declarada, mas que ainda não recebeu nenhum valor.

Se quisermos ver o valor de cada variável que declaramos, podemos usar `console.log` para isso, conforme listado no trecho de código a seguir:

```
console.log('num: ' + num);
console.log('myName: ' + myName);
console.log('trueValue: ' + trueValue);
console.log('price: ' + price);
console.log('nullVar: ' + nullVar);
console.log('und: ' + und);
```

O método `console.log` também aceita mais do que apenas argumentos. Em vez de `console.log('num: ' + num)`, também podemos usar `console.log('num: ', num)`. Enquanto a primeira opção concatenará o resultado em uma única string, a segunda nos permite adicionar uma descrição e visualizar o conteúdo da variável caso ela seja um objeto.

Há três maneiras de exibir valores de saída em JavaScript, que poderão ser usadas nos exemplos deste livro. A primeira é `alert('My text here')`, que apresenta uma janela de alerta no navegador; a segunda é `console.log('My text here')`, que exibe um texto na aba Console da ferramenta de depuração (Google Developer Tools ou Firebug, conforme o navegador que você estiver usando). A terceira é exibir o valor diretamente na página HTML sendo renderizada pelo navegador, usando `document.write('My text here')`. Você pode usar a opção com a qual se sentir mais à vontade.

Discutiremos as funções e os objetos mais adiante neste capítulo.

Escopo das variáveis

O escopo se refere ao local em que podemos acessar a variável no algoritmo (também pode ser em uma função quando trabalhamos com escopos de função). As variáveis podem ser locais ou globais.

Vamos observar um exemplo:

```
var myVariable = 'global';
myOtherVariable = 'global';
function myFunction() {
  var myVariable = 'local';
  return myVariable;
}
function myOtherFunction() {
  myOtherVariable = 'local';
  return myOtherVariable;
}
console.log(myVariable); // {1}
console.log(myFunction()); // {2}
console.log(myOtherVariable); // {3}
console.log(myOtherFunction()); // {4}
console.log(myOtherVariable); // {5}
```

O código anterior pode ser explicado assim:

- A linha {1} exibirá **global** porque estamos referenciando uma variável global.
- A linha {2} exibirá **local** porque declaramos a variável **myVariable** dentro da função **myFunction** como uma variável local, portanto o escopo está apenas no interior de **myFunction**.
- A linha {3} exibirá **global** porque estamos referenciando a variável global chamada **myOtherVariable**, inicializada na segunda linha do exemplo.
- A linha {4} exibirá **local**. Na função **myOtherFunction**, referenciamos a variável global **myOtherVariable** e lhe atribuímos o valor **local**, pois não declaramos a variável usando a palavra reservada **var**.
- Por esse motivo, a linha {5} exibirá **local** (pois alteramos o valor da variável em **myOtherFunction**).

Talvez você ouça falar que variáveis globais em JavaScript são prejudiciais, e isso é verdade. Em geral, a qualidade do código-fonte JavaScript é avaliada de acordo com o número de variáveis e funções globais (um número elevado é ruim). Portanto, sempre que possível, procure evitar as variáveis globais.

Operadores

Precisamos de operadores quando realizamos qualquer operação em uma linguagem de programação. A linguagem JavaScript também tem operadores aritméticos, de atribuição, de comparação, lógicos, bit a bit (bitwise) e unários, entre outros. Vamos observar esses operadores:

```
var num = 0; // {1}
num = num + 2;
num = num * 3;
num = num / 2;
num++;
num--;
num += 1; // {2}
num -= 2;
num *= 3;
num /= 2;
num %= 3;
console.log('num == 1 : ' + (num == 1)); // {3}
console.log('num === 1 : ' + (num === 1));
console.log('num != 1 : ' + (num != 1));
console.log('num > 1 : ' + (num > 1));
console.log('num < 1 : ' + (num < 1));
console.log('num >= 1 : ' + (num >= 1));
console.log('num <= 1 : ' + (num <= 1));
console.log('true && false : ' + (true && false)); // {4}
console.log('true || false : ' + (true || false));
console.log('!true : ' + (!true));
```

Na linha {1}, temos os operadores aritméticos. Na tabela a seguir, apresentamos os operadores e suas descrições:

| Operador aritmético | Descrição |
|---------------------|-------------------------------------------|
| + | Adição |
| - | Subtração |
| * | Multiplicação |
| / | Divisão |
| % | Módulo (resto de uma operação de divisão) |
| ++ | Incremento |
| -- | Decremento |

Na linha {2}, temos os operadores de atribuição. Na tabela a seguir, apresentamos os operadores e suas descrições:

| Operador de atribuição | Descrição |
|------------------------|------------|
| = | Atribuição |

| | |
|-----------------|-----------------------------------------------------------------------------------|
| <code>+=</code> | Atribuição de soma (<code>x += y</code>) == (<code>x = x + y</code>) |
| <code>-=</code> | Atribuição de subtração (<code>x -= y</code>) == (<code>x = x - y</code>) |
| <code>*=</code> | Atribuição de multiplicação (<code>x *= y</code>) == (<code>x = x * y</code>) |
| <code>/=</code> | Atribuição de divisão (<code>x /= y</code>) == (<code>x = x / y</code>) |
| <code>%=</code> | Atribuição de resto (<code>x %= y</code>) == (<code>x = x % y</code>) |

Na linha {3}, temos os operadores de comparação. Na tabela a seguir, apresentamos os operadores e suas descrições:

| Operador de comparação | Descrição |
|------------------------|-------------------------------------------------|
| <code>==</code> | Igual a |
| <code>===</code> | Igual a (tanto o valor quanto o tipo do objeto) |
| <code>!=</code> | Diferente de |
| <code>></code> | Maior que |
| <code>>=</code> | Maior ou igual a |
| <code><</code> | Menor que |
| <code><=</code> | Menor ou igual a |

Por fim, na linha {4}, temos os operadores lógicos. Na tabela a seguir, apresentamos os operadores e suas descrições:

| Operador lógico | Descrição |
|-------------------------|-----------|
| <code>&&</code> | E |
| <code> </code> | Ou |
| <code>!</code> | Negação |

A linguagem JavaScript também tem suporte para operadores bit a bit (bitwise), mostrados a seguir:

```
console.log('5 & 1:', (5 & 1));
console.log('5 | 1:', (5 | 1));
console.log('~ 5:', (~5));
console.log('5 ^ 1:', (5 ^ 1));
console.log('5 << 1:', (5 << 1));
console.log('5 >> 1:', (5 >> 1));
```

A tabela seguinte apresenta uma descrição mais detalhada dos operadores bit a bit:

| Operador bit a bit (bitwise) | Descrição |
|------------------------------|-------------------------------------------|
| <code>&</code> | E |
| <code> </code> | Ou |
| <code>~</code> | Negação |
| <code>^</code> | Ou exclusivo (Xor) |
| <code><<</code> | Deslocamento para a esquerda (left shift) |

>>

|Deslocamento para a direita (right shift)|

O operador **typeof** devolve o tipo da variável ou expressão. Por exemplo, observe o código a seguir:

```
console.log('typeof num:', typeof num);
console.log('typeof Packt:', typeof 'Packt');
console.log('typeof true:', typeof true);
console.log('typeof [1,2,3]:', typeof [1,2,3]);
console.log('typeof {name:John}:', typeof {name:'John'});
```

Eis a saída:

```
typeof num: number
typeof Packt: string
typeof true: boolean
typeof [1,2,3]: object
typeof {name:John}: object
```

De acordo com a especificação, há dois tipos de dados em JavaScript:

- **tipos de dados primitivos:** null (nulo), undefined (indefinido), string, number (número), boolean (booleano) e symbol (símbolo);
- **tipos de dados derivados/objetos:** objetos JavaScript, incluindo funções, arrays e expressões regulares.

A linguagem JavaScript também aceita o operador **delete**, que apaga uma propriedade de um objeto:

```
var myObj = {name: 'John', age: 21};
delete myObj.age;
console.log(myObj); //exibe Object {name: "John"}
```

Nos algoritmos deste livro, usaremos alguns desses operadores.

Verdadeiro e falso

Em JavaScript, **true** e **false** são um pouco complicados. Na maioria das linguagens, os valores booleanos **true** e **false** representam os resultados verdadeiro/falso. Em JavaScript, uma string como **Packt** é avaliada como **true**.

A tabela a seguir pode nos ajudar a compreender melhor o funcionamento de **true** e **false** em JavaScript:

| Tipo do valor | Resultado |
|---------------|-----------------------------------|
| undefined | false |
| null | false |
| Boolean | Verdadeiro é true e falso é false |

| | |
|--------|-----------------------------------------------------------------------------------------------------------------------------------|
| Number | O resultado é <code>false</code> para <code>+0</code> , <code>-0</code> ou <code>NaN</code> ; caso contrário, é <code>true</code> |
| String | O resultado é <code>false</code> se a string for vazia (o tamanho é 0); caso contrário, é <code>true</code> (tamanho ≥ 1) |
| Object | <code>true</code> |

Vamos considerar alguns exemplos e observar suas saídas:

```
function testTruthy(val) {
  return val ? console.log('truthy') : console.log('falsy');
}
testTruthy(true); // true
testTruthy(false); // false
testTruthy(new Boolean(false)); // true (objeto é sempre true)
testTruthy(''); // false
testTruthy('Packt'); // true
testTruthy(new String('')); // true (objeto é sempre true)
testTruthy(1); // true
testTruthy(-1); // true
testTruthy(NaN); // false
testTruthy(new Number(NaN)); // true (objeto é sempre true)
testTruthy({}); // true (objeto é sempre true)
var obj = { name: 'John' };
testTruthy(obj); // true
testTruthy(obj.name); // true
testTruthy(obj.age); // age (propriedade não existe)
```

Funções dos operadores de igualdade (`==` e `===`)

Os dois operadores de igualdade aceitos em JavaScript podem causar um pouco de confusão quando trabalhamos com eles.

Quando `==` é usado, os valores poderão ser considerados iguais mesmo se forem de tipos diferentes. Isso pode ser confuso, até mesmo para um desenvolvedor JavaScript mais experiente. Vamos analisar como `==` funciona usando a tabela a seguir:

| Type(x) | Type(y) | Resultado |
|------------------|------------------|----------------------------------|
| null | undefined | <code>true</code> |
| undefined | null | <code>true</code> |
| Number | String | <code>x == toNumber(y)</code> |
| String | Number | <code>toNumber(x) == y</code> |
| Boolean | Any | <code>toNumber(x) == y</code> |
| Any | Boolean | <code>x == toNumber(y)</code> |
| String ou Number | Object | <code>x == toPrimitive(y)</code> |
| Object | String ou Number | <code>toPrimitive(x) == y</code> |

Se `x` e `y` forem do mesmo tipo, então JavaScript usará o método `equals` para comparar os dois valores ou objetos. Qualquer outra combinação não listada na tabela resultará em `false`.

Os métodos `toNumber` e `toPrimitive` são internos e avaliam os valores de acordo com as tabelas a seguir.

Eis o método `toNumber`:

| Tipo do valor | Resultado |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <code>undefined</code> | É <code>NaN</code> |
| <code>null</code> | É <code>+0</code> |
| <code>Boolean</code> | Se o valor for <code>true</code> , o resultado será <code>1</code> ; se o valor for <code>false</code> , o resultado será <code>+0</code> |
| <code>Number</code> | É o valor do número |

Por fim, temos o método `toPrimitive`:

| Tipo do valor | Resultado |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Object</code> | Se <code>valueOf</code> devolver um valor primitivo, esse valor primitivo será devolvido; caso contrário, se <code>toString</code> devolver um valor primitivo, esse valor será devolvido; senão, um erro será devolvido. |

Vamos observar o resultado de alguns exemplos. Inicialmente, sabemos que a saída do código a seguir é `true` (tamanho da string > 1):

```
console.log('packt' ? true : false);
```

Mas e o código seguinte? Vamos observá-lo:

```
console.log('packt' == true);
```

A saída é `false`, então vamos entender o porquê:

- Inicialmente, o valor booleano é convertido com `toNumber`, portanto temos `packt == 1`.
- Em seguida, o valor de string é convertido com `toNumber`. Como a string é constituída de caracteres alfabéticos, `Nan` será devolvido, portanto temos `Nan == 1`, que é falso.

E o que dizer do código a seguir? Vamos analisá-lo:

```
console.log('packt' == false);
```

A saída também é `false`, e os motivos são estes:

- Inicialmente, o valor booleano é convertido com `toNumber`, portanto temos `packt == 0`.
- Em seguida, o valor de string é convertido com `toNumber`. Como a string é constituída de caracteres alfabéticos, `Nan` será devolvido,

portanto temos `NaN == 0`, que é falso.

E quanto ao operador `==?` Esse é muito mais simples. Se estivermos comparando dois valores de tipos diferentes, o resultado será sempre `false`. Se forem do mesmo tipo, eles serão comparados de acordo com a tabela a seguir:

| Type(x) | Valores | Resultado |
|---------|-------------------------------------------|-----------|
| Number | x tem o mesmo valor que y (mas não é NaN) | true |
| String | x e y têm caracteres idênticos | true |
| Boolean | x e y são ambos true ou são ambos false | true |
| Object | x e y referenciam o mesmo objeto | true |

Se `x` e `y` forem de tipos diferentes, o resultado será `false`. Vamos considerar alguns exemplos:

```
console.log('packt' === true); //false
console.log('packt' === 'packt'); //true
var person1 = {name: 'John'};
var person2 = {name: 'John'};
console.log(person1 === person2); //false, objetos diferentes
```

Estruturas de controle

A linguagem JavaScript tem um conjunto de estruturas de controle semelhante ao das linguagens C e Java. Instruções condicionais são tratadas com `if...else` e `switch`. Laços são tratados com as construções `while`, `do...while` e `for`.

Instruções condicionais

A primeira instrução condicional que analisaremos é a construção `if...else`. Há algumas maneiras de usar essa construção.

Podemos usar a instrução `if` se quisermos executar um bloco de código somente se a condição (expressão) for `true`, deste modo:

```
var num = 1;
if (num === 1) {
  console.log('num is equal to 1');
}
```

Podemos usar a instrução `if...else` se quisermos executar um bloco de código e a condição for `true`, ou outro bloco de código, somente caso a condição seja `false` (`else`), assim:

```
var num = 0;
if (num === 1) {
    console.log('num is equal to 1');
} else {
    console.log('num is not equal to 1, the value of num is ' + num);
}
```

A instrução **if...else** também pode ser representada por um operador ternário. Por exemplo, observe a instrução **if...else** a seguir:

```
if (num === 1) {
    num--;
} else {
    num++;
}
```

Essa instrução também pode ser representada assim:

```
(num === 1) ? num-- : num++;
```

Além do mais, se tivermos várias expressões, podemos usar **if...else** diversas vezes para executar blocos de código diferentes, de acordo com condições distintas, assim:

```
var month = 5;
if (month === 1) {
    console.log('January');
} else if (month === 2) {
    console.log('February');
} else if (month === 3) {
    console.log('March');
} else {
    console.log('Month is not January, February or March');
}
```

Por fim, temos a instrução **switch**. Se a condição que estivermos avaliando for a mesma que a anterior (porém a comparação é feita com valores diferentes), podemos usar a instrução **switch**:

```
var month = 5;
switch (month) {
    case 1:
        console.log('January');
        break;
    case 2:
        console.log('February');
        break;
    case 3:
        console.log('March');
        break;
```

```
    default:  
        console.log('Month is not January, February or March');  
    }  
}
```

Um aspecto muito importante em uma instrução **switch** é o uso das palavras reservadas **case** e **break**. A cláusula **case** determina se o valor de **switch** é igual ao valor da cláusula **case**. A instrução **break** impede que a instrução **switch** execute o restante da instrução (caso contrário, todos os scripts de todas as cláusulas **case** depois daquela com a qual uma correspondência foi feita seriam executados, até que uma instrução **break** fosse encontrada em uma das cláusulas **case**). Por fim, temos a instrução **default**, executada por padrão caso nenhuma das instruções **case** seja **true** (ou se a instrução **case** executada não tiver uma instrução **break**).

Laços

Os laços são usados com frequência quando trabalhamos com arrays (que serão o assunto do próximo capítulo). Especificamente, usaremos o laço **for** em nossos algoritmos.

O laço **for** é exatamente igual ao de C e de Java. É constituído de um contador de laço que, em geral, recebe um valor numérico; em seguida, a variável é comparada com outro valor (o script dentro do laço **for** será executado enquanto essa condição for verdadeira) e, por fim, o valor numérico é incrementado ou decrementado.

No exemplo a seguir, temos um laço **for**. Ele exibe o valor de **i** no console enquanto **i** for menor que **10**; **i** é iniciado com **0**, portanto o código a seguir exibirá os valores de **0** a **9**:

```
for (var i = 0; i < 10; i++) {  
    console.log(i);  
}
```

O próximo laço que veremos é o laço **while**. O bloco de código dentro do laço **while** será executado enquanto a condição for verdadeira. No código a seguir, temos uma variável **i**, inicializada com o valor **0**, e queremos que o valor de **i** seja exibido enquanto **i** for menor que **10** (ou menor ou igual a **9**). A saída mostrará os valores de **0** a **9**:

```
var i = 0;  
while (i < 10) {  
    console.log(i);  
}
```



```
i++;  
}
```

O laço **do...while** é muito parecido com o laço **while**. A única diferença é que, no laço **while**, a condição é avaliada antes da execução do bloco de código, enquanto no laço **do...while**, ela é avaliada depois de o bloco de código ter sido executado. O laço **do...while** garante que o bloco de código seja executado pelo menos uma vez. O código a seguir também exibe os valores de **0** a **9**:

```
var i = 0;  
do {  
    console.log(i);  
    i++;  
} while (i < 10);
```

Funções

As funções são muito importantes quando trabalhamos com JavaScript. Também faremos uso de funções em nossos exemplos.

O código a seguir mostra a sintaxe básica de uma função. Ela não tem argumentos nem a instrução **return**:

```
function sayHello() {  
    console.log('Hello!');  
}
```

Para executar esse código, basta usar a instrução a seguir:

```
sayHello();
```

Também podemos passar argumentos para uma função. Argumentos são variáveis com as quais se supõe que a função fará algo. O código a seguir mostra como usar argumentos com funções:

```
function output(text) {  
    console.log(text);  
}
```

Para usar essa função, podemos utilizar o seguinte código:

```
output('Hello!');
```

Você pode usar quantos argumentos quiser, assim:

```
output('Hello!', 'Other text');
```

Nesse caso, apenas o primeiro argumento será usado pela função; o segundo será ignorado. Uma função também pode devolver um valor:

```
function sum(num1, num2) {
```

```
    return num1 + num2;
}
```

Essa função calcula a soma de dois números especificados e devolve o resultado. Podemos usá-la da seguinte maneira:

```
var result = sum(1, 2);
output(result); // a saída é 3
```

Programação orientada a objetos em Javascript

Objetos JavaScript são coleções bem simples de pares nome-valor. Há duas maneiras de criar um objeto simples em JavaScript. A primeira é esta:

```
var obj = new Object();
```

A segunda é assim:

```
var obj = {};
```

Também podemos criar um objeto completo, desta maneira:

```
obj = {
  name: {
    first: 'Gandalf',
    last: 'the Grey'
  },
  address: 'Middle Earth'
};
```

Como podemos ver, para declarar um objeto JavaScript, pares *[chave, valor]* são usados, no quais a chave pode ser considerada um atributo do objeto, e o valor é o valor da propriedade. Todas as classes que criaremos neste livro serão objetos JavaScript, como **Stack**, **Set**, **LinkedList**, **Dictionary**, **Tree**, **Graph**, e assim por diante.

Em **POO** (Programação Orientada a Objetos), um objeto é uma instância de uma classe. Uma classe define as características do objeto. Em nossos algoritmos e estruturas de dados, criaremos algumas classes que representarão objetos. Eis o modo como podemos declarar uma classe (construtor) que representa um livro:

```
function Book(title, pages, isbn) {
  this.title = title;
  this.pages = pages;
  this.isbn = isbn;
}
```

Para instanciar essa classe, podemos usar o código a seguir:

```
var book = new Book('title', 'pag', 'isbn');
```

Então, podemos acessar as suas propriedades e atualizá-las deste modo:

```
console.log(book.title); // exibe o título do livro  
book.title = 'new title'; // atualiza o valor do título do livro  
console.log(book.title); // exibe o valor atualizado
```

Uma classe também pode conter funções (em geral, também são chamadas de **métodos**). Podemos declarar e usar uma função/método conforme mostra o código a seguir:

```
Book.prototype.printTitle = function() {  
    console.log(this.title);  
};  
book.printTitle();
```

Também podemos declarar funções diretamente na definição da classe:

```
function Book(title, pages, isbn) {  
    this.title = title;  
    this.pages = pages;  
    this.isbn = isbn;  
    this.printIsbn = function() {  
        console.log(this.isbn);  
    };  
}  
book.printIsbn();
```

No exemplo com `prototype`, a função `printTitle` será compartilhada entre todas as instâncias, e somente uma cópia será criada. Quando usamos uma definição baseada em classe, como no exemplo anterior, cada instância terá a sua própria cópia das funções. O uso do método `prototype` economiza memória e reduz o custo de processamento no que diz respeito a atribuir funções à instância. No entanto, você só pode declarar funções e propriedades `public` usando o método `prototype`. Com uma definição baseada em classe, você pode declarar funções e propriedades `private`, e os outros métodos da classe também poderão acessá-las. A ECMAScript 2015 (ES6) introduziu uma sintaxe simplificada muito semelhante ao exemplo baseado em classe, a qual é baseada em protótipo. Discutiremos melhor esse assunto mais adiante no Capítulo 2.

Depuração e ferramentas

Saber programar em JavaScript é importante, mas o mesmo vale para a

depuração de seu código. A depuração é muito útil para ajudar você a encontrar bugs em seu código, mas também pode ajudá-lo a executar o seu código mais lentamente, a fim de que seja possível ver tudo que está acontecendo (a pilha de métodos chamados, as atribuições de variáveis e assim por diante). É altamente recomendável que você invista tempo depurando o código-fonte deste livro a fim de observar todos os passos do algoritmo (isso também poderá ajudá-lo a compreender melhor o código).

O Firefox, o Safari, o Edge e o Chrome têm suporte para depuração. Um ótimo tutorial do Google que mostra como usar o Google Developer Tools para depurar JavaScript pode ser encontrado em <https://developer.chrome.com/devtools/docs/javascript-debugging>.

Você pode usar qualquer editor de texto de sua preferência. Entretanto, há também outras ferramentas ótimas que poderão ajudá-lo a ser mais produtivo quando trabalhar com JavaScript. São elas:

- **WebStorm:** é um IDE JavaScript muito eficaz, com suporte para as tecnologias web e os frameworks mais recentes. Embora pago, é possível fazer download de uma versão experimental de 30 dias (<http://www.jetbrains.com/webstorm>).
- **Sublime Text:** é um editor de texto leve, mas é possível personalizá-lo instalando plugins. Você pode comprar a licença para a equipe de desenvolvimento, mas também pode usá-lo gratuitamente (a versão experimental não expira); acesse <http://www.sublimetext.com>.
- **Atom:** é também um editor de texto gratuito, criado pelo GitHub. Tem ótimo suporte para JavaScript e pode ser igualmente personalizado por meio da instalação de plugins (<https://atom.io>).
- **Visual Studio Code:** é um editor de código gratuito, de código aberto, criado pela Microsoft e escrito com TypeScript. Tem a funcionalidade de preenchimento automático de JavaScript com o IntelliSense e oferece recursos embutidos para depuração diretamente a partir do editor. Também pode ser personalizado por meio da instalação de plugins (<https://code.visualstudio.com>).

Todos os editores mencionados anteriormente estão disponíveis para Windows, Linux e Mac OS.

Depuração com o VSCode

Para depurar código JavaScript ou ECMAScript diretamente do VSCode, primeiro é necessário instalar a extensão Debugger for Chrome (<https://goo.gl/QpXWGM>).

Em seguida, abra a extensão Web Server for Chrome e acesse o link para ver os exemplos do livro no navegador (o URL default é <http://127.0.0.1:8887/examples>).

A Figura 1.7 mostra como depurar diretamente no editor:

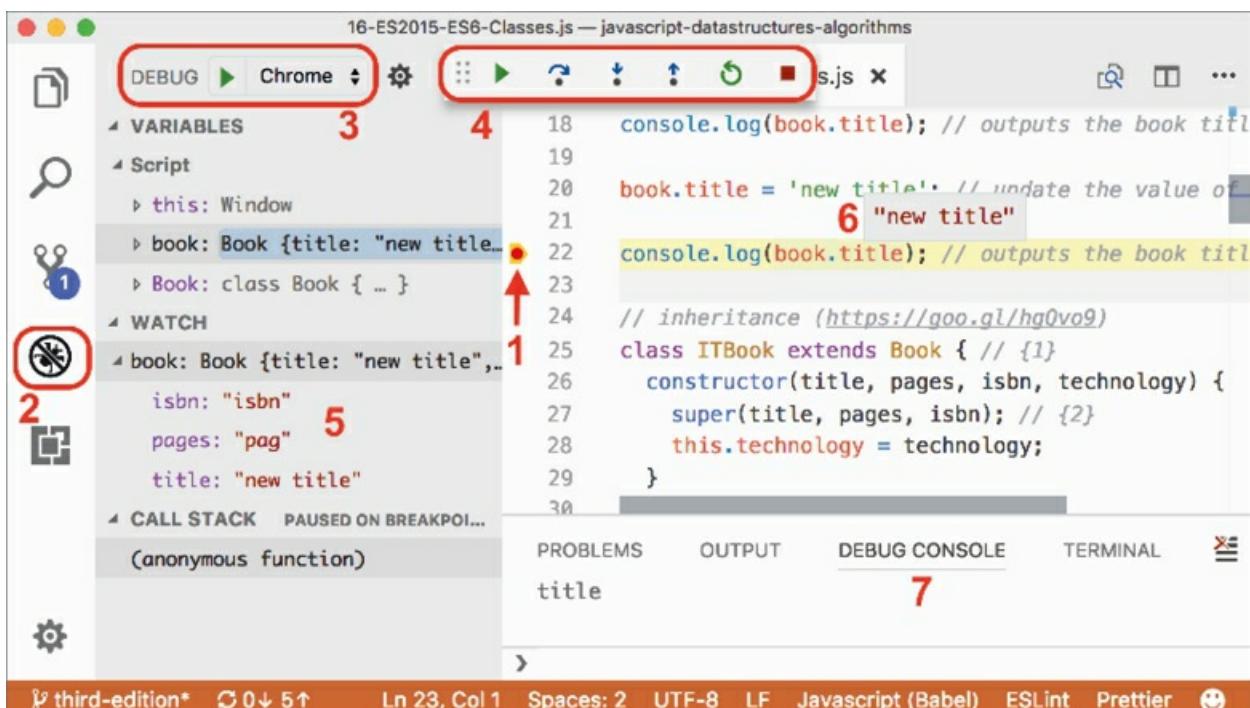


Figura 1.7

1. No editor, abra o arquivo JavaScript que você quer depurar, passe o ponteiro do mouse próximo aos números das linhas e clique na linha para adicionar um breakpoint (conforme mostrado pelo 1 na imagem de tela anterior). É nesse local que o depurador vai parar para que possamos analisar o código.
2. Depois que o Web Server estiver pronto e executando, clique na view de Debug (2), selecione Chrome (3) e clique no ícone Play (Executar) para iniciar o processo de depuração.
3. O Chrome será aberto automaticamente. Navegue para o exemplo desejado a fim de evocar o código que queremos depurar. Depois que a

linha na qual adicionamos o breakpoint for alcançada pelo depurador, o processo será interrompido e o editor receberá o foco.

4. Podemos controlar como o código é depurado usando a barra de ferramentas superior (4). É possível retomar o processo, ir para uma chamada de método ou para a próxima linha, reiniciar e parar o processo. É o mesmo comportamento que temos no depurador do Chrome e em outros navegadores.

5. A vantagem de usar essa funcionalidade de depuração embutida é que podemos fazer tudo a partir do editor (escrever código, depurar e testar). Além disso, temos as variáveis declaradas e a pilha de chamadas (call stack), podemos observar variáveis e expressões (5), passar o mouse sobre uma variável para ver o seu valor atual (6) e observar a saída do console (7).

O código-fonte deste livro foi desenvolvido usando o Visual Studio Code, e o pacote de código também contém tarefas de inicialização configuradas para que você possa depurar o código e fazer os testes diretamente do **VSCODE** (todos os detalhes estão no arquivo `.vscode/launch.json`). Todas as extensões recomendadas para executar o código-fonte deste livro também estão listadas no arquivo `.vscode/extensions.json`.

Resumo

Neste capítulo, aprendemos a configurar o ambiente de desenvolvimento para que possamos criar ou executar os exemplos deste livro.

Também vimos o básico sobre a linguagem JavaScript, necessário para dar início ao desenvolvimento dos algoritmos e das estruturas de dados incluídos neste livro.

No próximo capítulo, conheceremos as novas funcionalidades introduzidas no JavaScript a partir de 2015, e veremos como tirar proveito da tipagem estática e da verificação de erros usando TypeScript.

CAPÍTULO 2

Visão geral sobre ECMAScript e TypeScript

A linguagem JavaScript evolui a cada ano. Desde 2015, tem havido uma nova versão lançada a cada ano, que chamamos de **ECMAScript**, e, como JavaScript é uma linguagem muito potente, ela é usada também em desenvolvimentos corporativos. Um dos recursos que realmente ajuda nesse tipo de desenvolvimento (entre outros tipos de aplicação) são as variáveis tipadas, que agora temos graças ao **TypeScript** – um superconjunto do JavaScript.

Neste capítulo, conheceremos algumas funcionalidades que foram introduzidas no JavaScript a partir de 2015 e veremos as vantagens de usar uma versão tipada dessa linguagem em nossos projetos. Abordaremos os seguintes assuntos:

- introdução à ECMAScript;
- JavaScript no navegador *versus* no servidor;
- introdução ao TypeScript.

ECMAScript ou JavaScript?

Ao trabalhar com JavaScript, deparamos com o termo ECMAScript com muita frequência nos livros, em postagens de blog, nos cursos em vídeo e assim por diante. O que a ECMAScript tem a ver com o JavaScript, e há alguma diferença entre eles?

A ECMA é uma organização que padroniza informações. Resumindo uma longa história, muito tempo atrás, o JavaScript foi submetido à ECMA para que fosse padronizado. Isso resultou em um novo padrão de linguagem, que conhecemos como ECMAScript. O JavaScript é uma implementação dessa especificação (a mais popular), conhecida como **ActionScript**.

ES6, ES2015, ES7, ES2016, ES8, ES2017 e ES.Next

Como já sabemos, o JavaScript é uma linguagem que executa

principalmente nos navegadores (assim como nos servidores usando NodeJS, no desktop e em dispositivos móveis), e cada navegador pode implementar a própria versão das funcionalidades disponíveis no JavaScript (como veremos mais adiante neste livro). Essa implementação específica é baseada na ECMAScript. Assim, em sua maior parte, os navegadores oferecem as mesmas funcionalidades (novo código JavaScript executará em todos os navegadores); no entanto, o comportamento de cada funcionalidade poderá ser um pouco diferente de navegador para navegador.

Todo o código apresentado até agora neste capítulo é baseado na ECMAScript 5 (ES5 – ES é simplesmente uma abreviatura para ECMAScript), que se tornou um padrão em dezembro de 2009. A ECMAScript 2015 (ES2015) foi padronizada em junho de 2015, quase seis anos depois de sua versão anterior. O nome **ES6** se tornou popular antes do lançamento da ES2015.

O comitê responsável pela redação das especificações da ECMAScript tomou a decisão de adotar um modelo anual para definir novos padrões, segundo o qual novas funcionalidades seriam acrescentadas à medida que fossem aprovadas. Por esse motivo, a sexta edição da ECMAScript foi renomeada para ECMAScript 2015 (ES6).

Em junho de 2016, a sétima versão da ECMAScript foi padronizada. Ela é conhecida como **ECMAScript 2016** ou **ES2016 (ES7)**.

Em junho de 2017, a oitava versão da ECMAScript foi padronizada. Ela é conhecida como **ECMAScript 2017** ou **ES2017 (ES8)**. Quando este livro foi escrito, essa era a versão mais recente da ES.

Você também talvez veja **ES.Next** em alguns textos. Esse termo é uma referência à próxima versão da ECMAScript.

Nessa discussão, abordaremos algumas das novas funcionalidades introduzidas a partir da ES2015, as quais serão convenientes para o desenvolvimento de nossas estruturas de dados e algoritmos.

Tabela de compatibilidade

É importante saber que, apesar de as versões ES2015 a ES2017 já terem sido lançadas, suas funcionalidades talvez não sejam aceitas por todos os navegadores. Para ter uma boa experiência, é sempre melhor usar a versão

mais recente disponível para o navegador que você escolher.

Nos links a seguir, você poderá conferir quais funcionalidades estão disponíveis em cada navegador:

- ES2015 (ES6): <http://kangax.github.io/compat-table/es6/>
- ES2016+: <http://kangax.github.io/compat-table/es2016plus/>

Depois da ES5, a principal versão lançada da ES foi a ES2015. De acordo com a tabela de compatibilidade no link anterior, a maior parte de seus recursos está disponível nos navegadores modernos. Mesmo que alguns dos recursos das versões ES2016+ ainda não estejam disponíveis, podemos começar a usar a nova sintaxe e as novas funcionalidades hoje.

Por padrão, o Firefox acrescenta suporte para a ES à medida que a sua equipe disponibiliza a implementação das funcionalidades.

No Google Chrome, você pode ativar essas funcionalidades ativando a flag **Experimental JavaScript** (JavaScript experimental) acessando o URL `chrome://flags/#enable-javascript-harmony`, como vemos na Figura 2.1:

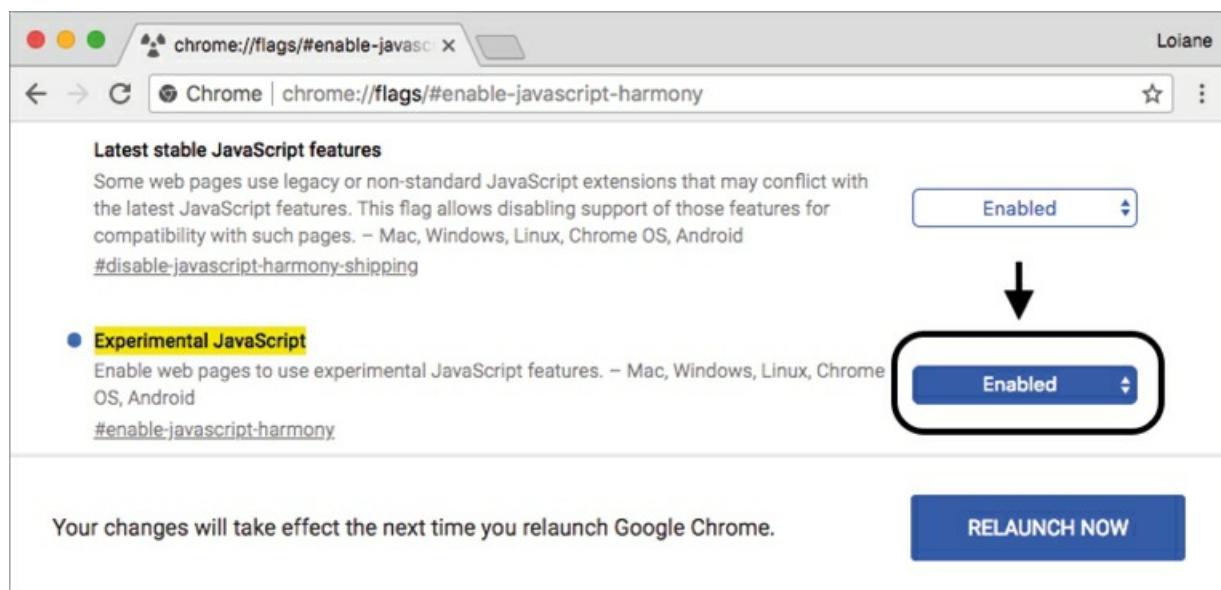


Figura 2.1

No Microsoft Edge, você pode acessar `about:flags` e selecionar a flag **Enable experimental JavaScript features** (Habilitar recursos experimentais de JavaScript) – um processo similar ao do Chrome.

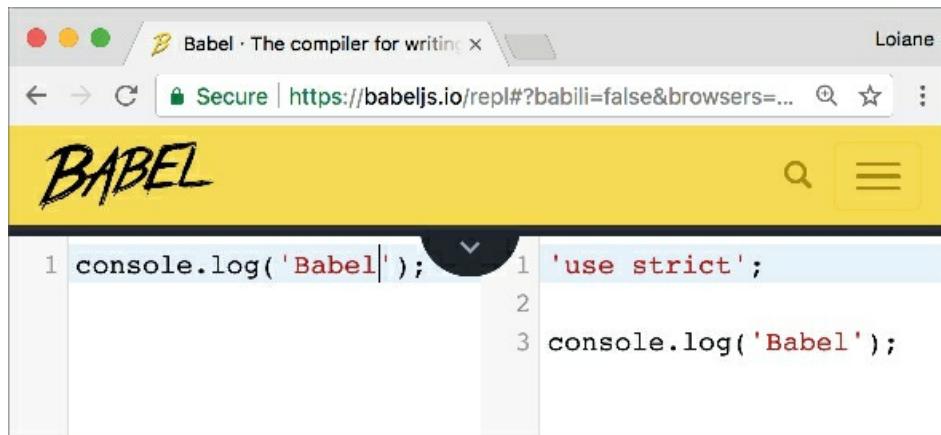
Mesmo com a flag **Enable Experimental JavaScript** (Habilitar JavaScript

experimental) ativada, alguns dos recursos das versões ES2016+ talvez não sejam aceitos pelo Chrome ou pelo Edge. O mesmo pode se aplicar ao Firefox. Para saber exatamente quais recursos já são aceitos em cada navegador, consulte a tabela de compatibilidade.

Usando o Babel.js

O Babel (<https://babeljs.io>) é um transpilador (transpiler) JavaScript, também conhecido como compilador de fonte para fonte (source-to-source compiler). Ele converte código JavaScript com recursos da linguagem ECMAScript em um código equivalente que use somente recursos de linguagem da especificação ES5, amplamente aceita.

Há muitas maneiras de usar o Babel.js. Uma delas é instalá-lo de acordo com a sua configuração (<https://babeljs.io/docs/setup>). Outra é usá-lo diretamente no navegador utilizando a sua opção **Try it out** (Experimente) em <https://babeljs.io/repl>, como mostra a Figura 2.2:

A screenshot of a web browser window titled "Babel · The compiler for writing next-generation JavaScript". The address bar shows a secure connection to https://babeljs.io/repl#?babili=false&browsers=... The main content area has a yellow header with the word "BABEL" in large letters. Below the header, there are two code editors. The left editor contains the following code: 1 console.log('Babel');. The right editor shows the transpiled code: 1 'use strict'; 2 3 console.log('Babel');. The browser's status bar at the bottom indicates "Loiane".

```
1 console.log('Babel');
1 'use strict';
2
3 console.log('Babel');
```

Figura 2.2

Junto com cada exemplo que será apresentado nas próximas seções, também forneceremos um link para que você possa executar e testar os exemplos no Babel.

Funcionalidades das versões ECMAScript 2015+

Nesta seção, mostraremos como usar algumas das novas funcionalidades da ES2015, as quais poderão ser úteis na programação JavaScript do dia a dia, e que também serão convenientes para simplificar os exemplos apresentados nos próximos capítulos deste livro.

Entre as funcionalidades, discutiremos:

- variáveis com **let** e **const**;
- templates literais;
- desestruturação (destructuring);
- operador de espalhamento (spread operator);
- funções de seta (arrow functions) usando **=>**;
- classes;
- módulos.

let e **const** no lugar de **var**

Até a ES5, podíamos declarar variáveis em qualquer ponto de nosso código, mesmo que sobrescrevêssemos a sua declaração, como no código a seguir:

```
var framework = 'Angular';
var framework = 'React';
console.log(framework);
```

A saída do código anterior é **React**, pois a última variável declarada, cujo nome é **framework**, recebeu esse valor. No código anterior, tínhamos duas variáveis com o mesmo nome; isso é muito perigoso e pode levar o código a gerar um resultado incorreto.

Outras linguagens, como C, Java e C#, não permitem esse comportamento.

Na ES2015, uma nova palavra reservada chamada **let** foi introduzida. **let** é a nova palavra reservada **var**, o que significa que podemos simplesmente substituir a palavra reservada **var** por **let**. No código a seguir, temos um exemplo:

```
let language = 'JavaScript!'; // {1}
let language = 'Ruby!'; // {2} - lança um erro
console.log(language);
```

A linha {2} lançará um erro porque uma variável chamada **language** já foi declarada no mesmo escopo (linha {1}). Discutiremos o **let** e o escopo das variáveis na próxima seção.

O código anterior pode ser testado e executado em <https://goo.gl/he0udZ>.

A ES2015 também introduziu a palavra reservada **const**. O seu comportamento é igual ao da palavra reservada **let**; a única diferença é que uma variável definida como **const** tem um valor somente para leitura, isto é, tem um valor constante.

Considere o código a seguir:

```
const PI = 3.141593;  
PI = 3.0; //lança um erro  
console.log(PI);
```

Se tentarmos atribuir um novo valor para **PI** ou até mesmo se a declararmos novamente usando **var PI** ou **let PI**, o código lançará um erro informando que **PI** é somente para leitura.

Vamos analisar outro exemplo com **const**. Declararemos um objeto como **const**:

```
const jsFramework = {  
  name: 'Angular'  
};
```

Vamos tentar alterar o **name** da variável **jsFramework**:

```
jsFramework.name = 'React';
```

Se tentarmos executar esse código, ele funcionará. No entanto, variáveis **const** são somente de leitura! Então por que é possível executar o código anterior? Para tipos que não sejam objetos, como número, booleano ou até mesmo string, isso significa que não podemos modificar os valores das variáveis. Ao trabalhar com objetos, um **const** somente para leitura permite que as propriedades do objeto recebam novos valores ou sejam atualizadas, mas a referência à variável em si (o endereço de referência na memória) não pode ser alterada, o que significa que ela não pode receber um novo valor.

Se tentarmos atribuir uma nova referência à variável **jsFramework**, como vemos a seguir, o compilador reclamará e lançará um erro ("**jsFramework**" **is read-only**).

```
// erro, não é possível atribuir outra referência ao objeto  
jsFramework = {  
  name: 'Vue'  
};
```

O código anterior pode ser executado em <https://goo.gl/YUQj3r>.

Escopo de variáveis com let e const

Para entender como as variáveis declaradas com as palavras reservadas **let** ou **const** funcionam, vamos usar o seguinte exemplo (você pode executá-lo usando o URL <https://goo.gl/NbsVvg>):

```
let movie = 'Lord of the Rings'; // {1}
//var movie = 'Batman v Superman'; // erro, movie já foi declarada
function starWarsFan() {
  const movie = 'Star Wars'; // {2}
  return movie;
}
function marvelFan() {
  movie = 'The Avengers'; // {3}
  return movie;
}
function blizzardFan() {
  const isFan = true;
  let phrase = 'Warcraft'; // {4}
  console.log('Before if: ' + phrase);
  if (isFan) {
    let phrase = 'initial text'; // {5}
    phrase = 'For the Horde!'; // {6}
    console.log('Inside if: ' + phrase);
  }
  phrase = 'For the Alliance!'; // {7}
  console.log('After if: ' + phrase);
}
console.log(movie); // {8}
console.log(starWarsFan()); // {9}
console.log(marvelFan()); // {10}
console.log(movie); // {11}
blizzardFan(); // {12}
```

Eis a saída do código anterior:

```
Lord of the Rings
Star Wars
The Avengers
The Avengers
Before if: Warcraft
Inside if: For the Horde!
After if: For the Alliance!
```

A seguir, apresentamos uma explicação de por que obtivemos essa saída:

- Na linha {1}, declaramos uma variável **movie** cujo valor é **Lord of the Rings** e exibimos o seu valor na linha {8}. Essa variável tem um escopo

global, conforme vimos na seção *Escopo das variáveis* no capítulo anterior.

- Na linha {9}, executamos a função `starWarsFan`. Nessa função, também declaramos uma variável chamada `movie` na linha {2}. A saída dessa função é `Star Wars` porque a variável da linha {2} tem um escopo local, o que significa que ela é válida somente dentro dessa função.
- Na linha {10}, executamos a função `marvelFan`. Nessa função, alteramos o valor da variável `movie` (linha {3}). Essa variável fez uma referência à variável global declarada na linha {1}. É por isso que obtivemos a saída `The Avengers` na linha {10} e na linha {11}, nas quais exibimos a variável global.
- Por fim, executamos a função `blizzardFan` na linha {12}. Nessa função, declaramos uma variável chamada `phrase` (linha {4}) no escopo da função. Então, na linha {5}, declaramos novamente uma variável chamada `phrase`, porém, dessa vez, essa variável tem como escopo somente a instrução `if`.
- Na linha {6}, alteramos o valor de `phrase`. Como ainda estamos dentro da instrução `if`, apenas a variável declarada na linha {5} terá o seu valor alterado.
- Em seguida, na linha {7}, alteramos novamente o valor de `phrase`, mas, como não estamos no bloco da instrução `if`, o valor da variável declarada na linha {4} é alterado.

Esse comportamento quanto aos escopos é igual ao de outras linguagens de programação, como Java ou C. Entretanto, isso só foi introduzido em JavaScript na ES2015 (ES6).

Observe que, no código apresentado na seção, estamos misturando `let` e `const`. Qual deles devemos usar? Alguns desenvolvedores (e algumas ferramentas lint) preferem usar `const` se a referência à variável não mudar. No entanto, essa é uma questão de preferência pessoal; não há uma escolha errada!

Templates literais

Os templates literais são um recurso interessante, pois podemos criar

strings sem a necessidade de concatenar os valores.

Por exemplo, considere os exemplos a seguir, escritos com ES5:

```
const book = {
  name: 'Learning JavaScript Data Structures and Algorithms'
};
console.log('You are reading ' + book.name + '.', '\n and this is a new line\n and so is this.');
```

Podemos melhorar a sintaxe da saída do `console.log` anterior com o código a seguir:

```
console.log(`You are reading ${book.name}.,
  and this is a new line
  and so is this.`);
```

Os templates literais devem estar entre crases (`). Para interpolar um valor de variável, basta defini-lo com um sinal de cifrão e chaves (`${}`), como fizemos com `book.name`.

Os templates literais também podem ser usados para strings multilinha. Não há mais necessidade de usar `\n`. Basta pressionar *Enter* no teclado para gerar uma quebra de linha na string, como foi feito com `and this is a new line` no exemplo anterior.

Essa funcionalidade será muito útil em nossos exemplos para simplificar a saída!

Os exemplos anteriores podem ser executados em <https://goo.gl/4N36cs>.

Funções de seta

As funções de seta (arrow functions) são uma ótima maneira de simplificar a sintaxe das funções na ES2015. Considere o exemplo a seguir:

```
var circleAreaES5 = function circleArea(r) {
  var PI = 3.14;
  var area = PI * r * r;
  return area;
};
console.log(circleAreaES5(2));
```

Podemos simplificar a sintaxe do código anterior com este código:

```
const circleArea = r => { // {1}
  const PI = 3.14;
  const area = PI * r * r;
  return area;
```

```
};

console.log(circleArea());
```

A principal diferença está na linha `{1}` do exemplo, na qual podemos omitir a palavra reservada `function` usando `=>`.

Se a função tiver uma única instrução, podemos usar uma versão mais simples, omitindo a palavra reservada `return` e as chaves, conforme mostrado no trecho de código a seguir:

```
const circleArea2 = r => 3.14 * r * r;
console.log(circleArea2(2));
```

Se a função não receber nenhum argumento, usamos parênteses vazios, comumente usados na ES5:

```
const hello = () => console.log('hello!');
hello();
```

Os exemplos anteriores podem ser executados em <https://goo.gl/nM414v>.

Valores default para parâmetros de funções

Com a ES2015, também é possível definir valores default para os parâmetros das funções. Eis um exemplo:

```
function sum(x = 1, y = 2, z = 3) {
  return x + y + z;
}
console.log(sum(4, 2)); // exibe 9
```

Como não estamos passando `z` como parâmetro, ele terá um valor igual a `3` como default. Desse modo, `4 + 2 + 3 == 9`.

Antes da ES2015, era necessário escrever a função anterior usando o código a seguir:

```
function sum(x, y, z) {
  if (x === undefined) x = 1;
  if (y === undefined) y = 2;
  if (z === undefined) z = 3;
  return x + y + z;
}
```

Ou, também, poderíamos ter escrito o código assim:

```
function sum() {
  var x = arguments.length > 0 && arguments[0] !== undefined ? arguments[0] : 1;
  var y = arguments.length > 1 && arguments[1] !== undefined ? arguments[1] : 2;
  var z = arguments.length > 2 && arguments[2] !== undefined ? arguments[2] : 3;
```

```
    return x + y + z;
}
```

As funções JavaScript também têm um objeto embutido chamado `arguments`. Esse objeto é um array dos argumentos usados quando a função é chamada. Podemos acessar dinamicamente os argumentos e usá-los, mesmo que não saibamos o nome deles.

Com a ES2015, podemos economizar algumas linhas de código usando a funcionalidade de valores default para parâmetros.

O exemplo anterior pode ser executado em <https://goo.gl/AP5EYb>.

Declarando os operadores de espalhamento e rest

Na ES5, podemos transformar arrays em parâmetros usando a função `apply()`.

A ES2015 tem o operador de espalhamento (spread operator), representado por `...`, para isso. Por exemplo, considere a função `sum` que declaramos na seção anterior. Podemos executar o código a seguir para passar os parâmetros `x`, `y` e `z`:

```
let params = [3, 4, 5];
console.log(sum(...params));
```

O código anterior é igual a este código escrito em ES5:

```
console.log(sum.apply(undefined, params));
```

O operador de espalhamento `(...)` também pode ser usado como um parâmetro rest em funções para substituir `arguments`. Considere o exemplo a seguir:

```
function restParamaterFunction(x, y, ...a) {
  return (x + y) * a.length;
}
console.log(restParamaterFunction(1, 2, 'hello', true, 7));
```

O código anterior é igual ao código a seguir (também exibe 9 no console):

```
function restParamaterFunction(x, y) {
  var a = Array.prototype.slice.call(arguments, 2);
  return (x + y) * a.length;
}
console.log(restParamaterFunction(1, 2, 'hello', true, 7));
```

O exemplo com o operador de espalhamento pode ser executado em <https://goo.gl/8equk5>, e o exemplo com o parâmetro rest pode ser executado em <https://goo.gl/LaJZqU>.

Propriedades melhoradas de objetos

A ES6 introduziu um conceito chamado **desestruturação de array** (array destructuring), que é uma maneira de inicializar variáveis de uma só vez. Por exemplo, considere o código a seguir:

```
let [x, y] = ['a', 'b'];
```

Executar o código anterior é o mesmo que fazer o seguinte:

```
let x = 'a';
let y = 'b';
```

A desestruturação de array também pode ser usada para trocar valores (fazer swap) de uma só vez, sem a necessidade de criar variáveis **temp**, deste modo:

```
[x, y] = [y, x];
```

O código anterior é o mesmo que:

```
var temp = x;
x = y;
y = temp;
```

Esse recurso será muito útil quando você conhecer os algoritmos de ordenação, pois essas trocas de valores são muito comuns.

Há também outra funcionalidade chamada **abreviação de propriedades** (property shorthand), que é outra maneira de executar a desestruturação de objetos. Por exemplo, considere o código a seguir:

```
let [x, y] = ['a', 'b'];
let obj = { x, y };
console.log(obj); // { x: "a", y: "b" }
```

O código anterior é o mesmo que o seguinte:

```
var x = 'a';
var y = 'b';
var obj2 = { x: x, y: y };
console.log(obj2); // { x: "a", y: "b" }
```

A última funcionalidade que discutiremos nesta seção se chama **nomes de método abreviados** (shorthand method names). Ela permite que os desenvolvedores declarem funções dentro de objetos, como se elas fossem

propriedades. Eis um exemplo:

```
const hello = {
  name: 'abcdef',
  printHello() {
    console.log('Hello');
  }
};
console.log(hello.printHello());
```

O código anterior também pode ser escrito assim:

```
var hello = {
  name: 'abcdef',
  printHello: function printHello() {
    console.log('Hello');
  }
};
console.log(hello.printHello());
```

Os três exemplos apresentados podem ser executados em:

- desestruturação de array: <https://goo.gl/VsLecp>
- troca (swap) de variáveis: <https://goo.gl/EyFAII>
- abreviatura de propriedades: <https://goo.gl/DKU2PN>

Programação orientada a objetos com classes

A ES2015 também introduziu uma maneira mais limpa de declarar classes. Vimos que podemos declarar uma classe chamada `Book` na seção sobre programação orientada a objetos, desta maneira:

```
function Book(title, pages, isbn) { // {1}
  this.title = title;
  this.pages = pages;
  this.isbn = isbn;
}
Book.prototype.printTitle = function() {
  console.log(this.title);
};
```

Com a ES2015, podemos simplificar a sintaxe e usar o código a seguir:

```
class Book { // {2}
  constructor(title, pages, isbn) {
    this.title = title;
    this.pages = pages;
    this.isbn = isbn;
  }
}
```

```

printIsbn() {
  console.log(this.isbn);
}
}

```

Podemos simplesmente usar a palavra reservada **class** e declarar uma classe com uma função **constructor**, além de outras funções – por exemplo, a função **printIsbn**. As classes da ES2015 são um açúcar sintático para a sintaxe baseada em protótipo. O código da classe **Book** declarada na linha {1} tem o mesmo comportamento e a mesma saída do código declarado na linha {2}:

```

let book = new Book('title', 'pag', 'isbn');
console.log(book.title); // exibe o título do livro
book.title = 'new title'; // atualiza o valor do título do livro
console.log(book.title); // exibe o título do livro

```

O exemplo anterior pode ser executado em <https://goo.gl/UhK1n4>.

Herança

Na ES2015, há também uma sintaxe simplificada para usar herança entre classes. Vamos observar um exemplo:

```

class ITBook extends Book { // {1}
  constructor(title, pages, isbn, technology) {
    super(title, pages, isbn); // {2}
    this.technology = technology;
  }
  printTechnology() {
    console.log(this.technology);
  }
}
let jsBook = new ITBook('Learning JS Algorithms', '200', '1234567890',
'JavaScript');
console.log(jsBook.title);
console.log(jsBook.printTechnology());

```

Podemos estender outra classe e herdar o seu comportamento usando a palavra reservada **extends** (linha {1}). No construtor, podemos também referenciar o **constructor** da superclasse usando a palavra reservada **super** (linha {2}).

Embora a sintaxe dessa nova forma de declarar classes em JavaScript seja muito semelhante a outras linguagens de programação como Java e

C/C++, é bom lembrar que a programação orientada a objetos em JavaScript é feita por meio de um protótipo.

O exemplo anterior pode ser executado em <https://goo.gl/hgQvo9>.

Trabalhando com getters e setters

Também é possível criar funções getters e setters para os atributos de classe com a ES2015. Embora os atributos da classe não sejam privados como em outras linguagens orientadas a objetos (o conceito de encapsulamento), é bom seguir um padrão de nomenclatura.

A seguir, apresentamos um exemplo de uma classe que declara uma função **get** e uma função **set**, junto com o seu uso:

```
class Person {
  constructor(name) {
    this._name = name; // {1}
  }
  get name() { // {2}
    return this._name;
  }
  set name(value) { // {3}
    this._name = value;
  }
}
let lotrChar = new Person('Frodo');
console.log(lotrChar.name); // {4}
lotrChar.name = 'Gandalf'; // {5}
console.log(lotrChar.name);
lotrChar._name = 'Sam'; // {6}
console.log(lotrChar.name);
```

Para declarar uma função **get** e uma função **set**, basta usar a palavra reservada **get** ou **set** na frente do nome da função (linhas {2} e {3}), o nome que queremos expor e que será usado. Podemos declarar os atributos da classe com o mesmo nome, ou podemos usar um underscore na frente do nome do atributo (linha {1}) para transparecer que esse atributo é privado.

Então, para usar as funções **get** ou **set**, podemos simplesmente referenciar seus nomes como se fossem atributos simples (linhas {4} e {5}).

O atributo **_name** não é privado, e ele continua podendo ser acessado (linha {6}). Esse assunto, porém, será discutido mais adiante neste livro.

Esse exemplo pode ser executado em <https://goo.gl/SMRYsv>.

Operador de exponencial

O operador de exponencial pode ser conveniente quando trabalhamos com matemática. Vamos usar a fórmula para calcular a área de um círculo como exemplo:

```
const area = 3.14 * r * r;
```

Poderíamos usar também a função `Math.pow` para escrever o mesmo código:

```
const area = 3.14 * Math.pow(r, 2);
```

A ES2016 introduziu o operador `**`, concebido para ser o novo operador de exponencial. Podemos calcular a área de um círculo usando o operador de exponencial, assim:

```
const area = 3.14 * (r ** 2);
```

Esse exemplo pode ser executado em <https://goo.gl/Z6dCFB>.

As versões ES2015+ também têm outras funcionalidades: entre elas, podemos listar os iteradores, os arrays tipados, `Set`, `Map`, `WeakSet`, `WeakMap`, chamadas de cauda (tail calls), `for..of`, `Symbol`, `Array.prototype.includes`, vírgulas no final, preenchimento de string, métodos estáticos de objetos e assim por diante. Abordaremos algumas dessas outras funcionalidades em outros capítulos deste livro.

Você pode conferir a lista de todas as funcionalidades disponíveis em JavaScript e em ECMAScript em <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.

Módulos

Os desenvolvedores de Node.js já têm familiaridade para trabalhar com módulos usando a instrução `require` (módulos **CommonJS**). Há também outro padrão JavaScript popular para módulos, que é o **AMD** (Asynchronous Module Definition, ou Definição de Módulo Assíncrono). O **RequireJS** é a implementação mais conhecida de AMD. A ES2015 introduziu uma funcionalidade oficial de módulo na especificação de

JavaScript. Vamos criar e usar alguns módulos.

O primeiro módulo que criaremos contém duas funções para calcular a área de figuras geométricas. Em um arquivo (**17-CalcArea.js**), adicione o código a seguir:

```
const circleArea = r => 3.14 * (r ** 2);
const squareArea = s => s * s;
export { circleArea, squareArea }; // {1}
```

Isso significa que estamos expondo as duas funções para que outros arquivos possam usá-las ({1}). Somente membros exportados são visíveis aos outros módulos ou arquivos.

No arquivo principal desse exemplo (**17-ES2015-ES6-Modules.js**), usaremos as funções declaradas no arquivo **17-CalcArea.js**. O trecho a seguir apresenta o código que consome as duas funções:

```
import { circleArea, squareArea } from './17-CalcArea'; // {2}
console.log(circleArea(2));
console.log(squareArea(2));
```

Em primeiro lugar, precisamos importar as funções que queremos usar nesse arquivo ({2}); depois de importá-las, elas poderão ser chamadas.

Se quiséssemos usar a função **circleArea**, poderíamos importar somente essa função também:

```
import { circleArea } from './17-CalcArea';
```

Basicamente, os módulos são um código JavaScript declarado em arquivos separados. Podemos importar as funções, as variáveis e as classes de outros arquivos diretamente no código JavaScript (sem a necessidade de importar vários arquivos no HTML antes – e na ordem correta – como costumávamos fazer há alguns anos, antes que o JavaScript se tornasse moderno e popular). Os módulos nos permitem organizar melhor o nosso código se estivermos criando uma biblioteca ou trabalhando em um projeto grande.

Há também a opção de usar o membro exportado com um nome diferente quando fazemos a importação, assim:

```
import { circleArea as circle } from './17-CalcArea';
```

Ou podemos renomear as funções quando elas forem exportadas:

```
export { circleArea as circle, squareArea as square };
```

Nesse caso, os membros exportados devem ser importados com o nome exposto, e não com o nome usado internamente, assim:

```
import { circle, square } from './17-CalcArea';
```

Há também algumas formas diferentes para importar as funções em um módulo distinto:

```
import * as area from './17-CalcArea';
console.log(area.circle(2));
console.log(area.square(2));
```

Nesse caso, podemos importar o módulo todo como uma única variável e chamar os membros exportados como se fossem atributos ou métodos de uma classe.

Também é possível adicionar a palavra reservada **export** na frente de cada função ou variável que queremos expor. Não precisamos ter uma declaração **export** no final do arquivo:

```
export const circleArea = r => 3.14 * (r ** 2);
export const squareArea = s => s * s;
```

Suponha que tivéssemos somente um único membro no módulo e quiséssemos exportá-lo. Poderíamos usar as palavras reservadas **export default** assim:

```
export default class Book {
  constructor(title) {
    this.title = title;
  }
  printTitle() {
    console.log(this.title);
  }
}
```

Para importar a classe anterior em um módulo diferente, use o código a seguir:

```
import Book from './17-Book';
const myBook = new Book('some title');
myBook.printTitle();
```

Observe que, nesse caso, não precisamos usar chaves (`{}`) em torno do nome da classe. Só usamos chaves se o módulo tiver mais de um membro exportado.

Usaremos módulos ao criar nossa biblioteca de estruturas de dados e de algoritmos mais adiante neste livro.

Para obter mais informações sobre os módulos na ES2015, por favor, acesse http://exploringjs.com/es6/ch_modules.html. Você também pode

analisar todo o código-fonte desse exemplo fazendo o download do pacote com o código-fonte deste livro.

Executando módulos ES2015 no navegador e com o Node.js

Vamos tentar executar o arquivo `17-ES2015-ES6-Modules.js` com o Node.js, mudando de diretório e então executando o comando `node`, assim:

```
cd path-source-bundle/examples/chapter01  
node 17-ES2015-ES6-Modules
```

Veremos um erro: `SyntaxError: Unexpected token import`. Isso ocorre porque, quando este livro foi escrito, o Node.js não tinha suporte para módulos ES2015 nativos. O Node.js usa a sintaxe do **CommonJS** com `require`, e isso significa que precisamos transpilar o nosso ES2015 para que o Node o entenda. Há diferentes ferramentas que podem ser usadas nessa tarefa. Para simplificar, usaremos o Babel CLI.

A configuração completa e a descrição detalhada do uso do Babel podem ser encontradas em <https://babeljs.io/docs/setup> e em <https://babeljs.io/docs/usage/cli/>.

A melhor abordagem seria criar um projeto local e configurá-lo para usar o Babel. Infelizmente, todos esses detalhes não estão no escopo deste livro (seria um assunto para um livro sobre o Babel). Em nosso exemplo, e para manter a simplicidade, usaremos o Babel CLI globalmente, instalando-o com o `npm`:

```
npm install -g babel-cli
```

Se você usa Linux ou Mac OS, talvez queira usar `sudo` na frente do comando para ter acesso de administrador (`sudo npm install -g babel-cli`).

A partir do diretório `chapter01`, compilaremos os três arquivos JavaScript com módulos que criamos antes para código CommonJS transpilado com o Babel, a fim de podermos usar o código com Node.js. Transpilaremos o arquivo para a pasta `chapter01/lib` usando os comandos a seguir:

```
babel 17-CalcArea.js --out-dir lib  
babel 17-Book.js --out-dir lib  
babel 17-ES2015-ES6-Modules.js --out-dir lib
```

Em seguida, vamos criar um arquivo JavaScript chamado **17-ES2015-ES6-Modules-node.js** para que usemos as funções de `area` e a classe `Book`:

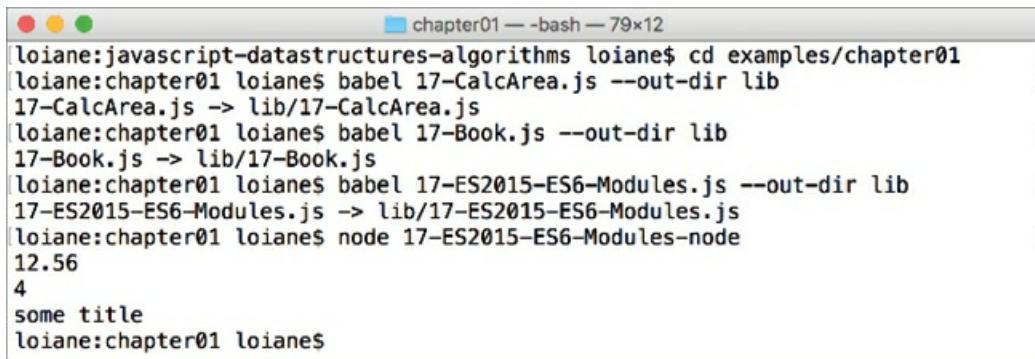
```
const area = require('./lib/17-CalcArea');
const Book = require('./lib/17-Book');
console.log(area.circle(2));
console.log(area.square(2));
const myBook = new Book('some title');
myBook.printTitle();
```

O código é, basicamente, o mesmo, mas a diferença é que, como o Node.js não aceita a sintaxe `import` (por enquanto), precisamos usar a palavra reservada `require`.

Para executar o código, podemos usar o comando a seguir:

```
node 17-ES2015-ES6-Modules-node
```

Na Figura 2.3, podemos ver os comandos e a saída, de modo que é possível confirmar que o código funciona com o Node.js:



```
loiane:javascript-datastructures-algorithms loiane$ cd examples/chapter01
loiane:chapter01 loiane$ babel 17-CalcArea.js --out-dir lib
17-CalcArea.js -> lib/17-CalcArea.js
loiane:chapter01 loiane$ babel 17-Book.js --out-dir lib
17-Book.js -> lib/17-Book.js
loiane:chapter01 loiane$ babel 17-ES2015-ES6-Modules.js --out-dir lib
17-ES2015-ES6-Modules.js -> lib/17-ES2015-ES6-Modules.js
loiane:chapter01 loiane$ node 17-ES2015-ES6-Modules-node
12.56
4
some title
loiane:chapter01 loiane$
```

Figura 2.3

Usando importações nativas da ES2015 no Node.js

Seria bom se pudéssemos usar as importações da ES2015 no Node.js para que não precisássemos transpilar o nosso código. A partir do Node 8.5, podemos usar importações da ES2015 no Node.js como uma funcionalidade experimental.

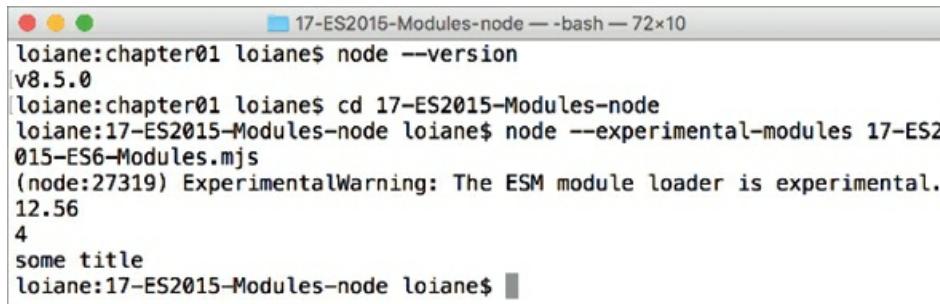
Neste exemplo, criaremos uma pasta em `chapter01`, chamada `17-ES2015-Modules-node`. Copiaremos os arquivos `17-CalcArea.js`, `17-Book.js` e `17-ES2015-ES6-Modules.js` para essa pasta e modificaremos a extensão, de `.js` para `.mjs` (a extensão `.mjs` é um requisito para que isso possa funcionar). No arquivo `17-ES2015-ES6-Modules.mjs`, atualizaremos as importações, acrescentando a extensão `.mjs`, assim:

```
import * as area from './17-CalcArea.mjs';
import Book from './17-Book.mjs';
```

Para executar o código, executaremos o comando `node`, passando `--experimental-modules` para ele, como vemos a seguir:

```
cd 17-ES2015-Modules-node
node --experimental-modules 17-ES2015-ES6-Modules.mjs
```

Na Figura 2.4, podemos ver os comandos e a saída:



A screenshot of a terminal window titled "17-ES2015-Modules-node — bash — 72x10". The terminal shows the following command-line session:

```
loiane:chapter01 loiane$ node --version
v8.5.0
loiane:chapter01 loiane$ cd 17-ES2015-Modules-node
loiane:17-ES2015-Modules-node loiane$ node --experimental-modules 17-ES2
015-ES6-Modules.mjs
(node:27319) ExperimentalWarning: The ESM module loader is experimental.
12.56
4
some title
loiane:17-ES2015-Modules-node loiane$
```

Figura 2.4

Quando este livro foi escrito, a versão para que o Node.js aceitasse a funcionalidade de importação da ES2015 era o Node 10 LTS.

Mais exemplos e informações sobre o suporte nativo às importações da ES2015 no Node.js podem ser encontrados em <https://github.com/nodejs/node-eps/blob/master/002-es-modules.md>.

Executando módulos ES2015 no navegador

Há diferentes abordagens para executar um código com ES2015 no navegador.

A primeira é disponibilizar o bundle tradicional (um arquivo JavaScript com o nosso código transpilado para ES5). Podemos criar um bundle usando ferramentas populares, como **Browserify** e **Webpack**. Nesse caso, criamos o arquivo de distribuição (bundle) e, em nosso arquivo HTML, importamos esse arquivo como qualquer outro código JavaScript:

```
<script src="./lib/17-ES2015-ES6-Modules-bundle.js"></script>
```

O suporte a módulos ES2015 finalmente foi incorporado nos navegadores no início de 2017. Quando este livro foi escrito, ainda era uma funcionalidade experimental, e não era aceita por todos os navegadores modernos. Para saber como é atualmente o suporte para esse recurso (e como ativá-lo em modo experimental), consulte

<http://caniuse.com/#feat=es6-module>, como mostra a Figura 2.5:



Figura 2.5

Para usar a palavra reservada `import` no navegador, inicialmente devemos atualizar o nosso código adicionando a extensão `.js` no `import`, assim:

```
import * as area from './17-CalcArea.js';
import Book from './17-Book.js';
```

Em segundo lugar, para importar os módulos que criamos, basta adicionar `type="module"` dentro da tag `script`:

```
<script type="module" src="17-ES2015-ES6-Modules.js"></script>
```

Se executarmos o código e abrirmos a aba **Developer Tools | Network** (Ferramentas do desenvolvedor | Rede), poderemos ver que todos os arquivos que criamos foram carregados (Figura 2.6):

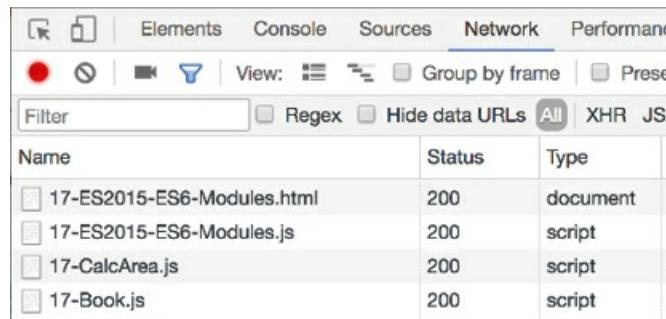


Figura 2.6

Para manter a compatibilidade com versões anteriores de navegadores que não aceitam essa funcionalidade, podemos usar `nomodule`:

```
<script nomodule src="./lib/17-ES2015-ES6-Modules-bundle.js"></script>
```

Até que esse recurso seja oficialmente aceito pela maioria dos navegadores modernos, continuará havendo a necessidade de usar uma ferramenta para gerar um bundle a fim de transpilar ES2015+.

Para saber mais sobre módulos ES2015 no navegador, por favor, acesse <https://goo.gl/cf1cGW> e <https://goo.gl/wBUJUo>.

Compatibilidade de versões anteriores a ES2015+

Preciso atualizar o meu código JavaScript atual para ES2015? A resposta é: só se você quiser! As versões ES2015+ são superconjuntos da linguagem JavaScript.

Tudo que foi padronizado como ES5 continuará funcionando como hoje. No entanto, você pode começar a usar versões ES2015+ para tirar proveito da nova sintaxe e deixar o seu código mais simples e mais fácil de ler.

Nos próximos capítulos do livro, usaremos as versões ES2015+ o máximo possível. Suponha que queremos criar uma biblioteca com estruturas de dados e algoritmos neste livro e, por padrão, queremos oferecer suporte aos desenvolvedores que queiram usar a nossa biblioteca no navegador (ES5) e no ambiente Node.js.

Por enquanto, a opção que temos é transpilar o nosso código para **UMD** (Universal Module Definition, ou Definição Universal de Módulo). Para obter mais informações sobre o UMD, acesse <https://github.com/umdjs/umd>. Conheceremos melhor o processo de transpilar código ES2015 com o Babel para UMD no Capítulo 4, *Pilhas*.

Junto com a sintaxe das versões ES2015+, nos exemplos que usam módulos, o bundle do código-fonte também inclui uma versão transpilada para que você possa executar o código em qualquer navegador.

Introdução ao TypeScript

O TypeScript é um superconjunto **gradualmente tipado** de JavaScript, com código aberto, criado e mantido pela Microsoft. Foi criado para permitir que os desenvolvedores potencializassem a linguagem JavaScript, além de facilitar que as aplicações fossem escaladas. Entre seus principais recursos está a possibilidade de usar tipagem em variáveis JavaScript. Os tipos em JavaScript permitem verificação estática, facilitando, assim, refatorar o

código e encontrar bugs. E, no final, o TypeScript compila gerando código JavaScript simples!

No que diz respeito ao escopo deste livro, com o TypeScript, podemos usar alguns conceitos de orientação a objetos que não estão disponíveis em JavaScript, como interfaces e propriedades privadas (isso pode ser útil ao trabalhar com estruturas de dados e algoritmos de ordenação). E, é claro, podemos também tirar proveito da funcionalidade de tipagem, que é muito importante para algumas estruturas de dados.

Todas essas funcionalidades estão disponíveis em **tempo de compilação**. Depois de escrever o nosso código, ele será compilado para JavaScript puro (ES5, ES2015+ e CommonJS, entre outras opções).

Para começar a usar o TypeScript, é necessário instalá-lo usando o **npm**:

```
npm install -g typescript
```

Em seguida, precisamos criar um arquivo com a extensão **.ts**, por exemplo, **hello-world.ts**:

```
let myName = 'Packt';
myName = 10;
```

O código anterior é um código ES2015 simples. Vamos agora compilá-lo usando o compilador **tsc**:

```
tsc hello-world
```

No Terminal, veremos o seguinte aviso:

```
hello-world.ts(2,1): error TS2322: Type '10' is not assignable to type 'string'.
```

Contudo, se verificarmos a pasta na qual criamos o arquivo, veremos que um arquivo **hello-world.js** foi criado com este conteúdo:

```
var myName = 'Packt';
myName = 10;
```

O código gerado anteriormente é ES5. Mesmo com o erro no Terminal (o qual, na verdade, não é um erro, mas um aviso), o compilador TypeScript gerou o código ES5 como deveria. Isso reforça o fato de que, embora o TypeScript faça toda a verificação de tipos e de erros em tempo de compilação, ele não impede que o compilador gere código JavaScript. Isso significa que os desenvolvedores podem tirar proveito de todas as validações quando escrevem o código, e podem obter um código JavaScript com menos chances de ter erros ou bugs.

Inferência de tipo

Quando trabalhamos com TypeScript, é muito comum ver um código como este:

```
let age: number = 20;
let existsFlag: boolean = true;
let language: string = 'JavaScript';
```

O TypeScript nos permite atribuir um tipo a uma variável. No entanto, o código anterior é verboso. O TypeScript tem inferência de tipos, o que significa que ele verificará e aplicará um tipo a uma variável automaticamente, com base no valor atribuído a ela. Vamos reescrever o código anterior com uma sintaxe mais limpa:

```
let age = 20; // number
let existsFlag = true; // boolean
let language = 'JavaScript'; // string
```

Com o código anterior, o TypeScript ainda saberá que `age` é um número, `existsFlag` é um booleano e `language` é uma string, portanto, não é necessário atribuir explicitamente um tipo a essas variáveis.

Então, quando devemos atribuir um tipo a uma variável? Se declararmos a variável e não a inicializarmos com um valor, é recomendável atribuir-lhe um tipo, como mostra o código a seguir:

```
let favoriteLanguage: string;
let langs = ['JavaScript', 'Ruby', 'Python'];
favoriteLanguage = langs[0];
```

Se não especificarmos um tipo para uma variável, ela será automaticamente tipada como `any`, o que significa que poderá receber qualquer valor, como ocorre em JavaScript.

Interfaces

Em TypeScript, há dois conceitos de interfaces. O primeiro está relacionado à atribuição de um tipo a uma variável. Considere o código a seguir:

```
interface Person {
  name: string;
  age: number;
}
function printName(person: Person) {
  console.log(person.name);
```

}

O primeiro conceito de interface em TypeScript é que uma interface deve existir. É uma descrição de atributos e de métodos que um objeto deve ter.

Isso permite que editores como o VSCode tenham preenchimento automático com o IntelliSense, como vemos na Figura 2.7:



Figura 2.7

Vamos agora experimentar usar a função `printName`:

```
const john = { name: 'John', age: 21 };
const mary = { name: 'Mary', age: 21, phone: '123-45678' };
printName(john);
printName(mary);
```

O código anterior não tem nenhum erro de compilação. A variável `john` tem `name` e `age`, conforme esperado pela função `printName`. A variável `mary` tem `name` e `age`, mas tem também informação em `phone`.

Então por que esse código funciona? O TypeScript tem um conceito chamado **duck typing** (tipagem pato). Se algo se parece com um pato, nada como um pato, faz quack como um pato, então deve ser um pato! No exemplo, a variável `mary` se comporta como a interface `Person`, portanto deve ser uma `Person`. Esse é um recurso eficaz do TypeScript.

Depois de executar o comando `tsc` novamente, veremos a saída a seguir no arquivo `hello-world.js`:

```
function printName(person) {
  console.log(person.name);
}
var john = { name: 'John', age: 21 };
var mary = { name: 'Mary', age: 21, phone: '123-45678' };
```

O código anterior é JavaScript puro. Os recursos de preenchimento de código e de verificação de tipos e erros estão disponíveis somente em tempo de compilação.

O segundo conceito de interface no TypeScript está relacionado à programação orientada a objetos. É o mesmo conceito presente em outras linguagens orientadas a objetos, como Java, C#, Ruby e assim por diante. Uma interface é um contrato. Nele, podemos definir o comportamento que as classes ou as interfaces que implementarão esse contrato devem ter. Considere o padrão ECMAScript. A ECMAScript é uma interface para a linguagem JavaScript. Ela diz à linguagem JavaScript quais funcionalidades ela deve ter, mas cada navegador pode ter uma implementação diferente para elas.

Considere o código a seguir:

```
interface Comparable {
    compareTo(b): number;
}

class MyObject implements Comparable {
    age: number;
    compareTo(b): number {
        if (this.age === b.age) {
            return 0;
        }
        return this.age > b.age ? 1 : -1;
    }
}
```

A interface **Comparable** diz à classe **MyObject** que ela deve implementar um método chamado **compareTo** que recebe um argumento. Nesse método, podemos implementar a lógica necessária. Nesse caso, estamos comparando dois números, mas poderíamos usar uma lógica diferente para comparar duas strings, ou até mesmo um objeto mais complexo com atributos distintos. Esse comportamento para uma interface não existe em JavaScript, mas é muito útil quando trabalhamos com algoritmos de ordenação, por exemplo.

Genéricos

Outro recurso eficaz do TypeScript, e útil em estruturas de dados e algoritmos, é o conceito de genérico (generic). Vamos modificar a interface **Comparable** para que possamos definir o tipo do objeto que o método **compareTo** deve receber como argumento:

```
interface Comparable<T> {
    compareTo(b: T): number;
```

}

Ao passar o tipo T dinamicamente para a interface **Comparable** usando o operador `<>`, podemos especificar o tipo do argumento da função `compareTo`:

```
class MyObject implements Comparable<MyObject> {
    age: number;
    compareTo(b: MyObject): number {
        if (this.age === b.age) {
            return 0;
        }
        return this.age > b.age ? 1 : -1;
    }
}
```

Esse recurso é útil para que possamos garantir que estamos comparando objetos do mesmo tipo; ao usar essa funcionalidade, também teremos o recurso de preenchimento de código no editor.

Outras funcionalidades do TypeScript

Essa foi uma introdução muito rápida ao TypeScript. A documentação do TypeScript é um ótimo local para conhecer todas as outras funcionalidades e explorar os detalhes sobre os assuntos que abordamos rapidamente neste capítulo; ela está disponível em <https://www.typescriptlang.org/docs/home.html>.

O TypeScript também tem um playground online (semelhante ao Babel) em <https://www.typescriptlang.org/play/index.html>, o qual pode ser usado para executar alguns exemplos de código.

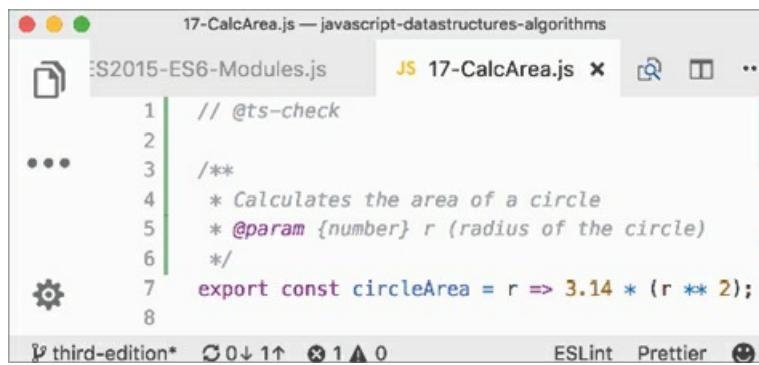
O bundle com o código-fonte deste livro apresenta também uma versão TypeScript da biblioteca com as estruturas de dados e os algoritmos JavaScript que desenvolveremos neste livro, como um recurso extra!

Verificações do TypeScript em tempo de compilação em arquivos JavaScript

Alguns desenvolvedores ainda preferem usar JavaScript puro para desenvolver o seu código, em vez de utilizar o TypeScript. No entanto, seria bom se pudéssemos usar alguns dos recursos de verificação de tipos e

de erros no JavaScript também!

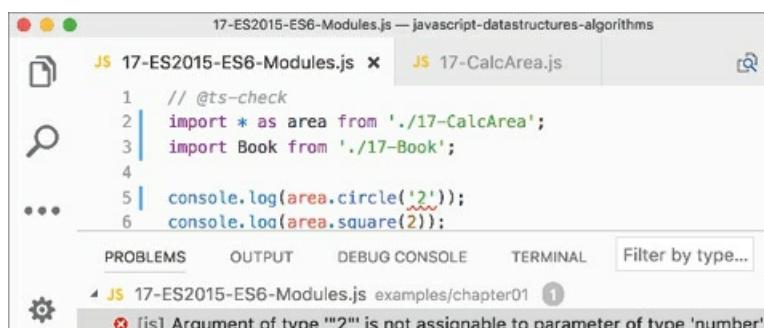
A boa notícia é que o TypeScript apresenta uma funcionalidade especial que nos permite ter essa verificação de erros e de tipos em tempo de compilação! Para usá-la, precisamos do TypeScript instalado globalmente em nosso computador. Na primeira linha dos arquivos JavaScript, se quisermos usar a verificação de tipos e de erros, basta acrescentar `// @ts-check`, conforme mostra a Figura 2.8.



```
// @ts-check
...
/**
 * Calculates the area of a circle
 * @param {number} r (radius of the circle)
 */
export const circleArea = r => 3.14 * (r ** 2);
```

Figura 2.8

A verificação de tipos é ativada quando adicionamos JSDoc (documentação do JavaScript) em nosso código. Então, se tentarmos passar uma string para o nosso método `circle` (ou `circleArea`), veremos um erro de compilação (Figura 2.9).



```
// @ts-check
import * as area from './17-CalcArea';
import Book from './17-Book';
...
console.log(area.circle('2'));
console.log(area.square(2));
```

Figura 2.9

Resumo

Neste capítulo, apresentamos uma visão geral de algumas das funcionalidades das versões ECMAScript 2015+, que nos ajudarão a simplificar a sintaxe dos exemplos que veremos no futuro. Também apresentamos o TypeScript para nos ajudar a tirar proveito da tipagem

estática e da verificação de erros.

No próximo capítulo, veremos a nossa primeira estrutura de dados, o array – a estrutura de dados mais básica que existe, aceita de modo nativo por muitas linguagens, incluindo o JavaScript.

CAPÍTULO 3

Arrays

Um **array** é a estrutura de dados mais simples possível em memória. Por esse motivo, todas as linguagens de programação têm um tipo de dado array incluído. A linguagem JavaScript também tem suporte nativo para arrays, embora a sua primeira versão tenha sido lançada sem esse recurso. Neste capítulo, exploraremos detalhadamente a estrutura de dados de array e seus recursos.

Um array armazena valores que são todos do mesmo tipo, sequencialmente. Embora o JavaScript nos permita criar arrays com valores de tipos distintos, obedeceremos às melhores práticas e partiremos do pressuposto de que não podemos fazer isso (a maioria das linguagens não tem essa capacidade).

Por que devemos usar arrays?

Vamos supor que precisamos armazenar a temperatura média de cada mês do ano para a cidade em que vivemos. Poderíamos usar algo semelhante ao código a seguir para armazenar essas informações:

```
const averageTempJan = 31.9;
const averageTempFeb = 35.3;
const averageTempMar = 42.4;
const averageTempApr = 52;
const averageTempMay = 60.8;
```

No entanto, essa não é a melhor abordagem. Se fôssemos armazenar a temperatura somente de um ano, poderíamos administrar as doze variáveis. O que ocorreria, porém, se precisássemos armazenar a temperatura média para mais de um ano? Felizmente, esse é o motivo pelo qual os arrays foram criados, e podemos representar facilmente as mesmas informações mencionadas antes da seguinte forma:

```
const averageTemp = [];
averageTemp[0] = 31.9;
averageTemp[1] = 35.3;
averageTemp[2] = 42.4;
```

```
averageTemp[3] = 52;  
averageTemp[4] = 60.8;
```

Também podemos representar o array **averageTemp** graficamente:

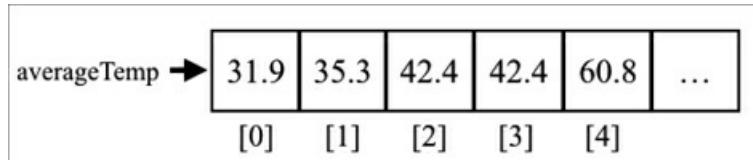


Figura 3.1

Criando e inicializando arrays

Declarar, criar e inicializar um array em JavaScript é realmente simples, conforme mostra o código a seguir:

```
let daysOfWeek = new Array(); // {1}  
daysOfWeek = new Array(7); // {2}  
daysOfWeek = new Array('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',  
'Friday', 'Saturday'); // {3}
```

Podemos apenas declarar e instanciar um novo array usando a palavra reservada **new** (linha **{1}**). Além disso, usando essa palavra reservada, podemos criar um array especificando o seu tamanho (linha **{2}**). Uma terceira opção é passar os elementos do array diretamente para o seu construtor (linha **{3}**).

Contudo, usar a palavra reservada **new** não é considerada a melhor prática. Se quisermos criar um array em JavaScript, podemos atribuir colchetes vazios (`[]`), como neste exemplo:

```
let daysOfWeek = [];
```

Também podemos inicializar o array com alguns elementos, assim:

```
let daysOfWeek = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',  
'Friday', 'Saturday'];
```

Se quisermos saber quantos elementos há no array (o seu tamanho), podemos usar a propriedade **length**. O código a seguir exibirá 7 como saída:

```
console.log(daysOfWeek.length);
```

Acessando elementos e fazendo uma iteração em um array

Para acessar uma posição em particular do array, podemos também usar

colchetes, passando o índice da posição que gostaríamos de acessar. Por exemplo, vamos supor que queremos exibir todos os elementos do array `daysOfWeek`. Para isso, precisamos percorrer o array com um laço e exibir os elementos, começando do índice **0**, assim:

```
for (let i = 0; i < daysOfWeek.length; i++) {  
    console.log(daysOfWeek[i]);  
}
```

Vamos analisar outro exemplo. Suponha que queremos descobrir quais são os 20 primeiros números da sequência de Fibonacci. Os dois primeiros números da sequência são **1** e **2**, e cada número subsequente é a soma dos dois números anteriores:

```
const fibonacci = [] // {1}  
fibonacci[1] = 1 // {2}  
fibonacci[2] = 1 // {3}  
for (let i = 3; i < 20; i++) {  
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2]; // // {4}  
}  
for (let i = 1; i < fibonacci.length; i++) { // {5}  
    console.log(fibonacci[i]); // {6}  
}
```

A seguir, apresentamos a explicação para o código anterior:

1. Na linha **{1}**, declaramos e criamos um array.
2. Nas linhas **{2}** e **{3}**, atribuímos os dois primeiros números da sequência de Fibonacci à segunda e à terceira posições do array (em JavaScript, a primeira posição do array é sempre referenciada com **0** (zero), mas, como não há **0** na sequência de Fibonacci, vamos ignorá-la).
3. Em seguida, tudo que temos de fazer é criar do terceiro ao vigésimo número da sequência (pois já sabemos quais são os dois primeiros números). Para isso, podemos usar um laço e atribuir a soma das duas posições anteriores do array à posição atual (linha **{4}**, começando pelo índice **3** do array, até o décimo nono índice).
4. Então, para ver a saída (linha **{6}**), basta percorrer o array com um laço, partindo da primeira posição até o seu tamanho (linha **{5}**).

Podemos usar `console.log` para exibir cada índice do array (linhas **{5}** e **{6}**), ou podemos também utilizar `console.log(fibonacci)` para exibir o próprio array. A maioria dos navegadores tem uma boa representação

de array em `console.log`.

Se você quiser gerar mais de 20 números da sequência de Fibonacci, basta alterar o número **20** para qualquer número desejado.

Acrescentando elementos

Acrescentar e remover elementos de um array não é tão difícil; contudo, essas operações podem ter suas sutilezas. Nos exemplos que usaremos nesta seção, vamos considerar que temos o array de números a seguir, inicializado com os números de **0** a **9**:

```
let numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

Inserindo um elemento no final do array

Se quisermos acrescentar um novo elemento nesse array (por exemplo, o número **10**), tudo que precisamos fazer é referenciar a última posição livre do array e atribuir-lhe um valor:

```
numbers[numbers.length] = 10;
```

Em JavaScript, um array é um objeto mutável. Podemos facilmente lhe acrescentar novos elementos. O objeto crescerá dinamicamente à medida que novos elementos forem adicionados. Em várias outras linguagens, por exemplo, em C e em Java, é preciso determinar o tamanho do array e, caso haja necessidade de adicionar mais elementos, um array totalmente novo deverá ser criado; não podemos simplesmente adicionar novos elementos ao array à medida que forem necessários.

Usando o método push

A API de JavaScript também tem um método chamado **push**, que nos permite acrescentar novos elementos no final de um array. Podemos acrescentar quantos elementos quisermos como argumentos do método **push**:

```
numbers.push(11);
numbers.push(12, 13);
```

O array **numbers** exibirá os números de **0** a **13** como saída.

Inserindo um elemento na primeira posição

Vamos agora supor que precisamos acrescentar um novo elemento ao array (número `-1`) e que gostaríamos de inseri-lo na primeira posição, e não na última. Para isso, inicialmente devemos deixar a primeira posição livre, deslocando todos os elementos para a direita. Podemos percorrer todos os elementos do array com um laço, começando pela última posição (o valor de `length` será o final do array), deslocando o elemento anterior (`i - 1`) para a nova posição (`i`) e, por fim, fazendo a atribuição do novo valor desejado à primeira posição (índice `0`). Podemos criar uma função para representar essa lógica e até mesmo adicionar um novo método diretamente no `Array prototype`, deixando o método `insertFirstPosition` disponível a todas as instâncias de array. O código a seguir representa a lógica descrita nesse caso:

```
Array.prototype.insertFirstPosition = function(value) {  
    for (let i = this.length; i >= 0; i--) {  
        this[i] = this[i - 1];  
    }  
    this[0] = value;  
};  
numbers.insertFirstPosition(-1);
```

Podemos representar essa ação por meio do seguinte diagrama (Figura 3.2):

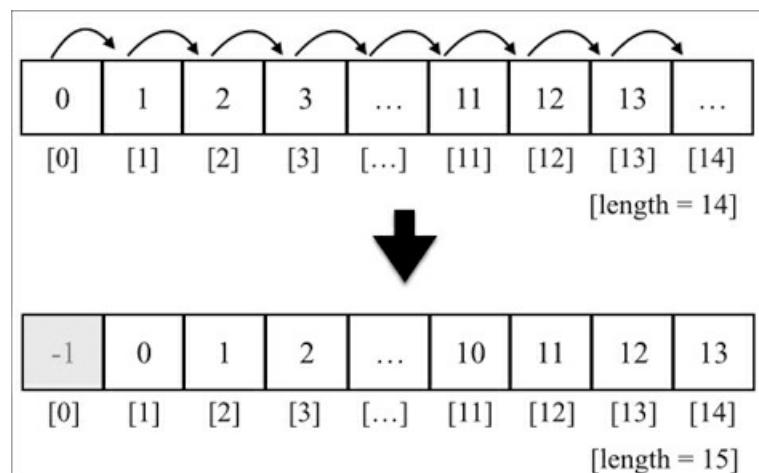


Figura 3.2

Usando o método `unshift`

A classe `array` de JavaScript também tem um método chamado `unshift`, que insere no início do array os valores passados como argumentos para o método (a lógica interna tem o mesmo comportamento do método `insertFirstPosition`):

```
numbers.unshift(-2);
numbers.unshift(-4, -3);
```

Assim, ao usar o método `unshift`, podemos adicionar o valor `-2` e, em seguida, `-3` e `-4` no início do array `numbers`. Esse array exibirá os números de `-4` a `13` como saída.

Removendo elementos

Até agora, vimos como adicionar elementos no array. Agora, vamos analisar a remoção de um valor de um array.

Removendo um elemento do final do array

Para remover um valor do final de um array, podemos utilizar o método `pop`:

```
numbers.pop();
```

Os métodos `push` e `pop` permitem que um array emule uma estrutura de dados `stack` básica, que será o assunto do próximo capítulo.

Nosso array exibirá os números de `-4` a `12` como saída. O tamanho de nosso array é `17`.

Removendo um elemento da primeira posição

Para remover um valor do início do array, podemos usar o código a seguir:

```
for (let i = 0; i < numbers.length; i++) {
  numbers[i] = numbers[i + 1];
}
```

Podemos representar o código anterior por meio do diagrama (Figura 3.3).

Deslocamos todos os elementos de uma posição para a esquerda. Entretanto, o tamanho (`length`) do array continua igual (`17`), o que significa que ainda temos um elemento extra em nosso array (com um valor `undefined`). Na última vez que o código do laço foi executado, `i+1` era

uma referência a uma posição inexistente. Em algumas linguagens, como Java, C/C++ ou C#, o código lançaria uma exceção, e seria necessário encerrar o nosso laço em `numbers.length - 1`.

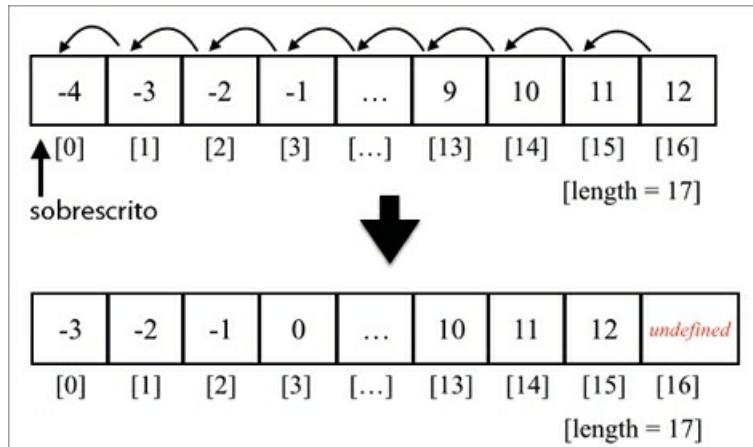


Figura 3.3

Simplesmente sobrescrevemos os valores originais do array, sem de fato tê-los removido (pois o tamanho do array continua o mesmo e temos esse elemento extra `undefined`).

Para remover o valor do array, podemos também criar um método `removeFirstPosition` com a lógica descrita nesta seção. No entanto, para realmente remover o elemento do array, precisamos criar outro array e copiar todos os valores diferentes de `undefined` do array original para o novo array e atribuí-lo ao nosso array. Para isso, podemos também criar um método `reIndex`, assim:

```
Array.prototype.reIndex = function(myArray) {
  const newArray = [];
  for(let i = 0; i < myArray.length; i++ ) {
    if (myArray[i] !== undefined) {
      // console.log(myArray[i]);
      newArray.push(myArray[i]);
    }
  }
  return newArray;
}
// remove a primeira posição manualmente e executa reIndex
Array.prototype.removeFirstPosition = function() {
  for (let i = 0; i < this.length; i++) {
    this[i] = this[i + 1];
  }
  return this.reIndex(this);
}
```

```
};  
numbers = numbers.removeFirstPosition();
```

O código anterior deve ser usado somente com finalidades pedagógicas, não devendo ser utilizado em projetos de verdade. Para remover o primeiro elemento de um array, devemos sempre usar o método `shift`, que será apresentado na próxima seção.

Usando o método `shift`

Para remover um elemento do início do array, use o método `shift`, deste modo:

```
numbers.shift();
```

Se considerarmos que o nosso array tem os valores de -4 a 12 e um tamanho igual a 17, após a execução do código anterior, ele conterá os valores de -3 a 12 e terá um tamanho igual a 16.

Os métodos `shift` e `unshift` permitem que um array emule uma estrutura de dados básica de fila (queue), que será o assunto do Capítulo 5, *Filas e deque*s.

Adicionando e removendo elementos de uma posição específica

Até agora, vimos como acrescentar elementos no final e no início de um array, e vimos também como remover elementos dessas posições. Mas e se quiséssemos também acrescentar ou remover elementos de qualquer posição de nosso array? Como poderíamos fazer isso?

O método `splice` pode ser usado para remover um elemento de um array, simplesmente especificando a posição/índice a partir do qual queremos fazer a remoção e a quantidade de elementos que gostaríamos de remover, assim:

```
numbers.splice(5,3);
```

Esse código removerá três elementos a partir do índice 5 de nosso array. Isso significa que `numbers[5]`, `numbers[6]` e `numbers[7]` serão removidos do array `numbers`. O conteúdo de nosso array será -3, -2, -1, 0, -1, 5, 6, 7, 8, 9, 10, 11 e 12 (pois os números -2, 3 e 4 foram removidos).

Assim como em arrays e objetos JavaScript, o operador `delete` também pode ser usado para remover um elemento de um array, por exemplo, `delete numbers[0]`. No entanto, a posição 0 do array terá o valor `undefined`, ou seja, será o mesmo que executar `numbers[0] = undefined`, e teríamos de reindexar o array. Por esse motivo, devemos sempre usar os métodos `splice`, `pop` ou `shift` para remover elementos.

Vamos agora supor que queremos inserir os números de 2 a 4 de volta no array a partir da posição 5. Podemos usar novamente o método `splice` para isso:

```
numbers.splice(5, 0, 2, 3, 4);
```

O primeiro argumento do método é o índice a partir do qual queremos remover ou inserir elementos. O segundo argumento é a quantidade de elementos que queremos remover (nesse caso, não queremos remover nenhum, portanto passamos o valor 0 (zero)). Do terceiro argumento em diante, temos os valores que gostaríamos de inserir no array (os elementos 2, 3 e 4). Os valores de -3 a 12 serão novamente exibidos na saída.

Por fim, execute o código a seguir:

```
numbers.splice(5, 3, 2, 3, 4);
```

Os valores de -3 a 12 serão exibidos na saída. Isso ocorre porque removemos três elementos a partir do índice 5, mas também acrescentamos os elementos 2, 3 e 4 a partir do índice 5.

Arrays bidimensionais e multidimensionais

No início deste capítulo, usamos o exemplo das medidas de temperatura. Vamos agora usar esse exemplo mais uma vez. Suponha que precisamos medir a temperatura de hora em hora durante alguns dias. Agora que já sabemos que é possível usar um array para armazenar as temperaturas, podemos facilmente escrever o código a seguir a fim de armazenar as temperaturas no decorrer de dois dias:

```
let averageTempDay1 = [72, 75, 79, 79, 81, 81];
let averageTempDay2 = [81, 79, 75, 75, 73, 72];
```

No entanto, essa não é a melhor abordagem; podemos ter algo mais apropriado! Uma **matriz** (um array bidimensional ou um *array de arrays*) pode ser usada para armazenar essas informações, na qual cada linha

representará o dia e cada coluna representará a medida da temperatura de hora em hora, assim:

```
let averageTemp = [];
averageTemp[0] = [72, 75, 79, 79, 81, 81];
averageTemp[1] = [81, 79, 75, 75, 73, 73];
```

A linguagem JavaScript aceita apenas arrays unidimensionais; ela não tem suporte para matrizes. Contudo, podemos implementá-las, ou implementar qualquer array multidimensional, usando array de arrays, como no código anterior. O mesmo código também pode ser escrito assim:

```
// dia 1
averageTemp[0] = [];
averageTemp[0][0] = 72;
averageTemp[0][1] = 75;
averageTemp[0][2] = 79;
averageTemp[0][3] = 79;
averageTemp[0][4] = 81;
averageTemp[0][5] = 81;
// dia 2
averageTemp[1] = [];
averageTemp[1][0] = 81;
averageTemp[1][1] = 79;
averageTemp[1][2] = 75;
averageTemp[1][3] = 75;
averageTemp[1][4] = 73;
averageTemp[1][5] = 73;
```

No código anterior, especificamos o valor de cada dia e de cada hora separadamente. Também podemos representar esse array bidimensional com o diagrama a seguir (Figura 3.4):

| | [0] | [1] | [2] | [3] | [4] | [5] |
|-----|-----|-----|-----|-----|-----|-----|
| [0] | 72 | 75 | 79 | 79 | 81 | 81 |
| [1] | 81 | 79 | 75 | 75 | 73 | 73 |

Figura 3.4

Cada linha representa um dia e cada coluna representa a temperatura de cada hora do dia.

Iterando pelos elementos de arrays bidimensionais

Se quisermos ver a saída da matriz, podemos criar uma função genérica

para fazer o log:

```
function printMatrix(myMatrix) {  
    for (let i = 0; i < myMatrix.length; i++) {  
        for (let j = 0; j < myMatrix[i].length; j++) {  
            console.log(myMatrix[i][j]);  
        }  
    }  
}
```

É necessário percorrer todas as linhas e colunas com um laço. Para isso, devemos usar um laço **for** aninhado, em que a variável **i** representa as linhas e **j** representa as colunas. Nesse caso, cada **myMatrix[i]** também representa um array e, desse modo, também precisamos iterar pelas posições dela no laço **for** aninhado.

Podemos exibir o conteúdo da matriz **averageTemp** com o código a seguir:

```
printMatrix(averageTemp);
```

Para exibir um array bidimensional no console do navegador, podemos usar também a instrução **console.table(averageTemp)**. Com ela, teremos uma saída mais elegante para o usuário.

Arrays multidimensionais

É possível também trabalhar com arrays multidimensionais em JavaScript. Por exemplo, vamos criar uma matriz 3 x 3. Cada célula contém a soma **i** (linha) + **j** (coluna) + **z** (profundidade) da matriz, deste modo:

```
const matrix3x3x3 = [];  
for (let i = 0; i < 3; i++) {  
    matrix3x3x3[i] = []; // precisamos inicializar cada array  
    for (let j = 0; j < 3; j++) {  
        matrix3x3x3[i][j] = [];  
        for (let z = 0; z < 3; z++) {  
            matrix3x3x3[i][j][z] = i + j + z;  
        }  
    }  
}
```

Não importa quantas dimensões temos na estrutura de dados; precisamos percorrer cada dimensão com um laço a fim de acessar a célula. Podemos representar uma matriz 3 x 3 x 3 com um diagrama em forma de cubo, assim (Figura 3.5).

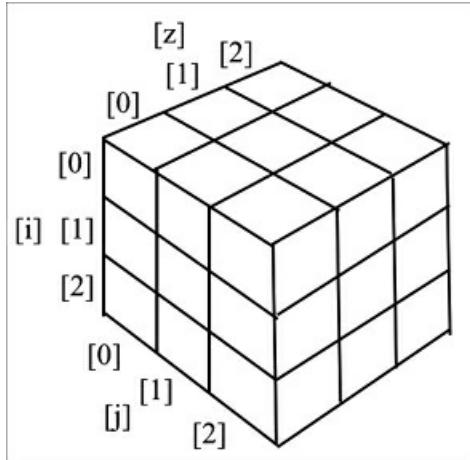


Figura 3.5

Para exibir o conteúdo dessa matriz, o código a seguir pode ser usado:

```
for (let i = 0; i < matrix3x3x3.length; i++) {
    for (let j = 0; j < matrix3x3x3[i].length; j++) {
        for (let z = 0; z < matrix3x3x3[i][j].length; z++) {
            console.log(matrix3x3x3[i][j][z]);
        }
    }
}
```

Se tivéssemos uma matriz $3 \times 3 \times 3 \times 3$, teríamos quatro instruções **for** aninhadas em nosso código, e assim sucessivamente. Raramente você precisará de um array de quatro dimensões em sua carreira de desenvolvedor. Os arrays bidimensionais são os mais comuns.

Referências para métodos de array em JavaScript

Os arrays em JavaScript são objetos modificados, o que significa que todo array que criarmos terá alguns métodos disponíveis para uso. Os arrays em JavaScript são bem interessantes, pois são muito eficazes e têm mais recursos disponíveis que os arrays primitivos em outras linguagens. Isso quer dizer que não precisaremos escrever funcionalidades básicas por conta própria, como acrescentar e remover elementos no/do meio da estrutura de dados.

A seguir, apresentamos uma lista dos métodos essenciais disponíveis em um objeto array. Já discutimos alguns métodos anteriormente:

| Método | Descrição |
|---------------------|------------------------------------------------------------------|
| <code>concat</code> | Junta vários arrays e devolve uma cópia dos arrays concatenados. |
| <code>every</code> | |

| | |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| | Itera por todos os elementos do array, verificando uma condição desejada (função) até que <code>false</code> seja devolvido. |
| <code>filter</code> | Cria um array com todos os elementos avaliados com <code>true</code> pela função especificada. |
| <code>forEach</code> | Executa uma função específica em cada elemento do array. |
| <code>join</code> | Reúne todos os elementos do array em uma string. |
| <code>indexOf</code> | Pesquisa o array em busca de elementos específicos e devolve a sua posição. |
| <code>lastIndexOf</code> | Devolve a posição do último item do array que corresponda ao critério de pesquisa. |
| <code>map</code> | Cria outro array a partir de uma função que contém o critério/condição e devolve os elementos do array que correspondam ao critério. |
| <code>reverse</code> | Inverte o array, de modo que o último item se torne o primeiro, e vice-versa. |
| <code>slice</code> | Devolve um novo array a partir do índice especificado. |
| <code>some</code> | Itera por todos os elementos do array, verificando a condição desejada (função) até que <code>true</code> seja devolvido. |
| <code>sort</code> | Organiza o array em ordem alfabética ou de acordo com a função especificada. |
| <code>toString</code> | Devolve o array na forma de uma string. |
| <code>valueOf</code> | É semelhante ao método <code>toString</code> e devolve o array na forma de uma string. |

Já discutimos os métodos `push`, `pop`, `shift`, `unshift` e `splice`. Vamos dar uma olhada nos novos métodos, os quais serão muito úteis nos capítulos subsequentes deste livro, quando implementaremos as nossas próprias estruturas de dados e algoritmos. Alguns desses métodos são muito úteis quando trabalhamos com **programação funcional**, que será discutida no Capítulo 14, *Designs de algoritmos e técnicas*.

Juntando vários arrays

Considere um cenário em que você tenha arrays diferentes e precise juntar todos eles em um único array. Poderíamos fazer uma iteração em cada array e acrescentar cada um dos elementos no array final. Felizmente, a linguagem JavaScript já tem um método chamado `concat` capaz de fazer isso para nós; esse método pode ser usado assim:

```
const zero = 0;
const positiveNumbers = [1, 2, 3];
const negativeNumbers = [-3, -2, -1];
let numbers = negativeNumbers.concat(zero, positiveNumbers);
```

Podemos passar quantos arrays e objetos/elementos quisermos para esse array. Os arrays serão concatenados no array especificado na ordem em que os argumentos forem passados para o método. Nesse exemplo, `zero` será concatenado a `negativeNumbers`, e então `positiveNumbers` será concatenado no array resultante.

O array `numbers` exibirá os valores `-3, -2, -1, 0, -1, 2 e 3` como saída.

Funções de iteração

Às vezes, precisamos iterar pelos elementos de um array. Vimos que um laço pode ser usado para isso, por exemplo, a instrução `for`, conforme mostraram alguns exemplos anteriores.

A linguagem JavaScript tem alguns métodos de iteração embutidos, que podem ser usados com arrays. Nos exemplos desta seção, precisaremos de um array e de uma função. Usaremos um array com os valores de `1` a `15`, além de uma função que devolverá `true` se o número for múltiplo de `2` (par), e `false` caso contrário.

Esse código é mostrado a seguir:

```
function isEven(x) {  
    // Devolve true se x for múltiplo de 2.  
    console.log(x);  
    return x % 2 === 0 ? true : false;  
}  
  
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];  
  
return (x % 2 == 0) ? true : false também pode ser representado por return  
(x % 2 == 0).
```

Para simplificar o nosso código, em vez de declarar funções usando a sintaxe `ES5`, usaremos a sintaxe `ES2015 (ES6)`, conforme vimos no Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*. Podemos reescrever a função `isEven` usando **funções de seta**:

```
const isEven = x => x % 2 === 0;
```

Iterando com o método `every`

O primeiro método que analisaremos é o método `every`, o qual itera pelos elementos do array até que a função devolva `false`, assim:

```
numbers.every(isEven);
```

Nesse caso, o nosso primeiro elemento do array `numbers` é o número `1`. O número `1` não é múltiplo de `2` (é um número ímpar), portanto a função `isEven` devolverá `false`, e essa será a única vez que a função será executada.

Iterando com o método some

A seguir, temos o método `some`, que apresenta o comportamento oposto ao método `every`; no entanto, o método `some` itera pelos elementos do array até que a função devolva `true`:

```
numbers.some(isEven);
```

Em nosso caso, o primeiro número par em nosso array `numbers` é 2 (o segundo elemento). O primeiro elemento da iteração é o número 1; ele devolverá `false`. Então, o segundo elemento da iteração será o número 2, que devolverá `true`, e a iteração será interrompida.

Iterando com forEach

Se precisarmos fazer a iteração em todo o array, independentemente de tudo mais, podemos usar a função `forEach`. O resultado será o mesmo que usar um laço `for` com o código da função dentro dele, assim:

```
numbers.forEach(x => console.log(x % 2 === 0));
```

Usando map e filter

A linguagem JavaScript também tem outros dois métodos iteradores que devolvem um novo array com um resultado. O primeiro é o método `map`, que vemos a seguir:

```
const myMap = numbers.map(isEven);
```

O array `myMap` terá os seguintes valores: `[false, true, false, true, false, true, false, true, false, true, false, true, false, true, false]`. Ele armazena os resultados da função `isEven`, passada para o método `map`. Desse modo, podemos facilmente saber se um número é par ou não. Por exemplo, `myMap[0]` devolve `false` porque 1 não é par, e `myMap[1]` devolve `true` porque 2 é par.

Também temos o método `filter`, o qual devolve um novo array com os elementos para os quais a função devolveu `true`, assim:

```
const evenNumbers = numbers.filter(isEven);
```

Em nosso caso, o array `evenNumbers` conterá os elementos que são múltiplos de 2: `[2, 4, 6, 8, 10, 12, 14]`.

Usando o método reduce

Por fim, temos o método `reduce`, que recebe uma função com os seguintes parâmetros: `previousValue`, `currentValue`, `index` e `array`. Os parâmetros `index` e `array` são opcionais, portanto não é necessário passá-los caso não sejam usados. Podemos usar essa função para devolver um valor que será somado a um acumulador, o qual será devolvido depois que o método `reduce` parar de executar. Isso pode ser muito útil se quisermos somar todos os valores de um array. Veja um exemplo:

```
numbers.reduce((previous, current) => previous + current);
```

A saída será igual a 120.

Esse trecho de código demonstra o uso do método `reduce` para somar todos os elementos de um array. A saída é 120.

Esse trecho de código demonstra o uso do método `reduce` para somar todos os elementos de um array. A saída é 120.

ECMAScript 6 e as novas funcionalidades de array

Como vimos no Capítulo 1, *JavaScript – uma visão geral rápida*, a linguagem JavaScript tem novas funcionalidades baseadas nas especificações da **ECMAScript 2015** (ES6 ou ES2015) ou em especificações mais recentes (2015+).

A seguir, apresentamos uma lista dos novos métodos adicionados na ES2015 e na ES2016:

| Método | Descrição |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>@@iterator</code> | Devolve um objeto iterador que contém os pares chave/valor do array; pode ser chamado sincronamente para obter a chave/valor dos elementos do array. |
| <code>copyWithin</code> | Copia uma sequência de valores do array na posição de um índice de início. |
| <code>entries</code> | Devolve <code>@@iterator</code> , que contém pares chave/valor. |
| <code>includes</code> | Devolve <code>true</code> caso um elemento seja encontrado no array, e <code>false</code> caso contrário. Foi adicionado na ES2016. |
| <code>find</code> | Busca um elemento no array, dada uma condição desejada (função de callback), e devolve o elemento caso seja encontrado. |
| <code>findIndex</code> | Busca um elemento no array, dada uma condição desejada (função de callback), e devolve o índice do elemento caso seja encontrado. |
| <code>fill</code> | Preenche o array com um valor estático. |
| <code>from</code> | Cria um novo array a partir de um array existente. |
| <code>keys</code> | Devolve <code>@@iterator</code> , contendo as chaves do array. |
| <code>of</code> | Cria um novo array a partir dos argumentos passados para o método. |

| | |
|---------------------|-----------------------------------------------------------------|
| <code>values</code> | Devolve <code>@@iterator</code> , contendo os valores do array. |
|---------------------|-----------------------------------------------------------------|

Junto com esses métodos, a API de Array também provê uma forma de iterar pelo array com o objeto `Iterator`, que pode ser obtido da instância do array e usado no laço `for...of`.

Iterando com o laço `for...of`

Vimos que podemos iterar por um array usando o laço `for` e o método `forEach`.

A ES2015 introduziu o laço `for...of` para iterar pelos valores de um array. Podemos observar um exemplo de uso do laço `for...of` a seguir:

```
for (const n of numbers) {
  console.log(n % 2 === 0 ? 'even' : 'odd');
}
```

Usando o objeto `@@iterator`

A classe `Array` também tem uma propriedade chamada `@@iterator`, introduzida na ES2015. Para usá-la, é necessário acessar a propriedade `Symbol.iterator` do array, assim:

```
let iterator = numbers[Symbol.iterator]();
console.log(iterator.next().value); // 1
console.log(iterator.next().value); // 2
console.log(iterator.next().value); // 3
console.log(iterator.next().value); // 4
console.log(iterator.next().value); // 5
```

Então, podemos chamar individualmente o método `next` do iterador para obter o próximo valor do array. Para o array `numbers`, devemos chamar `iterator.next().value` 15 vezes, pois temos 15 valores no array.

Podemos apresentar todos os 15 valores do array `numbers` usando o código a seguir:

```
iterator = numbers[Symbol.iterator]();
for (const n of iterator) {
  console.log(n);
}
```

Quando fizermos a iteração pelo array e não houver mais valores para iterar, o código `iterator.next()` devolverá `undefined`.

Métodos entries, keys e values de array

A ES2015 também introduziu três formas de obter iteradores de um array. A primeira que veremos é o método **entries**.

O método **entries** devolve **@@iterator**, que contém pares chave/valor. A seguir, apresentamos um exemplo de uso desse método:

```
let aEntries = numbers.entries(); // obtém um iterador de chave/valor
console.log(aEntries.next().value); // [0, 1] - posição 0, valor 1
console.log(aEntries.next().value); // [1, 2] - posição 1, valor 2
console.log(aEntries.next().value); // [2, 3] - posição 2, valor 3
```

Como o array **numbers** contém somente números, **key** será a posição do array e **value** será o valor armazenado no índice do array.

Também podemos usar o código a seguir como uma alternativa ao código anterior:

```
aEntries = numbers.entries();
for (const n of aEntries) {
  console.log(n);
}
```

Ser capaz de obter pares chave/valor será muito conveniente quando estivermos trabalhando com conjuntos, dicionários e mapas de hash (hash maps). Essa funcionalidade será bastante conveniente para nós em capítulos mais adiante neste livro.

O método **keys** devolve **@@iterator**, que contém as chaves do array. A seguir, apresentamos um exemplo de uso desse método:

```
const aKeys = numbers.keys(); // obtém um iterador de chaves
console.log(aKeys.next()); // {value: 0, done: false }
console.log(aKeys.next()); // {value: 1, done: false }
console.log(aKeys.next()); // {value: 2, done: false }
```

Para o array **numbers**, **keys** conterá os índices do array. Quando não houver mais valores para iterar, o código **aKeys.next()** devolverá **undefined** como **value** e **true** como **true**. Se **done** tiver um valor igual a **false**, isso significa que ainda há mais chaves para iterar no array.

O método **values** devolve **@@iterator**, que contém os valores do array. A seguir, apresentamos um exemplo de uso desse método:

```
const aValues = numbers.values();
console.log(aValues.next()); // {value: 1, done: false }
console.log(aValues.next()); // {value: 2, done: false }
console.log(aValues.next()); // {value: 3, done: false }
```

É bom lembrar que nem todas as funcionalidades da ES2015 são tratadas pelos navegadores. Por esse motivo, a melhor maneira de testar esse código é com o Babel. Os exemplos podem ser executados em <https://goo.gl/eojEGk>.

Usando o método from

O método `Array.from` cria outro array a partir de um array existente. Por exemplo, se quisermos copiar o array `numbers` para um novo array, podemos usar o código a seguir:

```
let numbers2 = Array.from(numbers);
```

Também é possível passar uma função para que possamos determinar quais valores queremos mapear. Considere o código a seguir:

```
let evens = Array.from(numbers, x => (x % 2 == 0));
```

O código anterior criou outro array chamado `evens`, com um valor `true` se o número for par no array original, e `false` caso contrário.

Usando o método Array.of

O método `Array.of` cria outro array a partir dos argumentos passados para o método. Por exemplo, considere o exemplo a seguir:

```
let numbers3 = Array.of(1);
let numbers4 = Array.of(1,2,3,4,5,6);
```

O código anterior é o mesmo que executar este código:

```
let numbers3 = [1];
let numbers4 = [1,2,3,4,5,6];
```

Também podemos usar esse método para fazer uma cópia de um array existente. Eis um exemplo:

```
let numbersCopy = Array.of(...numbers4);
```

O código anterior é o mesmo que usar `Array.from(numbers4)`. A diferença, nesse caso, é que estamos usando o operador de espalhamento, que vimos no Capítulo 1, *JavaScript – uma visão geral rápida*. Esse operador (...) fará o espalhamento de cada um dos valores do array `numbers4` nos argumentos.

Usando o método fill

O método **fill** preenche o array com um valor. Por exemplo, considere o array a seguir:

```
let numbersCopy = Array.of(1,2,3,4,5,6);
```

O array **numbersCopy** tem tamanho **6**, o que significa que temos seis posições. Vamos usar o código a seguir:

```
numbersCopy.fill(0);
```

Nesse caso, o array **numbersCopy** terá todas as suas posições com um valor **0** (**[0,0,0,0,0,0]**). Também podemos passar o índice de início a partir do qual queremos preencher o array, assim:

```
numbersCopy.fill(2, 1);
```

No exemplo anterior, todas as posições do array terão o valor **2**, a partir da posição **1** (**[0,2,2,2,2,2]**).

Também é possível passar o índice final até o qual queremos preencher o array:

```
numbersCopy.fill(1, 3, 5);
```

No exemplo anterior, preencheremos o array com o valor **1**, do índice **3** ao índice **5** (não inclusivo), resultando no seguinte array: **[0,2,2,1,1,2]**.

O método **fill** é ótimo se quisermos criar um array e inicializar seus valores, como vemos a seguir:

```
let ones = Array(6).fill(1);
```

O código anterior criará um array de tamanho **6** e todos os seus valores serão iguais a **1** (**[1,1,1,1,1,1]**).

Usando o método **copyWithin**

O método **copyWithin** copia uma sequência de valores do array para a posição de um índice de início. Por exemplo, considere o array a seguir:

```
let copyArray = [1, 2, 3, 4, 5, 6];
```

Vamos supor agora que queremos copiar os valores **4**, **5** e **6** para as primeiras três posições do array, resultando no array **[4,5,6,4,5,6]**. Podemos usar o código a seguir para obter esse resultado:

```
copyArray.copyWithin(0, 3);
```

Considere agora que queremos copiar os valores **4** e **5** (as posições **3** e **4**) para as posições **1** e **2**. Podemos usar o seguinte código para isso:

```
copyArray = [1, 2, 3, 4, 5, 6];
```

```
copyArray.copyWithin(1, 3, 5);
```

Nesse caso, copiaremos os elementos, partindo da posição 3 até a posição 5 (não inclusivo), para a posição 1 do array, resultando no array [1,4,5,4,5,6].

Ordenando elementos

Neste livro, veremos como implementar os algoritmos de pesquisa e ordenação mais usados. No entanto, o JavaScript também tem um método de ordenação e dois métodos de pesquisa disponíveis. Vamos analisá-los.

Em primeiro lugar, vamos usar o nosso array **numbers** e deixar os elementos fora de ordem (1, 2, 3, ... 15 já estão ordenados). Para isso, o método **reverse** pode ser aplicado: o último item será o primeiro e vice-versa, deste modo:

```
numbers.reverse();
```

Agora, portanto, o array **numbers** terá [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1] como saída. Em seguida, podemos aplicar o método **sort**:

```
numbers.sort();
```

No entanto, se exibirmos o array, o resultado será [1, 10, 11, 12, 13, 14, 15, 2, 3, 4, 5, 6, 7, 8, 9]. Ele não está ordenado corretamente. Isso ocorre porque o método **sort** deixa os elementos em ordem lexicográfica e pressupõe que todos os elementos são strings.

Podemos também implementar a nossa própria função de comparação. Como o nosso array tem elementos numéricos, podemos escrever o código a seguir:

```
numbers.sort((a, b) => a - b);
```

Esse código devolverá um número negativo se **b** for maior que **a**, um número positivo se **a** for maior que **b** e 0 (zero) se forem iguais. Isso significa que, se um valor negativo for devolvido, é sinal de que **a** é menor que **b**, o que será usado posteriormente pela função **sort** para organizar os elementos.

O código anterior também pode ser representado pelo código a seguir:

```
function compare(a, b) {
  if (a < b) {
    return -1;
```

```

    }
    if (a > b) {
        return 1;
    }
    // a deve ser igual a b
    return 0;
}
numbers.sort(compare);

```

Isso ocorre porque a função `sort` da classe `Array` de JavaScript pode receber um parâmetro chamado `compareFunction`, responsável pela ordenação do array. Em nosso exemplo, declaramos uma função que será responsável por comparar os elementos do array, resultando em um array em ordem crescente.

Ordenação personalizada

Podemos ordenar um array que contenha qualquer tipo de objeto, e podemos também criar uma `compareFunction` para comparar os elementos de acordo com o que for necessário. Por exemplo, suponha que haja um objeto `Person` com `name` e `age` e queremos ordenar o array de acordo com a idade (`age`) da pessoa. Podemos usar o código a seguir:

```

const friends = [
    { name: 'John', age: 30 },
    { name: 'Ana', age: 20 },
    { name: 'Chris', age: 25 }, // vírgula no final: ES2017
];
function comparePerson(a, b) {
    if (a.age < b.age) {
        return -1;
    }
    if (a.age > b.age) {
        return 1;
    }
    return 0;
}
console.log(friends.sort(comparePerson));

```

Nesse caso, a saída do código anterior será: `Ana (20), Chris (25) e John (30)`.

Ordenando strings

Suponha que tenhamos o seguinte array:

```
let names = ['Ana', 'ana', 'john', 'John'];
console.log(names.sort());
```

Como você imagina que será a saída? Eis a resposta:

```
["Ana", "John", "ana", "john"]
```

Por que **ana** vem depois de **John** se “a” vem antes no alfabeto? Isso ocorre porque o JavaScript compara os caracteres de acordo com o seu valor **ASCII**. Por exemplo, A, J, a e j têm os seguintes valores ASCII decimais: A: 65, J: 74, a: 97 e j: 106.

Portanto, J tem um valor menor que a, e, por esse motivo, vem antes no alfabeto.

Para obter mais informações sobre a tabela ASCII, acesse <http://www.asciitable.com>.

Porém, se passarmos uma **compareFunction** contendo o código para ignorar a diferença entre letras maiúsculas e minúsculas, a saída obtida será **["Ana", "ana", "john", "John"]**, assim:

```
names = ['Ana', 'ana', 'john', 'John']; // inicia o array com o estado original
console.log(names.sort((a, b) => {
  if (a.toLowerCase() < b.toLowerCase()) {
    return -1;
  }
  if (a.toLowerCase() > b.toLowerCase()) {
    return 1;
  }
  return 0;
}));
```

Nesse caso, a função **sort** não terá efeito algum; ela obedecerá à ordem atual das letras minúsculas e maiúsculas.

Se quisermos que as letras minúsculas venham antes no array ordenado, é preciso usar o método **localeCompare**:

```
names.sort((a, b) => a.localeCompare(b));
```

O resultado será **["ana", "Ana", "john", "John"]**.

Para caracteres com acento, podemos usar o método **localeCompare** também:

```
const names2 = ['Maève', 'Maeve'];
console.log(names2.sort((a, b) => a.localeCompare(b)));
```

A saída será `["Maeve", "Maève"]`.

Pesquisa

Temos duas opções para pesquisa: o método `indexOf`, que devolve o índice do primeiro elemento correspondente ao argumento passado, e `lastIndexOf`, que devolve o índice do último elemento encontrado, correspondente ao argumento passado. Vamos retomar o array `numbers` que estávamos usando antes:

```
console.log(numbers.indexOf(10));
console.log(numbers.indexOf(100));
```

No exemplo anterior, a saída no console será `9` para a primeira linha e `-1` (porque o valor não existe em nosso array) para a segunda linha. Um resultado semelhante pode ser obtido com o código a seguir:

```
numbers.push(10);
console.log(numbers.lastIndexOf(10));
console.log(numbers.lastIndexOf(100));
```

Adicionamos um novo elemento com o valor `10`, portanto a segunda linha exibirá `15` (o nosso array agora tem valores de `1` a `15` e `10`), e a terceira linha exibirá `-1` (porque o elemento `100` não existe em nosso array).

ECMAScript 2015 – os métodos `find` e `findIndex`

Considere o exemplo a seguir:

```
let numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15];
function multipleOf13(element, index, array) {
  return (element % 13 == 0);
}
console.log(numbers.find(multipleOf13));
console.log(numbers.findIndex(multipleOf13));
```

Os métodos `find` e `findIndex` recebem uma função de callback, a qual buscará um valor que satisfaça a condição presente na função de teste (callback). Nesse exemplo, estamos verificando se o array `numbers` contém algum múltiplo de `13`.

A diferença entre `find` e `findIndex` é que o método `find` devolve o primeiro valor do array que satisfaça a condição proposta. O método `findIndex`, por outro lado, devolve o índice do primeiro valor do array que satisfaça a condição. Caso o valor não seja encontrado, `undefined` será

devolvido.

ECMAScript 2016 – usando o método `includes`

O método `includes` devolve `true` caso um elemento seja encontrado no array, e `false` caso contrário. A seguir, apresentamos um exemplo de uso desse método:

```
console.log(numbers.includes(15));
console.log(numbers.includes(20));
```

Nesse exemplo, `includes(15)` devolverá `true` e `includes(20)` devolverá `false`, pois o elemento `20` não existe no array `numbers`.

Também é possível passar um índice de início a partir do qual queremos que o array faça a pesquisa do valor:

```
let numbers2 = [7,6,5,4,3,2,1];
console.log(numbers2.includes(4,5));
```

A saída do exemplo anterior será `false` porque o elemento `4` não existe após a posição `5`.

Convertendo um array em uma string

Por fim, chegamos aos dois últimos métodos: `toString` e `join`.

Se quisermos exibir todos os elementos do array em uma única string, podemos usar o método `toString`, assim:

```
console.log(numbers.toString());
```

Esse código exibirá os valores `1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15` e `10` no console.

Se quisermos separar os elementos com um separador diferente, por exemplo, `-`, o método `join` poderá ser usado para fazer exatamente isso, assim:

```
const numbersString = numbers.join('-');
console.log(numbersString);
```

Eis a saída:

```
1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-10
```

Isso pode ser útil se precisarmos enviar o conteúdo do array para um servidor ou se esse conteúdo tiver de ser decodificado (conhecendo o separador, a decodificação será mais fácil).

Há alguns recursos ótimos que você poderá usar para ampliar o seu conhecimento sobre arrays e seus métodos. O Mozilla também tem uma página excelente sobre arrays e seus métodos, com ótimos exemplos em https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array (<http://goo.gl/vu1diT>). A biblioteca Lo-Dash é igualmente muito útil quando trabalhamos com arrays em projetos JavaScript: <http://lodash.com>.

Classe TypedArray

Podemos armazenar qualquer tipo de dado em arrays JavaScript. Isso se deve ao fato de os arrays JavaScript não serem fortemente tipados como em outras linguagens como C e Java.

TypedArray foi criado para que pudéssemos trabalhar com arrays contendo um único tipo de dado. A sua sintaxe é `let myArray = new TypedArray(length)`, em que **TypedArray** deve ser substituído por uma classe **TypedArray**, conforme especificado na tabela a seguir:

| TypedArray | Descrição |
|-------------------|----------------------------------------------------------|
| Int8Array | Inteiro de 8 bits com sinal, usando complemento de dois |
| Uint8Array | Inteiro de 8 bits sem sinal |
| Uint8ClampedArray | Inteiro de 8 bits sem sinal |
| Int16Array | Inteiro de 16-bits com sinal, usando complemento de dois |
| Uint16Array | Inteiro de 16 bits sem sinal |
| Int32Array | Inteiro de 32-bits com sinal, usando complemento de dois |
| Uint32Array | Inteiro de 32 bits sem sinal |
| Float32Array | Número de ponto flutuante padrão IEEE com 32 bits |
| Float64Array | Número de ponto flutuante padrão IEEE com 64 bits |

Eis um exemplo:

```
let length = 5;
let int16 = new Int16Array(length);
let array16 = [];
array16.length = length;
for (let i=0; i<length; i++){
  int16[i] = i+1;
}
console.log(int16);
```

Arrays tipados são ótimos para trabalhar com APIs WebGL, manipular bits e lidar com arquivos e imagens. Esses arrays funcionam exatamente como os arrays simples; os mesmos métodos e funcionalidades que vimos neste capítulo também podem ser usados.

No link a seguir, você encontrará um bom tutorial sobre o uso de arrays tipados para manipular dados binários e suas aplicações em projetos do mundo real: <http://goo.gl/kZBsGx>.

Arrays em TypeScript

Todo o código-fonte deste capítulo é um código TypeScript válido. A diferença é que o TypeScript fará verificação de tipos em tempo de compilação para garantir que estejamos manipulando somente arrays nos quais todos os valores tenham o mesmo tipo de dado.

Se observarmos o código anterior, veremos que é o mesmo array **numbers**, conforme declaramos nas seções anteriores deste capítulo:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

Por causa da inferência de tipo, o TypeScript entende que a declaração do array **numbers** é igual a `const numbers: number[]`. Por esse motivo, não precisamos sempre declarar explicitamente o tipo da variável se ela for inicializada em sua declaração.

Se retomarmos o exemplo de ordenação do array **friends**, poderemos refatorar o código para o TypeScript a seguir:

```
interface Person {
  name: string;
  age: number;
}
// const friends: {name: string, age: number}[];
const friends = [
  { name: 'John', age: 30 },
  { name: 'Ana', age: 20 },
  { name: 'Chris', age: 25 }
];
function comparePerson(a: Person, b: Person) {
  // conteúdo da função comparePerson
}
```

Ao declarar a interface **Person**, garantimos que a função `comparePerson` receberá somente objetos que tenham **name** e **age**. O array **friends** não tem um tipo explícito; portanto, nesse caso, se quiséssemos, poderíamos

declarar explicitamente o seu tipo usando `const friends: Person[]`.

Em suma, se quisermos atribuir um tipo a nossas variáveis JavaScript usando TypeScript, basta usar `const` ou `let variableName: <type>[]`; por outro lado, se usarmos arquivos com uma extensão `.js`, poderemos também ter verificação de tipos adicionando o comentário `// @ts-check` na primeira linha do arquivo JavaScript, conforme vimos no Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*.

Em tempo de execução, a saída será exatamente a mesma que teríamos se estivéssemos usando JavaScript puro.

Resumo

Neste capítulo, descrevemos a estrutura de dados mais usada em programação: os arrays. Vimos como declarar, inicializar e atribuir valores, assim como acrescentar e remover elementos. Conhecemos os arrays bidimensionais e multidimensionais, bem como os principais métodos de um array, os quais serão muito úteis quando começarmos a criar os nossos próprios algoritmos em capítulos futuros.

Também conhecemos os novos métodos e funcionalidades acrescentados na classe `Array` nas especificações ECMAScript 2015 e 2016.

Por fim, vimos também como garantir que o array contenha somente valores do mesmo tipo usando TypeScript ou o seu recurso de verificação em tempo de compilação, para arquivos JavaScript.

No próximo capítulo, veremos as pilhas, que podem ser tratadas como arrays com um comportamento especial.

CAPÍTULO 4

Pilhas

No capítulo anterior, vimos como criar e usar arrays, que são o tipo mais comum de estrutura de dados em ciência da computação. Conforme aprendemos, é possível adicionar e remover elementos de um array em qualquer índice desejado. Às vezes, porém, precisaremos de alguma forma de estrutura de dados na qual tenhamos mais controle sobre o acréscimo e a remoção de itens. Há duas estruturas de dados que apresentam algumas semelhanças com os arrays, mas que nos oferecem mais controle sobre a adição e a remoção dos elementos. Essas estruturas de dados são as pilhas (stacks) e as **filas** (queues).

Os seguintes tópicos serão abordados neste capítulo:

- criação de nossa própria biblioteca de estrutura de dados JavaScript;
- a estrutura de dados de pilha;
- adição de elementos em uma pilha;
- remoção (pop) de elementos de uma pilha;
- como usar a classe **Stack**;
- o problema do decimal para binário.

Criação de uma biblioteca de estruturas de dados e algoritmos JavaScript

A partir deste capítulo, criaremos a nossa própria biblioteca de estruturas de dados e algoritmos JavaScript. O bundle com o código-fonte deste livro foi preparado para essa tarefa.

Depois de fazer o download do código-fonte, e com o Node.js instalado em seu computador conforme as instruções no Capítulo 1, *JavaScript – uma visão geral rápida*, vá para o diretório com a pasta do projeto e execute o comando `npm install`, como mostra a Figura 4.1:

```

loiane:development loiane$ cd javascript-datastructures-algorithms
loiane:javascript-datastructures-algorithms loiane$ npm install

> fsevents@1.1.2 install /Users/loiane/Documents/development/javascript-datastructures-algorithms/node_modules/fsevents
> node install

```

Figura 4.1

Com todas as dependências instaladas (`node_modules`), você terá acesso aos scripts que permitirão testar, gerar um relatório de cobertura de testes, além de gerar um arquivo chamado `PacktDataStructuresAlgorithms.min.js` contendo todo o código-fonte que criaremos a partir deste capítulo. A seguir, apresentamos os arquivos de nossa biblioteca e alguns dos arquivos que criaremos neste capítulo (Figura 4.2):

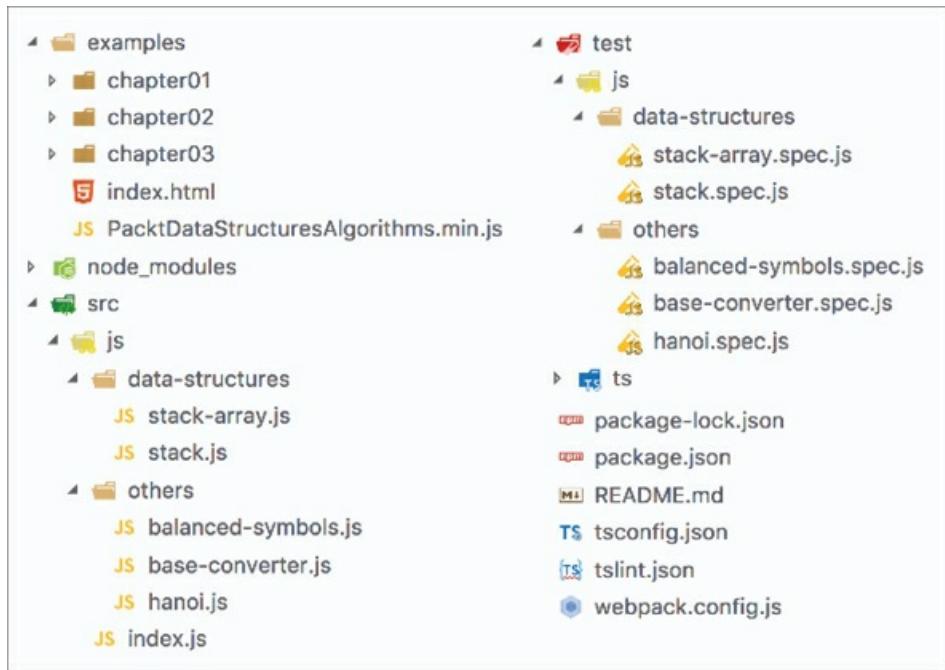


Figura 4.2

O código-fonte que criaremos neste capítulo pode ser encontrado no diretório `src/js`, organizado por categoria. Há também uma pasta `test`, na qual você encontrará um arquivo `spec.js` correspondente ao arquivo original na pasta `src`. Esses arquivos contêm o código de teste escrito com um framework de testes JavaScript chamado **Mocha** (<https://mochajs.org>). Para todo arquivo JavaScript, também teremos um arquivo TypeScript correspondente na pasta `ts`. Para executar

os testes, é possível usar o comando `npm run test`, e para executar os testes e ver o relatório de cobertura (o percentual do código-fonte coberto pelos testes), execute `npm run dev`. Se estiver usando o **Visual Studio Code** como editor, também encontrará scripts para depurar os códigos de teste. Basta adicionar seus breakpoints nos locais desejados e executar as tarefas de depuração Mocha TS ou Mocha JS. No arquivo `package.json`, você também encontrará o script `npm run webpack`, responsável por gerar o arquivo `PacktDataStructuresAlgorithms.min.js` usado por nossos exemplos de HTML. Esse script usa o **Webpack** (<https://webpack.github.io>), uma ferramenta que resolverá todas as dependências de módulos ECMAScript 2015+, transpilará o código-fonte usando o Babel, gerará o bundle com todos os arquivos JavaScript reunidos, além de deixar o código compatível com o navegador ou com o Node.js, conforme vimos no Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*. Mais informações sobre outros scripts disponíveis também podem ser encontradas no arquivo `README.md`.

Passos detalhados para fazer o download do bundle de código foram mencionados no *Prefácio* deste livro. Por favor, consulte esse texto. O bundle com o código do livro também está disponível no GitHub em <https://github.com/loiane/javascript-datastructures-algorithms>.

Estrutura de dados de pilha

Uma pilha é uma coleção ordenada de itens que obedece ao princípio **LIFO** (Last In First Out, isto é, o último a entrar é o primeiro a sair). A adição de novos itens ou a remoção de itens existentes ocorrem na mesma extremidade. O final da pilha é conhecido como topo, enquanto o lado oposto é conhecido como base. Os elementos mais novos ficam próximos ao topo, e os elementos mais antigos estão próximos da base.

Temos vários exemplos de pilhas na vida real, por exemplo, uma pilha de livros, como podemos ver na Figura 4.3, ou uma pilha de bandejas em uma lanchonete ou em uma praça de alimentação:



Figura 4.3

Uma pilha também é usada pelos compiladores em linguagens de programação, pela memória do computador para armazenar variáveis e chamadas de métodos, e também pelo histórico do navegador (o botão de retornar [back] do navegador).

Criando uma classe Stack baseada em array

Criaremos a nossa própria classe para representar uma pilha. Vamos começar pelo básico criando um arquivo `stack-array.js` e declarando a nossa classe `Stack`:

```
class Stack {  
    constructor() {  
        this.items = []; // {1}  
    }  
}
```

Precisamos de uma estrutura de dados que armazenará os elementos da pilha. Podemos usar um array para isso ({1}). A estrutura de dados de array nos permite adicionar ou remover elementos de qualquer posição da estrutura de dados. Como a pilha obedece ao princípio LIFO, limitaremos as funcionalidades que estarão disponíveis à inserção e remoção de elementos. Os métodos a seguir estarão disponíveis na classe `Stack`:

- `push(element(s))`: esse método adiciona um novo elemento (ou vários elementos) no topo da pilha.
- `pop()`: esse método remove o elemento que está no topo da pilha. Também devolve o elemento removido.

- **peek()**: esse método devolve o elemento que está no topo da pilha. A pilha não é modificada (o elemento não é removido; ele é devolvido apenas como informação).
- **isEmpty()**: esse método devolve **true** se a pilha não contiver nenhum elemento e **false** se o tamanho da pilha for maior que 0.
- **clear()**: esse método remove todos os elementos da pilha.
- **size()**: esse método devolve o número de elementos contidos na pilha. É semelhante à propriedade **length** de um array.

Push de elementos na pilha

O primeiro método que implementaremos é o método **push**, responsável pela adição de novos elementos na pilha, com um detalhe muito importante: só podemos adicionar novos itens no topo da pilha, isto é, no final dela. O método **push** é representado assim:

```
push(element) {
  this.items.push(element);
}
```

Como estamos usando um array para armazenar os elementos da pilha, podemos utilizar o método **push** da classe **Array** de JavaScript, sobre o qual discutimos no capítulo anterior.

Pop de elementos da pilha

A seguir, implementaremos o método **pop**, responsável pela remoção de itens da pilha. Como a pilha utiliza o princípio LIFO, o último item adicionado é aquele que será removido. Por esse motivo, podemos usar o método **pop** da classe **Array** de JavaScript, que também abordamos no capítulo anterior. O método **pop** é representado assim:

```
pop() {
  return this.items.pop();
}
```

Como os métodos **push** e **pop** serão os únicos métodos disponíveis para a adição e a remoção de itens da pilha, o princípio LIFO se aplicará à nossa classe **Stack**.

Dando uma espiada no elemento que está no topo da pilha

Vamos agora implementar alguns métodos auxiliares adicionais em nossa classe. Se quisermos saber qual foi o último elemento adicionado em nossa pilha, podemos usar o método `peek`. Esse método devolverá o item que está no topo da pilha:

```
peek() {  
    return this.items[this.items.length - 1];  
}
```

Como estamos usando internamente um array para armazenar os itens, o último item de um array pode ser obtido usando `length - 1`, como mostra o código que acabamos de ver.

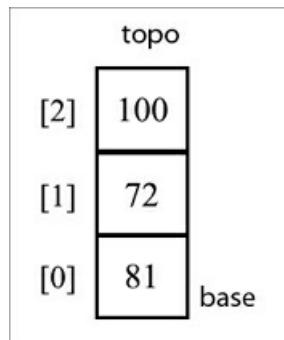


Figura 4.4

Por exemplo, no diagrama anterior (Figura 4.4), temos uma pilha com três itens; portanto, o tamanho do array interno é 3. A última posição usada no array interno é 2. Como resultado, `length - 1` ($3 - 1$) é igual a 2!

Verificando se a pilha está vazia

O próximo método que criaremos é `isEmpty`, que devolverá `true` se a pilha estiver vazia (nenhum elemento foi adicionado), e `false` caso contrário:

```
isEmpty() {  
    return this.items.length === 0;  
}
```

Ao usar o método `isEmpty`, podemos simplesmente verificar se o tamanho do array interno é 0.

De modo semelhante à propriedade `length` da classe `array`, também podemos implementar `length` em nossa classe `Stack`. Para coleções, em geral usamos o termo `size` no lugar de `length`. Novamente, como estamos

usando um array para armazenar os elementos internamente, basta devolver o valor de `length`:

```
size() {
    return this.items.length;
}
```

Limpando os elementos da pilha

Por fim, implementaremos o método `clear`, o qual simplesmente esvazia a pilha, removendo todos os seus elementos. Esta é a forma mais simples de implementar esse método:

```
clear() {
    this.items = [];
}
```

Uma implementação alternativa seria chamar o método `pop` até que a pilha esteja vazia.

Pronto! Nossa classe `Stack` está implementada.

Usando a classe Stack

Nesta seção, veremos como usar a classe `Stack`. Nossa primeira tarefa deve ser instanciar a classe `Stack` que acabamos de criar. Em seguida, podemos verificar se ela está vazia (a saída será `true` porque ainda não adicionamos nenhum elemento à nossa pilha):

```
const stack = new Stack();
console.log(stack.isEmpty()); // exibe true
```

A seguir, vamos adicionar alguns elementos na pilha (faremos um push dos números `5` e `8`; você pode adicionar elementos de qualquer tipo na pilha):

```
stack.push(5);
stack.push(8);
```

Se chamarmos o método `peek`, o número `8` será devolvido porque esse foi o último elemento que adicionamos na pilha:

```
console.log(stack.peek()); // exibe 8
```

Vamos adicionar outro elemento:

```
stack.push(11);
console.log(stack.size()); // exibe 3
console.log(stack.isEmpty()); // exibe false
```

Adicionamos o elemento **11**. Se chamarmos o método **size**, o resultado será o número **3**, pois temos três elementos em nossa pilha (**5, 8 e 11**). Além disso, se chamarmos o método **isEmpty**, a saída será **false** (temos três elementos em nossa pilha). Por fim, vamos acrescentar outro elemento:

```
stack.push(15);
```

O diagrama a seguir (Figura 4.5) mostra todas as operações de **push** que executamos até agora e o status atual de nossa pilha:

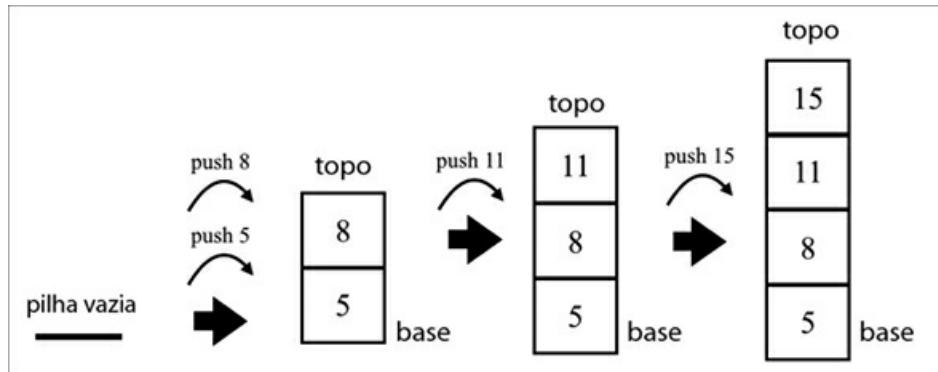


Figura 4.5

Em seguida, vamos remover dois elementos da pilha chamando o método **pop** duas vezes:

```
stack.pop();
stack.pop();
console.log(stack.size()); // exibe 2
```

Antes de chamar o método **pop** duas vezes, nossa pilha tinha quatro elementos. Após a execução do método **pop** duas vezes, a pilha agora tem apenas dois elementos: **5** e **8**. O diagrama a seguir (Figura 4.6) exemplifica a execução do método **pop**:

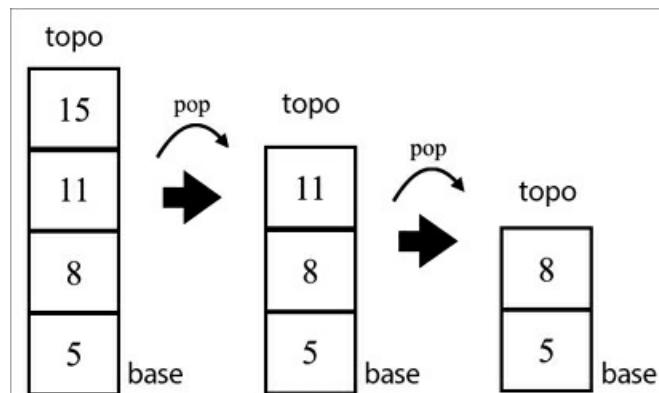


Figura 4.6

Criando uma classe JavaScript Stack baseada em objeto

O modo mais fácil de criar uma classe **Stack** usa um array para armazenar seus elementos. Ao trabalhar com um conjunto grande de dados (o que é muito comum em projetos reais), também é necessário analisar qual é o modo mais eficaz de manipular os dados. Quando trabalhamos com arrays, a maioria dos métodos tem uma complexidade de tempo $O(n)$; conhceremos melhor a complexidade dos algoritmos no último capítulo deste livro, o Capítulo 15, *Complexidade de algoritmos*. Isso significa que, para a maioria dos métodos, devemos iterar pelo array até encontrarmos o elemento que estamos procurando e, no cenário de pior caso, faremos a iteração por todas as posições do array, considerando que n é o tamanho do array. Se o array tiver mais elementos, demorará mais para iterar pelos elementos, em comparação com um array com menos elementos. Além disso, um array é um conjunto ordenado de elementos, e, para mantê-los assim, seria necessário ter mais espaço na memória também.

Não seria melhor se pudéssemos acessar diretamente o elemento, usar menos espaço de memória e continuar tendo todos os elementos organizados conforme fosse necessário? No cenário de uma estrutura de dados de pilha na linguagem JavaScript, também é possível usar um objeto JavaScript para armazenar os elementos da pilha, mantê-los em ordem e obedecer igualmente ao princípio LIFO. Vamos ver como podemos conseguir esse comportamento.

Começaremos declarando a classe **Stack** (arquivo **stack.js**) da seguinte maneira:

```
class Stack {
  constructor() {
    this.count = 0;
    this.items = {};
  }
  // métodos
}
```

Nessa versão da classe **Stack**, usaremos uma propriedade **count** para nos ajudar a manter o controle do tamanho da pilha (e, consequentemente, para nos ajudar também a adicionar e a remover elementos da estrutura de dados).

Push de elementos na pilha

Na versão baseada em array, podíamos adicionar vários elementos à classe **Stack** ao mesmo tempo. Como estamos trabalhando com um objeto, essa versão do método **push** nos permite fazer push somente de um único elemento de cada vez. Podemos ver o código do método **push** a seguir:

```
push(element) {  
    this.items[this.count] = element;  
    this.count++;  
}
```

Em JavaScript, um objeto é um conjunto de pares **chave** e **valor**. Para adicionar **element** à pilha, usaremos a variável **count** como a chave do objeto **items**, e **element** será o seu valor. Depois de fazer push do elemento na pilha, incrementamos **count**.

Podemos usar o mesmo exemplo anterior para usar a classe **Stack** e fazer push dos elementos 5 e 8:

```
const stack = new Stack();  
stack.push(5);  
stack.push(8);
```

Internamente, teremos os valores a seguir nas propriedades **items** e **count**:

```
items = {  
    0: 5,  
    1: 8  
};  
count = 2;
```

Verificando se a pilha está vazia e o seu tamanho

A propriedade **count** também funciona como o tamanho da pilha. Assim, para o método **size**, podemos simplesmente devolver a propriedade **count**:

```
size() {  
    return this.count;  
}
```

Para verificar se a pilha está vazia, podemos comparar se o valor de **count** é 0, assim:

```
isEmpty() {  
    return this.count === 0;  
}
```

Pop de elementos da pilha

Como não estamos usando um array para armazenar os elementos, teremos de implementar manualmente a lógica para remover um elemento. O método **pop** também devolve o elemento que foi removido da pilha. Esse método é implementado assim:

```
pop() {
  if (this.isEmpty()) { // {1}
    return undefined;
  }
  this.count--; // {2}
  const result = this.items[this.count]; // {3}
  delete this.items[this.count]; // {4}
  return result; // {5}
}
```

Inicialmente devemos verificar se a pilha está vazia ({1}) e, em caso afirmativo, devolveremos o valor **undefined**. Se a pilha não estiver vazia, decrementaremos a propriedade **count** ({2}) e armazenaremos o valor do topo da pilha ({3}) para que possamos devolvê-lo ({5}) depois que o elemento for removido ({4}).

Como estamos trabalhando com um objeto JavaScript, para remover um valor específico de um objeto, podemos usar o operador **delete** de JavaScript.

Vamos usar os valores internos a seguir para emular a ação **pop**:

```
items = {
  0: 5,
  1: 8
};
count = 2;
```

Para acessar o elemento do topo da pilha (último elemento adicionado: **8**), precisamos acessar a chave com o valor **1**. Então decrementamos a variável **count** de **2** para **1**. Podemos acessar **items[1]**, apagá-lo e devolver o seu valor.

Dando uma espiada no topo e limpando a pilha

Na última seção, vimos que, para acessar o elemento armazenado no topo da pilha, é necessário decrementar a propriedade **count** de **1**. Vamos então ver o código do método **peek**:

```

peek() {
  if (this.isEmpty()) {
    return undefined;
  }
  return this.items[this.count - 1];
}

```

Para limpar a pilha, basta reiniciá-la com os mesmos valores usados no construtor:

```

clear() {
  this.items = [];
  this.count = 0;
}

```

Também poderíamos usar a lógica a seguir para remover todos os elementos da pilha, respeitando o comportamento de LIFO:

```

while (!this.isEmpty()) {
  this.pop();
}

```

Criando o método `toString`

Na versão com array, não precisamos nos preocupar com um método `toString`, pois a estrutura de dados usará o método já oferecido pelo array. Para essa versão com objeto, criaremos um método `toString` para que possamos exibir o conteúdo da pilha, de modo semelhante a um array:

```

toString() {
  if (this.isEmpty()) {
    return '';
  }
  let objString = `${this.items[0]}`; // {1}
  for (let i = 1; i < this.count; i++) { // {2}
    objString = `${objString},${this.items[i]}`; // {3}
  }
  return objString;
}

```

Se a pilha estiver vazia, basta devolver uma string vazia. Se não estiver, inicializaremos a string com o primeiro elemento, que está na base da pilha (**{1}**). Então faremos uma iteração por todas as chaves da pilha (**{2}**) até o seu topo, adicionando uma vírgula (,), seguida do próximo elemento (**{3}**). Se a pilha contiver um único elemento, o código das linhas **{2}** e **{3}** não será executado.

Com o método `toString`, concluímos essa versão da classe `Stack`. Esse é

também um exemplo de como ter diferentes versões de código. Para o desenvolvedor que usar a classe **Stack**, não importa se a versão com array ou com objeto será usada; ambas têm a mesma funcionalidade, mas, internamente, o comportamento é muito diferente.

Com exceção do método `toString`, todos os outros métodos que criamos têm complexidade $O(1)$, o que significa que podemos acessar diretamente o elemento no qual estamos interessados e executar uma ação com ele (`push`, `pop` ou `peek`).

Protegendo os elementos internos da estrutura de dados

Ao criar uma estrutura de dados ou um objeto que outros desenvolvedores poderão usar também, devemos proteger os elementos internos para que somente os métodos que expusermos sejam usados para modificar a estrutura interna. No caso da classe **Stack**, queremos garantir que os elementos sejam adicionados no topo da pilha e que não seja possível adicionar elementos em sua base nem em qualquer outra posição aleatória (no meio da pilha). Infelizmente, as propriedades `items` e `count` que declaramos na classe **Stack** não estão protegidas, e esse comportamento se deve ao modo como as classes funcionam em JavaScript.

Experimente executar o código a seguir:

```
const stack = new Stack();
console.log(Object.getOwnPropertyNames(stack)); // {1}
console.log(Object.keys(stack)); // {2}
console.log(stack.items); // {3}
```

Veremos `["count", "items"]` como saída para as linhas `{1}` ou `{2}`. Isso significa que as variáveis `count` e `items` são públicas, pois podemos acessá-las facilmente, como mostra a linha `{3}`. Com esse comportamento, podemos atribuir um novo valor às propriedades `count` ou `items`.

Neste capítulo, usamos a sintaxe da **ES2015 (ES6)** para criar a classe **Stack**. As classes ES2015 são baseadas em protótipo. Embora uma classe baseada em protótipo economize memória e escale melhor que as classes baseadas em funções, essa abordagem não nos permite declarar propriedades (variáveis) ou métodos `private`. Nesse caso, queremos usar a classe **Stack** para ter acesso somente aos métodos que estamos expondo na classe. Vamos analisar outras abordagens que nos permitem ter

propriedades **private** em JavaScript.

Convenção de nomenclatura com underscore

Alguns desenvolvedores preferem usar a convenção de nomenclatura com underscore para marcar um atributo como **private** em JavaScript:

```
class Stack {  
    constructor() {  
        this._count = 0;  
        this._items = {};  
    }  
}
```

Essa convenção consiste em inserir um underscore (_) como prefixo no nome do atributo. No entanto, essa opção é apenas uma convenção; ela não protege os dados e dependemos do bom senso do desenvolvedor que usará o nosso código.

Classes ES2015 com símbolos no escopo

A ES2015 introduziu um novo tipo primitivo chamado **Symbol** que é imutável e pode ser usado como propriedade de um objeto. Vamos ver como podemos usá-lo para declarar a propriedade **items** na classe **Stack** (usaremos um array para armazenagem a fim de simplificar o código):

```
const _items = Symbol('stackItems'); //{1}  
class Stack {  
    constructor () {  
        this[_items] = []; //{2}  
    }  
    // métodos de Stack  
}
```

No código anterior, declaramos a variável **_items** como um **Symbol** (linha {1}) e inicializamos o seu valor no **constructor** da classe (linha {2}). Para acessar a variável **_items**, basta substituir todas as ocorrências de **this.items** por **this[_items]**.

Essa abordagem oferece uma propriedade **private** falsa para a classe, pois o método **Object.getOwnPropertySymbols** também foi introduzido na ES6 e pode ser usado para obter todos os símbolos de propriedades declaradas na classe. Eis um exemplo de como podemos explorar e fazer um hack na classe **Stack**:

```

const stack = new Stack();
stack.push(5);
stack.push(8);
let objectSymbols = Object.getOwnPropertySymbols(stack);
console.log(objectSymbols.length); // exibe 1
console.log(objectSymbols); // [Symbol()]
console.log(objectSymbols[0]); // Symbol()
stack[objectSymbols[0]].push(1);
stack.print(); //exibe 5, 8, 1

```

Como podemos ver no código anterior, é possível obter o símbolo `_items` acessando `stack[objectSymbols[0]]`. Como a propriedade `_items` é um array, podemos executar qualquer operação de array, por exemplo, remover ou acrescentar um elemento no meio dele (o mesmo aconteceria se estivéssemos usando um objeto para armazenar os elementos). Todavia, não é isso que queremos, pois estamos trabalhando com uma pilha.

Vamos então ver uma terceira opção.

Classes ES2015 com WeakMap

Há um tipo de dado que podemos usar para garantir que a propriedade seja `private` em uma classe, e ele se chama `WeakMap`. Exploraremos a estrutura de dados de `mapa` em detalhes no Capítulo 8, *Dicionários e hashes*, mas, por enquanto, precisamos saber que um `WeakMap` é capaz de armazenar um par chave/valor, no qual a chave é um objeto e o valor pode ser um dado de qualquer tipo.

Vamos ver como será a aparência da classe `Stack` se usarmos `WeakMap` para armazenar o atributo `items` (versão com array):

```

const items = new WeakMap(); // {1}
class Stack {
  constructor () {
    items.set(this, []); // {2}
  }
  push(element){
    const s = items.get(this); // {3}
    s.push(element);
  }
  pop(){
    const s = items.get(this);
    const r = s.pop();
    return r;
  }
}

```

```
//outros métodos  
}
```

O trecho de código anterior pode ser interpretado assim:

- Na linha {1}, declaramos a variável `items` como um `WeakMap`.
- Na linha {2}, definimos o valor de `items` no construtor, especificando `this` (referência à classe `Stack`) como a chave do `WeakMap` e o array que representa a pilha como o valor.
- Na linha {3}, obtivemos o valor de `items` acessando o valor do `WeakMap`, isto é, passando `this` como a chave (que foi definida na linha {2}).

Agora sabemos que a propriedade `items` é realmente privada na classe `Stack`. Com essa abordagem, o código não é fácil de ler e não será possível herdar as propriedades que são `private` se estendermos essa classe; não podemos ter tudo!

Proposta para campos de classe na ECMAScript

O TypeScript tem um modificador `private` para propriedades e métodos de classe. No entanto, esse modificador funciona somente em tempo de compilação (como a verificação de tipos e de erros do TypeScript, sobre a qual já discutimos em capítulos anteriores). Depois que o código é transpilado para JavaScript, o atributo será igualmente público.

O fato é que não é possível declarar propriedades ou métodos `private` como podemos fazer em outras linguagens de programação. Há abordagens diferentes com as quais podemos alcançar o mesmo resultado, mas cada uma delas tem os seus prós e contras no que diz respeito a uma sintaxe mais simples ou ao desempenho.

Qual abordagem é a melhor? Isso dependerá de como você usará os algoritmos apresentados neste livro nos projetos da vida real. Dependerá do volume de dados com o qual você lidará ou do número necessário de instâncias das classes que criamos, entre outras restrições. Em última instância, a decisão será sua.

Quando este livro foi escrito, havia uma proposta para adicionar propriedades `private` em classes JavaScript. Com essa proposta, poderemos declarar campos de classes JavaScript diretamente no corpo da classe e inicializar as propriedades. Eis um exemplo:

```
class Stack {  
    #count = 0;  
    #items = 0;  
    // métodos de Stack  
}
```

Será possível declarar propriedades **private** prefixando as propriedades com um símbolo de sustenido (#). Esse comportamento é muito semelhante à privacidade dos atributos com o **WeakMap**. Desse modo, esperamos que, em breve, não precisaremos aplicar hacks nem comprometer a legibilidade do código para usar atributos **private** nas classes.

Para obter mais informações sobre a proposta de campos de classe, por favor, acesse <https://github.com/tc39/proposal-class-fields>.

Resolvendo problemas usando pilhas

As pilhas têm uma variedade de aplicações nos problemas do mundo real. Elas podem ser usadas para problemas de backtracking, a fim de lembrar as tarefas ou os caminhos visitados, e para desfazer ações (veremos como aplicar esse exemplo quando discutirmos grafos e problemas de backtracking mais adiante neste livro). As linguagens de programação Java e C# usam pilhas para armazenar variáveis e chamadas de métodos e quando houver uma exceção de stack overflow (transbordamento de pilha) que possa ser lançada, em especial quando trabalhamos com algoritmos recursivos (que serão discutidos posteriormente neste livro também).

Agora que já sabemos como usar a classe **Stack**, vamos utilizá-la para resolver alguns problemas de ciência da computação. Nesta seção, discutiremos o problema de decimal para binário; nessa ocasião, transformaremos também o algoritmo em um conversor de base.

Convertendo números decimais para binários

Já temos familiaridade com a base decimal. No entanto, a representação binária é muito importante em ciência da computação, pois tudo em um computador é representado por dígitos binários (0 e 1). Sem a capacidade de converter números decimais para binários e vice-versa, seria um pouco difícil se comunicar com um computador.

Para converter um número decimal em uma representação binária, podemos dividir o número por 2 (binário é um sistema numérico de base 2) até que o resultado da divisão seja 0. Como exemplo, converteremos o número 10 em dígitos binários:

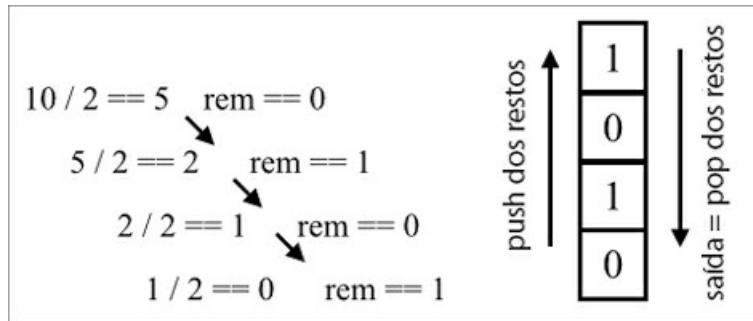


Figura 4.7

Essa conversão é uma das primeiras operações que você aprenderá na faculdade (nas aulas de ciência da computação). Eis o nosso algoritmo:

```
function decimalToBinary(decNumber) {
  const remStack = new Stack();
  let number = decNumber;
  let rem;
  let binaryString = '';
  while (number > 0) { // {1}
    rem = Math.floor(number % 2); // {2}
    remStack.push(rem); // {3}
    number = Math.floor(number / 2); // {4}
  }
  while (!remStack.isEmpty()) { // {5}
    binaryString += remStack.pop().toString();
  }
  return binaryString;
}
```

No código anterior, enquanto o resultado da divisão não for zero (linha {1}), vamos obter o resto da divisão (módulo – mod), e fazemos o push desse valor na pilha (linhas {2} e {3}); por fim, atualizamos o número que será dividido por 2 (linha {4}). Uma observação importante: JavaScript tem um tipo de dado numérico, mas não há uma distinção entre inteiros e números de ponto flutuante. Por esse motivo, devemos usar a função `Math.floor` a fim de obter somente o valor inteiro das operações de divisão. Por fim, fazemos pop dos elementos da pilha até que ela esteja vazia, concatenando os elementos removidos em uma string (linha {5}).

Podemos testar o algoritmo anterior e exibir o seu resultado no console usando o código a seguir:

```
console.log(decimalToBinary(233)); // 11101001
console.log(decimalToBinary(10)); // 1010
console.log(decimalToBinary(1000)); // 1111101000
```

Algoritmo conversor de base

Podemos modificar o algoritmo anterior para que ele funcione como um conversor de decimal para as bases entre 2 e 36. Em vez de dividir o número decimal por 2, podemos passar a base desejada como argumento para o método e usá-la nas operações de divisão, como mostra o algoritmo a seguir:

```
function baseConverter(decNumber, base) {
  const remStack = new Stack();
  const digits = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'; // {6}
  let number = decNumber;
  let rem;
  let baseString = '';
  if (!(base >= 2 && base <= 36)) {
    return '';
  }
  while (number > 0) {
    rem = Math.floor(number % base);
    remStack.push(rem);
    number = Math.floor(number / base);
  }
  while (!remStack.isEmpty()) {
    baseString += digits[remStack.pop()]; // {7}
  }
  return baseString;
}
```

Há mais um detalhe que devemos modificar. Na conversão de decimal para binário, os restos serão 0 ou 1; na conversão de decimal para octogonal, os restos variarão de 0 a 8, e na conversão de decimal para hexadecimal, os restos poderão ser de 0 a 9, além das letras de A a F (valores de 10 a 15). Por esse motivo, precisamos converter esses valores também (linhas **{6}** e **{7}**). Assim, começando na base 11, cada letra do alfabeto representará a sua base. A letra A representa a base 11, B representa a base 12, e assim sucessivamente.

Podemos usar o algoritmo anterior e exibir o seu resultado no console,

assim:

```
console.log(baseConverter(100345, 2)); // 1100001111111001  
console.log(baseConverter(100345, 8)); // 303771  
console.log(baseConverter(100345, 16)); // 187F9  
console.log(baseConverter(100345, 35)); // 2BW0
```

Você verá também os exemplos de parênteses balanceados (problema do palíndromo) e da Torre de Hanói quando fizer o download do código-fonte deste livro.

Resumo

Neste capítulo, conhecemos a estrutura de dados de pilha. Implementamos o nosso próprio algoritmo para representar uma pilha usando arrays e um objeto JavaScript, e vimos como adicionar e remover elementos dela usando os métodos **push** e **pop**.

Além disso, comparamos as diferentes sintaxes que podem ser usadas para criar a classe **Stack**, e apresentamos os prós e contras de cada uma. Também discutimos o modo de resolver um dos problemas mais famosos de ciência da computação usando pilhas.

No próximo capítulo, conheceremos as filas, que são muito semelhantes às pilhas, porém usam um princípio que não é a LIFO.

CAPÍTULO 5

Filas e dequees

Já vimos como as pilhas funcionam. As filas (queues) são estruturas de dados muito semelhantes, mas, em vez de LIFO, elas usam um princípio diferente, que conheceremos neste capítulo. Veremos também como funcionam os dequees (filas de duas pontas ou fila duplamente terminada): uma estrutura de dados que mistura princípios de pilha e de fila.

Os seguintes tópicos serão abordados neste capítulo:

- a estrutura de dados de fila;
- a estrutura de dados de deque;
- adição de elementos em uma fila e em um deque;
- remoção de elementos de uma fila e de um deque;
- simulação de filas circulares com o jogo de Batata Quente;
- verificação se uma frase é um palíndromo com um deque.

Estrutura de dados de fila

Uma **fila** é uma coleção ordenada de itens baseada em **FIFO** (First In First Out, isto é, o primeiro que entra é o primeiro que sai), também conhecido como princípio do **first-come first-served** (o primeiro a chegar é o primeiro a ser servido). A adição de novos elementos em uma fila é feita na cauda (tail) e a remoção, na frente. O elemento mais recente adicionado na fila deve esperar no final dela.

O exemplo mais conhecido de uma fila na vida real é a típica fila que se forma ocasionalmente (Figura 5.1).

Temos filas no cinema, na lanchonete e no caixa de um supermercado, por exemplo. A primeira pessoa que estiver na fila será a primeira a ser atendida.

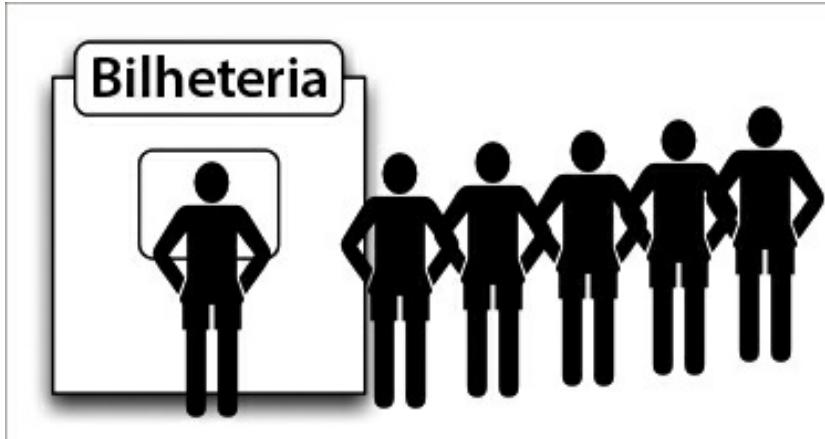


Figura 5.1

Um exemplo muito conhecido em ciência da computação é a fila de impressão. Suponha que precisamos imprimir cinco documentos. Abrimos cada um dos documentos e clicamos no botão para imprimir. Cada um será enviado para a fila da impressora. O primeiro documento para o qual solicitamos a impressão será impresso antes, e assim por diante, até que todos tenham sido impressos.

Criando a classe Queue

Criaremos agora a nossa própria classe para representar uma fila. Vamos começar pelo básico e declarar a nossa classe:

```
class Queue {  
    constructor() {  
        this.count = 0; // {1}  
        this.lowestCount = 0; // {2}  
        this.items = {}; // {3}  
    }  
}
```

Inicialmente, precisamos ter uma estrutura de dados que armazenará os elementos da fila. Podemos usar um array para isso, como fizemos na classe `Stack` em um dos exemplos do capítulo anterior; no entanto, usaremos um objeto para armazenar nossos elementos ({3}); isso nos permitirá escrever uma estrutura de dados mais eficiente para acessar seus elementos. Você também notará que as classes `Queue` e `Stack` são muito parecidas – somente os princípios para adição e remoção de elementos são diferentes.

Para nos ajudar a controlar o tamanho da fila, declaramos uma

propriedade `count` também (`{1}`). E, como removeremos os elementos da frente da fila, também precisamos de uma variável para nos ajudar a manter o controle do primeiro elemento. Para isso, declaramos a variável `lowestCount` (`{2}`).

Em seguida, devemos declarar os métodos disponíveis em uma fila:

- `enqueue(element)`: esse método adiciona um novo elemento no final da fila.
- `dequeue()`: esse método remove o primeiro elemento da fila (o item que está na frente). Também devolve o elemento removido.
- `peek()`: esse método devolve o primeiro elemento da fila – é o primeiro item adicionado e o primeiro que será removido da fila. A fila não é modificada (o elemento não é removido, mas será devolvido apenas como informação – é muito semelhante ao método `peek` da classe `Stack`). Funciona igualmente como o método `front`, como é conhecido em outras linguagens.
- `isEmpty()`: esse método devolve `true` se a fila não contiver nenhum elemento, e `false` se o tamanho for maior que `0`.
- `size()`: esse método devolve o número de elementos contidos na fila. É semelhante à propriedade `length` do array.

Inserção de elementos na fila

O primeiro método que implementaremos é o método `enqueue`. Esse método será responsável pela adição de novos elementos na fila, com um detalhe muito importante: só podemos adicionar novos itens no final da fila.

```
enqueue(element) {  
    this.items[this.count] = element;  
    this.count++;  
}
```

O método `enqueue` tem a mesma implementação que o método `push` da classe `Stack`. Como a propriedade `items` é um objeto JavaScript, ela é uma coleção de pares `chave` e `valor`. Para adicionar um elemento à fila, usaremos a variável `count` como chave do objeto `items`, e `element` será o seu valor. Depois de inserir o elemento na fila, incrementaremos `count`.

Remoção de elementos da fila

A seguir, implementaremos o método `dequeue`, responsável pela remoção de itens da fila. Como a fila utiliza o princípio FIFO, o primeiro item adicionado na fila será o item a ser removido:

```
dequeue() {
  if (this.isEmpty()) {
    return undefined;
  }
  const result = this.items[this.lowestCount]; // {1}
  delete this.items[this.lowestCount]; // {2}
  this.lowestCount++; // {3}
  return result; // {4}
}
```

Inicialmente devemos verificar se a fila está vazia e, em caso afirmativo, devolveremos o valor `undefined`. Se a fila não estiver vazia, armazenaremos o valor da frente da fila (`{1}`) para que possamos devolvê-lo (`{4}`) depois que o elemento tiver sido removido (`{2}`). Também precisamos incrementar a propriedade `lowestCount` de 1 (`{3}`).

Vamos usar os valores internos a seguir para emular a ação de remoção da fila:

```
items = {
  0: 5,
  1: 8
};
count = 2;
lowestCount = 0;
```

Para acessar o elemento da frente da fila (o primeiro elemento adicionado: 5), precisamos acessar a chave com o valor 0. Podemos acessar `items[0]`, apagá-lo e devolver o seu valor. Nesse cenário, depois de remover o primeiro elemento, a propriedade `items` conterá somente um elemento (1: 8), que será o próximo a ser removido se o método `dequeue` for chamado. Assim, incrementamos a variável `lowestCount` de 0 para 1.

Como os métodos `enqueue` e `dequeue` são os únicos métodos disponíveis para a adição e a remoção de itens da fila, garantimos que o princípio FIFO será aplicado em nossa classe `Queue`.

Dando uma espiada no elemento que está na frente da fila

Vamos agora implementar alguns métodos auxiliares adicionais em nossa classe. Se quisermos saber qual é o elemento que está na frente em nossa fila, podemos usar o método `peek`. Esse método devolverá o item que está na frente da fila (usando `lowestCount` como chave para obter o valor do elemento):

```
peek() {
    if (this.isEmpty()) {
        return undefined;
    }
    return this.items[this.lowestCount];
}
```

Verificando se a pilha está vazia e o seu tamanho

O próximo método é `isEmpty`, que devolverá `true` se a pilha estiver vazia, e `false` caso contrário:

```
isEmpty() {
    return this.count - this.lowestCount === 0;
}
```

Para calcular quantos elementos há na fila, basta calcular a diferença entre as chaves `count` e `lowestCount`.

Suponha que a propriedade `count` tenha valor 2 e `lowestCount` seja igual a 0. Isso significa que temos dois elementos na fila. Em seguida, removemos um elemento dela. A propriedade `lowestCount` será atualizada com o valor 1 e `count` continuará com um valor igual a 2. Agora a fila terá somente um elemento, e assim por diante.

Assim, para implementar o método `size`, basta devolver esta diferença:

```
size() {
    return this.count - this.lowestCount;
}
```

Também podemos escrever método `isEmpty` assim:

```
isEmpty() {
    return this.size() === 0;
}
```

Limpando a fila

Para limpar todos os elementos da fila, podemos chamar o método `dequeue` até que ele devolva `undefined`, ou podemos simplesmente

reiniciar o valor das propriedades da classe `Queue` com os mesmos valores declarados em seu construtor:

```
clear() {
  this.items = [];
  this.count = 0;
  this.lowestCount = 0;
}
```

Criando o método `toString`

Pronto! Nossa classe `Queue` está implementada. Assim como fizemos na classe `Stack`, também podemos acrescentar o método `toString`:

```
toString() {
  if (this.isEmpty()) {
    return '';
  }
  let objString = `${this.items[this.lowestCount]}`;
  for (let i = this.lowestCount + 1; i < this.count; i++) {
    objString = `${objString},${this.items[i]}`;
  }
  return objString;
}
```

Na classe `Stack`, começamos a iterar pelos valores dos itens a partir do índice zero. Como o primeiro índice da classe `Queue` pode não ser zero, começamos iterando a partir do índice `lowestCount`.

Agora realmente terminamos!

As classes `Queue` e `Stack` são muito parecidas. A única diferença está nos métodos `dequeue` e `peek`, que se deve à distinção entre os princípios FIFO e LIFO.

Usando a classe `Queue`

Inicialmente devemos instanciar a classe `Queue` que criamos. Em seguida, podemos verificar se ela está vazia (a saída será `true` porque ainda não adicionamos nenhum elemento em nossa fila):

```
const queue = new Queue();
console.log(queue.isEmpty()); // exibe true
```

Em seguida, vamos adicionar alguns elementos na fila (faremos a inserção

dos elementos 'John' e 'Jack' com `enqueue` – você poderá adicionar qualquer tipo de elemento na fila):

```
queue.enqueue('John');
queue.enqueue('Jack');
console.log(queue.toString()); // John,Jack
```

Vamos acrescentar outro elemento:

```
queue.enqueue('Camila');
```

Executaremos também outros comandos:

```
console.log(queue.toString()); // John,Jack,Camila
console.log(queue.size()); // exibe 3
console.log(queue.isEmpty()); // exibe false
queue.dequeue(); // remove John
queue.dequeue(); // remove Jack
console.log(queue.toString()); // Camila
```

Se pedirmos para exibir o conteúdo da fila, veremos **John**, **Jack** e **Camila**. O tamanho da fila será igual a 3, pois temos três elementos na fila (e ela também não estará vazia).

O diagrama a seguir exemplifica todas as operações de `enqueue` executadas até agora e o status atual de nossa fila:

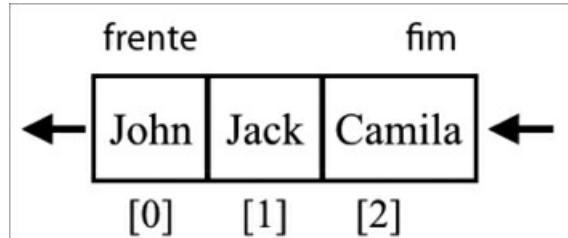


Figura 5.2

Em seguida, pedimos para remover dois elementos da fila com `dequeue` (esse método é executado duas vezes). O diagrama a seguir exemplifica a execução do método `dequeue`:

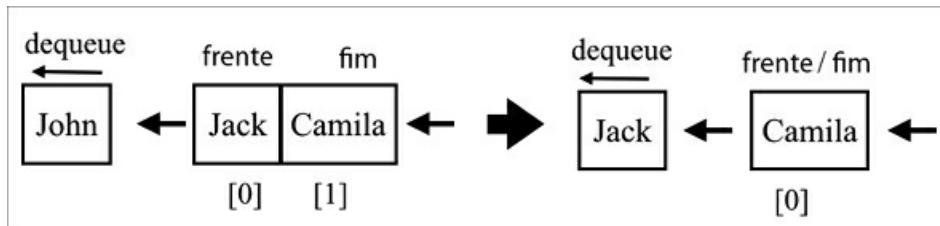


Figura 5.3

Por fim, quando pedimos para exibir o conteúdo da fila novamente, vemos

apenas o elemento **Camila**. Os dois primeiros elementos inseridos na fila foram removidos; o último elemento inserido na estrutura de dados será o último a ser removido da fila. Isso significa que estamos seguindo o princípio FIFO.

Estrutura de dados de deque

A estrutura de dados de **deque**, também conhecida como **fila de duas pontas** (double-ended queue), é uma fila especial que nos permite inserir e remover elementos do final ou da frente da fila.

Um exemplo de um deque na vida real é a fila típica em cinemas, lanchonetes e assim por diante. Por exemplo, uma pessoa que acabou de comprar um ingresso poderia retornar para a frente da fila somente para pedir uma informação rápida. Se a pessoa que estiver no final da fila estiver com pressa, ela poderia também sair da fila.

Em ciência da computação, uma aplicação comum de um deque é na armazenagem de uma lista de operações para desfazer ações (undo). Sempre que um usuário executar uma operação no software, um push dessa operação será feito no deque (exatamente como em uma pilha). Quando o usuário clicar no botão Undo (Desfazer), uma operação de pop será efetuada no deque, o que significa que essa operação será removida do final. Depois de um número predefinido de operações, as operações mais antigas serão removidas da frente do deque. Como o deque implementa os princípios tanto de FIFO quanto de LIFO, podemos dizer também que o deque combina as estruturas de dados de fila e de pilha.

Criando a classe Deque

Como sempre, começaremos declarando a classe **Deque** e o seu **construtor**:

```
class Deque {  
    constructor() {  
        this.count = 0;  
        this.lowestCount = 0;  
        this.items = {};  
    }  
}
```

Como o deque é uma fila especial, percebemos que ele compartilha alguns

trechos de código com o construtor, tem as mesmas propriedades internas e tem os métodos a seguir: `isEmpty`, `clear`, `size` e `toString`.

Pelo fato de o deque permitir inserir e remover elementos das duas extremidades, teremos também os métodos a seguir:

- `addFront(element)`: esse método adiciona um novo elemento na frente do deque.
- `addBack(element)`: esse método adiciona um novo elemento no fim do deque (a mesma implementação do método `enqueue` da classe `Queue`).
- `removeFront()`: esse método remove o primeiro elemento do deque (a mesma implementação do método `dequeue` da classe `Queue`).
- `removeBack()`: esse método remove o último elemento do deque (a mesma implementação do método `pop` da classe `Stack`).
- `peekFront()`: esse método devolve o primeiro elemento do deque (a mesma implementação do método `peek` da classe `Queue`).
- `peekBack()`: esse método devolve o último elemento do deque (a mesma implementação do método `peek` da classe `Stack`).

A classe `Deque` também implementa os métodos `isEmpty`, `clear`, `size` e `toString` (você pode analisar o código-fonte completo fazendo o download do bundle com os códigos-fontes deste livro).

Adicionando elementos na frente do deque

Como já implementamos a lógica de alguns métodos, manteremos o foco somente na lógica do método `addFront`. Apresentamos a seguir o código desse método:

```
addFront(element) {  
    if (this.isEmpty()) { // {1}  
        this.addBack(element);  
    } else if (this.lowestCount > 0) { // {2}  
        this.lowestCount--;  
        this.items[this.lowestCount] = element;  
    } else {  
        for (let i = this.count; i > 0; i--) { // {3}  
            this.items[i] = this.items[i - 1];  
        }  
        this.count++;  
    }  
}
```

```

    this.lowestCount = 0;
    this.items[0] = element; // {4}
}
}

```

Ao adicionar um elemento na frente do deque, há três cenários.

O primeiro cenário é aquele em que o deque está vazio ({1}). Nesse caso, chamamos o método **addBack**. O elemento será adicionado no final do deque, que, nesse caso, também será a frente. O método **addBack** já tem a lógica necessária para incrementar a propriedade **count**, portanto podemos reutilizá-la a fim de evitar um código duplicado.

O segundo cenário é aquele em que um elemento é removido da frente do deque ({2}), o que significa que a propriedade **lowestCount** terá um valor maior ou igual a 1. Nesse caso, basta decrementar a propriedade **lowestCount** e atribuir o elemento à chave desse objeto (posição).

Considere os valores internos a seguir para a classe **Deque**:

```

items = {
  1: 8,
  2: 9
};
count = 3;
lowestCount = 1;

```

Se quisermos adicionar o elemento 7 na frente do deque, teremos o segundo cenário. Nesse exemplo, o valor de **lowestCount** será decrementado (o novo valor será 0 – zero), e adicionaremos o valor 7 à chave 0.

O terceiro e último cenário é aquele em que **lowestCount** é igual a 0 (zero). Poderíamos atribuir uma chave com um valor negativo e atualizar a lógica usada para calcular o tamanho do deque para avaliar também as chaves negativas. Nesse caso, a operação para adicionar um novo valor continuaria tendo o menor custo de processamento. Por motivos pedagógicos, trataremos esse cenário como se estivéssemos trabalhando com arrays. Para adicionar um novo elemento na primeira chave ou posição, devemos mover todos os elementos para a próxima posição ({3}) a fim de deixar o primeiro **index** livre. Como não queremos perder nenhum valor existente, começamos a iterar pelos valores existentes na propriedade **items** a partir de seu último índice, atribuindo-lhe o elemento que está em **index - 1**. Depois que todos os elementos tiverem sido

movidos, a primeira posição estará livre e poderemos sobrescrever qualquer valor existente com o elemento que queremos adicionar no deque (`{4}`).

Usando a classe Deque

Depois de instanciar a classe `Deque`, podemos chamar os seus métodos:

```
const deque = new Deque();
console.log(deque.isEmpty()); // exibe true
deque.addBack('John');
deque.addBack('Jack');
console.log(deque.toString()); // John,Jack
deque.addBack('Camila');
console.log(deque.toString()); // John,Jack,Camila
console.log(deque.size()); // exibe 3
console.log(deque.isEmpty()); // exibe false
deque.removeFront(); // remove John
console.log(deque.toString()); // Jack,Camila
deque.removeBack(); // Camila decide sair
console.log(deque.toString()); // Jack
deque.addFront('John'); // John retorna para pedir uma informação
console.log(deque.toString()); // John,Jack
```

Com a classes `Deque`, podemos chamar operações das classes `Stack` e `Queue`. Poderíamos também usar a classe `Deque` para implementar uma fila de prioridades, mas exploraremos esse assunto no Capítulo 11, *Heap binário e heap sort*.

Resolvendo problemas usando filas e deque

Agora que já sabemos como usar as classes `Queue` e `Deque`, vamos utilizá-las para resolver alguns problemas de ciência da computação. Nesta seção, discutiremos uma simulação do jogo de Batata Quente (Hot Potato) com filas, além de verificar se uma frase é um palíndromo usando deque.

Fila circular – Batata Quente

Como as filas são aplicadas com frequência em ciência da computação e em nossas vidas, há algumas versões modificadas em relação à fila padrão que implementamos neste capítulo. Uma das versões modificadas é a **fila circular**. Um exemplo de fila circular é o jogo de Batata Quente (Hot

Potato). Nesse jogo, as crianças se organizam em um círculo e passam a batata quente para o seu vizinho o mais rápido possível. Em determinado ponto do jogo, a batata quente para de ser passada pelo círculo de crianças e a criança que tiver a batata quente em mãos deverá sair do círculo. Essa ação será repetida até que reste apenas uma criança (a vencedora).

Neste exemplo, implementaremos uma simulação do jogo de Batata Quente:

```
function hotPotato(elementsList, num) {
  const queue = new Queue(); // {1}
  const eliminatedList = [];
  for (let i = 0; i < elementsList.length; i++) {
    queue.enqueue(elementsList[i]); // {2}
  }
  while (queue.size() > 1) {
    for (let i = 0; i < num; i++) {
      queue.enqueue(queue.dequeue()); // {3}
    }
    eliminatedList.push(queue.dequeue()); // {4}
  }
  return {
    eliminated: eliminatedList,
    winner: queue.dequeue() // {5}
  };
}
```

Para implementar uma simulação desse jogo, usaremos a classe `Queue` que implementamos no início do capítulo ({1}). Vamos obter uma lista de nomes e enfileirar todos eles ({2}). Dado um número, devemos iterar pela fila. Removemos um item do início da fila e o adicionamos no final ({3}) para simular a batata quente (se você passou a batata quente para o seu vizinho, então não correrá o risco de ser eliminado de imediato). Uma vez que o número for alcançado, a pessoa que tiver a batata quente será eliminada (removida da fila – {4}). Quando restar apenas uma pessoa, ela será declarada a vencedora (linha {5}).

Podemos usar o código a seguir para testar o algoritmo `hotPotato`:

```
const names = ['John', 'Jack', 'Camila', 'Ingrid', 'Carl'];
const result = hotPotato(names, 7);
result.eliminated.forEach(name => {
  console.log(`#${name} was eliminated from the Hot Potato game.`);
});
console.log(`The winner is: ${result.winner}`);
```

Eis a saída do algoritmo anterior:

```
Camila was eliminated from the Hot Potato game.  
Jack was eliminated from the Hot Potato game.  
Carl was eliminated from the Hot Potato game.  
Ingrid was eliminated from the Hot Potato game.  
The winner is: John
```

O diagrama a seguir (Figura 5.4) simula esse resultado:

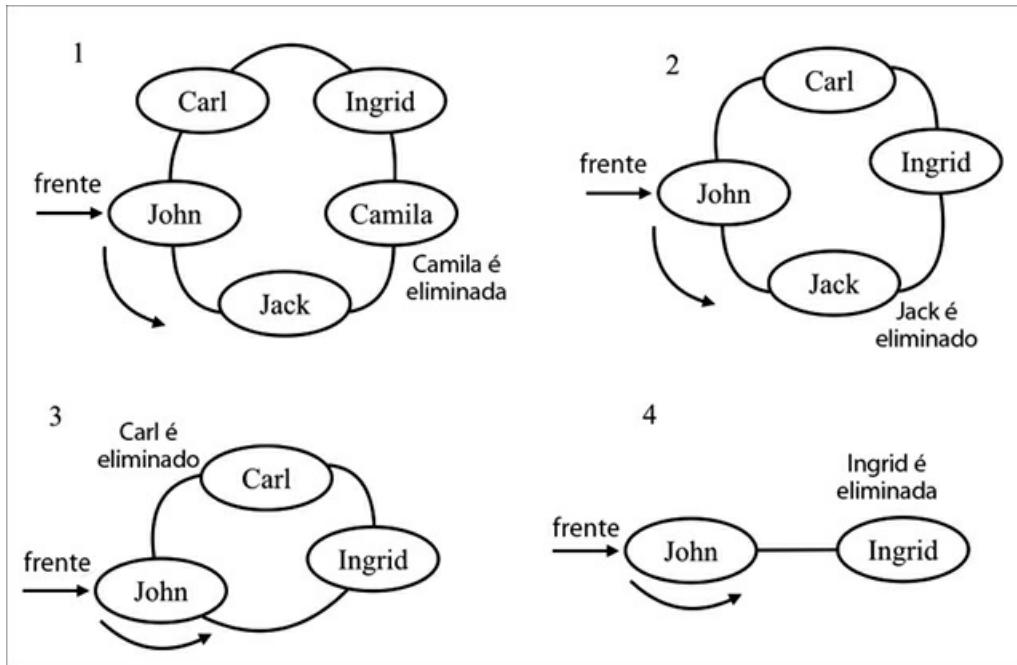


Figura 5.4

Você pode alterar o número passado para a função `hotPotato` a fim de simular cenários diferentes.

Verificador de palíndromo

A seguir, apresentamos a definição de **palíndromo** de acordo com a Wikipédia:

Um palíndromo é uma palavra, frase, número ou outra sequência de caracteres que é lido igualmente de trás para frente ou de frente para trás, por exemplo, madam ou racecar.¹

Há diferentes algoritmos que podem ser usados para verificar se uma frase ou uma string é um palíndromo. O modo mais fácil é inverter a string e compará-la com a string original. Se as duas strings forem iguais, teremos um palíndromo. Também podemos usar uma pilha para fazer isso, mas a

maneira mais fácil de resolver esse problema com uma estrutura de dados é usando um deque.

O algoritmo a seguir utiliza um deque para solucionar esse problema:

```
function palindromeChecker(aString) {
  if (aString === undefined || aString === null ||
      (aString !== null && aString.length === 0)) { // {1}
    return false;
  }
  const deque = new Deque(); // {2}
  const lowerString = aString.toLocaleLowerCase().split(' ').join(''); // {3}
  let isEqual = true;
  let firstChar, lastChar;
  for (let i = 0; i < lowerString.length; i++) { // {4}
    deque.addBack(lowerString.charAt(i));
  }
  while (deque.size() > 1 && isEqual) { // {5}
    firstChar = deque.removeFront(); // {6}
    lastChar = deque.removeBack(); // {7}
    if (firstChar !== lastChar) {
      isEqual = false; // {8}
    }
  }
  return isEqual;
}
```

Antes de começar a analisar a lógica do algoritmo, devemos verificar se a string passada como parâmetro é válida ({1}). Se não for, devolveremos **false**.

Nesse algoritmo, usamos a classe **Deque** que implementamos neste capítulo ({2}). Como podemos receber uma string com letras tanto minúsculas quanto maiúsculas, transformamos todas as letras em minúsculas e removemos também todos os espaços ({3}). Se quiser, você também poderá remover todos os caracteres especiais, como !?()- e assim por diante. Para manter a simplicidade do algoritmo, desconsideraremos essa parte.

Em seguida, inserimos todos os caracteres da string no **deque** usando **enqueue** ({4}). Enquanto tivermos elementos no **deque** (se restar apenas um caractere, será um palíndromo) e a string for um palíndromo ({5}), removemos um elemento da frente ({6}) e um de trás ({7}). Para ser um palíndromo, os dois caracteres removidos do **deque** devem ser iguais. Se os caracteres não coincidirem, a string não será um palíndromo ({8}).

Podemos usar o código a seguir para testar o algoritmo `palindromeChecker`:

```
console.log('a', palindromeChecker('a'));
console.log('aa', palindromeChecker('aa'));
console.log('kayak', palindromeChecker('kayak'));
console.log('level', palindromeChecker('level'));
console.log('Was it a car or a cat I saw', palindromeChecker('Was it a car or a
cat I saw'));
console.log('Step on no pets', palindromeChecker('Step on no pets'));
```

A saída de todos os exemplos anteriores é `true`.

Filas de tarefas em JavaScript

Como estamos usando JavaScript neste livro, por que não explorar um pouco mais o funcionamento da linguagem?

Quando abrimos uma nova aba no navegador, uma fila de tarefas é criada. Isso ocorre porque apenas uma única thread trata todas as tarefas de uma única aba, e ela é chamada de **laço de eventos** (event loop). O navegador é responsável por várias tarefas, como renderizar o HTML, executar comandos com código JavaScript, tratar a interação com o usuário (entrada de usuário, cliques de mouse e assim por diante) e executar e processar requisições assíncronas. Saiba mais sobre os laços de eventos no link <https://goo.gl/ayF840>.

É muito bom saber que uma linguagem popular e eficaz como o JavaScript utiliza uma estrutura de dados tão básica para lidar com controles internos.

Resumo

Neste capítulo, conhecemos a estrutura de dados de fila (queue). Implementamos o nosso próprio algoritmo para representar uma fila e vimos como adicionar e remover elementos dela usando os métodos `enqueue` e `dequeue`, de acordo com o princípio de FIFO. Também conhecemos a estrutura de dados de deque, aprendemos a adicionar elementos na frente e no final do deque e a remover elementos da frente ou do final dessa estrutura.

Além disso, discutimos como resolver dois problemas famosos usando as estruturas de dados de fila e de deque: o jogo de Batata Quente (usando uma fila modificada: a fila circular) e um verificador de palíndromo usando

um deque.

No próximo capítulo, conhceremos as listas ligadas: uma estrutura de dados dinâmica e mais complexa.

¹ N.T.: Tradução livre com base no original em inglês.

CAPÍTULO 6

Listas ligadas

No Capítulo 3, *Arrays*, conhecemos a estrutura de dados de array. Um array (também podemos chamá-lo de lista) é uma estrutura de dados muito simples que armazena uma sequência de dados. Neste capítulo, aprenderemos a implementar e a usar uma lista ligada, que é uma estrutura de dados dinâmica; isso significa que podemos adicionar ou remover itens do modo que quisermos, e ela aumentará conforme for necessário.

Os seguintes tópicos serão abordados neste capítulo:

- a estrutura de dados da lista ligada;
- adição de elementos em uma lista ligada;
- remoção de elementos de uma lista ligada;
- como usar a classe `LinkedList`;
- listas duplamente ligadas;
- listas ligadas circulares;
- listas ligadas ordenadas;
- implementação de uma pilha com listas ligadas.

Estrutura de dados da lista ligada

Arrays (ou listas) provavelmente são a estrutura de dados mais comum usada para armazenar uma coleção de elementos. Conforme mencionamos antes neste livro, cada linguagem tem a própria implementação de arrays. Essa estrutura de dados é muito conveniente e oferece uma sintaxe prática com `[]` para acessar seus elementos. No entanto, ela apresenta uma desvantagem: o tamanho do array é fixo (na maioria das linguagens), e inserir ou remover itens do início ou do meio do array é custoso, pois os elementos têm de sofrer um deslocamento (apesar de termos aprendido que JavaScript tem métodos na classe `Array` que farão isso para nós; é isso que acontece internamente também).

As listas ligadas armazenam uma coleção sequencial de elementos; no

entanto, de modo diferente dos arrays, nas listas ligadas os elementos não são posicionados de forma contígua na memória. Cada elemento é constituído de um nó que armazena o elemento propriamente dito, além de uma referência (também conhecida como ponteiro ou ligação) que aponta para o próprio elemento. O diagrama a seguir (Figura 6.1) exemplifica a estrutura de uma lista ligada:

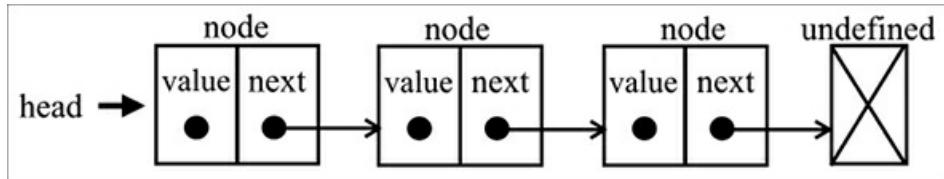


Figura 6.1

Uma das vantagens de uma lista ligada em relação a um array convencional é que não é necessário deslocar os elementos quando eles são adicionados ou removidos. Entretanto, precisamos usar ponteiros quando trabalhamos com uma lista ligada, e, por esse motivo, é preciso prestar atenção especial na implementação desse tipo de lista. Em um array, podemos acessar diretamente qualquer elemento em qualquer posição; em uma lista ligada, se quisermos acessar um elemento no meio, será necessário partir do início (cabeça ou **head**) e iterar pela lista até encontrarmos o elemento desejado.

Temos alguns exemplos de listas ligadas no mundo real. O primeiro é uma fila de pessoas dançando conga. Cada pessoa é um elemento, e as mãos seriam o ponteiro que faz a ligação com a próxima pessoa na fila da conga. Você pode adicionar pessoas à fila – basta encontrar a posição em que se deseja adicionar essa pessoa, desfazer a conexão e então inserir a nova pessoa e fazer a conexão novamente.

Outro exemplo seria uma caça ao tesouro. Você tem uma pista, e esta será o ponteiro para o lugar em que a próxima pista poderá ser encontrada. Com essa ligação, você irá para o próximo local e obterá outra pista que levará para a próxima. A única maneira de obter uma pista que está no meio da lista é seguir a lista desde o início (partindo da primeira).

Temos outro exemplo, que talvez seja o mais popularmente usado para exemplificar as listas ligadas: um trem. Um trem é constituído de uma série de carros (também conhecidos como vagões). Cada um dos carros ou vagões está ligado a outro. Você pode facilmente desconectar um vagão, mudá-lo de lugar, ou ainda adicionar ou remover um vagão. A Figura 6.2

mostra um trem. Cada vagão é um elemento da lista e a ligação entre os vagões é o ponteiro:

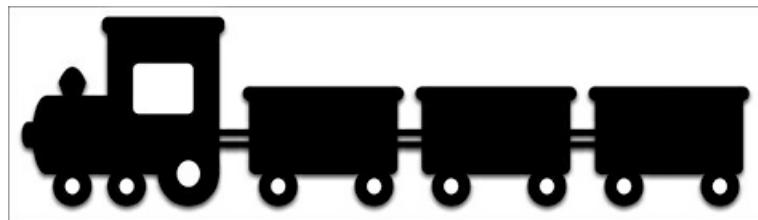


Figura 6.2

Neste capítulo, discutiremos a lista ligada, assim como algumas de suas variantes; vamos, porém, começar pela estrutura de dados mais simples.

Criando a classe `LinkedList`

Agora que você já sabe o que é uma lista ligada, vamos começar a implementar a nossa estrutura de dados. Eis o esqueleto de nossa classe `LinkedList`:

```
import { defaultEquals } from '../util';
import { Node } from './models/linked-list-models'; // {1}
export default class LinkedList {
  constructor>equalsFn = defaultEquals) {
    this.count = 0; // {2}
    this.head = undefined; // {3}
    this.equalsFn = equalsFn; // {4}
  }
}
```

Na estrutura de dados `LinkedList`, começamos declarando a propriedade `count` ({2}), que armazena o número de elementos que temos na lista.

Implementaremos um método chamado `indexOf`, o qual nos permitirá encontrar um elemento específico na lista ligada. Para uma comparação de igualdade entre os elementos da lista ligada, usaremos uma função que será chamada internamente como `equalsFn` ({4}). O desenvolvedor que usará a classe `LinkedList` poderá passar uma função personalizada que compare dois objetos JavaScript ou dois valores. Se nenhuma função personalizada for passada, essa estrutura de dados usará a função `defaultEquals` declarada no arquivo `util.js` (de modo que possamos reutilizá-la em outras estruturas de dados e algoritmos que criaremos em capítulos futuros) como a função default para comparação de igualdade. Eis a função

`defaultEquals`:

```
export function defaultEquals(a, b) {
  return a === b;
}
```

O valor `default` dos parâmetros e a importação do módulo para a função `defaultEquals` fazem parte das funcionalidades da ECMAScript 2015 (ES6), conforme vimos no Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*.

Como essa estrutura de dados é dinâmica, precisamos armazenar também uma referência ao primeiro elemento. Para isso, podemos armazenar a referência de `this` em uma variável que chamaremos de `head` ({3}).

Para representar a cabeça (`head`) e outros elementos da lista ligada, precisamos de uma classe auxiliar chamada `Node` ({1}). Essa classe representa o item que queremos adicionar na lista. Ela contém um atributo `element`, que é o valor que queremos adicionar na lista, e um atributo `next`, que é o ponteiro que faz a ligação para o próximo nó da lista. A classe `Node` está declarada no arquivo `models/linked-list-models.js` (visando à reutilização) e o seu código está apresentado a seguir:

```
export class Node {
  constructor(element) {
    this.element = element;
    this.next = undefined;
  }
}
```

Em seguida, temos os métodos da classe `LinkedList`. Vejamos qual é a responsabilidade de cada método antes de implementá-los:

- `push(element)`: esse método adiciona um novo elemento no final da lista.
- `insert(element, position)`: esse método insere um novo elemento em uma posição específica da lista.
- `getElementAt(index)`: esse método devolve o elemento que está em uma posição específica da lista. Se o elemento não estiver na lista, `undefined` será devolvido.
- `remove(element)`: esse método remove um elemento da lista.
- `indexOf(element)`: esse método devolve o índice do elemento na lista.

Se o elemento não estiver na lista, `-1` será devolvido.

- `removeAt(position)`: esse método remove um item de uma posição específica da lista.
- `isEmpty()`: esse método devolve `true` se a lista ligada não contiver nenhum elemento, e `false` se o tamanho da lista ligada for maior que `0`.
- `size()`: esse método devolve o número de elementos contidos na lista ligada. É semelhante à propriedade `length` do array.
- `toString()`: esse método devolve uma representação em string da lista ligada. Como a lista usa uma classe `Node` como elemento, devemos sobrescrever o método `toString` default herdado da classe `Object` de JavaScript a fim de exibir somente os valores dos elementos.

Inserindo elementos no final da lista ligada

Ao adicionar um elemento no final de um objeto `LinkedList`, podemos ter dois cenários: um em que a lista está vazia e estamos adicionando o seu primeiro elemento, ou outro em que a lista não está vazia e estamos concatenando elementos a ela.

Eis a implementação do método `push`:

```
push(element) {  
    const node = new Node(element); // {1}  
    let current; // {2}  
    if (this.head == null) { // {3}  
        this.head = node;  
    } else {  
        current = this.head; // {4}  
        while (current.next != null) { // {5} obtém o último item  
            current = current.next;  
        }  
        // e atribui o novo elemento a next para criar a ligação  
        current.next = node; // {6}  
    }  
    this.count++; // {7}  
}
```

A primeira tarefa é criar um novo `Node` passando `element` como o seu valor ({1}).

Vamos implementar o primeiro cenário: adicionar um elemento quando a lista está vazia. Quando criamos um objeto `LinkedList`, `head` (cabeça)

apontará para `undefined` (ou poderia ser `null` também):

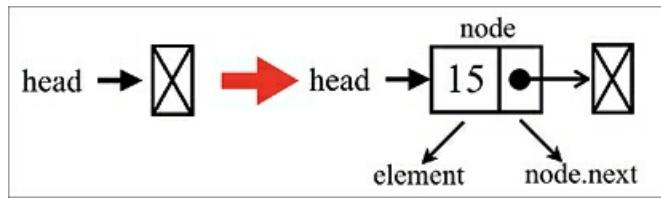


Figura 6.3

Se o elemento `head` for `undefined` ou `null` (a lista está vazia {3}), é sinal de que estamos adicionando o primeiro elemento à lista. Então tudo que temos a fazer é atribuir o `node` a `head`. O próximo elemento `node` será automaticamente `undefined`.

O último nó da lista sempre terá um valor `undefined` ou `null` como o próximo elemento.

Desse modo, temos o primeiro cenário coberto. Vamos passar para o segundo, que consiste em adicionar um elemento no final da lista quando ela não está vazia.

Para adicionar um elemento no final da lista, inicialmente devemos localizar o último elemento. Lembre-se de que temos somente uma referência ao primeiro elemento {4}, portanto é necessário iterar pela lista até encontrarmos o último item. Para isso, precisamos de uma variável que aponte para o item atual (`current`) na lista {2}.

Quando percorremos a lista com um laço, sabemos que alcançamos o seu final quando o ponteiro `current.next` for `undefined` ou `null` {5}. Então, tudo que temos a fazer é ligar o ponteiro `next` do elemento `current` (que é o último) ao nó que queremos adicionar na lista {6}.

`this.head == null` {3} é equivalente a `(this.head === undefined || head === null)`, e `current.next != null` {5} é equivalente a `(current.next !== undefined && current.next !== null)`. Para obter mais informações sobre os operadores de igualdade `==` e `===` de JavaScript, por favor, consulte o Capítulo 1, *JavaScript – uma visão geral rápida*.

O diagrama a seguir (Figura 6.4) exemplifica a inserção de um elemento no final de uma lista ligada quando ela não está vazia:

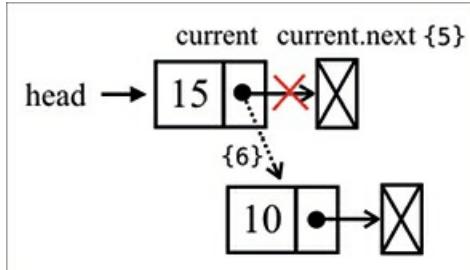


Figura 6.4

Quando criamos uma instância de `Node`, seu ponteiro `next` sempre será `undefined`. Não há problema nisso, pois sabemos que esse será o último item da lista.

E, por fim, não podemos nos esquecer de incrementar o tamanho da lista para que possamos ter controle sobre ela e obter facilmente o seu tamanho (`{7}`).

Podemos usar e testar a estrutura de dados que criamos até agora com o código a seguir:

```
const list = new LinkedList();
list.push(15);
list.push(10);
```

Removendo elementos de uma posição específica da lista ligada

Vamos agora ver como podemos remover elementos de `LinkedList`. Implementaremos dois métodos: o primeiro remove um elemento de uma posição específica (`removeAt`), enquanto o segundo é baseado no valor do elemento (apresentaremos o método `remove` mais adiante). Como no caso do método `push`, há dois cenários para remover elementos de uma lista ligada. O primeiro é aquele em que removemos o primeiro elemento, e o segundo é aquele em que removemos qualquer elemento que não seja o primeiro.

O código de `removeAt` é mostrado a seguir:

```
removeAt(index) {
  // verifica valores fora do intervalo
  if (index >= 0 && index < this.count) { // {1}
    let current = this.head; // {2}
    //remove o primeiro item
    if (index === 0) { // {3}
      this.head = current.next;
    } else {
```

```

let previous; // {4}
for (let i = 0; i < index; i++) { // {5}
    previous = current; // {6}
    current = current.next; // {7}
}
// faz a ligação de previous com o next de current: pula esse elemento para
removê-lo
previous.next = current.next; // {8}
}
this.count--; // {9}
return current.element;
}
return undefined; // {10}
}

```

Vamos explorar esse código passo a passo. Como o método receberá o `index` (posição) do nó que deve ser removido, precisamos verificar se `index` é válido ({1}). Uma posição válida variará de `index 0` (inclusive) até o tamanho da lista (`count - 1`, pois `index` começa de zero). Se a posição não for válida, devolveremos `undefined` ({10}), o que significa que nenhum elemento foi removido da lista).

Vamos escrever o código para o primeiro cenário – queremos remover o primeiro elemento da lista (`index === 0`, em {3}). O diagrama a seguir exemplifica essa ação:

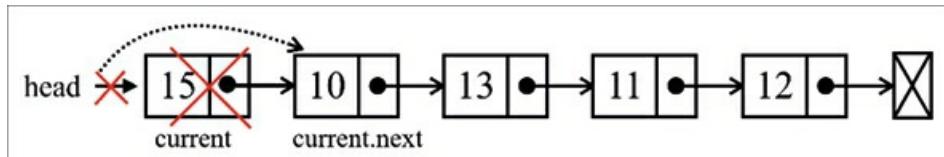


Figura 6.5

Assim, se quisermos remover o primeiro elemento, basta fazer `head` apontar para o segundo elemento da lista. Fazemos uma referência ao primeiro elemento da lista usando a variável `current` ({2}); também usaremos esse valor para iterar pela lista, mas discutiremos isso em breve). Assim, a variável `current` é uma referência ao primeiro elemento da lista. Se atribuirmos `head` para `current.next`, estaremos removendo o primeiro elemento. Poderíamos também atribuir `head` diretamente para `head.next` (sem usar a variável `current`, como alternativa).

Vamos agora supor que queremos remover o último item ou um item do meio da lista. Para isso, devemos iterar pelos nós da lista ligada até

chegarmos à posição desejada (`{5}`). Um detalhe importante: a variável `current` sempre fará referência ao elemento atual da lista que estivermos percorrendo com o laço (`{7}`). Devemos também ter uma referência ao elemento que estiver antes de `current` (`{6}`); nós o chamaremos de `previous` (`{4}`).

Depois de iterar até a posição desejada, a variável `current` armazenará o nó que queremos remover da lista ligada. Assim, para remover o nó `current`, tudo que temos a fazer é ligar `previous.next` a `current.next` (`{8}`). Desse modo, o nó `current` ficará perdido na memória do computador e estará disponível para limpeza pelo coletor de lixo (garbage collector).

Para compreender melhor o funcionamento do coletor de lixo em JavaScript, acesse https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management.

Vamos tentar entender melhor esse código usando alguns diagramas. Suponha que queremos remover o último elemento:

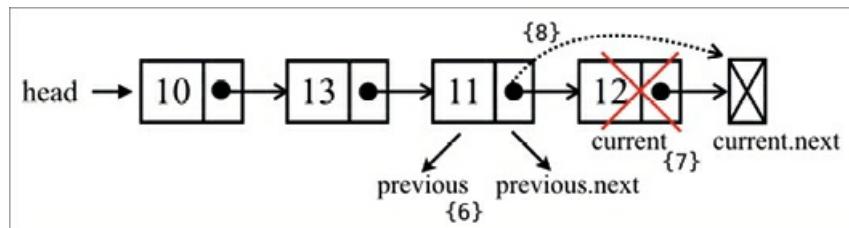


Figura 6.6

No caso do último elemento, quando saímos do laço na linha `{8}`, a variável `current` será uma referência ao último nó da lista (aquele que queremos remover). O valor de `current.next` será `undefined` (porque ele é o último nó). Como mantemos também uma referência ao nó anterior com `previous` (um nó antes do atual), `previous.next` apontará para `current`; portanto, para remover `current`, tudo que temos a fazer é alterar o valor de `previous.next` para `current.next`.

Vamos ver agora se a mesma lógica se aplica a um elemento que está no meio da lista:

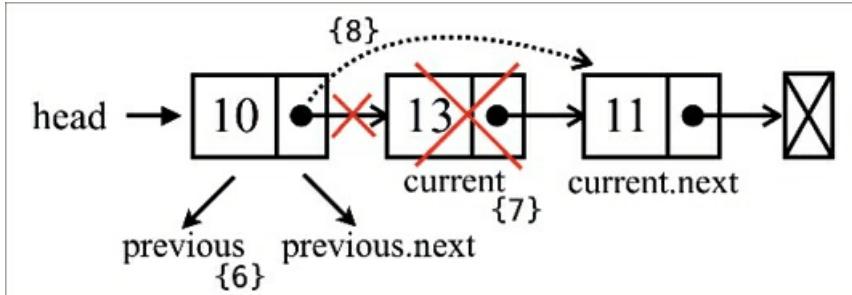


Figura 6.7

A variável `current` é uma referência ao nó que queremos remover. A variável `previous` é uma referência ao nó que vem antes do elemento que queremos remover; assim, para remover o nó `current`, tudo que temos a fazer é ligar `previous.next` a `current.next`, e, portanto, nossa lógica funciona nos dois casos.

Percorrendo a lista com um laço até alcançar a posição desejada

No método `remove`, devemos percorrer a lista com um laço até alcançar o `index` (posição) desejado. O trecho de código para alcançar o `index` desejado com um laço é comum nos métodos da classe `LinkedList`. Por esse motivo, podemos refatorá-lo e extrair a sua lógica em um método separado para que ele seja reutilizado em lugares diferentes. Desse modo, vamos criar o método `getElementAt`:

```
getElementAt(index) {
  if (index >= 0 && index <= this.count) { // {1}
    let node = this.head; // {2}
    for (let i = 0; i < index && node != null; i++) { // {3}
      node = node.next;
    }
    return node; // {4}
  }
  return undefined; // {5}
}
```

Somente para garantir que percorreremos a lista com um laço até encontrarmos uma posição válida, devemos verificar se o `index` passado como parâmetro é uma posição válida ({1}). Se uma posição inválida for passada como parâmetro, devolveremos `undefined`, pois a posição não existirá na `LinkedList` ({5}). Em seguida, inicializaremos a variável `node` com o primeiro elemento, que é o `head` ({2}) – essa variável será usada para fazer a iteração pela lista. Também podemos renomear a variável `node`

para `current` se quisermos manter o padrão usado nos outros métodos da classe `LinkedList`.

A seguir, percorremos a lista com um laço até alcançar o `index` desejado (`{3}`). Ao sair do laço, o elemento `node` (`{4}`) referenciará o elemento na posição `index`. Você também pode usar `i = 1; i <= index` no laço para alcançar o mesmo resultado.

Refatorando o método `remove`

Podemos refatorar o método `remove` e usar o método `getElementAt` criado. Para isso, podemos substituir as linhas de `{4}` a `{8}`, assim:

```
if (index === 0) {  
    // lógica para a primeira posição  
} else {  
    const previous = this.getElementAt(index - 1);  
    current = previous.next;  
    previous.next = current.next;  
}  
this.count--; // {9}
```

Inserindo um elemento em qualquer posição

A seguir, implementaremos o método `insert`, que possibilita inserir um `element` em qualquer posição. Vamos analisar a sua implementação:

```
insert(element, index) {  
    if (index >= 0 && index <= this.count) { // {1}  
        const node = new Node(element);  
        if (index === 0) { // adiciona na primeira posição  
            const current = this.head;  
            node.next = current; // {2}  
            this.head = node;  
        } else {  
            const previous = this.getElementAt(index - 1); // {3}  
            const current = previous.next; // {4}  
            node.next = current; // {5}  
            previous.next = node; // {6}  
        }  
        this.count++; // atualiza o tamanho da lista  
        return true;  
    }  
    return false; // {7}  
}
```

Como estamos lidando com posições (índices), devemos verificar se os valores não estão fora do intervalo (`{1}`), exatamente como fizemos no método `remove`. Se o valor estiver fora do intervalo, devolveremos `false` para informar que nenhum item foi adicionado na lista (`{7}`).

Se a posição for válida, trataremos os diferentes cenários. O primeiro é aquele em que precisamos adicionar `element` no início da lista, isto é, na *primeira posição*. O diagrama a seguir (Figura 6.8) exemplifica esse cenário:

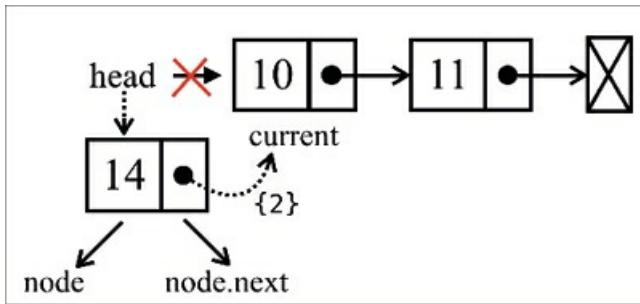


Figura 6.8

No diagrama anterior, temos a variável `current` que faz referência ao primeiro elemento da lista. O que precisamos fazer é definir o valor de `node.next` para `current` (o primeiro elemento da lista, ou `head` diretamente). Agora temos `head` e também `node.next` apontando para `current`. Em seguida, basta alterar a referência `head` para `node` (`{2}`), e teremos um novo elemento na lista.

Vamos agora cuidar do segundo cenário: adicionar `element` no meio ou no final da lista. Em primeiro lugar, devemos percorrer a lista com um laço até que a posição desejada seja alcançada (`{3}`). Nesse caso, avançaremos até `index - 1`, isto é, uma posição antes do local em que queremos inserir o novo `node`.

Ao sair do laço, a variável `previous` referenciará um elemento antes do `index` em que queremos inserir o novo elemento, e a variável `current` (`{4}`) referenciará um `element` após a posição em que gostaríamos de inserir o novo elemento. Nesse caso, queremos adicionar o novo item entre `previous` e `current`. Portanto, em primeiro lugar, devemos fazer uma ligação entre o novo elemento (`node`) e `current` (`{5}`) e, então, precisamos alterar a ligação entre `previous` e `current`. Devemos fazer `previous.next` apontar para `node` (`{6}`) em vez de apontar para `current`.

Vamos ver o código em ação usando um diagrama (Figura 6.9):

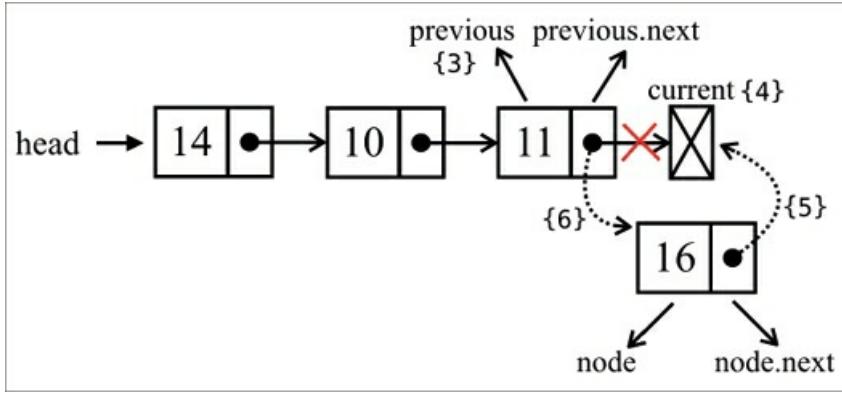


Figura 6.9

Se tentarmos adicionar um novo **element** na última posição, **previous** será uma referência ao último item da lista, e **current** será **undefined**. Nesse caso, **node.next** apontará para **current**, **previous.next** apontará para **node** e teremos um novo **element** na lista.

Vamos ver agora como adicionar um novo **element** no meio da lista, com a ajuda do diagrama a seguir (Figura 6.10).

Nesse caso, estamos tentando inserir o novo **element** (**node**) entre os elementos **previous** e **current**. Inicialmente, devemos definir o valor do ponteiro **node.next** para **current**. Em seguida, temos de definir o valor de **previous.next** para **node**. Por fim, teremos um novo **element** na lista!

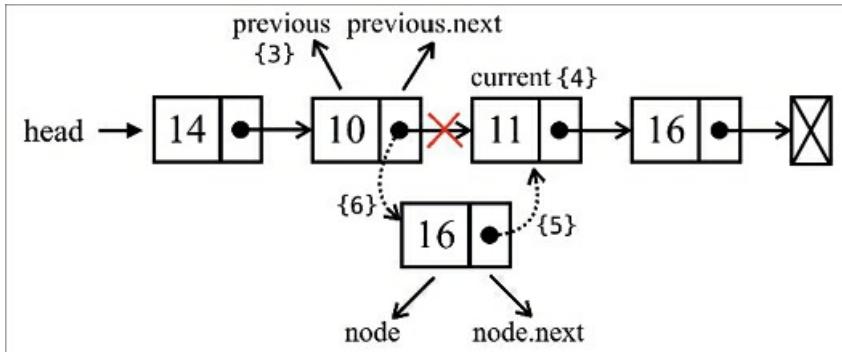


Figura 6.10

É muito importante ter variáveis que referenciem os nós a serem controlados para que a ligação entre eles não seja perdida. Poderíamos trabalhar com apenas uma variável (**previous**), mas seria mais difícil controlar as ligações entre os nós. Por esse motivo, é melhor declarar uma variável extra para nos ajudar com essas referências.

Método `indexOf`: devolvendo a posição de um elemento

O próximo método que implementaremos é o método `indexOf`. Esse método recebe o valor de um elemento e devolve a sua posição caso ele seja encontrado. Do contrário, `-1` será devolvido.

Vamos analisar a sua implementação:

```
indexOf(element) {  
    let current = this.head; // {1}  
    for (let i = 0; i < this.count && current != null; i++) { // {2}  
        if (this.equalsFn(element, current.element)) { // {3}  
            return i; // {4}  
        }  
        current = current.next; // {5}  
    }  
    return -1; // {6}  
}
```

Como sempre, precisamos de uma variável que nos ajude a iterar pela lista; essa variável é `current`, e o seu primeiro valor é `head` ({1}).

Em seguida, iteramos pelos elementos ({2}), começando de `head` (índice `0`) até que o tamanho da lista (a variável `count`) seja alcançado. Somente por garantia, podemos verificar se a variável `current` é `null` ou `undefined` a fim de evitar erros em tempo de execução.

Em cada iteração, verificaremos se o elemento que estamos procurando é o elemento no nó `current` ({3}). Nesse caso, usaremos a função de igualdade que passamos para o construtor da classe `LinkedList`. O valor default de `equalFn` é apresentado a seguir:

```
function defaultEquals(a, b) {  
    return a === b;  
}
```

Portanto, seria o mesmo que usar `element === current.element` na linha {3}. No entanto, se o elemento for um objeto complexo, permitimos que o desenvolvedor passe uma função personalizada para `LinkedList` a fim de comparar os elementos.

Se o elemento que estamos procurando for o elemento em `current`, devolveremos a sua posição ({4}). Se não for, iteramos para o próximo nó da lista ({5}).

O laço não será executado se a lista estiver vazia ou se alcançarmos o final dela. Se o valor não for encontrado, devolveremos `-1` ({6}).

Removendo um elemento da lista ligada

Com o método `indexOf` criado, podemos implementar outros métodos, por exemplo, o método `remove`:

```
remove(element) {  
    const index = this.indexOf(element);  
    return this.removeAt(index);  
}
```

Já temos implementado um método que remove um elemento de uma dada posição (`removeAt`). Agora que temos o método `indexOf`, se passarmos o valor do elemento, podemos determinar a sua posição e chamar o método `removeAt` passando a posição encontrada. Será muito mais simples, além de mais fácil, caso haja necessidade de modificar o código do método `removeAt` – ele será alterado para os dois métodos (esse é um aspecto interessante da reutilização de código). Desse modo, não será preciso manter dois métodos para remover um item da lista – só precisamos de um! Além do mais, as restrições de limites também serão verificadas pelo método `removeAt`.

Métodos `isEmpty`, `size` e `getHead`

Os métodos `isEmpty` e `size` são iguais àqueles implementados nas classes que criamos em capítulos anteriores. De qualquer modo, vamos analisá-los:

```
size() {  
    return this.count;  
}
```

O método `size` devolve o número de elementos da lista. De modo diferente das classes que implementamos em capítulos anteriores, a variável `count` da lista é controlada internamente, pois `LinkedList` é uma classe implementada do zero.

O método `isEmpty` devolverá `true` se não houver nenhum elemento na lista, e `false` caso contrário: Este código é mostrado a seguir:

```
isEmpty() {  
    return this.size() === 0;  
}
```

Por fim, temos o método `getHead`:

```
getHead() {
```

```
    return this.head;
}
```

A variável `head` é uma variável *privada* da classe `LinkedList` (conforme vimos, o JavaScript ainda não oferece suporte para propriedades realmente privadas, mas, como aprendizado, estamos considerando as propriedades das instâncias como privadas, pois supomos que os desenvolvedores que usarão nossas classes acessarão somente seus métodos). Assim, se precisarmos iterar pela lista fora da implementação da classe, podemos disponibilizar um método para obter o primeiro elemento da lista.

Método `toString`

O método `toString` converte o objeto `LinkedList` em uma string. Eis a implementação do método `toString`:

```
toString() {
  if (this.head == null) { // {1}
    return '';
  }
  let objString = `${this.head.element}`; // {2}
  let current = this.head.next; // {3}
  for (let i = 1; i < this.size() && current != null; i++) { // {4}
    objString = `${objString},${current.element}`;
    current = current.next;
  }
  return objString; // {5}
}
```

Em primeiro lugar, se a lista estiver vazia (`head` será `null` ou `undefined`), devolveremos uma string vazia ({1}). Também podemos usar `if (this.isEmpty())` para essa verificação.

Se a lista não estiver vazia, inicializamos a string que será devolvida no final do método (`objString`) com o valor do primeiro elemento ({2}). Em seguida, iteramos por todos os outros elementos da lista ({4}), adicionando os valores à nossa string. Se a lista tiver somente um elemento, a validação `current != null` falhará porque a variável `current` terá um valor igual a `undefined` (ou `null`), portanto o algoritmo não adicionará nenhum outro valor em `objString`.

No final, devolvemos a string com o conteúdo da lista ({5}).

Listas duplamente ligadas

Há diferentes tipos de listas ligadas. Nesta seção, discutiremos a **lista duplamente ligada**. A diferença entre uma lista duplamente ligada e uma lista ligada comum é que, nessa última, fazemos a ligação somente de um nó para o próximo, enquanto, em uma lista duplamente ligada, temos uma ligação dupla: uma para o próximo elemento e outra para o elemento anterior, como mostra o diagrama a seguir:

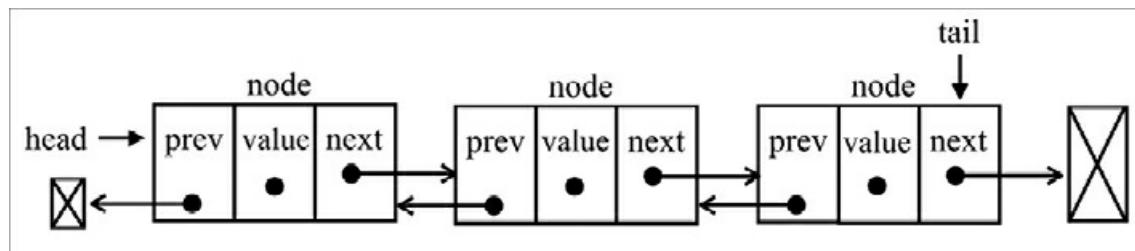


Figura 6.11

Vamos começar com as alterações necessárias para implementar a classe **DoublyLinkedList**:

```
class DoublyNode extends Node { // {1}
    constructor(element, next, prev) {
        super(element, next); // {2}
        this.prev = prev; // {3} NOVO
    }
}
class DoublyLinkedList extends LinkedList { // {4}
    constructor>equalsFn = defaultEquals) {
        super>equalsFn); // {5}
        this.tail = undefined; // {6} NOVO
    }
}
```

Como a classe **DoublyLinkedList** é um tipo especial de **LinkedList**, estenderemos essa classe ({4}). Isso significa que a classe **DoublyLinkedList** herdará (terá acesso a) todas as propriedades e métodos da classe **LinkedList**. Então, para começar, no **constructor** de **DoublyLinkedList**, devemos chamar ({5}) o **constructor** de **LinkedList**, que inicializará as propriedades **equalsFn**, **count** e **head**. Além disso, precisamos manter também uma referência ao último elemento da lista (**tail**, em {6}).

A lista duplamente ligada nos oferece duas maneiras de iterar por ela: do

início para o fim ou vice-versa. Também podemos acessar o próximo elemento ou o elemento anterior de um nó em particular. Por causa desse comportamento, para cada nó, temos de manter também o controle sobre o seu nó anterior. Assim, além das propriedades `element` e `next` da classe `Node`, `DoubleLinkedList` usará um nó especial chamado `DoublyNode` com uma propriedade chamada `prev` ({3}). `DoublyNode` estende a classe `Node`, portanto podemos herdar as propriedades `element` e `next` ({1}). Como estamos usando a herança, devemos chamar o construtor de `Node` no construtor da classe `DoublyNode` ({2}).

Na lista ligada simples, quando iteramos pela lista e ultrapassamos o elemento desejado, é necessário retornar ao início da lista e reiniciar a iteração. Essa é uma das vantagens da lista duplamente ligada.

Como podemos observar no código anterior, as diferenças entre as classes `LinkedList` e `DoublyLinkedList` estão marcadas com `NOVO`.

Inserindo um novo elemento em qualquer posição

Inserir um novo `element` em uma lista duplamente ligada é muito semelhante à inserção em uma lista ligada. A diferença é que, na lista ligada, controlamos apenas um ponteiro (`next`), enquanto, na lista duplamente ligada, temos de controlar as propriedades `next` e `prev` (o elemento anterior). Na classe `DoublyLinkedList`, sobrescreveremos o método `insert`, o que significa que aplicaremos um comportamento diferente daquele da classe `LinkedList`.

Eis o algoritmo para inserir um novo elemento em qualquer posição:

```
insert(element, index) {
    if (index >= 0 && index <= this.count) {
        const node = new DoublyNode(element);
        let current = this.head;
        if (index === 0) {
            if (this.head === null) { // {1} NOVO
                this.head = node;
                this.tail = node;
            } else {
                node.next = this.head; // {2}
                current.prev = node; // {3} NOVO
                this.head = node; // {4}
            }
        }
```

```

} else if (index === this.count) { // último item - NOVO
  current = this.tail; // {5}
  current.next = node; // {6}
  node.prev = current; // {7}
  this.tail = node; // {8}
} else {
  const previous = this.getElementAt(index - 1); // {9}
  current = previous.next; // {10}
  node.next = current; // {11}
  previous.next = node; // {12}
  current.prev = node; // {13} NOVO
  node.prev = previous; // {14} NOVO
}
this.count++;
return true;
}
return false;
}

```

Vamos analisar o primeiro cenário: inserir um novo **element** na primeira posição da lista (no início dela). Se a lista estiver vazia ({1}), basta fazer **head** e **tail** apontarem para o novo nó. Caso contrário, a variável **current** será uma referência ao primeiro **element** da lista. Como fizemos no caso da lista ligada, definimos **node.next** para **current** ({2}), e **head** apontará para **node** ({4}) – será o primeiro **element** da lista). A diferença agora é que devemos definir também um valor para o ponteiro anterior dos elementos. O ponteiro **current.prev** apontará para o novo elemento (**node**, em {3}), em vez de apontar para **undefined**. Como o ponteiro **node.prev** já é **undefined**, não é necessário alterar nada.

O diagrama a seguir mostra esse processo:

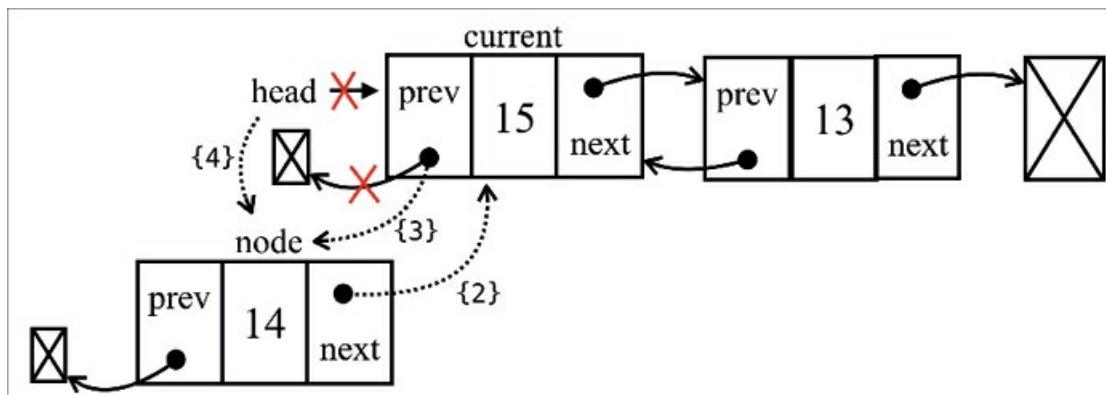


Figura 6.12

Vamos agora analisar outro cenário: suponha que queremos adicionar um novo `element` como o último da lista. Como estamos controlando também o ponteiro para o último `element`, esse é um caso especial. A variável `current` referenciará o último `element` ({5}). Então começamos a fazer as ligações: o ponteiro `current.next` (que aponta para `undefined`) apontará para `node` ({6}): `node.next` já estará apontando para `undefined` por causa do construtor). O ponteiro `node.prev` referenciará `current` ({7}). Então restará apenas uma tarefa a ser feita: atualizar `tail`, que apontará para `node` em vez de apontar para `current` ({8}).

O diagrama a seguir (Figura 6.13) mostra todas essas ações:

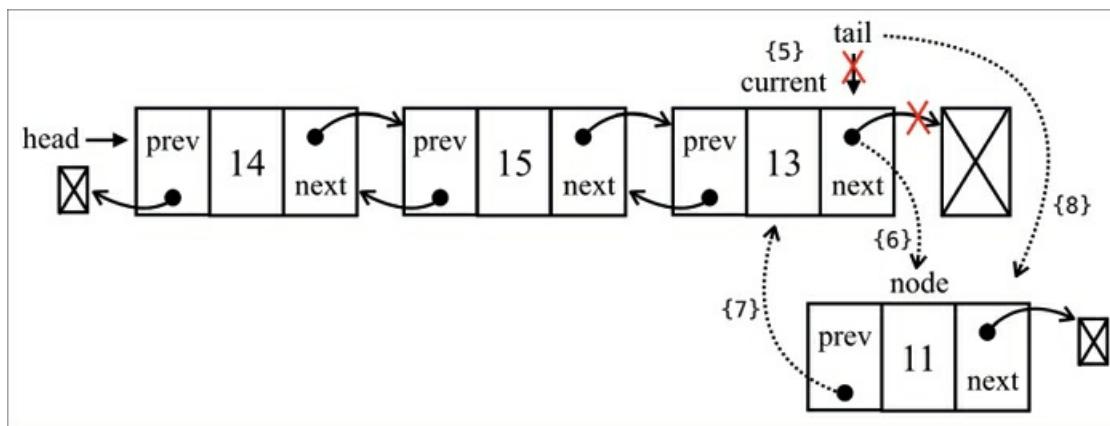


Figura 6.13

Temos então o terceiro cenário: inserir um novo `element` no meio da lista. Como no caso dos métodos anteriores, faremos uma iteração pela lista até alcançarmos a posição desejada ({9}). O método `getElementAt` é herdado da classe `LinkedList`, e não será necessário sobrescrever o seu comportamento. Vamos inserir o novo `element` entre os elementos `current` ({10}) e `previous`. Inicialmente, `node.next` apontará para `current` ({11}) e `previous.next` apontará para `node` ({12}), para não perdermos a ligação entre os nós. Em seguida, devemos corrigir todas as ligações: `current.prev` apontará para `node` ({13}) e `node.prev` apontará para `previous` ({14}). O diagrama a seguir (Figura 6.14) exemplifica esse processo:

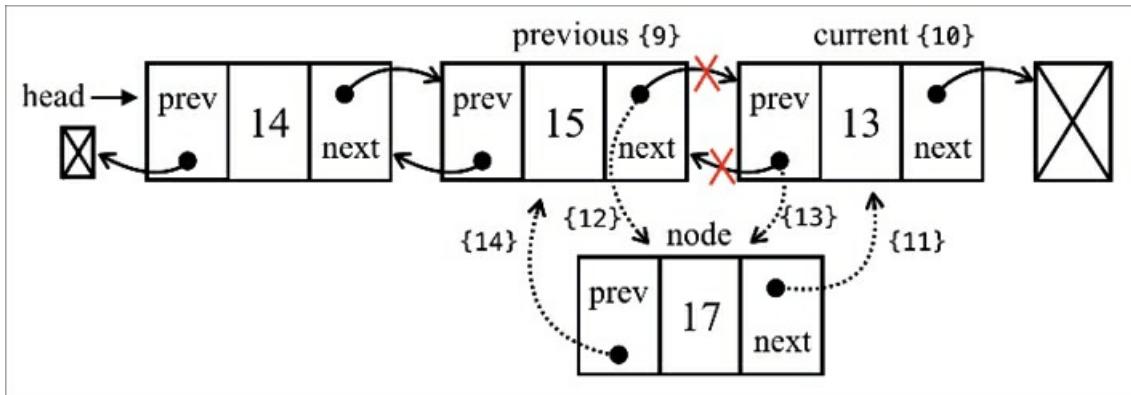


Figura 6.14

Podemos fazer algumas melhorias nos dois métodos que implementamos: `insert` e `remove`. No caso de um resultado negativo, poderíamos inserir os elementos no final da lista. Há também uma possível melhoria quanto ao desempenho: por exemplo, se `position` for maior que `length/2`, será melhor iterar a partir do final da lista em vez do início (ao fazer isso, a iteração percorrerá menos elementos na lista).

Removendo elementos de qualquer posição

Remover elementos de uma lista duplamente ligada também é muito semelhante à remoção em uma lista ligada. A única diferença é que devemos definir o ponteiro para o elemento anterior também. Vamos analisar a implementação:

```
removeAt(index) {
    if (index >= 0 && index < this.count) {
        let current = this.head;
        if (index === 0) {
            this.head = current.next; // {1}
            // se houver apenas um item, atualizamos tail também - NOVO
            if (this.count === 1) { // {2}
                this.tail = undefined;
            } else {
                this.head.prev = undefined; // {3}
            }
        } else if (index === this.count - 1) { // último item - NOVO
            current = this.tail; // {4}
            this.tail = current.prev; // {5}
            this.tail.next = undefined; // {6}
        }
    }
}
```

```

    } else {
        current = this.getElementAt(index); // {7}
        const previous = current.prev; // {8}
        // faz a ligação de previous com o next de current - pula esse elemento para
        // removê-lo
        previous.next = current.next; // {9}
        current.next.prev = previous; // {10} NOVO
    }
    this.count--;
    return current.element;
}
return undefined;
}

```

Devemos tratar três cenários: remover um elemento do início da lista, remover um elemento do meio da lista e remover o último elemento.

Vamos analisar a remoção do primeiro elemento. A variável **current** é uma referência ao primeiro **element** da lista, isto é, aquele que queremos remover. Tudo que temos a fazer é alterar a referência de **head**; em vez de apontar para **current**, ele apontará para o próximo **element** (**current.next**, em {1}). No entanto, devemos atualizar também o ponteiro para o elemento anterior de **current.next** (pois o ponteiro **prev** do primeiro **element** é uma referência para **undefined**); assim, alteramos a referência de **head.prev** para **undefined** ({3} – pois **head** também aponta para o novo primeiro **element** da lista; ou podemos usar também **current.next.prev**). Como precisamos controlar também a referência em **tail**, podemos verificar se o **element** que estamos tentando remover é o primeiro e, em caso afirmativo, tudo que temos a fazer é definir **tail** igualmente com **undefined** ({2}).

O diagrama a seguir (Figura 6.15) mostra a remoção do primeiro elemento de uma lista duplamente ligada:

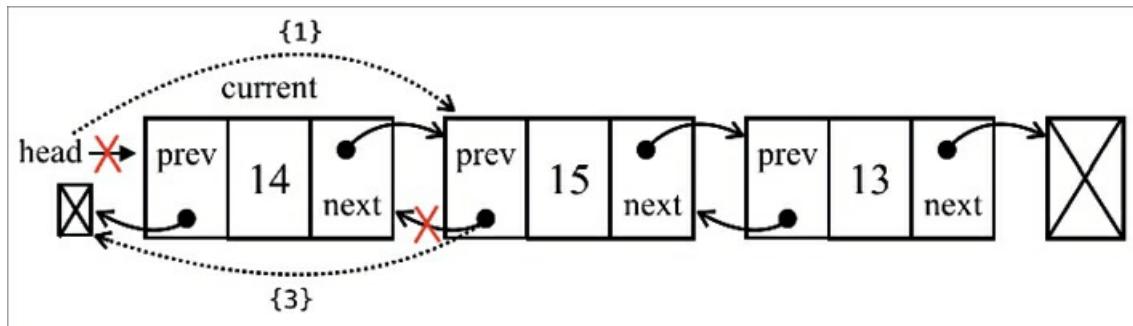


Figura 6.15

No próximo cenário, removeremos um elemento da última posição. Como já temos a referência ao último elemento (`tail`), não precisamos iterar pela lista para alcançá-lo; assim, podemos atribuir a referência de `tail` à variável `current` também (linha {4}). Em seguida, devemos atualizar a referência de `tail` para o penúltimo elemento da lista ({5} – `current.prev` ou `tail.prev` também funcionam). Agora que `tail` aponta para o penúltimo elemento, tudo que precisamos fazer é atualizar o ponteiro de `next` para `undefined` ({6}, isto é, `tail.next = undefined`). O diagrama a seguir (Figura 6.16) mostra essa ação:

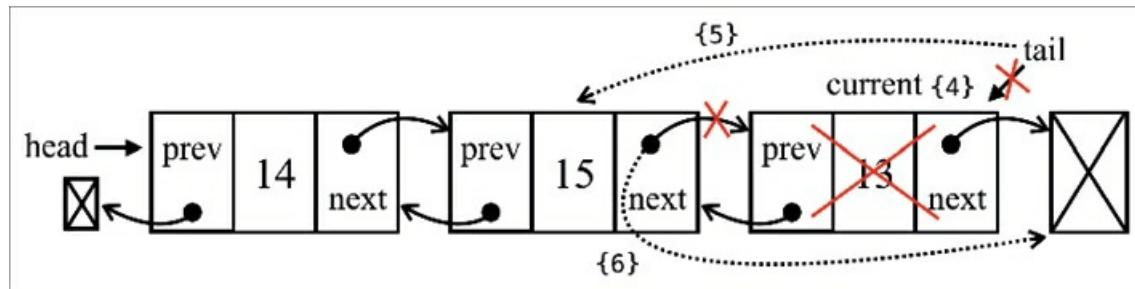


Figura 6.16

O terceiro e último cenário trata a remoção de um elemento do meio da lista. Inicialmente, devemos iterar até alcançarmos a posição desejada ({7}). O elemento que queremos remover será referenciado pela variável `current` ({7}). Assim, para removê-lo, podemos desprezá-lo na lista, atualizando as referências de `previous.next` e `current.next.prev`. Desse modo, `previous.next` apontará para `current.next` ({9}) e `current.next.prev` apontará para `previous` ({10}), conforme mostra o diagrama a seguir (Figura 6.17).

Para verificar a implementação dos demais métodos da lista duplamente ligada, consulte o código-fonte do livro. O link para download desse código-fonte foi mencionado no Prefácio do livro; o código também pode ser acessado em <http://github.com/loiane/javascript-datastructures-algorithms>.

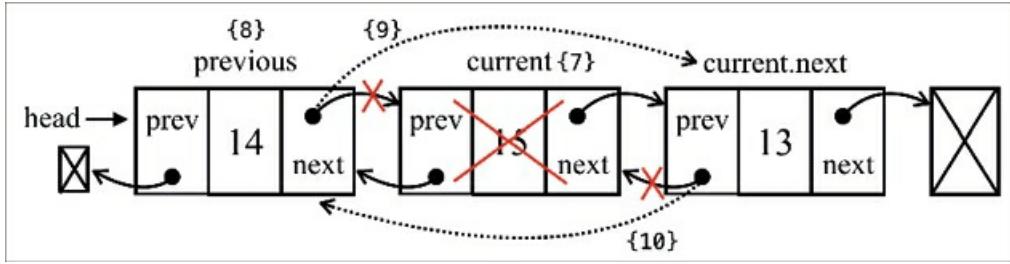


Figura 6.17

Listas ligadas circulares

Uma lista ligada circular pode ter apenas uma direção de referência (como na lista ligada) ou uma referência dupla (como na lista duplamente ligada). A única diferença entre uma lista ligada circular e uma lista ligada é que o ponteiro para o próximo item do último elemento (`tail.next`) não faz uma referência a `undefined`, mas ao primeiro elemento (`head`), como podemos ver no seguinte diagrama:

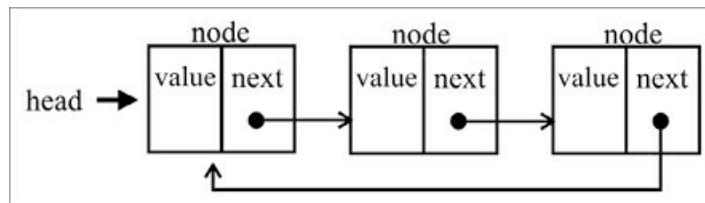


Figura 6.18

Em uma lista circular duplamente ligada, `tail.next` aponta para o elemento `head` e `head.prev` aponta para o elemento `tail`:

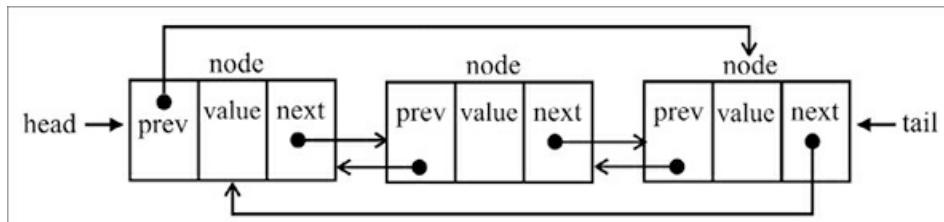


Figura 6.19

Vamos analisar o código que cria a classe `CircularLinkedList`:

```
circularLinkedList extends LinkedList {
    constructor>equalsFn = defaultEquals) {
        super>equalsFn);
    }
}
```

A classe `CircularLinkedList` não precisa de nenhuma propriedade adicional, portanto basta estender a classe `LinkedList` e sobrescrever os métodos necessários para aplicar o comportamento especial.

Sobrescreveremos a implementação dos métodos `insert` e `removeAt` nas próximas seções.

Inserindo um novo elemento em qualquer posição

A lógica para inserir um `element` em uma lista `ligada` circular é a mesma usada para inserir um `element` em uma lista ligada comum. A diferença na lista ligada circular é que precisamos também ligar a referência `next` do último nó ao nó apontado por `head`. A seguir, apresentamos o método `insert` da classe `CircularLinkedList`:

```
insert(element, index) {
    if (index >= 0 && index <= this.count) {
        const node = new Node(element);
        let current = this.head;
        if (index === 0) {
            if (this.head == null) {
                this.head = node; // {1}
                node.next = this.head; // {2} NOVO
            } else {
                node.next = current; // {3}
                current = this.getElementAt(this.size()); // {4}
                // atualiza o último elemento
                this.head = node; // {5}
                current.next = this.head; // {6} NOVO
            }
        } else { // sem alterações neste cenário
            const previous = this.getElementAt(index - 1);
            node.next = previous.next;
            previous.next = node;
        }
        this.count++;
        return true;
    }
    return false;
}
```

Vamos detalhar os diferentes cenários. O primeiro cenário é aquele em que queremos inserir um novo `element` na primeira posição da lista. Se a lista estiver vazia, atribuímos o novo `element` criado a `head` ({1}), como

fizemos na classe `LinkedList`; também precisamos fazer a ligação do último `node` com `head` `{2}`). Nesse caso, o último `element` da lista é o `node` que criamos e que apontará para si mesmo, pois ele é também o `head`. O diagrama a seguir (Figura 6.20) exemplifica o primeiro cenário:

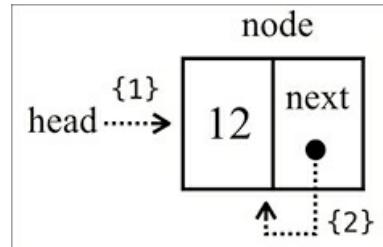


Figura 6.20

O segundo cenário é aquele em que inserimos um `element` na primeira posição em uma lista não vazia, portanto, a primeira tarefa é atribuir a `node.next` a referência de `head` (variável `current`). Essa é a lógica que usamos na classe `LinkedList`. Entretanto, em `CircularLinkedList`, ainda precisamos fazer o último nó da lista apontar para o novo `element` em `head`, portanto temos de obter a referência ao último `element`. Para isso, usaremos o método `getElementAt`, passando o tamanho da lista como parâmetro `{2}`). Atualizamos o `element` em `head` com o novo `element` e fazemos a ligação do último nó (`current`) com o novo `head` `{3}`).

O diagrama a seguir (Figura 6.21) exemplifica o segundo cenário:

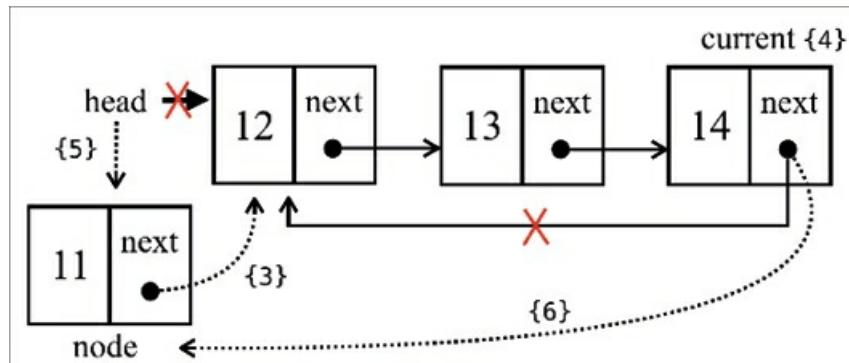


Figura 6.21

Se quisermos inserir um novo `element` no meio da lista, o código será o mesmo da classe `LinkedList`, pois nenhuma alteração será aplicada no último nem no primeiro nó da lista.

Removendo elementos de qualquer posição

Para remover um **element** de uma lista ligada circular, só precisaremos nos preocupar com o segundo cenário, que altera o elemento **head** da lista. Apresentamos a seguir o código do método **removeAt**:

```
removeAt(index) {
    if (index >= 0 && index < this.count) {
        let current = this.head;
        if (index === 0) {
            if (this.size() === 1) {
                this.head = undefined;
            } else {
                const removed = this.head; // {1}
                current = this.getElementAt(this.size()); // {2} NOVO
                this.head = this.head.next; // {3}
                current.next = this.head; // {4}
                current = removed; // {5}
            }
        } else {
            // não há necessidade de atualizar o último elemento da lista circular
            const previous = this.getElementAt(index - 1);
            current = previous.next;
            previous.next = current.next;
        }
        this.count--;
        return current.element; // {6}
    }
    return undefined;
}
```

O primeiro cenário para remover um **element** é a remoção desse elemento de uma lista com um único **node**. Nesse caso, basta atribuir **undefined** a **head**. Não há alterações nesse caso, se compararmos com a classe **LinkedList**.

O segundo cenário consiste em remover o primeiro **element** de uma lista não vazia. Como a referência de **head** mudará, também precisamos atualizar a referência da propriedade **next** do último nó; desse modo, inicialmente manteremos uma referência ao elemento **head** atual, que será removido da lista ({1}). Como fizemos no método de inserção, também será necessário obter a referência ao último nó da lista ({2}), que será armazenado na variável **current**. Depois que tivermos uma referência a todos os nós necessários, podemos começar a fazer as novas ligações.

Atualizamos o `element` em `head`, ligando `head` ao segundo `element` (`head.next`, em {3}), e então fazemos a ligação entre o último `element` (`current.next`) e o novo `head` ({4}). Podemos atualizar a referência à variável `current` ({5}) para que possamos devolver o seu valor ({6}) para informação.

O diagrama a seguir exemplifica essas ações:

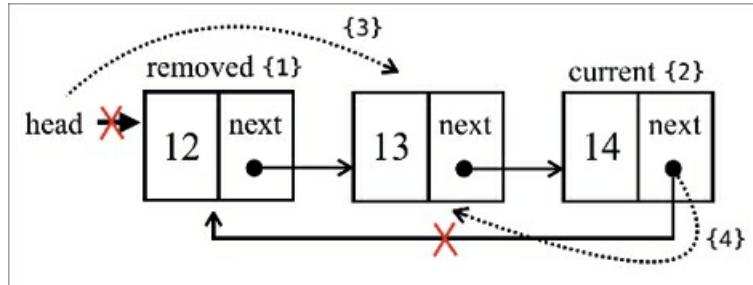


Figura 6.22

Listas ligadas ordenadas

Uma **lista ligada ordenada** é uma lista que mantém seus elementos ordenados. Para manter todos os elementos ordenados, em vez de aplicar um algoritmo de ordenação, inseriremos `element` em sua posição correta a fim de manter a lista sempre ordenada.

Vamos começar declarando a classe `SortedLinkedList`:

```
const Compare = {
  LESS_THAN: -1,
  BIGGER_THAN: 1
};
function defaultCompare(a, b) {
  if (a === b) { // {1}
    return 0;
  }
  return a < b ? Compare.LESS_THAN : Compare.BIGGER_THAN; // {2}
}
class SortedLinkedList extends LinkedList {
  constructor>equalsFn = defaultEquals, compareFn = defaultCompare) {
    super>equalsFn);
    this.compareFn = compareFn; // {3}
  }
}
```

A classe `SortedLinkedList` herdará todas as propriedades e os métodos da

classe `LinkedList`, mas, como essa classe tem um comportamento especial, precisaremos de uma função para comparar os elementos. Por esse motivo, também temos de declarar `compareFn` ({3}), que será usado para comparar elementos. Essa função usará a função `defaultCompare` por padrão. Se os elementos tiverem a mesma referência, ela devolverá 0 ({1}). Se o primeiro elemento for menor que o segundo, ela devolverá -1; caso contrário, devolverá 1. Para deixar o código mais elegante, podemos declarar uma constante `Compare` para representar cada um desses valores. Se o `element` sendo comparado for mais complexo, podemos criar uma função de comparação personalizada e passá-la também para o construtor da classe `SortedLinkedList`.

Inserindo elementos na ordem

Sobrescreveremos o método `insert` com o código apresentado a seguir:

```
insert(element, index = 0) { // {1}
  if (this.isEmpty()) {
    return super.insert(element, 0); // {2}
  }
  const pos = this.getIndexNextSortedElement(element); // {3}
  return super.insert(element, pos); // {4}
}
getIndexNextSortedElement(element) {
  let current = this.head;
  let i = 0;
  for (; i < this.size() && current; i++) {
    const comp = this.compareFn(element, current.element); // {5}
    if (comp === Compare.LESS_THAN) { // {6}
      return i;
    }
    current = current.next;
  }
  return i; // {7}
}
```

Como não queremos permitir a inserção de elementos em qualquer índice, começaremos atribuindo um valor default ao parâmetro `index` ({1}); assim, podemos simplesmente chamar `list.insert(myElement)`, sem a necessidade de passar o parâmetro `index`. Se o parâmetro `index` for passado para o método, seu valor será ignorado, pois a posição em que `element` vai ser inserido será controlada internamente. O motivo para

termos de fazer isso dessa forma é porque não queremos reescrever todos os métodos da classe `LinkedList` novamente; desse modo, sobrescreveremos somente o comportamento do método `insert`. Se quiser, você poderá criar uma classe `SortedLinkedList` do zero, copiando todo o código da classe `LinkedList`; no entanto, isso dificultará um pouco mais a manutenção do código, uma vez que teremos de fazer alterações em duas classes, em vez de uma só.

Se a lista estiver vazia, basta chamar o método `insert` de `LinkedList` passando 0 (zero) como `index` ({2}). Se a lista não estiver vazia, temos de obter o que é conhecido como o `index` correto para inserir `element` ({3}), e chamaremos o método `insert` de `LinkedList`, passando a posição a fim de manter a lista ordenada ({4}).

Para obter o índice correto a fim de inserir `element`, criamos um método chamado `getIndexNextSortedElement`. Nesse método, iteramos pela lista até encontrar uma posição para inserir `element` ou até que todos os elementos tenham `sido percorridos`. Nesse último cenário, o `index` devolvido ({7}) será o tamanho (`size`) da lista (`element` será inserido no final da lista). Para comparar os elementos, usaremos `compareFn` ({5}), passado para o construtor da classe. Quando o `element` que queremos inserir na lista for menor que o elemento `current` da `lista`, teremos encontrado a posição para a inserção ({6}).

É isso! Podemos reutilizar o método `insert` de `LinkedList` internamente. Todos os demais métodos, como `remove`, `indexOf` e `on` serão os mesmos de `LinkedList`.

Criando a classe `StackLinkedList`

Também podemos usar a classe `LinkedList` e suas variantes como estruturas de dados internas a fim de criar outras estruturas de dados como `pilha`, `fila` e `deque`. Nesta seção, veremos como criar a estrutura de dados de `pilha` (discutida no Capítulo 4, *Pilhas*).

A estrutura da classe `StackLinkedList` e os métodos `push` e `pop` são declarados assim:

```
class StackLinkedList {
    constructor() {
        this.items = new DoublyLinkedList(); // {1}
```

```

    }
    push(element) {
      this.items.push(element); // {2}
    }
    pop() {
      if (this.isEmpty()) {
        return undefined;
      }
      return this.items.removeAt(this.size() - 1); // {3}
    }
  }
}

```

Na classe **StackLinkedList**, em vez de usar um array ou um objeto JavaScript para armazenar **items**, usamos uma **DoublyLinkedList** ({1}). O motivo para usar uma lista duplamente ligada no lugar de uma lista ligada é que, para a pilha, os elementos serão inseridos no final da lista ({2}) e a remoção também será no final ({3}). Nossa classe **DoublyLinkedList** mantém uma referência ao último **element** da lista (**tail**), portanto não é necessário iterar por todos os elementos da lista para acessá-lo; há um acesso direto ao primeiro e ao último elementos, reduzindo o esforço de processamento e mantendo o custo em **O(1)**, como em nosso implementação original da **Stack**.

Poderíamos também melhorar a classe **LinkedList**, mantendo uma referência ao **element** em **tail** e usando essa versão aperfeiçoada no lugar de **DoublyLinkedList**.

Podemos inspecionar o código dos outros métodos de **Stack**, conforme apresentados a seguir:

```

peek() {
  if (this.isEmpty()) {
    return undefined;
  }
  return this.items.getElementAt(this.size() - 1).element;
}
isEmpty() {
  return this.items.isEmpty();
}
size() {
  return this.items.size();
}
clear() {
  this.items.clear();
}

```

```
toString() {  
    return this.items.toString();  
}
```

Estamos chamando os métodos da classe `DoublyLinkedList` para todos os demais métodos. Usar a estrutura de dados de lista ligada internamente na implementação de pilha é mais fácil, pois não precisamos criar o código do zero, mantendo o custo de processamento e deixando o código mais legível!

Podemos aplicar a mesma lógica e criar uma classe `Queue` e uma classe `Deque` usando `DoublyLinkedList`, ou até mesmo a classe `LinkedList`!

Resumo

Neste capítulo, conhecemos a estrutura de dados da lista ligada e suas variantes: a lista duplamente ligada, a lista ligada circular e a lista ligada ordenada. Vimos como adicionar e remover elementos de qualquer posição e como iterar por uma lista ligada. Vimos também que a principal vantagem de uma lista ligada em relação a um array está no fato de ser possível adicionar e remover elementos facilmente da lista, sem precisar deslocar seus elementos. Assim, sempre que precisar adicionar e remover muitos elementos, a melhor opção será usar uma lista ligada no lugar de um array.

Aprendemos também a criar uma pilha usando uma lista ligada interna para armazenar seus elementos, em vez de usar um array ou um objeto, e conhecemos também as vantagens de usar outra estrutura de dados para tirar proveito das operações disponíveis, sem ser necessário escrever toda a lógica do zero.

No próximo capítulo, conheceremos os conjuntos: uma estrutura de dados que armazena elementos únicos.

CAPÍTULO 7

Conjuntos

Estruturas de dados sequenciais como os arrays (listas), as pilhas, as filas e as listas ligadas, a essa altura, devem soar familiares a você. Neste capítulo, descreveremos uma estrutura de dados chamada conjuntos (sets) – que também é uma estrutura de dados sequencial, porém não permite valores duplicados. Veremos como criar uma estrutura de dados para conjuntos, como adicionar e remover valores, além de fazer buscas para saber se um valor já existe. Você aprenderá também a realizar operações matemáticas como união, intersecção e diferença. Por fim, veremos como usar a classe **Set** nativa da ECMAScript 2015 (ES2015).

Neste capítulo, veremos:

- como criar uma classe **Set** do zero;
- como realizar operações matemáticas com um **Set**;
- a classe **Set** nativa da ECMAScript 2015.

Estruturando um conjunto de dados

Um conjunto (**set**) é uma coleção não ordenada de itens, composta de elementos únicos (isto é, que não podem ser repetidos). Essa estrutura de dados usa o mesmo conceito matemático dos conjuntos finitos, porém aplicado a uma estrutura de dados em ciência da computação.

Vamos analisar o conceito matemático de conjuntos antes de explorar a sua implementação em ciência da computação. Em matemática, um conjunto é uma coleção de objetos distintos.

Por exemplo, temos o conjunto de números naturais, que é composto dos números inteiros maiores ou iguais a 0 – isto é, $N = \{0, 1, 2, 3, 4, 5, 6, \dots\}$. A lista de objetos no conjunto deve estar entre {} (chaves).

Temos também o conceito de conjunto nulo. Um conjunto sem elementos é chamado de conjunto **nulo** ou **conjunto vazio**. Um exemplo seria o conjunto dos números primos entre 24 e 29. Como não há nenhum

número primo (um número natural maior que 1 sem nenhum divisor positivo que não seja 1 e o próprio número) entre 24 e 29, o conjunto será vazio. Representaremos um conjunto vazio com { }.

Você também pode pensar em um conjunto como um array sem elementos repetidos e sem o conceito de ordem.

Em matemática, um conjunto também apresenta algumas operações básicas como união, intersecção e diferença. Essas operações serão discutidas neste capítulo.

Criando uma classe Set

A ECMAScript 2015 introduziu a classe **Set** como parte da API JavaScript, e você aprenderá a usá-la mais adiante neste capítulo. Faremos a nossa própria implementação da classe **Set** baseada na classe **Set** da **ES2015**. Implementaremos também algumas operações de conjunto como união, intersecção e diferença, que não estão presentes na classe **ES2015** nativa.

Para começar, apresentamos a seguir a declaração da classe **Set** com o seu construtor:

```
class Set {
  constructor() {
    this.items = {};
  }
}
```

Um detalhe muito importante nesse caso é que estamos usando um objeto para representar o nosso conjunto (**items**), em vez de utilizar um array. No entanto, poderíamos também ter usado um array nessa implementação. A abordagem que adotaremos neste capítulo é bem semelhante àquela com o *objeto items*, usada no Capítulo 4, *Pilhas*, e no Capítulo 5, *Filas e deque*s. Além do mais, os objetos em JavaScript não permitem que haja duas propriedades diferentes na mesma chave, o que garante que os elementos em nosso conjunto sejam únicos.

Em seguida, devemos declarar os métodos disponíveis a um conjunto (tentaremos simular a mesma classe **Set** implementada na ECMAScript 2015):

- **add(element)**: adiciona um novo **element** ao conjunto.
- **delete(element)**: remove **element** do conjunto.

- **has(element)**: devolve `true` se `element` estiver no conjunto, e `false` caso contrário.
- **clear()**: remove todos os elementos do conjunto.
- **size()**: devolve quantos elementos estão contidos no conjunto. É semelhante à propriedade `length` de um array.
- **values()**: devolve um array com todos os valores (elementos) que estão no conjunto.

Método `has(element)`

O primeiro método que implementaremos é o método `has(element)`, que será implementado antes porque vai ser usado em outros métodos, como `add` e `remove`, para verificar se o elemento já está no conjunto. Podemos observar a sua implementação a seguir:

```
has(element){
  return element in items;
};
```

Como estamos usando um objeto para armazenar todos os valores do conjunto, o operador `in` de JavaScript pode ser utilizado para verificar se o elemento especificado é uma propriedade do objeto `items`.

No entanto, há uma maneira melhor de implementar esse método, que é a seguinte:

```
has(element) {
  return Object.prototype.hasOwnProperty.call(this.items, element);
}
```

O protótipo de `Object` tem o método `hasOwnProperty`, que devolve um booleano informando se o objeto tem a propriedade especificada diretamente no objeto ou não, enquanto o operador `in` devolve um booleano informando se o objeto tem a propriedade especificada na cadeia do objeto.

Poderíamos também ter usado `this.items.hasOwnProperty(element)` em nosso código. No entanto, algumas ferramentas `lint`, como o `ESLint` (<https://eslint.org>), lançam um erro se tentarmos usar esse código. O erro ocorre porque nem todos os objetos herdam de `Object.prototype`, e, mesmo para os objetos que herdam de

`Object.prototype`, o método `hasOwnProperty` poderia ser encoberto por algo diferente e o código poderia não funcionar. Para evitar qualquer problema, é mais seguro usar `Object.prototype.hasOwnProperty.call`.

Método add

O próximo método que implementaremos é o método **add**, com o código a seguir:

```
add(element) {  
    if (!this.has(element)) {  
        this.items[element] = element; // {1}  
        return true;  
    }  
    return false;  
}
```

Dado um `element`, podemos verificar se ele já está presente no conjunto. Se não estiver, adicionaremos `element` ao conjunto (`{1}`) e devolveremos `true` para informar que ele foi adicionado. Se o elemento já estiver no conjunto, simplesmente devolveremos `false` para informar que não foi adicionado.

Estamos adicionando `element` como a chave e o valor porque, se o armazenarmos como a chave também, isso nos ajudará a procurá-lo.

Métodos delete e clear

A seguir, implementaremos o método **delete**:

```
delete(element) {  
    if (this.has(element)) {  
        delete this.items[element]; // {1}  
        return true;  
    }  
    return false;  
}
```

No método **delete**, verificaremos se o `element` especificado está presente no conjunto. Em caso afirmativo, removemos `element` do conjunto (`{1}`) e devolvemos `true` para informar que ele foi removido; caso contrário, devolvemos `false`.

Como estamos usando um objeto `items` para armazenar o objeto que

representa o conjunto, podemos simplesmente utilizar o operador **delete** para remover a propriedade desse objeto (**{1}**).

Para usar a classe **Set**, podemos executar o código a seguir como exemplo:

```
const set = new Set();
set.add(1);
set.add(2);
```

Somente por curiosidade, se exibirmos a propriedade **this.items** no console (**console.log**) depois de executar o código anterior, esta será a saída no Google Chrome:

```
Object {1: 1, 2: 2}
```

Como podemos observar, esse é um objeto com duas propriedades. O nome da propriedade é o valor que adicionamos no conjunto, e é também o seu valor.

Se quisermos remover todos os valores do conjunto, o método **clear** poderá ser usado, cujo código é:

```
clear() {
  this.items = {};
}
```

Tudo que precisamos fazer para reiniciar o objeto **items** é atribuir-lhe um objeto vazio novamente (**{2}**). Também poderíamos iterar pelo conjunto e remover todos os elementos, um a um, usando o método **remove**, porém seria muito trabalhoso, e temos uma forma mais fácil de fazer isso.

Método **size**

O próximo método que implementaremos é o método **size** (que devolve a quantidade de elementos presente no conjunto). Há três maneiras de implementar esse método.

O primeiro método consiste em usar uma variável **length** e controlá-la sempre que os métodos **add** ou **remove** forem usados, como fizemos nas classes **LinkedList**, **Stack** e **Queue** em capítulos anteriores.

No segundo método, usamos uma função embutida da classe **Object** de JavaScript (ECMAScript 2015+), assim:

```
size() {
  return Object.keys(this.items).length; // {1}
};
```

A classe **Object** de JavaScript contém um método chamado **keys** que devolve um array com todas as propriedades de um dado objeto. Nesse caso, podemos usar a propriedade **length** desse array (**{1}**) para devolver a quantidade de propriedades existente no objeto **items**. Esse código funcionará somente nos navegadores modernos (como IE9+, FF4+, Chrome5+, Opera12+, Safari5+ e assim por diante).

O terceiro método consiste em extrair cada propriedade do objeto **items** manualmente, contar quantas propriedades há e devolver esse número. Esse método funcionará em qualquer navegador e é equivalente ao código anterior, conforme vemos a seguir:

```
sizeLegacy() {
  let count = 0;
  for(let key in this.items) { // {2}
    if(this.items.hasOwnProperty(key)) { // {3}
      count++; // {4}
    }
  }
  return count;
};
```

Inicialmente iteramos por todas as propriedades do objeto **items** (**{2}**) e verificamos se essa propriedade é realmente uma propriedade de nosso objeto (para que ele não seja contado mais de uma vez – **{3}**). Em caso afirmativo, incrementamos a variável **count** (**{4}**) e, no final do método, devolvemos esse número.

Não podemos simplesmente usar a instrução **for-in**, iterar pelas propriedades do objeto **items** e incrementar o valor da variável **count**. Também precisamos usar o método **hasOwnProperty** (para conferir se o objeto **items** tem essa propriedade) porque o protótipo do objeto contém propriedades adicionais (herdadas da classe-base **Object** de JavaScript, que não são usadas nessa estrutura de dados).

Método **values**

Para implementar o método **values**, podemos usar também um método embutido da classe **Object** chamado **values**, assim:

```
values() {
  return Object.values(this.items);
```

}

O método `Object.values()` devolve um array com os valores de todas as propriedades de um dado objeto. Foi acrescentado na ECMAScript 2017 e está disponível somente nos navegadores modernos.

Se quisermos ter um código que possa ser executado em qualquer navegador, podemos usar o código a seguir, que é equivalente ao anterior:

```
valuesLegacy() {  
    let values = [];  
    for(let key in this.items) { // {1}  
        if(this.items.hasOwnProperty(key)) { // {2}  
            values.push(key);  
        }  
    }  
    return values;  
};
```

Desse modo, inicialmente iteramos por todas as propriedades do objeto `items` ({1}), adicionamos essas propriedades em um array ({2}) e o devolvemos. O método é semelhante ao método `sizeLegacy` que desenvolvemos, mas, em vez de contar as propriedades, elas são adicionadas em um array.

Usando a classe Set

Agora que concluímos a implementação de nossa estrutura de dados para conjunto, veremos como ela pode ser usada. Vamos testar o código executando alguns comandos a fim de testar a nossa classe `Set`, assim:

```
const set = new Set();  
set.add(1);  
console.log(set.values()); // exibe [1]  
console.log(set.has(1)); // exibe true  
console.log(set.size()); // exibe 1  
set.add(2);  
console.log(set.values()); // exibe [1, 2]  
console.log(set.has(2)); // true  
console.log(set.size()); // 2  
set.delete(1);  
console.log(set.values()); // exibe [2]  
set.delete(2);  
console.log(set.values()); // exibe []
```

Agora temos então uma implementação muito semelhante à classe `Set` da

ECMAScript 2015.

Operações em conjuntos

O conjunto é um dos conceitos mais básicos em matemática e é muito importante também em ciência da computação. Uma das principais aplicações em ciência da computação se dá em **bancos de dados**, que estão na base da maioria das aplicações. Os conjuntos são usados no design e no processamento de consultas (queries). Ao criar uma consulta para obter um conjunto de dados em um banco de dados relacional (Oracle, Microsoft SQL Server, MySQL e assim por diante), fazemos o design da consulta usando a notação de conjunto, e o banco de dados também devolverá um conjunto de dados. Quando criamos uma consulta SQL, podemos especificar se queremos ler todos os dados de uma tabela ou apenas um subconjunto deles. Também podemos obter dados que são comuns a duas tabelas, os quais estão presentes apenas em uma tabela (e não na outra) ou nas duas (entre outras operações). Essas operações são conhecidas no mundo SQL como junções (joins), e a base das **junções SQL** são as operações em conjuntos.

Para saber mais sobre as operações de junção SQL, acesse <http://www.sql-join.com/sql-join-types>.

Podemos realizar as seguintes operações em conjuntos:

- **União**: dados dois conjuntos, devolve um novo conjunto com elementos dos dois conjuntos especificados.
- **Intersecção**: dados dois conjuntos, devolve um novo conjunto com os elementos presentes em ambos os conjuntos.
- **Diferença**: dados dois conjuntos, devolve um novo conjunto com todos os elementos presentes no primeiro conjunto, mas não no segundo.
- **Subconjunto**: confirma se um dado conjunto é um subconjunto de outro.

União de conjuntos

Nesta seção, discutiremos o conceito matemático de união. A união dos conjuntos **A** e **B** é representada por:

$$A \cup B$$

Figura 7.1

A união é definida assim:

$$A \cup B = x | x \in A \vee x \in B$$

Figura 7.2

Isso significa que **x** (o elemento) está presente em **A** ou **x** está presente em **B**. O diagrama a seguir exemplifica a operação de união:

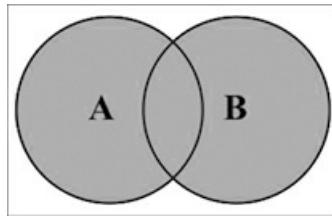


Figura 7.3

Vamos agora implementar o método **union** em nossa classe **Set** com o código a seguir:

```
union(otherSet) {
  const unionSet = new Set(); // {1}
  this.values().forEach(value => unionSet.add(value)); // {2}
  otherSet.values().forEach(value => unionSet.add(value)); // {3}
  return unionSet;
}
```

Inicialmente devemos criar outro conjunto para representar a união dos dois conjuntos ({1}). Em seguida, acessamos todos os **values** do primeiro conjunto (a instância atual da classe **Set**), iteramos por eles e adicionamos todos os valores no conjunto que representa a **union** ({2}). Então fazemos exatamente o mesmo, mas com o segundo conjunto ({3}). Por fim, devolvemos o resultado.

Como o método **values** que implementamos devolve um array, podemos usar o método **forEach** da classe **Array** para iterar por todos os elementos do array. Somente para lembrar, o método **forEach** foi introduzido na ECMAScript 2015. Esse método recebe um parâmetro (**value**) que representa cada valor do array, e tem também uma função de callback que executa uma lógica de programação. No código

anterior, estamos usando também funções de seta (`=>`) em vez de declarar explicitamente `function(value) { unionSet.add(value) }`. O código parece moderno e sucinto com as funcionalidades da ECMAScript que vimos no Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*.

Também podemos escrever o método `union` conforme vemos a seguir, sem o método `forEach` e as funções de seta, mas, sempre que possível, tentaremos usar as funcionalidades das versões ES2015+:

```
union(otherSet) {
  const unionSet = new Set(); // {1}
  let values = this.values(); // {2}
  for (let i = 0; i < values.length; i++){
    unionSet.add(values[i]);
  }
  values = otherSet.values(); // {3}
  for (let i = 0; i < values.length; i++){
    unionSet.add(values[i]);
  }
  return unionSet;
};
```

Vamos testar o código anterior da seguinte maneira:

```
const setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);
const setB = new Set();
setB.add(3);
setB.add(4);
setB.add(5);
setB.add(6);
const unionAB = setA.union(setB);
console.log(unionAB.values());
```

O resultado será `[1, 2, 3, 4, 5, 6]`. Observe que o elemento `3` está presente tanto em `setA` quanto em `setB`, mas aparece somente uma vez no conjunto resultante.

É importante observar que os métodos `union`, `intersection` e `difference` que estamos implementando neste capítulo não modificam a instância atual da classe `Set` nem o `otherSet` passado como parâmetro. Os métodos ou funções que não têm efeito colateral são chamados de funções puras. Uma função pura não modifica a instância atual nem os

parâmetros: ela apenas gera um novo resultado. Esse é um conceito muito importante do paradigma de programação funcional, que será apresentado mais adiante no livro.

Intersecção de conjuntos

Nesta seção, discutiremos o conceito matemático de intersecção. A intersecção entre os conjuntos **A** e **B** é representada por:

$$A \cap B$$

Figura 7.4

Eis a sua definição:

$$A \cap B = x | x \in A \wedge x \in B$$

Figura 7.5

Isso significa que **x** (o elemento) está presente tanto em **A** quanto em **B**, havendo, assim, um compartilhamento de um ou mais elementos entre **A** e **B**. O diagrama a seguir (Figura 7.6) exemplifica a operação de intersecção:

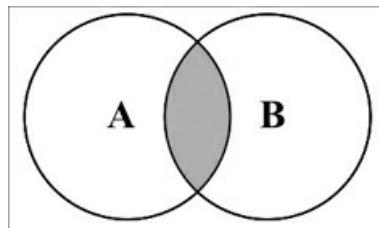


Figura 7.6

Vamos agora implementar o método `intersection` em nossa classe `Set`, deste modo:

```
intersection(otherSet) {
  const intersectionSet = new Set(); // {1}
  const values = this.values();
  for (let i = 0; i < values.length; i++) { // {2}
    if (otherSet.has(values[i])) { // {3}
      intersectionSet.add(values[i]); // {4}
    }
  }
  return intersectionSet;
}
```

No método `intersection`, devemos encontrar todos os elementos da

instância atual da classe `Set` que também estejam presentes na instância de `Set` especificada (`otherSet`). Assim, em primeiro lugar, criamos outra instância de `Set` para que possamos devolvê-la com os elementos comuns (`{1}`). Em seguida, iteramos por todos os valores (`values`) da instância atual da classe `Set` (`{2}`) e verificamos se o valor está presente na instância `otherSet` também (`{3}`). Podemos usar o método `has`, que implementamos antes neste capítulo, a fim de verificar se o elemento está presente na instância de `Set`. Então, se o valor estiver presente também na outra instância, nós o adicionamos à variável `intersectionSet` criada (`{4}`), e a devolvemos.

Vamos fazer alguns testes, assim:

```
const setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);
const setB = new Set();
setB.add(2);
setB.add(3);
setB.add(4);
const intersectionAB = setA.intersection(setB);
console.log(intersectionAB.values());
```

O resultado será `[2, 3]`, pois os valores `2` e `3` estão presentes nos dois conjuntos.

Aperfeiçoando o método `intersection`

Suponha que tenhamos os dois conjuntos a seguir:

- `setA` com valores `[1, 2, 3, 4, 5, 6, 7]`
- `setB` com valores `[4, 6]`

Usando o método `intersection` que criamos, precisaríamos iterar pelos valores de `setA` sete vezes, que é o número de elementos nesse conjunto, e comparar esses sete valores somente com os dois elementos de `setB`. Seria melhor se tivéssemos o mesmo resultado e tivéssemos de iterar somente duas vezes por `setB`. Menos iterações significa um custo menor de processamento, portanto vamos otimizar o nosso código a fim de iterar pelo conjunto com menos elementos, assim:

```
intersection(otherSet) {
  const intersectionSet = new Set(); // {1}
```

```

const values = this.values(); // {2}
const otherValues = otherSet.values(); // {3}
let biggerSet = values; // {4}
let smallerSet = otherValues; // {5}
if (otherValues.length - values.length > 0) { // {6}
  biggerSet = otherValues;
  smallerSet = values;
}
smallerSet.forEach(value => { // {7}
  if (biggerSet.includes(value)) {
    intersectionSet.add(value);
  }
});
return intersectionSet;
}

```

Desse modo, inicialmente criaremos outro conjunto para armazenar o resultado de nossa `intersection` ({1}). Vamos obter também os valores da instância atual do conjunto ({2}) e do conjunto especificado, passado como parâmetro para o método `intersection` ({3}). Em seguida, supomos que a instância atual é o conjunto com mais elementos ({4}), e o conjunto especificado é o conjunto com menos elementos ({5}). Comparamos o tamanho dos dois conjuntos ({6}) e, caso o conjunto especificado tenha mais elementos que a instância atual, trocamos os valores de `biggerSet` e `smallerSet`. Por fim, iteramos pelo conjunto menor ({7}) para calcular os valores comuns entre os dois conjuntos e devolvemos o resultado.

Diferença entre conjuntos

Nesta seção, discutiremos o conceito matemático de diferença. A diferença entre os conjuntos **A** e **B** é representada por $A - B$, sendo definida como:

$$A - B = \{x | x \in A \wedge x \notin B\}$$

Figura 7.7

Isso significa que **x** (o elemento) está presente em **A**, mas **x** não está presente em **B**. O diagrama a seguir exemplifica a operação de diferença entre os conjuntos **A** e **B**:

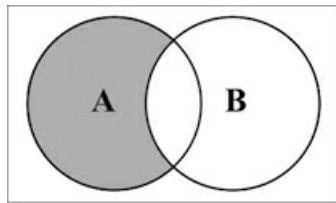


Figura 7.8

Vamos agora implementar o método **difference** em nossa classe **Set** usando o código a seguir:

```
difference(otherSet) {
  const differenceSet = new Set(); // {1}
  this.values().forEach(value => { // {2}
    if (!otherSet.has(value)) { // {3}
      differenceSet.add(value); // {4}
    }
  });
  return differenceSet;
}
```

O método **intersection** devolverá todos os elementos presentes nos dois conjuntos. O método **difference** devolverá todos os elementos que estão presentes em **A**, mas não em **B**. Inicialmente criaremos o nosso conjunto de resultados (**{1}**), pois não queremos modificar o conjunto atual nem o conjunto especificado. Em seguida, fazemos uma iteração por todos os valores da instância do conjunto atual (**{2}**). Verificamos se **value** (o elemento) está presente no conjunto especificado (**{3}**); se ele não estiver em **otherSet**, adicionamos **value** no conjunto resultante.

Vamos fazer alguns testes (com os mesmos conjuntos que usamos na seção sobre **intersection**):

```
const setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);
const setB = new Set();
setB.add(2);
setB.add(3);
setB.add(4);
const differenceAB = setA.difference(setB);
console.log(differenceAB.values());
```

O resultado será **[1]** porque 1 é o único elemento que está presente apenas em **setA**. Se executarmos **setB.difference(setA)**, teremos **[4]** como resultado porque 4 é o único elemento que está presente somente em **setB**.

Não podemos otimizar o método `difference` como fizemos com o método `intersection`, pois a diferença entre `setA` e `setB` pode ser diferente da diferença entre `setB` e `setA`.

Subconjunto

A última operação de conjunto que discutiremos é o subconjunto. Um exemplo do conceito matemático de subconjunto é dizer que **A** é um subconjunto de (ou está incluído em) **B**, e isso é representado por:

$$A \subseteq B$$

Figura 7.9

O conjunto é definido assim:

$$\forall x x \in A \Rightarrow x \in B$$

Figura 7.10

Isso significa que, para todo **x** (elemento) que estiver em **A**, ele também *deve estar presente* em **B**. O diagrama a seguir exemplifica o caso em que **A** é um subconjunto de **B**:

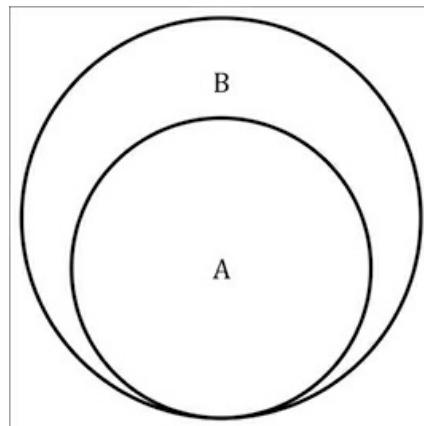


Figura 7.11

Vamos agora criar o método `isSubsetOf` em nossa classe **Set** usando o código a seguir:

```
isSubsetOf(otherSet) {  
    if (this.size() > otherSet.size()) { // {1}  
        return false;  
    }
```

```

let isSubset = true; // {2}
this.values().every(value => { // {3}
  if (!otherSet.has(value)) { // {4}
    isSubset = false; // {5}
    return false;
  }
  return true; // {6}
});
return isSubset; // {7}
}

```

A primeira verificação que precisamos fazer é conferir o tamanho da instância atual da classe **Set**. Se a instância atual tiver mais elementos que a instância **otherSet**, ela não será um subconjunto ({1}). Um subconjunto deve ter um número menor ou igual de elementos que o conjunto com o qual está sendo comparado.

Em seguida, supomos que a instância atual é um subconjunto do conjunto especificado ({2}). Iteramos por todos os elementos do conjunto atual ({3}) e verificamos se o elemento também está presente em **otherSet** ({4}). Se houver algum elemento que não esteja presente em **otherSet**, é sinal de que esse conjunto não é um subconjunto, portanto devolveremos **false** ({5}). Se todos os elementos estiverem também presentes em **otherSet**, a linha {5} não será executada e devolveremos **true** ({7}), pois a flag **isSubset** não será alterada.

Em **isSubsetMethod**, não usamos o método **forEach**, utilizado em **union**, **intersection** e **difference**. Estamos usando o método **every**, que também faz parte da classe **Array** de JavaScript, e foi introduzido na ES2015. No Capítulo 3, *Arrays*, vimos que o método **forEach** é chamado para cada **value** do array. No caso da lógica de subconjuntos, podemos parar a iteração em **values** se encontrarmos um **value** que não esteja em **otherSet**, o que significa que a instância atual não é um subconjunto. O método **every** será chamado enquanto a função de callback devolver **true** ({6}). Se a função de callback devolver **false**, o laço será interrompido, e é por isso que estamos também alterando o valor da flag **isSubset** na linha {5}.

Vamos testar o código anterior:

```

const setA = new Set();
setA.add(1);
setA.add(2);

```

```
const setB = new Set();
setB.add(1);
setB.add(2);
setB.add(3);
const setC = new Set();
setC.add(2);
setC.add(3);
setC.add(4);
console.log(setA.isSubsetOf(setB));
console.log(setA.isSubsetOf(setC));
```

Temos três conjuntos: `setA` é um subconjunto de `setB` (portanto, a saída é `true`); no entanto, `setA` não é subconjunto de `setC` (`setC` contém apenas o valor 2 de `setA`, e não os valores 1 e 2), portanto o resultado será `false`.

ECMAScript 2015 – a classe Set

A ECMAScript 2015 introduziu a classe `Set` como parte da API de JavaScript. Desenvolvemos a nossa classe `Set` com base na classe `Set` da ES2015.

Você pode ver os detalhes da implementação da classe `Set` da ECMAScript 2015 em https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set (ou em <http://goo.gl/2li2a5>).

Vamos agora observar como podemos usar também a classe `Set` nativa.

Utilizaremos os mesmos exemplos que usamos para testar a nossa classe `Set`, assim:

```
const set = new Set();
set.add(1);
console.log(set.values()); // exibe @Iterator
console.log(set.has(1)); // exibe true
console.log(set.size); // exibe 1
```

A diferença entre a nossa classe `Set` e a classe `Set` da ES2015 é que o método `values` devolve um `Iterator` (que conhecemos no Capítulo 3, *Arrays*) em vez de devolver o array com os valores. Outra diferença está no fato de termos desenvolvido um método `size` que devolve o número de valores armazenados em `Set`. A classe `Set` da ES2015 tem uma propriedade chamada `size`.

Também podemos chamar o método `delete` para remover um elemento de

set usando o código a seguir:

```
set.delete(1);
```

O método **clear** também reinicia a estrutura de dados de **Set**. Ele tem a mesma funcionalidade que nós implementamos.

Operações com a classe Set da ES2015

Desenvolvemos as operações matemáticas como união, intersecção, diferença e subconjunto em nossa classe **Set**. Infelizmente, a classe **Set** nativa da ES2015 não contém essas funcionalidades. No entanto, podemos escrever nossas próprias funções com funcionalidades semelhantes caso seja necessário.

Usaremos os dois conjuntos a seguir em nossos exemplos:

```
const setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);
const setB = new Set();
setB.add(2);
setB.add(3);
setB.add(4);
```

Simulando a operação de união

Para somar todos os elementos dos dois conjuntos, podemos criar uma função que devolverá um novo conjunto com todos os elementos de **set1** e de **set2**. Devemos iterar por **set1** (`{1}`) e por **set2** (`{2}`) e adicionar todos os elementos no conjunto de união usando **add**, como vemos no código a seguir:

```
const union = (set1, set2) => {
  const unionAb = new Set();
  set1.forEach(value => unionAb.add(value));
  set2.forEach(value => unionAb.add(value));
  return unionAb;
};
console.log(union(setA, setB)); // {1, 2, 3, 4}
```

Simulando a operação de intersecção

Para simular a operação de intersecção, podemos também implementar

uma função para nos ajudar a criar outro conjunto com os elementos comuns presentes tanto em `setA` quanto em `setB`, conforme mostrado a seguir:

```
const intersection = (set1, set2) => {
  const intersectionSet = new Set();
  set1.forEach(value => {
    if (set2.has(value)) {
      intersectionSet.add(value);
    }
  });
  return intersectionSet;
};
console.log(intersection(setA, setB)); // {2, 3}
```

O código anterior faz o mesmo que a função `intersection` que desenvolvemos, porém não está otimizado (pois desenvolvemos também uma versão otimizada).

Simulando a operação de diferença

Enquanto a operação de intersecção é implementada com a criação de um novo conjunto contendo os elementos comuns presentes tanto em `setA` quanto em `setB`, a operação de diferença é feita por meio da criação de um novo conjunto com os elementos que estão em `setA`, mas não estão em `setB`. Observe o código a seguir:

```
const difference = (set1, set2) => {
  const differenceSet = new Set();
  set1.forEach(value => {
    if (!set2.has(value)) { // {1}
      differenceSet.add(value);
    }
  });
  return differenceSet;
};
console.log(difference(setA, setB));
```

A única diferença entre as funções `intersection` e `difference` está na linha `{1}`, pois queremos adicionar somente os diferentes elementos de `Set` contidos em `setA`, mas não em `setB`.

Usando o operador de espalhamento

Há uma maneira mais simples de simular as operações de união,

intersecção e diferença, usando o **operador de espalhamento** (spread operator), também introduzido na ES2015, e que conhecemos no Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*.

O processo é constituído de três passos:

1. Converter os conjuntos em arrays.
2. Executar a operação desejada.
3. Converter o resultado de volta em um conjunto.

Vamos ver como podemos executar a operação de *união de conjuntos* usando o operador de espalhamento:

```
console.log(new Set([...setA, ...setB]));
```

A classe **Set** da ES2015 também aceita que um array seja diretamente passado para o seu construtor a fim de inicializar o conjunto com alguns valores, portanto aplicamos o operador de espalhamento em **setA** (**...setA**); esse operador transformará os valores desse conjunto em um array (espalhará seus valores), e fazemos o mesmo com **setB**.

Como **setA** tem os valores [1, 2, 3] e **setB** é igual a [2, 3, 4], o código anterior é o mesmo que executar `new Set([1, 2, 3, 2, 3, 4])`, mas somente os valores únicos serão adicionados.

Vamos agora ver como podemos executar a operação de *intersecção de conjuntos* usando o operador de espalhamento:

```
console.log(new Set([...setA].filter(x => setB.has(x))));
```

O código anterior também transformará **setA** em um array e usará o método **filter**, que devolve um novo array com os valores que correspondem à função de callback – a qual, nesse caso, verifica se o elemento também está presente em **setB**. O array devolvido será usado para inicializar o construtor do **Set** resultante.

Por fim, vamos ver como podemos executar a operação de *diferença entre conjuntos* usando o operador de espalhamento:

```
console.log(new Set([...setA].filter(x => !setB.has(x))));
```

É o mesmo código usado na operação de intersecção, mas estamos interessados somente nos elementos que não estão em **setB**.

Você pode usar a versão de sua preferência para executar as operações de conjunto usando a classe **Set** nativa da ES2015!

Multiconjuntos ou bags

Como já vimos, a estrutura de dados de conjunto não permite elementos duplicados. Entretanto, em matemática, temos um conceito chamado multiconjunto (multiset), que permite que o mesmo elemento seja inserido no conjunto, mesmo que ele já tenha sido inserido antes. Os **multiconjuntos** (ou **bags**) podem ser muito úteis para contar quantos vezes o elemento está presente em um conjunto. Eles são frequentemente aplicados em sistemas de banco de dados.

Não discutiremos a estrutura de dados de bag neste livro. No entanto, você pode verificar o código-fonte e os exemplos se fizer o download do bundle de código deste livro, ou pode acessar <https://github.com/loiane/javascript-datastructures-algorithms>.

Resumo

Neste capítulo, aprendemos a implementar uma classe **Set** do zero, a qual é semelhante à classe **Set** definida na ECMAScript 2015. Também discutimos alguns métodos que, em geral, não estão presentes nas implementações da estrutura de dados para conjuntos em outras linguagens de programação, como união, intersecção, diferença e subconjunto. Implementamos uma classe **Set** completa, se comparada à implementação atual de **Set** em outras linguagens de programação.

No próximo capítulo, discutiremos os hashes e os dicionários, que são estruturas de dados não sequenciais.

CAPÍTULO 8

Dicionários e hashes

No capítulo anterior, conhecemos os conjuntos. Neste capítulo, continuaremos a nossa discussão sobre estruturas de dados que armazenam valores únicos (valores não repetidos) usando dicionários e hashes.

Em um conjunto, estamos interessados no próprio valor como o elemento principal. Em um dicionário (ou mapa), armazenamos valores em pares [chave, valor]. O mesmo vale para os hashes (armazenam valores em pares [chave, valor]), mas o modo como implementamos essas estruturas de dados é um pouco diferente, pois os dicionários podem armazenar apenas um valor único por chave, como veremos a seguir.

Neste capítulo, veremos:

- a estrutura de dados de dicionário;
- a estrutura de dados de tabela hash;
- tratamento de colisões em tabelas hash;
- as classes **Map**, **WeakMap** e **WeakSet** da ECMAScript 2015.

Estrutura de dados de dicionário

Como já vimos, um conjunto é uma coleção de elementos distintos (elementos não repetidos). Um **dicionário** é usado para armazenar pares [chave, valor], em que a chave pode ser usada para encontrar um elemento em particular. Um dicionário é muito parecido com um conjunto; um conjunto armazena uma coleção de elementos [chave, chave], enquanto um dicionário armazena uma coleção de elementos [chave, valor]. Um dicionário também é conhecido como **mapa** (map), **tabela de símbolos** e **array associativo**.

Em ciência da computação, os dicionários são usados frequentemente para armazenar endereços de referência de objetos. Por exemplo, se abrirmos **Chrome | Developer tools** (Chrome | Ferramentas do desenvolvedor) na aba **Memory** (Memória) e executarmos um **snapshot**, poderemos ver

alguns objetos e seus respectivos endereços de referência na memória (representados por @<número>). Podemos observar esse cenário na Figura 8.1:

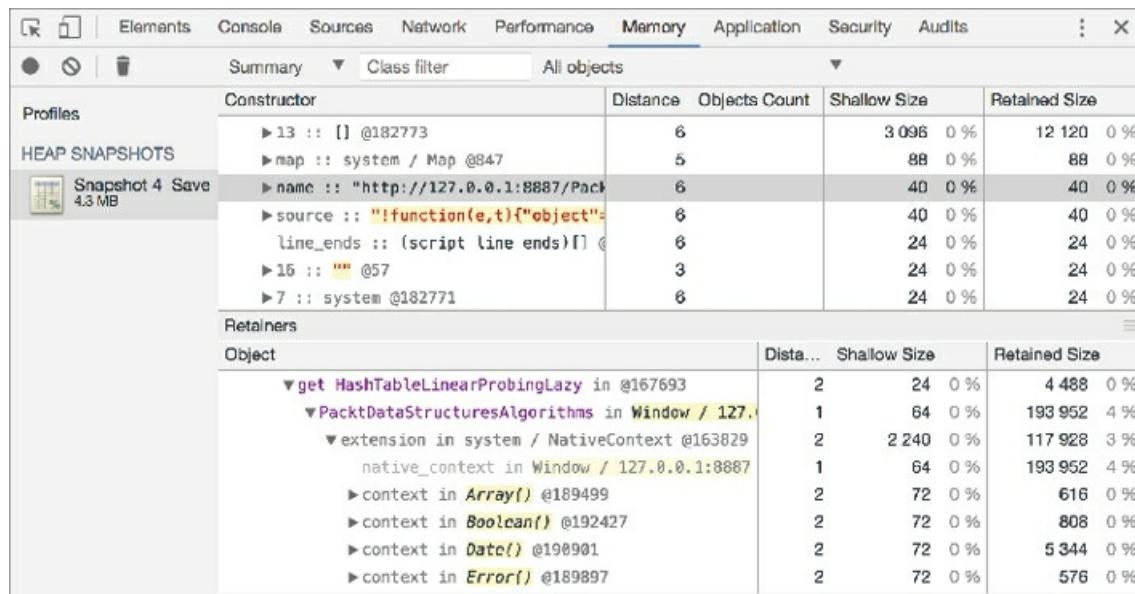


Figura 8.1

Neste capítulo, descreveremos alguns exemplos de uso da estrutura de dados de dicionário em um projeto do mundo real: uma agenda com endereços.

Criando a classe Dictionary

De modo semelhante à classe **Set**, a ECMAScript 2015 contém igualmente uma implementação da classe **Map**, também conhecida como dicionário.

A classe que implementaremos neste capítulo é baseada na implementação de **Map** da ECMAScript 2015. Você notará que ela é muito semelhante à classe **Set** (mas, em vez de armazenar um par [chave, chave], armazenaremos um par [chave, valor]).

A seguir, apresentamos a estrutura básica de nosso **Dictionary**:

```
import { defaultToString } from '../util';
export default class Dictionary {
  constructor(toStrFn = defaultToString) {
    this.toStrFn = toStrFn; // {1}
    this.table = {}; // {2}
  }
}
```

De modo semelhante à classe **Set**, também armazenaremos os elementos da classe **Dictionary** em uma instância de **Object**, em vez de usar um array (propriedade **table**, em {2}). Armazenaremos os pares [chave, valor] como **table[key] = {key, value}**.

O JavaScript nos permite acessar a propriedade de um objeto usando colchetes ([]), passando o nome da propriedade como a “posição”. É por isso que ele é chamado de array associativo! Já usamos dicionários antes neste livro, no Capítulo 4, *Pilhas*, no Capítulo 5, *Filas e deque*s, e no Capítulo 7, *Conjuntos*.

Em um dicionário, o ideal seria armazenar chaves do tipo string e qualquer tipo de **value** (de tipos primitivos como números, uma string, até objetos complexos). No entanto, como JavaScript não é uma linguagem fortemente tipada, não podemos garantir que **key** será uma string. Por esse motivo, precisamos transformar qualquer que seja o objeto passado como **key** em uma string para facilitar a busca e a obtenção dos valores da classe **Dictionary** (essa mesma lógica também pode ser aplicada à classe **Set** do capítulo anterior). Para isso, precisaremos de uma função para transformar **key** em uma string ({1}). Por padrão, usaremos a função **defaultToString** declarada no arquivo **utils.js** (podemos reutilizar as funções desse arquivo em qualquer estrutura de dados que criarmos).

Como estamos usando a funcionalidade de parâmetro default da ES2015, **toStrFn** é um parâmetro opcional. Se for necessário, também podemos passar uma função personalizada para especificar como gostaríamos de transformar a **key** em uma string.

A função **defaultToString** é declarada assim:

```
export function defaultToString(item) {
  if (item === null) {
    return 'NULL';
  } else if (item === undefined) {
    return 'UNDEFINED';
  } else if (typeof item === 'string' || item instanceof String) {
    return `${item}`;
  }
  return item.toString(); // {1}
}
```

Observe que é importante que, se a variável `item` for um objeto, ela tenha um método `toString` implementado; caso contrário, você poderá ter resultados inesperados, como `[object Object]`, que não é muito elegante.

Se `key (item)` for uma string, basta devolvê-la; caso contrário, chamaremos o método `toString` de `item`.

A seguir, devemos declarar os métodos disponíveis em um mapa/dicionário; são eles:

- `set(key, value)`: esse método adiciona um novo elemento ao dicionário. Se `key` já existir, seu valor será sobrescrito com o novo valor.
- `remove(key)`: esse método remove `value` do dicionário usando `key` como o parâmetro de busca.
- `hasKey(key)`: esse método devolve `true` se `key` estiver presente no dicionário, e `false` caso contrário.
- `get(key)`: esse método devolve um `value` específico do dicionário usando `key` como o parâmetro de busca.
- `clear()`: esse método remove todos os valores do dicionário.
- `size()`: esse método devolve a quantidade de valores contida no dicionário. É semelhante à propriedade `length` da classe `Array`.
- `isEmpty()`: esse método devolve `true` se `size` for igual a zero, e `false` caso contrário.
- `keys()`: esse método devolve um array com todas as chaves contidas no dicionário.
- `values()`: esse método devolve um array com todos os valores contidos no dicionário.
- `keyValues()`: esse método devolve um array com todos os pares de valores [chave, valor] contidos no dicionário.
- `forEach(callbackFn)`: esse método itera pelos valores (`value`) do dicionário. A função `callbackFn` tem dois parâmetros: `key` e `value`. Esse método também pode ser interrompido caso a função de callback devolva `false` (é semelhante ao método `every` da classe `Array`).

Verificando se uma chave está presente no dicionário

O primeiro método que implementaremos é o método `hasKey(key)`. Implementaremos esse método antes porque ele será usado em outros métodos, como `set` e `remove`. Podemos ver o seu código a seguir:

```
hasKey(key) {  
    return this.table[this.toStrFn(key)] != null;  
}
```

O JavaScript nos permite usar somente strings como `key`/propriedade dos objetos. Caso tenhamos um objeto complexo passado como `key`, será necessário transformá-lo em uma string. Por esse motivo, sempre chamamos a função `toStrFn`. Caso haja um valor para a `key` especificada (uma posição na tabela que não seja `null` ou `undefined`), devolveremos `true`; caso contrário, devolveremos `false`.

Definindo uma chave e um valor no dicionário, e a classe ValuePair

A seguir, temos o método `set`, conforme declarado:

```
set(key, value) {  
    if (key != null && value != null) {  
        const tableKey = this.toStrFn(key); // {1}  
        this.table[tableKey] = new ValuePair(key, value); // {2}  
        return true;  
    }  
    return false;  
}
```

Esse método recebe `key` e `value` como parâmetros. Se `key` e `value` não forem `undefined` ou `null`, geramos a string que representa `key` ({1}) e criamos um par de valores, atribuindo-o à propriedade formada pela string da chave (`tableKey`) no objeto `table` ({2}). Se `key` e `value` forem válidos, também devolvemos `true` para informar que o dicionário foi capaz de armazená-los; caso contrário, devolvemos `false`.

Esse método pode ser usado para adicionar um novo valor ou atualizar um valor existente.

Na linha {2}, instanciamos a classe `ValuePair`. Eis a declaração da classe `ValuePair`:

```
class ValuePair {  
    constructor(key, value) {  
        this.key = key;
```

```

        this.value = value;
    }
    toString() {
        return `[#${this.key}: ${this.value}]`;
    }
}

```

Como estamos transformando **key** em uma string para armazenar **value** no dicionário, armazenaremos também a **key** original para informação. Por esse motivo, em vez de simplesmente armazenar **value** no dicionário, armazenaremos os dois valores: a **key** original e **value**. Para facilitar a exibição do dicionário no método **toString** mais tarde, criaremos também um método **toString** para a classe **ValuePair**.

Removendo um valor do dicionário

A seguir, implementaremos o método **remove**. Esse método é muito semelhante ao **remove** da classe **Set**; a única diferença é que buscaremos inicialmente a **key** (em vez de **value**), assim:

```

remove(key) {
    if (this.hasKey(key)) {
        delete this.table[this.toStrFn(key)];
        return true;
    }
    return false;
}

```

Então usamos o operador **delete** de JavaScript para remover a **key** (transformada em uma string) do objeto **table**. Caso possamos remover **value** do dicionário, também devolvemos **true**; caso contrário, devolveremos **false**.

Obtendo um valor do dicionário

Se quisermos buscar uma **key** em particular no dicionário e obter o seu **value**, podemos usar o método a seguir:

```

get(key) {
    const valuePair = this.table[this.toStrFn(key)]; // {1}
    return valuePair == null ? undefined : valuePair.value; // {2}
}

```

O método **get** obterá o objeto armazenado na propriedade **key** especificada ({1}). Se o objeto com o par de valores existir, devolveremos o

seu **value**; caso contrário, devolveremos **undefined** ({2}).

Uma maneira diferente de implementar esse mesmo método seria verificar se o **value** que gostaríamos de obter existe (procurando a sua **key**) antes; se o resultado for positivo, acessamos o objeto **table** e devolvemos o valor desejado. Eis a segunda opção para o método **get**:

```
get(key) {
  if (this.hasKey(key)) {
    return this.table[this.toStrFn(key)];
  }
  return undefined;
}
```

No entanto, nessa segunda abordagem, estariámos obtendo a string para **key** e acessando o objeto **table** duas vezes: a primeira vez no método **hasKey** e a segunda, dentro da instrução **if**. É um pequeno detalhe, mas a primeira abordagem tem um custo menor de processamento.

Métodos **keys**, **values** e **valuePairs**

Agora que criamos os métodos mais importantes da classe **Dictionary**, vamos implementar alguns métodos auxiliares que serão muito úteis.

A seguir, criaremos o método **valuePairs**, que devolverá um array com todos os objetos **ValuePair** presentes no dicionário. Eis o código desse método:

```
keyValues() {
  return Object.values(this.table);
}
```

O código é bem simples – chamamos o método embutido **values** da classe **Object** de JavaScript, introduzido na ECMAScript 2017, que conhecemos no Capítulo 1, *JavaScript – uma visão geral rápida*.

Como o método **Object.values** pode não estar disponível ainda em todos os navegadores, podemos também usar o código a seguir como alternativa:

```
keyValues() {
  const valuePairs = [];
  for (const k in this.table) { // {1}
    if (this.hasKey(k)) {
      valuePairs.push(this.table[k]); // {2}
    }
  }
  return valuePairs;
```

```
};
```

No código anterior, precisamos iterar por todas as propriedades do objeto `table` ({1}). Somente para garantir que `key` existe, usamos a função `hasKey` para essa verificação e, então, adicionamos o `ValuePair` do objeto `table` ({2}) no array resultante. Nesse método, como já estamos acessando a propriedade (`key`) do objeto `table` diretamente, não precisamos transformá-la em uma string com a função `toString`.

Não podemos simplesmente usar a instrução `for-in` e iterar pelas propriedades do objeto `table`. Precisamos usar também o método `hasKey` (para conferir se o objeto `table` tem essa propriedade), pois o protótipo do objeto contém propriedades adicionais. (As propriedades são herdadas da classe-base `Object` de JavaScript, mas o objeto continua tendo propriedades que não são de nosso interesse nessa estrutura de dados.)

O próximo método que criaremos é o método `Keys`, o qual devolve todas as chaves (as originais) usadas para identificar um valor na classe `Dictionary`, da seguinte maneira:

```
keys() {
  return this.keyValues().map(valuePair => valuePair.key);
}
```

Chamamos o método `keyValues` que criamos, o qual devolve um array de instâncias de `ValuePair`. Então iteramos por elas. Como estamos interessados somente na propriedade `key` de `ValuePair`, devolvemos apenas a sua `key`.

No código anterior, usamos o método `map` da classe `Array` para iterar pelos `ValuePairs`. O método `map` transforma um dado `value` em outro valor. Nesse caso, estamos transformando cada `ValuePair` em sua `key`. A lógica usada no método `keys` também pode ser escrita assim:

```
const keys = [];
const valuePairs = this.keyValues();
for (let i = 0; i < valuePairs.length; i++) {
  keys.push(valuePairs[i].key);
}
return keys;
```

O método `map` nos permite executar a mesma lógica e obter o mesmo resultado que as cinco linhas do código anterior; depois que estivermos

acostumados com a sua sintaxe, será mais fácil ainda ler o código anterior e entender o que ele faz.

O método `map` foi introduzido na ES2015 (ES6), conforme vimos no Capítulo 3, *Arrays*. O método `keys` que criamos também usa o paradigma da programação funcional, que conheceremos melhor mais adiante neste livro.

De modo semelhante ao método `keys`, temos também o método `values`. Esse método devolve um array com todos os valores armazenados no dicionário. O seu código é muito parecido com o código do método `keys`; contudo, em vez de devolver a propriedade `key` da classe `ValuePair`, devolvemos a propriedade `value`, assim:

```
values() {  
    return this.keyValues().map(valuePair => valuePair.value);  
}
```

Iterando pelos ValuePairs do dicionário com `forEach`

Até agora, não criamos nenhum método que nos permita iterar pelos valores armazenados nas estruturas de dados. Implementaremos o método `forEach` para a classe `Dictionary`, mas podemos usar essa mesma lógica igualmente nas outras estruturas de dados que criamos antes no livro.

Eis a implementação do método `forEach`:

```
forEach(callbackFn) {  
    const valuePairs = this.keyValues(); // {1}  
    for (let i = 0; i < valuePairs.length; i++) { // {2}  
        const result = callbackFn(valuePairs[i].key, valuePairs[i].value); // {3}  
        if (result === false) {  
            break; // {4}  
        }  
    }  
}
```

Inicialmente obtemos o array de `ValuePairs` do dicionário ({1}). Em seguida, iteramos por cada `ValuePair` ({2}) e chamamos a função `callbackFn` ({3}) passada como parâmetro para o método `forEach`, além de armazenar o seu resultado. Caso a função de callback devolva `false`, interrompemos a execução do método `forEach` ({4}), saindo do laço `for` que faz a iteração por `valuePairs`.

Métodos clear, size, isEmpty e toString

O método `size` devolve quantos valores estão armazenados no dicionário. Podemos obter todas as `keys` do objeto `table` usando o método `Object.keys` (fizemos o mesmo no método `keyValues`). O código do método `size` é declarado assim:

```
size() {
  return Object.keys(this.table).length;
}
```

Poderíamos também ter chamado o método `keyValues` e devolvido o `length` do array devolvido (`return this.keyValues().length`).

Para verificar se o dicionário está vazio, podemos obter o seu tamanho (`size`) e conferir se é igual a zero. Se `size` for zero, é sinal de que o dicionário está vazio. Essa lógica é implementada no método `isEmpty`, da seguinte maneira:

```
isEmpty() {
  return this.size() === 0;
}
```

Para limpar o dicionário, basta atribuir uma nova instância de objeto a `table`:

```
clear() {
  this.table = {};
}
```

Por fim, criaremos também o método `toString`, assim:

```
toString() {
  if (this.isEmpty()) {
    return '';
  }
  const valuePairs = this.keyValues();
  let objString = `${valuePairs[0].toString()}`; // {1}
  for (let i = 1; i < valuePairs.length; i++) {
    objString = `${objString},${valuePairs[i].toString()}`; // {2}
  }
  return objString; // {3}
}
```

No método `toString`, se o dicionário estiver vazio, devolvemos uma string vazia. Caso contrário, adicionamos o seu primeiro `ValuePair` na string resultante chamando o método `toString` de `ValuePair` ({1}). Então, se houver algum `value` no array, ele também será adicionado à string

resultante ({2}); essa string será devolvida no final do método ({3}).

Usando a classe Dictionary

Para usar a classe **Dictionary**, inicialmente devemos criar uma instância e, em seguida, adicionaremos três emails a ela. Usaremos essa instância **dictionary** para exemplificar uma agenda de endereços de emails.

Vamos executar um código que utilize a classe que criamos:

```
const dictionary = new Dictionary();
dictionary.set('Gandalf', 'gandalf@email.com');
dictionary.set('John', 'johnsnow@email.com');
dictionary.set('Tyrion', 'tyrion@email.com');
```

Se executarmos o código a seguir, veremos uma saída igual a **true**:

```
console.log(dictionary.hasKey('Gandalf'));
```

O código seguinte exibirá 3, pois adicionamos três elementos em nossa instância de dicionário:

```
console.log(dictionary.size());
```

Vamos agora executar estas linhas de código:

```
console.log(dictionary.keys());
console.log(dictionary.values());
console.log(dictionary.get('Tyrion'));
```

Eis a saída, na respectiva ordem:

```
["Gandalf", "John", "Tyrion"]
["gandalf@email.com", "johnsnow@email.com", "tyrion@email.com"]
tyrion@email.com
```

Por fim, vamos executar mais algumas linhas de código:

```
dictionary.remove('John');
```

Execute as linhas a seguir também:

```
console.log(dictionary.keys());
console.log(dictionary.values());
console.log(dictionary.keyValues());
```

Eis a saída:

```
["Gandalf", "Tyrion"]
["gandalf@email.com", "tyrion@email.com"]
[{"key: "Gandalf", value: "gandalf@email.com"}, {"key: "Tyrion", value:
"tyrion@email.com"}]
```

Como removemos um **value**, a instância **dictionary** agora contém somente dois tipos de **value**. A linha em destaque exemplifica o modo

como o objeto **table** está estruturado internamente.

Para chamar o método **forEach**, podemos usar o código a seguir:

```
dictionary.forEach((k, v) => {
  console.log('forEach: ', `key: ${k}, value: ${v}`);
});
```

Veremos a seguinte saída:

```
forEach: key: Gandalf, value: gandalf@email.com
forEach: key: Tyrion, value: tyrion@email.com
```

Tabela hash

Nesta seção, conhiceremos a classe **HashTable**, também conhecida como **HashMap**: uma implementação com hash da classe **Dictionary**.

O **hashing** consiste em encontrar um valor em uma estrutura de dados o mais rápido possível. Nos capítulos anteriores, vimos que, se quisermos obter um valor de uma estrutura de dados (usando um método **get**), será necessário iterar por ela até que esse valor seja encontrado. Quando usamos uma função de hash, já sabemos em que posição o valor se encontra, portanto, podemos simplesmente o acessar. Uma função de hash é uma função que, dada uma **key**, devolve o endereço em que o valor está na tabela.

Em ciência da computação, a tabela hash tem vários casos de uso. Ela pode ser usada como arrays associativos, pois é uma implementação do dicionário, e também pode ser usada para indexar um banco de dados. Ao criar uma tabela em um banco de dados relacional como MySQL, Microsoft Server, Oracle e outros, é sempre uma boa prática criar um **index** para permitir uma pesquisa mais rápida da chave (**key**) do registro. Nesse caso, uma tabela hash pode ser criada para armazenar a **key** e a referência ao registro na tabela. Outro uso muito comum é a utilização de tabelas hash para representar objetos. A linguagem JavaScript usa uma tabela hash internamente para representar cada objeto. Nesse caso, cada propriedade e cada método (membros) do objeto são armazenados como tipos de objeto **key**, e cada **key** aponta para o respectivo membro do objeto.

Como exemplo, vamos continuar usando a agenda de endereços de email que vimos na seção anterior. A função de hash que usaremos neste livro é a função mais comum existente, chamada de função de **hash lóose-lose**, em

que simplesmente somamos os valores ASCII de cada caractere da chave:

| Nome/Chave | Função de hash | Valor do hash | Tabela hash |
|------------|----------------------------------------|---------------|-------------|
| Gandalf | $71 + 97 + 110 + 100 + 97 + 108 + 102$ | 685 | [...] |
| John | $74 + 111 + 104 + 110$ | 399 | [399] |
| Tyrion | $84 + 121 + 114 + 105 + 111 + 110$ | 645 | [645] |

Figura 8.2

Criando uma classe HashTable

Usaremos também um array associativo (objeto) para representar a nossa estrutura de dados, como fizemos com a classe `Dictionary`.

Como sempre, vamos começar pela estrutura de nossa classe, usando o código a seguir:

```
class HashTable {
    constructor(toStrFn = defaultToString) {
        this.toStrFn = toStrFn;
        this.table = {};
    }
}
```

Em seguida, devemos acrescentar alguns métodos em nossa classe. Implementaremos três métodos básicos para todas as classes:

- `put(key, value)`: esse método adiciona um novo item à tabela hash (ou pode atualizá-la também).
- `remove(key)`: esse método remove o `value` da tabela hash usando `key`.
- `get(key)`: esse método devolve um `value` específico encontrado com `key`.

Criando uma função de hash

O primeiro método que implementaremos antes desses três métodos é

`hashCode`. Eis o código dele:

```
loseloseHashCode(key) {
  if (typeof key === 'number') { // {1}
    return key;
  }
  const tableKey = this.toStrFn(key); // {2}
  let hash = 0; // {3}
  for (let i = 0; i < tableKey.length; i++) {
    hash += tableKey.charCodeAt(i); // {4}
  }
  return hash % 37; // {5}
}
hashCode(key) {
  return this.loseloseHashCode(key);
}
```

O método `hashCode` simplesmente chama o método `loseloseHashCode`, passando `key` como parâmetro.

No método `loseloseHashCode`, inicialmente verificamos se `key` é um `number` ({1}). Se for, ele será apenas devolvido com `return`. Na sequência, geramos um número baseado na soma do valor de cada caractere ASCII que compõe a `key`. Assim, em primeiro lugar, temos de transformar `key` em uma string ({2}) caso ela seja um objeto, e não uma string. Iniciamos a variável `hash` que armazenará a soma ({3}). Então iteramos pelos caracteres de `key` e somamos o valor ASCII do caractere correspondente na tabela ASCII na variável `hash` ({3}). Para isso, podemos usar o método `charCodeAt` da classe `String` de JavaScript. Por fim, devolvemos o valor de `hash`. Para trabalhar com números menores, devemos usar o resto da divisão (%) do número de `hash` utilizando um número arbitrário ({5}) – isso evitará correr o risco de trabalhar com números grandes, que não caibam em uma variável numérica.

Para obter mais informações sobre a tabela ASCII, consulte <http://www.asciitable.com>.

Inserindo uma chave e um valor na tabela hash

Agora que temos a nossa função `hashCode`, podemos implementar o método `put`, assim:

```
put(key, value) {
```

```

if (key != null && value != null) { // {1}
    const position = this.hashCode(key); // {2}
    this.table[position] = new ValuePair(key, value); // {3}
    return true;
}
return false;
}

```

O método `put` tem uma lógica semelhante à logica do método `set` da classe `Dictionary`. Poderíamos ter chamado esse método de `set` também; no entanto, a maioria das linguagens de programação usa o método `put` na estrutura de dados `HashTable`, portanto adotaremos a mesma convenção de nomenclatura.

Em primeiro lugar, verificamos se `key` e `value` são válidos ({1}); caso não sejam, devolvemos `false` para informar que o valor não foi adicionado (nem atualizado). Para o parâmetro `key` especificado, devemos encontrar uma posição na tabela usando a função `hashCode` que criamos ({2}). Então, tudo que temos a fazer é criar uma instância de `ValuePair` com `key` e `value` ({3}). De modo semelhante à classe `Dictionary`, armazenaremos também a `key` original para informação.

Obtendo um valor da tabela hash

Obter um `value` da instância de `HashTable` também é simples. Implementaremos o método `get` para isso, do seguinte modo:

```

get(key) {
    const valuePair = this.table[this.hashCode(key)];
    return valuePair == null ? undefined : valuePair.value;
}

```

Em primeiro lugar, obtemos a posição do parâmetro `key` especificado, usando a função `hashCode` que criamos. Essa função devolverá a posição de `value`, e tudo que temos de fazer é acessar essa posição no array `table` e devolver o seu `value`.

As classes `HashTable` e `Dictionary` são muito parecidas. A diferença está no fato de que, na classe `Dictionary`, armazenamos o `ValuePair` na propriedade `key` de `table` (depois de ter sido transformado em uma string); na classe `HashTable`, geramos um número a partir da `key` (`hash`) e armazenamos o `ValuePair` na posição (ou propriedade) `hash`.

Removendo um valor da tabela hash

O último método que implementaremos para `HashTable` é o método `remove`, que vemos a seguir:

```
remove(key) {  
    const hash = this.hashCode(key); // {1}  
    const valuePair = this.table[hash]; // {2}  
    if (valuePair != null) {  
        delete this.table[hash]; // {3}  
        return true;  
    }  
    return false;  
}
```

Para remover um `value` da `HashTable`, inicialmente devemos saber qual posição devemos acessar, portanto, obtemos o `hash` usando a função `hashCode` ({1}). Lemos o `valuePair` armazenado na posição `hash` ({2}) e, caso ele não seja `null` nem `undefined`, nós o removemos usando o operador `delete` de JavaScript ({3}). Também devolveremos `true` se a remoção for bem-sucedida, e `false` caso contrário.

Em vez de usar o operador `delete` de JavaScript, podemos também atribuir `null` ou `undefined` à posição `hash` removida.

Usando a classe HashTable

Vamos testar a classe `HashTable` executando alguns exemplos:

```
const hash = new HashTable();  
hash.put('Gandalf', 'gandalf@email.com');  
hash.put('John', 'johnsnow@email.com');  
hash.put('Tyrion', 'tyrion@email.com');  
console.log(hash.hashCode('Gandalf') + ' - Gandalf');  
console.log(hash.hashCode('John') + ' - John');  
console.log(hash.hashCode('Tyrion') + ' - Tyrion');
```

Se inspecionarmos o conteúdo da tabela hash depois de executar o código anterior, veremos o resultado a seguir:

```
19 - Gandalf  
29 - John  
16 - Tyrion
```

O diagrama seguinte representa a estrutura de dados de `HashTable` com esses três elementos:

| Nome/Chave | Função de hash | Tabela hash |
|------------|----------------|--------------------------------|
| Gandalf | 19 | [...] |
| John | 29 | [16] [...] |
| Tyrion | 16 | [19] [...] [29] [...] |

Figura 8.3

Vamos agora testar o método **get** executando o código a seguir:

```
console.log(hash.get('Gandalf')) // gandalf@email.com
console.log(hash.get('Loiane')) // undefined
```

Como **Gandalf** é uma **key** presente na **HashTable**, o método **get** devolverá o seu **value**. Pelo fato de **Loiane** não ser uma **key** existente, se tentarmos acessar a sua posição no array (uma posição gerada pela função **hash**), o seu **value** será **undefined** (inexistente).

A seguir, vamos tentar remover **Gandalf** da **HashTable**, assim:

```
hash.remove('Gandalf');
console.log(hash.get('Gandalf'));
```

O método **hash.get('Gandalf')** devolverá **undefined** como saída no console, pois **Gandalf** não está mais presente na tabela.

Tabela hash versus conjunto hash

Uma tabela hash é o mesmo que um mapa hash. Discutimos essa estrutura de dados neste capítulo.

Em algumas linguagens de programação, também temos a implementação de **conjunto hash** (hash set). A estrutura de dados de um conjunto hash é composta de um conjunto; contudo, para inserir, remover ou acessar elementos, usamos uma função **hashCode**. Podemos reutilizar todo o código que implementamos neste capítulo em um conjunto hash; a diferença é que, em vez de adicionar um par chave-valor, apenas o valor

será inserido, e não a chave. Por exemplo, poderíamos usar um conjunto hash para armazenar todas as palavras em inglês (sem as suas definições). De modo semelhante à estrutura de dados **set**, o conjunto hash armazena somente valores únicos – não haverá valores repetidos.

Tratando colisões nas tabelas hash

Às vezes, chaves diferentes podem ter o mesmo valor de hash. Chamaremos a isso de **colisão**, pois tentaremos atribuir diferentes pares chave-valor à mesma posição na instância de **HashTable**. Por exemplo, vamos observar a saída resultante do código a seguir:

```
const hash = new HashTable();
hash.put('Ygritte', 'ygritte@email.com');
hash.put('Jonathan', 'jonathan@email.com');
hash.put('Jamie', 'jamie@email.com');
hash.put('Jack', 'jack@email.com');
hash.put('Jasmine', 'jasmine@email.com');
hash.put('Jake', 'jake@email.com');
hash.put('Nathan', 'nathan@email.com');
hash.put('Athelstan', 'athelstan@email.com');
hash.put('Sue', 'sue@email.com');
hash.put('Aethelwulf', 'aethelwulf@email.com');
hash.put('Sargerás', 'sargerás@email.com');
```

Ao chamar o método **hash.hashCode** para cada nome mencionado, veremos a saída a seguir:

```
4 - Ygritte
5 - Jonathan
5 - Jamie
7 - Jack
8 - Jasmine
9 - Jake
10 - Nathan
7 - Athelstan
5 - Sue
5 - Aethelwulf
10 - Sargerás
```

Observe que **Nathan** tem o mesmo valor de hash que **Sargerás** (10). **Jack** tem o mesmo valor de hash que **Athelstan** (7). **Jonathan**, **Jamie**, **Sue** e **Aethelwulf** (5) também têm o mesmo valor de hash.

O que acontecerá com a instância de **HashTable**? Quais valores teremos aí

depois de executar o código anterior?

Para nos ajudar a descobrir, vamos implementar o método `toString`:

```
toString() {
  if (this.isEmpty()) {
    return '';
  }
  const keys = Object.keys(this.table);
  let objString = `${keys[0]} => ${this.table[keys[0]].toString()}`;
  for (let i = 1; i < keys.length; i++) {
    objString = `${objString}, ${keys[i]} => ${this.table[keys[i]].toString()}`;
  }
  return objString;
}
```

Como não sabemos quais posições do array da tabela têm valores, podemos usar uma lógica semelhante à do método `toString` de `Dictionary`.

Depois de chamar `console.log(hashTable.toString())`, veremos a seguinte saída no console:

```
{4 => [#Ygritte: ygritte@email.com]}
{5 => [#Aethelwulf: aethelwulf@email.com]}
{7 => [#Athelstan: athelstan@email.com]}
{8 => [#Jasmine: jasmine@email.com]}
{9 => [#Jake: jake@email.com]}
{10 => [#Sargerás: sargerás@email.com]}
```

Jonathan, **Jamie** e **Sue** e **Aethelwulf** têm o mesmo valor de hash, isto é, 5. Como **Aethelwulf** foi o último a ser adicionado, ele ocupará a posição 5 da `HashTable`. Inicialmente **Jonathan** ocupará essa posição, em seguida **Jamie** a sobrescreverá e depois **Sue** fará o mesmo; por fim, **Aethelwulf** a sobrescreverá novamente. O mesmo acontecerá com os demais elementos que apresentam uma colisão.

A ideia de usar uma estrutura de dados para armazenar todos esses valores obviamente não é perdê-los, mas preservá-los de alguma forma. Por esse motivo, devemos lidar com essa situação quando ela ocorrer. Há algumas técnicas para tratar colisões: encadeamento separado (separate chaining), sondagem linear (linear probing) e hashing duplo (double hashing). Discutiremos as duas primeiras técnicas neste livro.

Encadeamento separado

A técnica de **encadeamento separado** (separate chaining) consiste em criar uma lista ligada para cada posição da tabela e armazenar aí os elementos. É a técnica mais simples que há para tratar colisões; no entanto, ela exige memória extra, além daquela ocupada pela instância de **HashTable**.

Por exemplo, se usarmos o encadeamento separado no código utilizado para testes na seção anterior e o representarmos em um diagrama (Figura 8.4), a saída seria esta (os valores foram omitidos do diagrama para simplificá-lo).

Na posição 5, teríamos uma instância de **LinkedList** com quatro elementos; nas posições 7 e 10, teríamos instâncias de **LinkedList** com dois elementos e, nas posições 4, 8 e 9, teríamos instâncias de **LinkedList** com um único elemento.

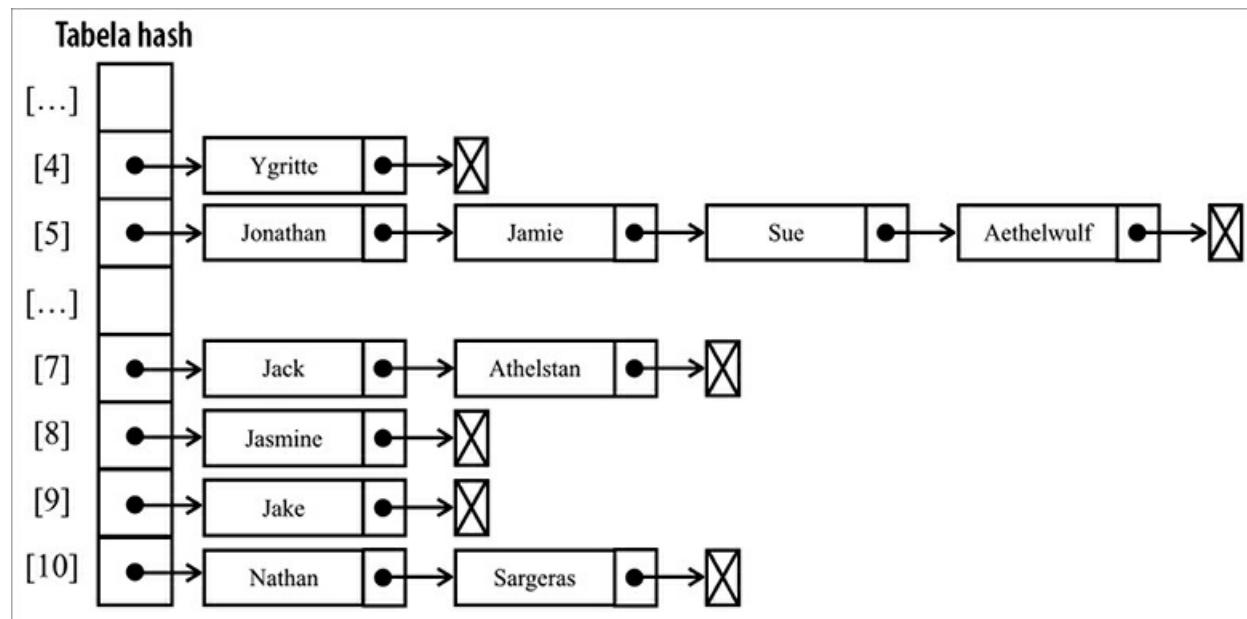


Figura 8.4

No encadeamento separado e na sondagem linear, será necessário substituir três métodos: **put**, **get** e **remove**. Esses três métodos serão diferentes para cada técnica distinta que decidirmos implementar.

Como sempre, vamos começar declarando a estrutura básica de **HashTableSeparateChaining**:

```

class HashTableSeparateChaining {
    constructor(toStrFn = defaultToString) {
        this.toStrFn = toStrFn;
        this.table = {};
    }
}

```

```
}
```

Método put

Vamos implementar o primeiro método – o método **put** – assim:

```
put(key, value) {
  if (key != null && value != null) {
    const position = this.hashCode(key);
    if (this.table[position] == null) { // {1}
      this.table[position] = new LinkedList(); // {2}
    }
    this.table[position].push(new ValuePair(key, value)); // {3}
    return true;
  }
  return false;
}
```

Nesse método, verificamos se a posição na qual estamos tentando adicionar o valor já contém outros valores ({1}). Se essa for a primeira vez que estamos adicionando um elemento nessa posição, vamos inicializá-la com uma instância da classe **LinkedList** ({2} – que conhecemos no Capítulo 6, *Listas ligadas*). Então adicionamos a instância de **ValuePair** à instância de **LinkedList** usando o método **push** ({3}), que também implementamos no Capítulo 6, *Listas ligadas*.

Método get

A seguir, implementaremos o método **get** para obter um valor, dada uma chave, usando o código a seguir:

```
get(key) {
  const position = this.hashCode(key);
  const linkedList = this.table[position]; // {1}
  if (linkedList != null && !linkedList.isEmpty()) { // {2}
    let current = linkedList.getHead(); // {3}
    while (current != null) { // {4}
      if (current.element.key === key) { // {5}
        return current.element.value; // {6}
      }
      current = current.next; // {7}
    }
  }
  return undefined; // {8}
}
```

A primeira verificação que devemos fazer é conferir se há algum `value` na posição desejada. Para isso, acessamos a `linkedList` na `position` do hash (`{1}`) e verificamos se há uma instância de `linkedList` ou se ela está vazia (`{2}`). Se não houver nenhum `value`, devolvemos `undefined` para informar que o valor não foi encontrado na instância de `HashTable` (`{8}`). Se houver um valor em `position`, saberemos que a instância desse objeto é uma instância de `LinkedList`. Agora, tudo que temos a fazer é procurar o `element` que queremos encontrar fazendo uma iteração pela lista. Para isso, é necessário obter a referência à cabeça (`head`) da lista (`{3}`), que é o primeiro `element` da `linkedList`, e, em seguida, podemos iterar por ela até encontrarmos o final da lista (`{4}` – na última iteração, `current.next` será `null`).

O `Node` de `linkedList` contém o ponteiro `next` e `element` como propriedades. A propriedade `element` é uma instância de `ValuePair`, portanto ela tem as propriedades `value` e `key`. Para acessar a propriedade `key` do `Node` da `LinkedList`, podemos usar `current.element.key` e compará-lo para ver se é a `key` que estamos procurando (`{5}`). Se for o mesmo atributo `key`, devemos devolver o valor de `Node` (`{6}`); se não for, continuaremos iterando pela lista, acessando o próximo `element` (`{7}`). Essa lógica nos permite procurar todos os atributos `key` em qualquer posição da `LinkedList`.

Outra abordagem para esse algoritmo é esta: em vez de fazer a busca de `key` no método `get`, poderíamos instanciar a `LinkedList` no método `put`, passando um `equalsFn` personalizado para o construtor de `LinkedList`, o qual comparará somente a propriedade `key` do elemento (que é uma instância de `ValuePair`). Apenas para lembrar, por padrão, a `LinkedList` usará o operador `==` para comparar suas instâncias de `element`, o que significa que ela comparará a referência da instância de `ValuePair`. Nesse caso, no método `get`, usariamos o método `indexOf` para procurar a `key` desejada, o qual devolverá uma posição maior ou igual a zero caso o elemento esteja presente na `LinkedList`. Com a `position`, podemos usar `getElementAt` para obter a instância de `ValuePair` da `LinkedList`.

Método `remove`

Remover um valor da instância de `HashTableSeparateChaining` é um

pouco diferente do método `remove` que implementamos antes neste capítulo. Agora que estamos usando `LinkedList`, temos de remover o `element` da `LinkedList`. Vamos analisar a implementação do método `remove`:

```
remove(key) {
    const position = this.hashCode(key);
    const linkedList = this.table[position];
    if (linkedList != null && !linkedList.isEmpty()) {
        let current = linkedList.getHead();
        while (current != null) {
            if (current.element.key === key) { // {1}
                linkedList.remove(current.element); // {2}
                if (linkedList.isEmpty()) { // {3}
                    delete this.table[position]; // {4}
                }
                return true; // {5}
            }
            current = current.next; // {6}
        }
    }
    return false; // {7}
}
```

No método `remove`, fazemos o mesmo que fizemos no método `get` para encontrar o `element` que estamos procurando. Ao iterar pela instância de `LinkedList`, se o `element` de `current` na lista for a chave que estamos procurando ({1}), usaremos o método `remove` para removê-lo da `LinkedList` ({2}). Então, faremos uma validação adicional: se a lista estiver vazia ({3}), não há mais nenhum elemento aí), removeremos `position` de `table` usando o operador `delete` ({4}), de modo que possamos pular essa `position` sempre que procurarmos um `element`. Por fim, devolveremos `true` para informar que o elemento foi removido ({5}), ou devolveremos `false` no final para informar que o `element` não estava presente em `HashTableSeparateChaining` ({7}). Se não for o elemento que estávamos procurando, faremos uma iteração para o próximo elemento da `LinkedList` ({6}), como foi feito no método `get`.

Ao substituir esses três métodos, teremos uma instância de `HashTableSeparateChaining` com uma técnica de encadeamento separada para tratar colisões.

Sondagem linear

Outra técnica de resolução de colisão é a **sondagem linear** (linear probing). É chamada de linear porque a colisão é tratada de modo que os valores serão armazenados diretamente na tabela, e não em uma estrutura de dados separada.

Ao tentar adicionar um novo elemento, se a **position** do hash já estiver ocupada, tentaremos usar **position + 1**. Se **position + 1** estiver ocupada, tentaremos **position + 2**, e assim sucessivamente, até que uma posição livre seja encontrada na tabela hash. Vamos supor que temos uma tabela hash com alguns valores já inseridos, e queremos adicionar uma nova chave e um novo valor. Calculamos o hash para essa nova chave e verificamos se a **position** do hash já está ocupada. Se não estiver, adicionamos o valor na posição correta. Se estiver ocupada, iteramos pelo hash até encontrarmos uma posição livre.

O diagrama a seguir (Figura 8.5) mostra esse processo:

| #1: adicionar Jamie - hash 5 | | |
|------------------------------|-------------|------|
| índice | chave-valor | hash |
| 4 | Ygritte | 4 |
| 5 | Jonathan | 5 |
| 6 | | |
| 7 | Jack | 7 |
| 8 | Jasmine | 8 |
| 9 | Jake | 9 |
| 10 | Nathan | 10 |
| 11 | | |

| #2: adicionar Athelstan - hash 7 | | |
|----------------------------------|-------------|------|
| índice | chave-valor | hash |
| 4 | Ygritte | 4 |
| 5 | Jonathan | 5 |
| 6 | Jamie | 5 |
| 7 | Jack | 7 |
| 8 | Jasmine | 8 |
| 9 | Jake | 9 |
| 10 | Nathan | 10 |
| 11 | | |

Figura 8.5

Quando removemos uma chave-valor da tabela hash, não será suficiente simplesmente remover o elemento da **position**, conforme implementamos nas estruturas de dados anteriores neste capítulo. Se somente removermos o elemento, isso poderá fazer com que uma posição vazia seja encontrada

quando buscamos outro elemento com o mesmo hash (**position**), resultando em um bug no algoritmo.

Há duas opções na técnica de sondagem linear. A primeira é a abordagem da *remoção soft* (soft delete). Usamos um valor especial (flag) para sinalizar que a chave-valor foi apagada (*remoção preguiçosa [lazy] ou soft*), em vez de realmente apagar o elemento. Com o passar do tempo, porém, a tabela hash será manipulada e poderá acabar com várias posições apagadas. Com isso, a eficiência da tabela se deteriorará gradualmente, e as buscas de chaves-valores se tornarão mais lentas com o tempo. Poder acessar e encontrar rapidamente uma chave é um dos principais motivos para usarmos uma tabela hash. A Figura 8.6 mostra esse processo.

A segunda abordagem exige verificar se é necessário mover um ou mais elementos uma **position** para trás. Ao procurar uma chave, essa abordagem evita encontrar uma posição vazia, mas, se for necessário mover elementos, significa que precisaremos deslocar as chaves-valores na tabela hash. A Figura 8.7 exemplifica esse processo:

| índice chave-valor hash | | |
|-------------------------|----------------|---|
| 4 | Ygritte | 4 |
| 5 | <i>Apagado</i> | |
| 6 | <i>Apagado</i> | |
| 7 | <i>Apagado</i> | |
| 8 | Jasmine | 8 |
| 9 | <i>Apagado</i> | |
| 10 | <i>Apagado</i> | |
| 11 | Athelstan | 7 |

encontrar Athelstan - hash 7

Apagado, vai para a próxima posição

Ocupado, e não é a chave, vai para a próxima posição

Apagado, vai para a próxima posição

Apagado, vai para a próxima posição

Encontrado!



Figura 8.6

| índice | chave-valor | hash |
|--------|-------------|------|
| 4 | Ygritte | 4 |
| 5 | jonathan | 5 |
| 6 | Jamie | 5 |
| 7 | Jack | 7 |
| 8 | Jasmine | 8 |
| 9 | Jake | 9 |
| 10 | Nathan | 10 |
| 11 | Athelstan | 7 |
| 12 | Sue | 5 |
| 13 | Aethelwulf | 5 |
| 14 | Sargeras | 10 |
| 15 | | |

Figura 8.7

As duas abordagens têm seus prós e contras. Neste capítulo, implementaremos a segunda abordagem (mover um ou mais elementos uma `position` para trás). Para ver a implementação da abordagem de remoção preguiçosa (classe `HashTableLinearProbingLazy`), consulte o código-fonte deste livro. O link para download desse código-fonte foi mencionado no *Prefácio* do livro; o código também pode ser acessado em <http://github.com/loiane/javascript-datastructures-algorithms>.

Método put

Vamos prosseguir e implementar os três métodos que devemos sobrescrever. O primeiro será o método `put`, que vemos a seguir:

```
put(key, value) {
```

```

if (key != null && value != null) {
  const position = this.hashCode(key);
  if (this.table[position] == null) { // {1}
    this.table[position] = new ValuePair(key, value); // {2}
  } else {
    let index = position + 1; // {3}
    while (this.table[index] != null) { // {4}
      index++; // {5}
    }
    this.table[index] = new ValuePair(key, value); // {6}
  }
  return true;
}
return false;
}

```

Como sempre, começaremos obtendo a **position** do hash gerada pelo método **hashCode**. Em seguida, vamos conferir se a **position** contém um elemento ({1}). Se não contiver (esse é o cenário mais simples), adicionaremos o elemento ({2} – que é uma instância da classe **ValuePair**) nesse local.

Se a **position** já estiver ocupada, devemos encontrar a próxima **position** livre (**position** é **undefined** ou **null**); desse modo, criamos uma variável **index** e lhe atribuímos **position + 1** ({3}). Em seguida, verificamos se a **position** está ocupada ({4}) e, em caso afirmativo, incrementamos **index** ({5}), até encontrarmos uma **position** que não esteja ocupada. Depois do laço **while**, o **index** apontará para uma posição livre. Então tudo que precisaremos fazer é atribuir o valor desejado a essa **position** ({6}).

Em algumas linguagens, é necessário definir o tamanho do array. Uma das preocupações ao usar a sondagem linear é a situação em que não há mais posições disponíveis no array. Quando o algoritmo alcançar o final do array, será necessário retornar ao início e continuar iterando pelos seus elementos – se for preciso, também deveremos criar outro array maior e copiar os elementos para esse array. Não temos de nos preocupar com isso em JavaScript, pois não é preciso definir um tamanho para o array (e estamos usando um array associativo); ele pode crescer automaticamente, conforme necessário. Isso faz parte da funcionalidade embutida em JavaScript.

Vamos simular as inserções na tabela hash:

1. Tentaremos inserir **Ygritte**. O valor de hash é 4 e, como a tabela hash acabou de ser criada, a **position 4** está vazia, portanto podemos inserir o nome aí.
2. Tentaremos inserir **Jonathan** na **position 5**. Ela também está vazia, portanto podemos inserir esse nome.
3. Tentaremos inserir **Jamie** na **position 5**, que também tem um valor de hash igual a 5. A **position 5** já está ocupada por **Jonathan**, portanto precisamos acessar **position + 1** ($5 + 1$). A **position 6** está livre, portanto podemos inserir **Jamie** aí.
4. Tentaremos inserir **Jack** na **position 7**. Ela está vazia, portanto podemos inserir o nome sem que haja colisão.
5. Tentaremos inserir **Jasmine** na **position 8**. Ela está vazia, portanto podemos inserir o nome sem que haja colisão.
6. Tentaremos inserir **Jake** na **position 9**. Ela está vazia, portanto podemos inserir o nome sem que haja colisão.
7. Tentaremos inserir **Nathan** na **position 10**. Ela está vazia, portanto podemos inserir o nome sem que haja colisão.
8. Tentaremos inserir **Athelstan** na **position 7**. A **position 7** já está ocupada por **Jack**, portanto precisamos acessar **position + 1** ($7 + 1$). A **position 8** também não está livre, portanto iteramos até a próxima posição livre, que é **11**, e inserimos **Athelstan**.
9. Tentaremos inserir **Sue** na **position 5**. As posições de **5** a **11** estão todas ocupadas, portanto vamos para a **position 12** e inserimos **Sue**.
10. Tentaremos inserir **Aethelwulf** na **position 5**. As posições de **5** a **12** estão todas ocupadas, portanto vamos para a **position 13** e inserimos **Aethelwulf**.
11. Tentaremos inserir **Sargeras** na **position 10**. As posições de **10** a **13** estão todas ocupadas, portanto vamos para a **position 14** e inserimos **Sargeras**.

Método **get**

Agora que adicionamos os nossos elementos, vamos implementar a função **get** para que possamos obter os seus valores, assim:

```

get(key) {
  const position = this.hashCode(key);
  if (this.table[position] != null) { // {1}
    if (this.table[position].key === key) { // {2}
      return this.table[position].value; // {3}
    }
    let index = position + 1; // {4}
    while (this.table[index] != null && this.table[index].key !== key) { // {5}
      index++;
    }
    if (this.table[index] != null && this.table[index].key === key) { // {6}
      return this.table[position].value; // {7}
    }
  }
  return undefined; // {8}
}

```

Para obter o valor de uma chave, inicialmente devemos verificar se a **key** existe ({1}). Se não existir, é sinal de que o valor não está na tabela hash, portanto podemos devolver **undefined** ({8}). Se existir, devemos verificar se o valor que estamos procurando é aquele que está na **position** original do hash ({2}). Se for, basta devolver o seu valor ({3}).

Caso contrário, iteramos pelos elementos de **HashTableLinearProbing**, começando pela próxima **position** do hash ({4}). Continuaremos procurando nas próximas posições da instância de **HashTableLinearProbing** até encontrar uma posição que contenha o elemento que estamos procurando, ou até que uma posição vazia seja localizada ({5}). Ao sair do laço **while**, verificamos se a **key** do elemento coincide com a **key** que estamos procurando ({6}); em caso afirmativo, devolveremos o seu **value** ({7}). Se, depois de iterar pela **table**, a posição **index** for **undefined** ou **null**, é sinal de que a chave não existe, e devolveremos **undefined** ({8}).

Método remove

O método **remove** é muito parecido com o método **get**, sendo declarado da seguinte maneira:

```

remove(key) {
  const position = this.hashCode(key);
  if (this.table[position] != null) {
    if (this.table[position].key === key) {
      delete this.table[position]; // {1}
    }
  }
}

```

```

        this.verifyRemoveSideEffect(key, position); // {2}
        return true;
    }
    let index = position + 1;
    while (this.table[index] != null && this.table[index].key !== key) {
        index++;
    }
    if (this.table[index] != null && this.table[index].key === key) {
        delete this.table[index]; // {3}
        this.verifyRemoveSideEffect(key, index); // {4}
        return true;
    }
}
return false;
}

```

No método **get**, quando encontramos a **key** que estamos procurando, devolvemos o seu valor. No método **remove**, apagamos o elemento da tabela hash com **delete**. Podemos encontrar o elemento diretamente na **position** original do hash ({1}) ou em uma posição diferente caso tenha havido um tratamento de colisão ({3}). Como não sabemos se há mais elementos com o mesmo hash em uma position diferente, temos de verificar se a remoção tem algum efeito colateral. Se houver, é necessário mover o elemento que sofreu colisão uma **position** para trás a fim de que não tenhamos posições vazias ({2} e {4}). Para isso, criaremos um método auxiliar com essa lógica, declarado da seguinte maneira:

```

verifyRemoveSideEffect(key, removedPosition) {
    const hash = this.hashCode(key); // {1}
    let index = removedPosition + 1; // {2}
    while (this.table[index] != null) { // {3}
        const posHash = this.hashCode(this.table[index].key); // {4}
        if (posHash <= hash || posHash <= removedPosition) { // {5}
            this.table[removedPosition] = this.table[index]; // {6}
            delete this.table[index];
            removedPosition = index;
        }
        index++;
    }
}

```

O método **verifyRemoveSideEffect** recebeu dois parâmetros: a **key** que foi removida e a **position** da qual a chave foi removida. Inicialmente obtemos o hash da **key** removida ({1} – poderíamos também passar esse valor como parâmetro para esse método). Então começamos a iterar pela

`table`, partindo da próxima `position` (`{2}`), até encontrarmos uma posição vazia (`{3}`). Quando uma posição livre for encontrada, é sinal de que os elementos estão nos lugares corretos e nenhum deslocamento (ou outros deslocamentos) será necessário. Ao iterar pelos elementos seguintes, temos de calcular o `hash` do elemento da `position` atual (`{4}`). Se o `hash` do elemento atual for menor ou igual ao `hash` original (`{5}`), ou se o hash do elemento atual for menor ou igual a `removedPosition` (que é o `hash` da última `key` removida), é sinal de que devemos mover o elemento atual para `removedPosition` (`{6}`). Ao fazer isso, podemos apagar o elemento atual com `delete` (pois ele foi copiado para `removedPosition`). Também é necessário atualizar a `removedPosition` com o `index` atual, e repetimos o processo.

Vamos considerar a tabela hash que criamos para exemplificar o método `put`. Suponha que queremos remover `Jonathan` dessa tabela. Vamos simular essa remoção na tabela hash:

1. Encontraremos e removeremos `Jonathan` da `position` 5. A `position` 5 agora está livre. Verificaremos os efeitos colaterais.
2. Acessamos a `position` 6, onde armazenamos `Jamie`, também com `hash` 5. Seu `hash` $5 \leq \text{hash } 5$, portanto copiaremos `Jamie` para a `position` 5 e apagaremos `Jamie`. A `position` 6 agora está livre e verificaremos a próxima `position`.
3. Acessamos a `position` 7, onde armazenamos `Jack`, com `hash` 7. Seu `hash` $7 > \text{hash } 5$, e `hash` $7 > \text{removedPosition } 6$, portanto não é necessário movê-lo. A próxima `position` também está ocupada, portanto vamos verificará-la.
4. Acessamos a `position` 8, onde armazenamos `Jasmine`, com `hash` 8. Seu `hash` $8 > \text{hash } 5$ de `Jasmine`, e `hash` $8 > \text{removedPosition } 6$, portanto não é necessário movê-la. A próxima `position` também está ocupada, portanto vamos verificará-la.
5. Acessamos a `position` 9, onde armazenamos `Jake`, com `hash` 9. Seu `hash` $9 > \text{hash } 5$, e `hash` $9 > \text{removedPosition } 6$, portanto não é necessário movê-lo. A próxima `position` também está ocupada, portanto vamos verificará-la.
6. Repetiremos esse mesmo processo até a `position` 12.

7. Acessamos a **position** 12, onde armazenamos **Sue**, com **hash** 5. Sua **hash** 5 <= **hash** 5, e **hash** 5 <= **removedPosition** 6, portanto copiaremos **Sue** para a **position** 6 e apagaremos **Sue** da **position** 12. A **position** 12 agora está livre. A próxima **position** também está ocupada, portanto vamos verificá-la.
8. Acessamos a **position** 13, onde armazenamos **Aethelwulf**, com **hash** 5. Seu **hash** 5 <= **hash** 5, e **hash** 5 <= **removedPosition** 12, portanto copiaremos **Aethelwulf** para a **position** 12 e apagaremos a **position** 13. A **position** 13 agora está livre. A próxima **position** também está ocupada, portanto vamos verificá-la.
9. Acessamos a **position** 14, onde armazenamos **Sargeras** com **hash** 10. O **hash** 10 > **hash** 5 de **Aethelwulf**, mas **hash** 10 <= **removedPosition** 13, portanto copiaremos **Sargeras** para a **position** 13 e apagaremos a **position** 14. A **position** 14 agora está livre. A próxima **position** está livre, portanto a execução terminou.

Criando funções melhores de hash

A função de hash lose-lose que implementamos não é uma boa função de hash, conforme podemos concluir (há muitas colisões). Teríamos muitas colisões se usássemos essa função. Uma boa função de hash apresenta determinados fatores: o tempo para inserir e acessar um elemento (desempenho), além de uma baixa probabilidade de colisões. Podemos encontrar várias implementações diferentes na internet, ou podemos fazer a nossa própria implementação.

Outra função de hash simples, que podemos implementar e é melhor que a função de hash lose lose, é a **djb2**, que vemos a seguir:

```
djb2HashCode(key) {
  const tableKey = this.toStrFn(key); // {1}
  let hash = 5381; // {2}
  for (let i = 0; i < tableKey.length; i++) { // {3}
    hash = (hash * 33) + tableKey.charCodeAt(i); // {4}
  }
  return hash % 1013; // {5}
}
```

Depois de transformar a chave em uma string ({1}), o método **djb2HashCode** inicializa a variável **hash** com um número primo ({2}), a

maioria das implementações usa 5381); em seguida, iteramos pelos caracteres da string que representa a **key** ({3}), multiplicamos o valor do **hash** por 33 (usado como um número mágico) e somamos com o valor ASCII do caractere ({4}).

Por fim, usamos o resto da divisão do total por outro número primo aleatório ({5}), maior que o tamanho que achamos que a instância de **HashTable** poderá ter. Em nosso cenário, vamos considerar um tamanho igual a 1.000.

Se executarmos as inserções da seção de *Sondagem linear* novamente, o resultado a seguir será obtido se **djb2HashCode** for usado no lugar de **loseLoseHashCode**:

```
807 - Ygritte
288 - Jonathan
962 - Jamie
619 - Jack
275 - Jasmine
877 - Jake
223 - Nathan
925 - Athelstan
502 - Sue
149 - Aethelwulf
711 - Sargeras
```

Não há colisões!

Essa não é a melhor função de hash que existe, mas é uma das mais recomendadas pela comunidade.

Há também algumas técnicas para criar funções de hash para chaves numéricas. Você pode ver uma lista delas e as implementações em <http://goo.gl/VtdN2x>.

Classe Map da ES2015

A ECMAScript 2015 introduziu uma classe **Map** como parte da API de JavaScript. Desenvolvemos a nossa classe **Dictionary** com base na classe **Map** da ES2015.

Você pode ver os detalhes da implementação da classe **Map** da ECMAScript 2015 em <https://developer.mozilla.org/en->

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Map (ou em <http://goo.gl/dm8VP6>).

Vamos ver como podemos usar a classe **Map** nativa também. Usaremos os mesmos exemplos que utilizamos para testar a nossa classe **Dictionary**:

```
const map = new Map();
map.set('Gandalf', 'gandalf@email.com');
map.set('John', 'johnsnow@email.com');
map.set('Tyrion', 'tyrion@email.com');
console.log(map.has('Gandalf')); // true
console.log(map.size); // 3
console.log(map.keys()); // MapIterator {"Gandalf", "John", "Tyrion"}
console.log(map.values()); // MapIterator {"gandalf@email.com",
"johnsnow@email.com", "tyrion@email.com"}
console.log(map.get('Tyrion')) // tyrion@email.com
```

A diferença entre a nossa classe **Dictionary** e a classe **Map** da ES2015 está no fato de os métodos **values** e **keys** devolverem um **Iterator** (que conhecemos no Capítulo 3, *Arrays*) em vez de devolver o array com os valores ou as chaves. Outra diferença é que desenvolvemos um método **size** para devolver o número de valores armazenados no mapa. A classe **Map** da ES2015 tem uma propriedade chamada **size**.

Podemos também chamar o método **delete** para remover um elemento do mapa usando o código a seguir:

```
map.delete('John');
```

O método **clear** também reinicia a estrutura de dados de **map**. Essa é a mesma funcionalidade que implementamos na classe **Dictionary**.

Classes WeakMap e WeakSet da ES2015

Junto com as duas novas estruturas de dados **Set** e **Map**, a ES2015 introduziu também uma versão dessas classes com tipos fracos: **WeakMap** e **WeakSet**.

Basicamente, a única diferença entre as classes **Map** ou **Set** e suas versões fracas são:

- As classes **WeakSet** ou **WeakMap** não têm os métodos **entries**, **keys** e **values**.
- É possível usar somente objetos como chaves.

O motivo para a criação e o uso dessas duas classes tem a ver com

desempenho. Como **WeakSet** e **WeakMap** têm tipos fracos (usam objetos como chave), não há nenhuma referência forte para as chaves. Esse comportamento permite que o coletor de lixo de JavaScript limpe uma entrada completa do mapa ou do conjunto.

Outra vantagem das versões fracas é que só poderemos obter um valor se tivermos a sua chave. Como essas classes não têm os métodos iteradores (**entries**, **keys** e **values**), não há maneiras de obter um valor, a menos que você saiba qual é a chave. Isso confirma a nossa opção de usar a classe **WeakMap** para encapsular as propriedades privadas das classes ES2015, conforme vimos no Capítulo 4, *Pilhas*.

O código a seguir é um exemplo do que podemos fazer com a classe **WeakMap**:

```
const map = new WeakMap();
const ob1 = { name: 'Gandalf' }; // {1}
const ob2 = { name: 'John' };
const ob3 = { name: 'Tyrion' };
map.set(ob1, 'gandalf@email.com'); // {2}
map.set(ob2, 'johnsnow@email.com');
map.set(ob3, 'tyrion@email.com');
console.log(map.has(ob1)); // true {3}
console.log(map.get(ob3)); // tyrion@email.com {4}
map.delete(ob2); // {5}
```

Podemos ainda usar o método **set** da classe **WeakMap** ({2}). No entanto, como ele não nos permite usar string nem qualquer outro tipo de dado primitivo (valores numéricos, string ou booleanos), devemos transformar o nome em um objeto ({1}).

Para procurar um valor específico ({3}), acessá-lo ({4}) e apagá-lo ({5}), devemos passar o objeto criado como uma chave.

A mesma lógica se aplica à classe **WeakSet**.

Resumo

Neste capítulo, conhecemos os dicionários e os métodos para adicionar, remover e acessar os elementos, entre outros. Também vimos a diferença entre um dicionário e um conjunto.

Descrevemos o hashing, discutimos como criar uma estrutura de dados de tabela hash (ou mapa hash), vimos como adicionar, remover e acessar elementos, e como criar funções de hash. Aprendemos a lidar com colisões

em uma tabela hash usando duas técnicas diferentes.

Também discutimos o uso da classe `Map` da ES2105, assim como das classes `WeakMap` e `WeakSet`.

No próximo capítulo, veremos a técnica da recursão.

CAPÍTULO 9

Recursão

Nos capítulos anteriores, conhecemos diferentes estruturas de dados iterativas. Começando pelo próximo capítulo, passaremos a usar um método especial para facilitar a escrita dos algoritmos usados para acessar as estruturas de dados de árvore (tree) e **grafo** (graph), que é a **recursão**. Contudo, antes de começar a explorar as árvores e os grafos, é preciso entender como a recursão funciona.

Neste capítulo, abordaremos o seguinte:

- como entender a recursão;
- como calcular o fatorial de um número;
- a sequência de Fibonacci;
- a pilha de chamadas (call stack) de JavaScript.

Entendendo a recursão

Há um famoso ditado de programação que diz o seguinte:

“Para entender a recursão, é preciso entender antes a recursão.”

- Autor desconhecido

A **recursão** é um método para resolução de problemas que consiste em solucionar partes menores do mesmo problema até resolvermos o problema original, mais amplo. Em geral, ela envolve chamar a própria função.

Um método ou função será recursivo se ele puder chamar a si mesmo diretamente, assim:

```
function recursiveFunction(someParam){  
    recursiveFunction(someParam);  
}
```

Uma função também é chamada de recursiva se puder chamar a si mesma indiretamente, desta maneira:

```
function recursiveFunction1(someParam){  
    recursiveFunction2(someParam);
```

```
}
```

```
function recursiveFunction2(someParam){
```

```
    recursiveFunction1(someParam);
```

```
}
```

Suponha que tenhamos de executar `recursiveFunction`. Qual seria o resultado? Nesse caso, a função seria executada indefinidamente. Por esse motivo, toda função recursiva deve ter um **caso de base**, que é uma condição para a qual nenhuma chamada recursiva será feita (um **ponto de parada**) a fim de evitar uma recursão infinita.

Retornando ao ditado mencionado sobre programação, ao compreender o que é a recursão, resolveremos o problema original. Se traduzirmos esse ditado de programação em código JavaScript, poderemos escrevê-lo assim:

```
function understandRecursion(doIunderstandRecursion) {
```

```
    const recursionAnswer = confirm('Do you understand recursion?');
```

```
    if (recursionAnswer === true) { // caso de base ou ponto de parada
```

```
        return true;
```

```
    }
```

```
    understandRecursion(recursionAnswer); // chamada recursiva
```

```
}
```

A função `understandRecursion` continuará chamando a si mesma até que `recursionAnswer` seja sim (`true`). Ter `recursionAnswer` igual a sim é o caso de base no código anterior.

Vamos analisar alguns algoritmos recursivos famosos na próxima seção.

Calculando o fatorial de um número

Em nosso primeiro exemplo de recursão, veremos como calcular o fatorial de um número. O fatorial de um número n é definido por $n!$, e é o resultado da multiplicação dos números de 1 a n .

O fatorial de 5 é representado por $5!$ e é igual a $5 * 4 * 3 * 2 * 1$, resultando em 120.

Fatorial iterativo

Se tentarmos representar os passos para calcular o fatorial de qualquer número n , podemos defini-los assim: $(n) * (n - 1) * (n - 2) * (n - 3) * \dots * 1$.

É possível escrever uma função para calcular o fatorial de um número

usando um laço, como mostrado a seguir:

```
function factorialIterative(number) {  
    if (number < 0) return undefined;  
    let total = 1;  
    for (let n = number; n > 1; n--) {  
        total = total * n;  
    }  
    return total;  
}  
console.log(factorialIterative(5)); // 120
```

Podemos iniciar o cálculo do fatorial começando com o dado `number` e decrementando `n` até que ele tenha um valor igual a 2, pois o fatorial de 1 é 1 e já estará incluso na variável `total`. O fatorial de zero também é 1. O fatorial de números negativos não será calculado.

Fatorial recursivo

Vamos agora tentar reescrever a função `factorialIterative` usando recursão. Antes, porém, vamos determinar todos os passos usando uma definição recursiva.

O fatorial de 5 é calculado com $5 * 4 * 3 * 2 * 1$. O fatorial de 4 ($n - 1$) é calculado com $4 * 3 * 2 * 1$. Calcular $(n - 1)$ é um subproblema que resolvemos para calcular $n!$, que é o problema original, portanto, podemos definir o fatorial de 5 assim:

1. `factorial(5) = 5 * factorial(4)`: podemos calcular $5!$ como $5 * 4!$.
2. `factorial(5) = 5 * (4 * factorial(3))`: precisamos resolver o subproblema de calcular $4!$, que podemos calcular como $4 * 3!$.
3. `factorial(5) = 5 * 4 * (3 * factorial(2))`: precisamos resolver o subproblema de calcular $3!$, que podemos calcular como $3 * 2!$.
4. `factorial(5) = 5 * 4 * 3 * (2 * factorial(1))`: precisamos resolver o subproblema de calcular $2!$, que podemos calcular como $2 * 1!$.
5. `factorial(5) = 5 * 4 * 3 * 2 * (1)`: precisamos resolver o subproblema de calcular $1!$.
6. `factorial(1)` ou `factorial(0)` devolve 1. $1!$ é igual a 1.

Poderíamos dizer também que $1! = 1 * 0!$ e $0!$ também é 1 .

A função **factorial** usando recursão é declarada da seguinte maneira:

```
function factorial(n) {
    if (n === 1 || n === 0) { // caso de base
        return 1;
    }
    return n * factorial(n - 1); // chamada recursiva
}
console.log(factorial(5)); // 120
```

Pilha de chamadas

Conhecemos a estrutura de dados de pilha no Capítulo 4, *Pilhas*. Vamos vê-la em ação em uma aplicação real usando recursão. Sempre que uma função é chamada por um algoritmo, ela vai para o topo da **pilha de chamadas** (call stack). Ao usar recursão, as chamadas de função serão empilhadas umas sobre as outras, por causa da possibilidade de uma chamada depender do resultado da própria chamada anterior.

Podemos ver a **Call Stack** em ação usando o navegador, conforme mostra a Figura 9.1:

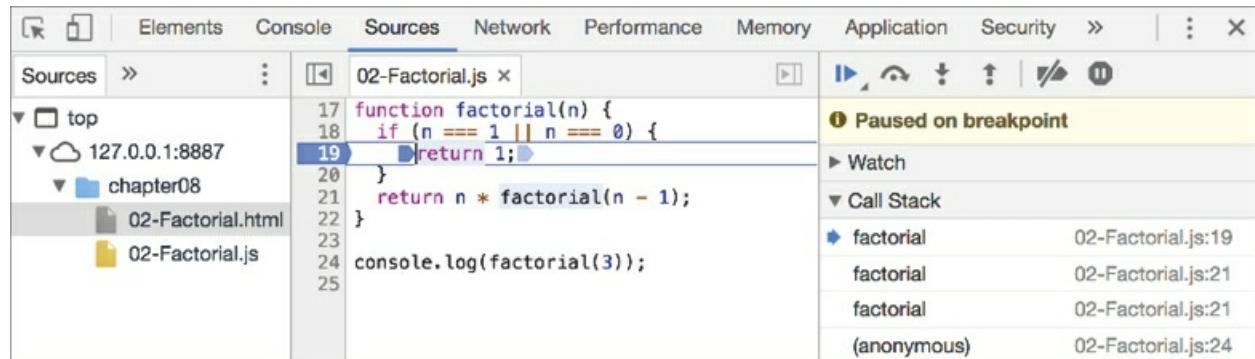


Figura 9.1

Se executarmos **factorial(3)**, abrirmos as ferramentas do desenvolvedor no navegador, acessarmos **Sources** (Fontes) e adicionarmos um breakpoint no arquivo **Factorial.js**, quando **n** tiver um valor igual a **1**, poderemos ver a **Call Stack** com três chamadas da função **factorial**. Se continuarmos a execução, veremos também que, como resultado de **factorial(1)** retornar, a **Call Stack** começará a remover as chamadas de **factorial**.

Também podemos acrescentar **console.trace()** no início da função para

ver igualmente o resultado no console do navegador:

```
function factorial(n) {  
    console.trace();  
    // lógica da função  
}
```

Quando **factorial(3)** for chamada, veremos o resultado a seguir no console:

```
factorial @ 02-Factorial.js:18  
(anonymous) @ 02-Factorial.js:25 // chamada de console.log(factorial(3))
```

Quando **factorial(2)** for chamada, veremos o resultado a seguir no console:

```
factorial @ 02-Factorial.js:18  
factorial @ 02-Factorial.js:22 // factorial(3) está esperando factorial(2)  
(anonymous) @ 02-Factorial.js:25 // chamada de console.log(factorial(3))
```

Por fim, quando **factorial(1)** for chamada, veremos o resultado a seguir no console:

```
factorial @ 02-Factorial.js:18  
factorial @ 02-Factorial.js:22 // factorial(2) está esperando factorial(1)  
factorial @ 02-Factorial.js:22 // factorial(3) está esperando factorial(2)  
(anonymous) @ 02-Factorial.js:25 // chamada de console.log(factorial(3))
```

Podemos representar os passos executados e as ações na pilha de chamadas com o diagrama a seguir:

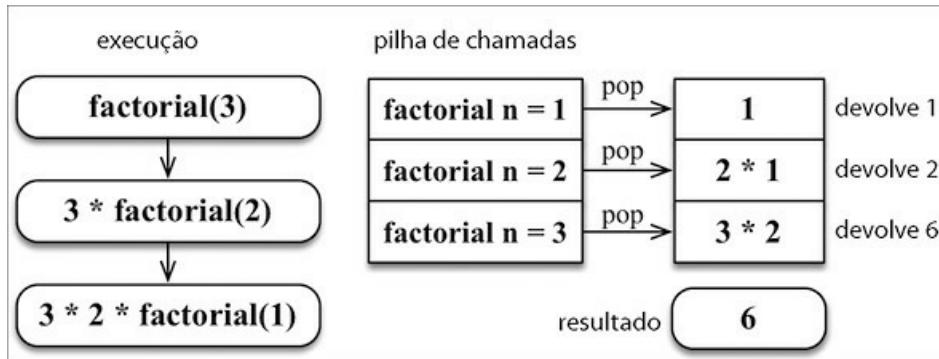


Figura 9.2

Quando **factorial(1)** devolver 1, as chamadas começarão a ser desempilhadas e os resultados serão devolvidos até que **3 * factorial(2)** seja calculado.

Limitação do tamanho da pilha de chamadas em JavaScript

O que acontecerá se esquecermos de adicionar um caso de base para interromper as chamadas recursivas de uma função? Ela não será executada indefinidamente; o navegador lançará um erro, que é conhecido como **erro de stack overflow** (erro de transbordamento de pilha).

Cada navegador tem as próprias limitações, e podemos usar o código a seguir para executar alguns testes:

```
let i = 0;
function recursiveFn() {
    i++;
    recursiveFn();
}
try {
    recursiveFn();
} catch (ex) {
    console.log('i = ' + i + ' error: ' + ex);
}
```

No **Chrome versão 65**, a função é executada 15.662 vezes, e o navegador lança o erro **RangeError: Maximum call stack size exceeded** (tamanho máximo da pilha de chamadas excedido). No **Firefox versão 59**, a função é executada 188.641 vezes, e o navegador lança o erro **InternalError: too much recursion** (excesso de recursão). No **Edge versão 41**, a função é executada 17.654 vezes.

Conforme o seu sistema operacional e o navegador, os valores poderão ser diferentes, mas serão próximos a esses.

A ECMAScript 2015 tem **tail call optimization** (otimização de chamadas finais ou otimização de chamadas de cauda). Se uma chamada de função for a última ação em uma função (em nosso exemplo, é a linha em destaque), ela será tratada com um “jump”, e não com uma “subroutine call” (chamada de sub-rotina). Isso significa que o nosso código pode ser executado indefinidamente na ECMAScript 2015. Por isso que é muito importante ter um caso de base para interromper a recursão.

Para obter mais informações sobre a otimização de chamadas finais, acesse <https://goo.gl/3Rxq7L>.

Sequência de Fibonacci

A **sequência de Fibonacci** é outro problema que podemos resolver usando recursão. Essa sequência é composta de uma série de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 e assim por diante. O número 2 é encontrado por meio da soma $1 + 1$. O número 3 é encontrado somando $1 + 2$, 5 é encontrado somando $2 + 3$, e assim sucessivamente! A sequência de Fibonacci pode ser definida assim:

1. O número de Fibonacci na posição 0 é 0.
2. O número de Fibonacci na posição 1 ou 2 é 1.
3. O número de Fibonacci na posição n (para $n > 2$) é o Fibonacci de ($n - 1$) + Fibonacci de ($n - 2$).

Fibonacci iterativo

Implementamos a função **fibonacci** de modo iterativo, assim:

```
function fibonacciIterative(n) {  
    if (n < 1) return 0;  
    if (n <= 2) return 1;  
    let fibNMinus2 = 0;  
    let fibNMinus1 = 1;  
    let fibN = n;  
    for (let i = 2; i <= n; i++) { // n >= 2  
        fibN = fibNMinus1 + fibNMinus2; // f(n-1) + f(n-2)  
        fibNMinus2 = fibNMinus1;  
        fibNMinus1 = fibN;  
    }  
    return fibN;  
}
```

Fibonacci recursivo

A função **fibonacci** pode ser escrita da seguinte maneira:

```
function fibonacci(n){  
    if (n < 1) return 0; // {1}  
    if (n <= 2) return 1; // {2}  
    return fibonacci(n - 1) + fibonacci(n - 2); // {3}  
}
```

No código anterior, temos os casos de base ({1} e {2}) e a lógica para calcular o Fibonacci para $n > 2$ ({3}).

Se tentarmos descobrir o **fibonacci(5)**, este será o resultado das chamadas efetuadas:

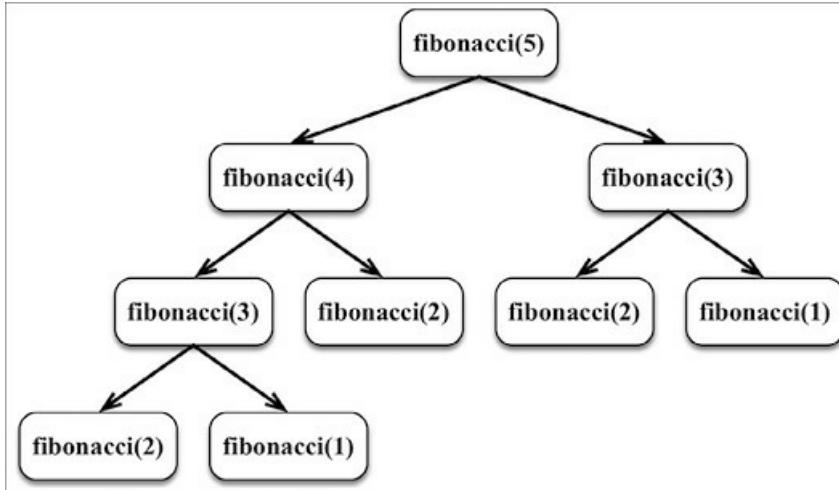


Figura 9.3

Fibonacci com memoização

Há também uma terceira abordagem chamado **memoização** (memoization), que podemos usar para escrever a função `fibonacci`. A memoização é uma técnica de otimização em que armazenamos os valores de resultados anteriores, de modo semelhante a um cache. Se analisarmos as chamadas feitas para calcular `fibonacci(5)`, perceberemos que `fibonacci(3)` foi calculado duas vezes; portanto, podemos armazenar o seu resultado de modo que, quando a processarmos novamente, já teremos esse resultado.

O código a seguir representa a função `fibonacci` escrita com memoização:

```

function fibonacciMemoization(n) {
  const memo = [0, 1]; // {1}
  const fibonacci = (n) => {
    if (memo[n] != null) return memo[n]; // {2}
    return memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo); // {3}
  };
  return fibonacci;
}
  
```

No código anterior, declaramos um array `memo` que fará o cache de todos os resultados calculados ({1}). Se o resultado já tiver sido calculado, ele será devolvido ({2}); caso contrário, calcularemos o resultado e o adicionaremos ao cache ({3}).

Por que usar recursão? É mais rápido?

Vamos fazer uma comparação entre as três funções `fibonacci` distintas que escrevemos neste capítulo:

| Teste | | Ops/seg |
|-----------------------|---------------------------------------|----------------------------------------|
| Iterativa | <code>fibonacciIterative(25)</code> | 38.699.512 ±2,11% mais rápido |
| Recursiva | <code>fibonacci(25)</code> | 1.420 ±1,01% 100% mais lento |
| Com memoização | <code>fibonacciMemoization(25)</code> | 27.697.365 ±3,16% 29% mais lento |

Figura 9.4

A versão **Iterativa** é muito mais rápida que as versões **Recursivas**, portanto isso significa que a recursão é mais lenta. No entanto, observe novamente o código das três versões diferentes. A recursão é mais fácil de entender e, em geral, exige também menos código. Além disso, em alguns algoritmos, a solução **Iterativa** pode não estar disponível, e, com a eliminação da chamada de cauda (tail call), a penalidade da recursão pode até desaparecer.

Desse modo, em geral usamos a recursão porque é mais fácil resolver problemas com ela.

Os testes da função `fibonacci` estão disponíveis em <https://jsperf.com/fibonacci-comparison-jsbook>.

Resumo

Neste capítulo, aprendemos a escrever versões iterativas e recursivas de dois algoritmos famosos: o fatorial de um número e a sequência de Fibonacci. Vimos que, caso um algoritmo recursivo precise calcular o mesmo resultado mais de uma vez, podemos usar uma técnica de otimização chamada memoização.

Também comparamos o desempenho das versões iterativa e recursiva do algoritmo de Fibonacci, e aprendemos que, apesar de a versão iterativa poder ser mais rápida, um algoritmo recursivo é mais legível, sendo mais fácil entender o que ele faz.

No próximo capítulo, conheceremos a estrutura de dados de árvore e criaremos a classe `Tree`, na qual a maioria dos métodos usa recursão.

CAPÍTULO 10

Árvores

Até agora neste livro, descrevemos algumas estruturas de dados sequenciais. A primeira estrutura de dados não sequencial que abordamos no livro foi a **tabela hash**. Neste capítulo, conheceremos outra estrutura de dados não sequencial chamada árvore (**tree**), muito útil para armazenar informações que devam ser encontradas facilmente.

Neste capítulo, veremos:

- a terminologia de árvores;
- como criar uma árvore binária de busca;
- como percorrer uma árvore;
- adição e remoção de nós;
- a árvore AVL.

Estrutura de dados de árvore

Uma árvore é um modelo abstrato de uma estrutura hierárquica. O exemplo mais comum de uma árvore na vida real seria o de uma árvore genealógica ou o organograma de uma empresa, como podemos ver na Figura 10.1.

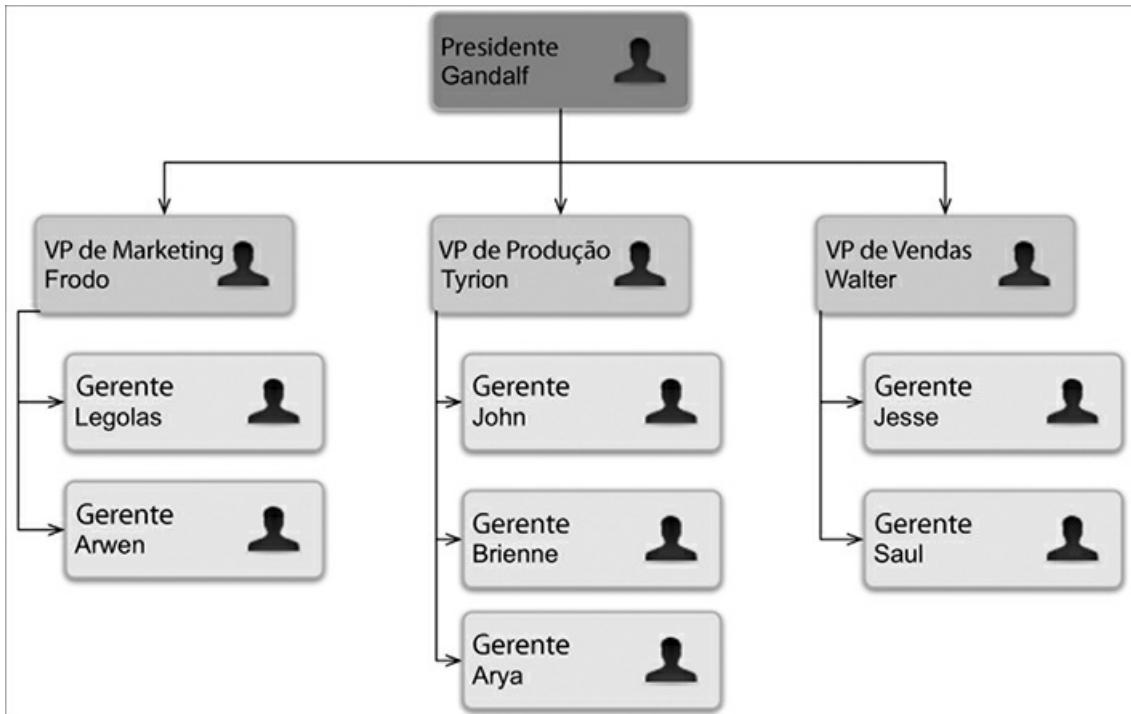


Figura 10.1

Terminologia de árvores

Uma árvore é constituída de **nós** (ou nodos) com um relacionamento pai-filho. Todo nó tem um pai (exceto o primeiro nó no topo) e zero ou mais filhos, como mostra a figura a seguir:

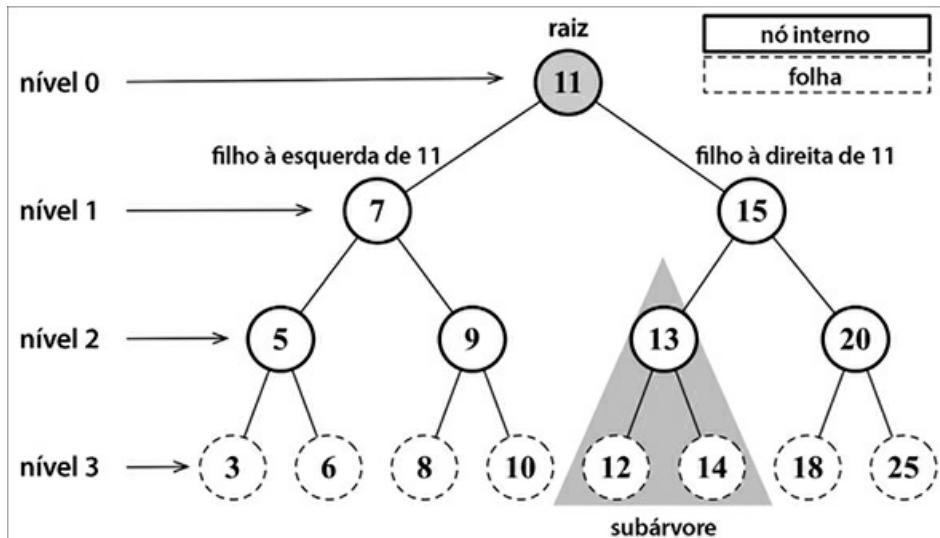


Figura 10.2

O nó no topo de uma árvore é chamado de **raiz** (11). É o nó que não tem

pai. Cada elemento da árvore é chamado de nó. Há **nós internos** e **nós externos**. Um nó interno é um nó que tem pelo menos um filho (7, 5, 9, 15, 13 e 20 são nós internos). Um nó que não tem filhos é chamado de nó externo ou **folha** (3, 6, 8, 10, 12, 14, 18 e 25 são folhas).

Um nó pode ter ancestrais e descendentes. Os ancestrais de um nó (exceto a raiz) são o pai, o avô, o bisavô, e assim sucessivamente. Os descendentes de um nó são os filhos (filho), os netos (neto), os bisnetos (bisneto), e assim por diante. Por exemplo, o nó 5 tem 7 e 11 como seus ancestrais, e 3 e 6 como seus descendentes.

Outra terminologia usada em árvores é o termo **subárvore**. Uma subárvore é composta de um nó e seus descendentes. Por exemplo, os nós 13, 12 e 14 formam uma subárvore na árvore do diagrama anterior.

A profundidade de um nó corresponde ao número de ancestrais. Por exemplo, o nó 3 tem profundidade igual a 3 porque tem três ancestrais (5, 7 e 11).

A altura de uma árvore corresponde à profundidade máxima dos nós. Uma árvore também pode ser dividida em níveis. A raiz está no **nível 0**, seus filhos estão no **nível 1**, e assim sucessivamente. A árvore do diagrama anterior tem altura igual a 3 (a profundidade máxima é 3, conforme mostra o **nível 3** na figura anterior).

Agora que vimos os termos mais importantes relacionados às árvores, podemos conhecê-las melhor.

Árvore binária e árvore binária de busca

Um nó em uma **árvore binária** tem no máximo dois filhos: um filho à esquerda e um filho à direita. Essa definição nos permite escrever algoritmos mais eficazes para inserir, pesquisar e remover nós na/da árvore. As árvores binárias são muito usadas em ciência da computação.

Uma **BST** (**Binary Search Tree**, ou Árvore Binária de Busca) é uma árvore binária, mas permite armazenar somente nós com valores menores do lado esquerdo e nós com valores maiores do lado direito. O diagrama da seção anterior exemplifica uma árvore binária de busca.

Essa é a estrutura de dados com a qual trabalharemos neste capítulo.

Criando as classes Node e BinarySearchTree

Vamos começar criando a nossa classe **Node** que representará cada nó de nossa árvore binária de busca, usando o código a seguir:

```
export class Node {  
    constructor(key) {  
        this.key = key; // {1} valor do nó  
        this.left = null; // referência ao nó que é o filho à esquerda  
        this.right = null; // referência ao nó que é o filho à direita  
    }  
}
```

O diagrama a seguir exemplifica como uma **BST** (Binary Search Tree, ou Árvore Binária de Busca) é organizada no que concerne à estrutura de dados:

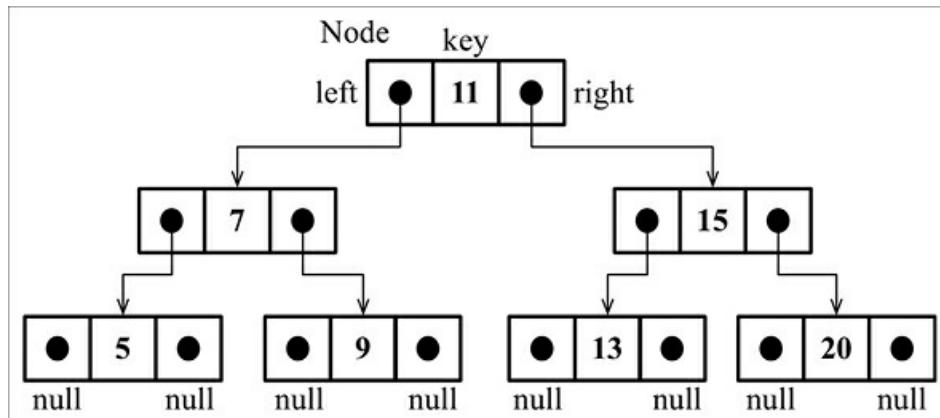


Figura 10.3

Assim como nas listas ligadas, trabalharemos novamente com ponteiros (referências) para representar a conexão entre os nós (chamadas de arestas [edges] na terminologia de árvore). Quando trabalhamos com as listas duplamente ligadas, cada nó tinha dois ponteiros: um para indicar o próximo nó e outro para indicar o nó anterior. Ao trabalhar com árvores, usaremos a mesma abordagem, isto é, trabalharemos também com dois ponteiros. No entanto, um ponteiro referenciará o filho à **esquerda**, enquanto o outro apontará para o filho à **direita**. Por esse motivo, precisaremos de uma classe **Node** que representará cada nó da árvore. Um pequeno detalhe que vale a pena mencionar é que, em vez de chamar o nó propriamente dito de nó ou de item, como fizemos nos capítulos anteriores, nós o chamaremos de **key** ({1}). Uma chave (key) é o termo pelo qual um nó é conhecido na terminologia de árvores.

A seguir, declararemos a estrutura básica de nossa classe `BinarySearchTree`:

```
import { Compare, defaultCompare } from '../util';
import { Node } from './models/node';
export default class BinarySearchTree {
    constructor(compareFn = defaultCompare) {
        this.compareFn = compareFn; // usado para comparar os valores dos nós
        this.root = null; // {1} nó raiz do tipo Node
    }
}
```

Seguiremos o mesmo padrão que usamos na classe `LinkedList` (do Capítulo 6, *Listas ligadas*). Isso significa que declararemos também uma variável para que possamos controlar o primeiro nó da estrutura de dados. No caso de uma árvore, em vez de declarar `head`, temos `root` ({1}).

Na sequência, devemos implementar alguns métodos. A seguir, apresentamos os métodos que criaremos em nossa classe `BinarySearchTree`:

- `insert(key)`: esse método insere uma nova chave na árvore.
- `search(key)`: esse método busca a chave na árvore e devolve `true` se ela existir, e `false` se o nó não existir.
- `inOrderTraverse()`: esse método visita todos os nós da árvore usando um percurso em-ordem (in-order).
- `preOrderTraverse()`: esse método visita todos os nós da árvore usando um percurso pré-ordem (pre-order).
- `postOrderTraverse()`: esse método visita todos os nós da árvore usando um percurso pós-ordem (post-order).
- `min()`: esse método devolve a chave/valor mínimo da árvore.
- `max()`: esse método devolve a chave/valor máximo da árvore.
- `remove(key)`: esse método remove a chave da árvore.

Implementaremos cada um desses métodos nas próximas seções.

Inserindo uma chave na BST

Os métodos que criaremos neste capítulo são um pouco mais complexos do que aqueles implementados nos capítulos anteriores. Usaremos muita recursão em nossos métodos. Se você não tiver familiaridade com recursão,

consulte o Capítulo 9, *Recursão*.

O código a seguir é a primeira parte do algoritmo usado para inserir uma nova **key** em uma árvore:

```
insert(key) {
    if (this.root == null) { // {1}
        this.root = new Node(key); // {2}
    } else {
        this.insertNode(this.root, key); // {3}
    }
}
```

Para inserir um novo nó (ou **key**) em uma árvore, há dois passos que devemos seguir.

O primeiro é verificar se a inserção constitui um caso especial. O caso especial para a BST é o cenário em que o nó que estamos tentando adicionar é o primeiro nó da árvore ({1}). Se for, tudo que temos a fazer é apontar **root** para esse novo nó ({2}) criando uma instância da classe **Node** e atribuindo-a à propriedade **root**. Por causa das propriedades do construtor de **Node**, basta passar o valor que queremos adicionar na árvore (**key**), e seus ponteiros **left** e **right** terão automaticamente o valor **null**.

O segundo passo consiste na adição do nó em uma posição que não seja o **root**. Nesse caso, precisaremos de um método auxiliar ({3}) para nos ajudar a fazer isso; esse método deve ser declarado assim:

```
insertNode(node, key) {
    if (this.compareFn(key, node.key) === Compare.LESS_THAN) { // {4}
        if (node.left == null) { // {5}
            node.left = new Node(key); // {6}
        } else {
            this.insertNode(node.left, key); // {7}
        }
    } else {
        if (node.right == null) { // {8}
            node.right = new Node(key); // {9}
        } else {
            this.insertNode(node.right, key); // {10}
        }
    }
}
```

A função **insertNode** nos ajudará a encontrar o lugar correto para inserir um novo nó. A lista a seguir descreve o que esse método faz:

- Se a árvore não estiver vazia, devemos encontrar um local para adicionar o novo nó. Por esse motivo, chamaremos o método `insertNode` passando o nó raiz e a chave que queremos inserir como parâmetros ({3}).
- Se a chave do nó for menor que a chave do nó atual (nesse caso, é a raiz [{4}]), devemos verificar o filho à esquerda do nó. Observe que estamos usando aqui a função `compareFn`, que pode ser passada no construtor da classe BST para comparar os valores, pois `key` pode não ser um número, mas um objeto complexo. Se não houver um nó à esquerda ({5}), a nova `key` será inserida nesse local ({6}). Caso contrário, devemos descer um nível na árvore chamando `insertNode` recursivamente ({7}). Nesse caso, o nó com o qual faremos a comparação na próxima vez será o filho à esquerda do nó atual (subárvore do nó à esquerda).
- Se a chave do nó for maior que a chave do nó atual e não houver nenhum filho à direita ({8}), a nova chave será inserida nesse local ({9}). Caso contrário, também chamaremos o método `insertNode` recursivamente, porém o novo nó a ser comparado será o filho à direita ({10} – a subárvore do nó à direita).

Vamos aplicar essa lógica em um exemplo para que possamos compreender melhor esse processo. Considere o seguinte cenário: temos uma nova árvore e estamos tentando inserir a sua primeira chave. Nesse caso, executaremos este código:

```
const tree = new BinarySearchTree();
tree.insert(11);
```

Nesse cenário, teremos um único nó em nossa árvore e a propriedade `root` apontará para ele. O código que será executado está nas linhas {1} e {2} de nosso código-fonte.

Vamos supor agora que já temos a árvore a seguir (Figura 10.4):

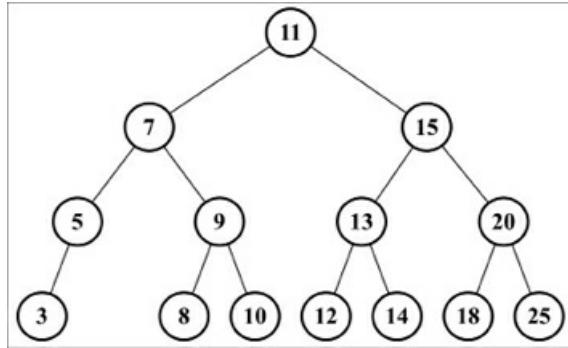


Figura 10.4

O código para criar a árvore do diagrama anterior é uma continuação do código anterior (no qual inserimos a chave 11), conforme vemos a seguir:

```

tree.insert(7);
tree.insert(15);
tree.insert(5);
tree.insert(3);
tree.insert(9);
tree.insert(8);
tree.insert(10);
tree.insert(13);
tree.insert(12);
tree.insert(14);
tree.insert(20);
tree.insert(18);
tree.insert(25);
  
```

Gostaríamos de inserir uma nova chave cujo valor é 6, portanto executaremos o código a seguir também:

```
tree.insert(6);
```

Eis os passos que serão executados:

1. A árvore não está vazia, portanto o código da linha {3} será executado. O código chamará o método `insertNode(root, key[6])`.
2. O algoritmo verificará a linha {4} (`key[6] < root[11]` é `true`), depois verificará a linha {5} (`node.left[7]` não é `null`) e, por fim, passará para a linha {7} chamando `insertNode(node.left[7], key[6])`.
3. Entraremos no método `insertNode` novamente, porém com parâmetros diferentes. O código verificará a linha {4} de novo (`key[6] < node[7]` é `true`), depois verificará a linha {5} (`node.left[5]` não é `null`) e, por fim, passará para a linha {7} chamando `insertNode(node.left[5], key[6])`.

4. Executaremos o método `insertNode` mais uma vez. O código verificará a linha `{4}` novamente (`key[6] < node[5]` é `false`), depois passará para a linha `{8}` (`node.right` é `null` – o nó 5 não tem nenhum filho à direita como descendente) e, por fim, a linha `{9}` será executada, inserindo a chave **6** como o filho à direita do nó 5.
5. Depois disso, as chamadas dos métodos serão desempilhadas e a execução terminará.

Eis o resultado após a chave **6** ter sido inserida na árvore:

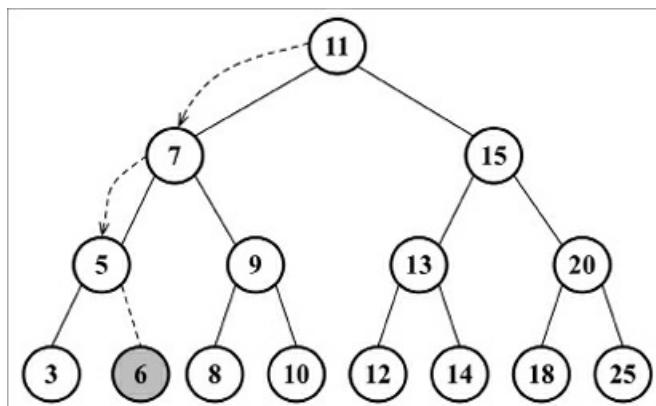


Figura 10.5

Percorrendo uma árvore

Percorrer uma árvore (ou caminhar por ela) é o processo de visitar todos os seus nós e executar uma operação em cada um deles. Entretanto, como devemos fazer isso? Devemos começar do topo da árvore ou da parte inferior? Do lado esquerdo ou do lado direito? Há três abordagens distintas que podem ser usadas para visitar todos os nós de uma árvore: em-ordem (in-order), pré-ordem (pre-order) e pós-ordem (post-order).

Nas próximas seções, exploraremos melhor os usos e as implementações desses três tipos de percurso em árvores.

Percorso em-ordem

Um percurso **em-ordem** (in-order) visita todos os nós de uma BST em ordem crescente, o que significa que todos os nós serão visitados, do menor para o maior. Uma aplicação do percurso em-ordem seria ordenar uma árvore. Vamos observar a sua implementação:

```
inOrderTraverse(callback) {
    this.inOrderTraverseNode(this.root, callback); // {1}
}
```

O método `inOrderTraverse` recebe uma função `callback` como parâmetro, a qual pode ser usada para executar a ação desejada quando visitamos o nó (esse padrão é conhecido como visitante [visitor]; para mais informações sobre esse assunto, consulte http://en.wikipedia.org/wiki/Visitor_pattern). Como a maior parte dos algoritmos que estamos implementando para a BST é recursiva, usaremos um método auxiliar que receberá o nó `root` da árvore (ou subárvore) e a função `callback` ({1}). O método auxiliar está listado a seguir:

```
inOrderTraverseNode(node, callback) {
    if (node != null) { // {2}
        this.inOrderTraverseNode(node.left, callback); // {3}
        callback(node.key); // {4}
        this.inOrderTraverseNode(node.right, callback); // {5}
    }
}
```

Para percorrer uma árvore usando o método em-ordem, devemos inicialmente verificar se o `node` da árvore passado como parâmetro é `null` ({2} – esse é o ponto em que a recursão é interrompida, ou seja, é o caso de base do algoritmo de recursão).

Em seguida, visitamos o nó à esquerda ({3}) chamando a mesma função recursivamente. Então visitamos o nó raiz ({4}) executando uma ação nesse nó (`callback`) e depois visitamos o nó à direita ({5}).

Vamos tentar executar esse método usando a árvore da seção anterior como exemplo, assim:

```
const printNode = (value) => console.log(value); // {6}
tree.inOrderTraverse(printNode); // {7}
```

Em primeiro lugar, devemos criar uma função de callback ({6}). Tudo que faremos é exibir o valor do nó no console do navegador. Então podemos chamar o método `inOrderTraverse` passando a nossa função de callback como parâmetro ({7}). Quando esse código for executado, a saída a seguir será exibida no console (cada número será mostrado em uma linha diferente):

```
3 5 6 7 8 9 10 11 12 13 14 15 18 20 25
```

O diagrama a seguir mostra o caminho que o método `inOrderTraverse` seguiu:

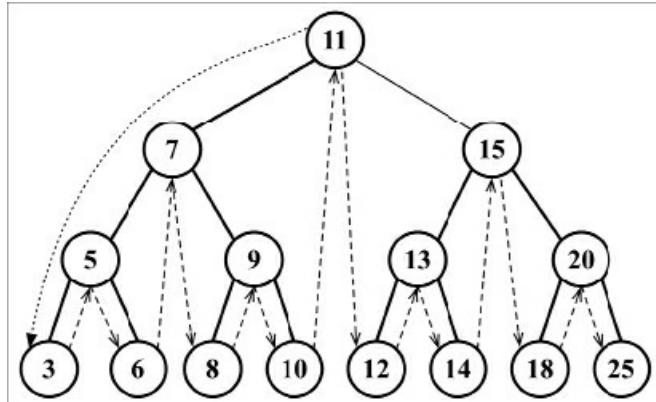


Figura 10.6

Percorso pré-ordem

Um percurso **pré-ordem** (pre-order) visita o nó antes de visitar seus descendentes. Uma aplicação do percurso pré-ordem seria exibir um documento estruturado.

Vamos analisar a sua implementação:

```
preOrderTraverse(callback) {  
    this.preOrderTraverseNode(this.root, callback);  
}
```

Eis a implementação do método `preOrderTraverseNode`:

```
preOrderTraverseNode(node, callback) {  
    if (node != null) {  
        callback(node.key); // {1}  
        this.preOrderTraverseNode(node.left, callback); // {2}  
        this.preOrderTraverseNode(node.right, callback); // {3}  
    }  
}
```

A diferença entre os percursos em-ordem e pré-ordem é que o percurso pré-ordem visita o nó raiz antes ({1}), depois o nó à esquerda ({2}) e, por fim, o nó à direita ({3}), enquanto o percurso em-ordem executa as linhas na seguinte ordem: {2}, {1} e {3}.

A saída a seguir será exibida no console (cada número será mostrado em uma linha diferente):

```
11 7 5 3 6 9 8 10 15 13 12 14 20 18 25
```

O diagrama seguinte mostra o caminho percorrido pelo método `preOrderTraverse`:

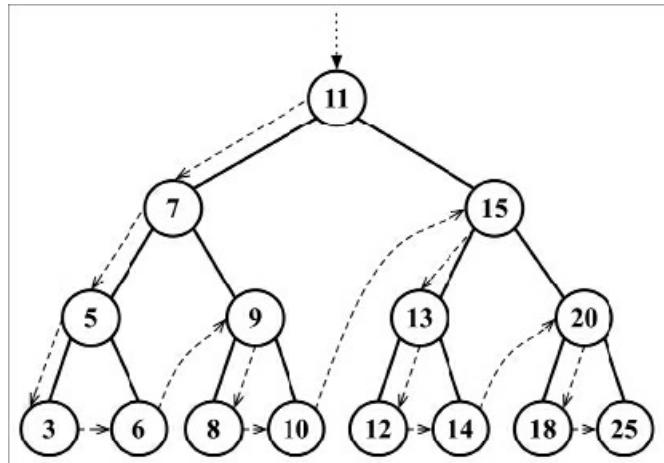


Figura 10.7

Percorso pós-ordem

Um percurso **pós-ordem** (post-order) visita o nó depois de visitar os seus descendentes. Uma aplicação do percurso pós-ordem poderia ser calcular o espaço usado por um arquivo em um diretório e em seus subdiretórios.

Vamos analisar a sua implementação:

```
postOrderTraverse(callback) {  
    this.postOrderTraverseNode(this.root, callback);  
}
```

Eis a implementação de `postOrderTraverseNode`:

```
postOrderTraverseNode(node, callback) {  
    if (node != null) {  
        this.postOrderTraverseNode(node.left, callback); // {1}  
        this.postOrderTraverseNode(node.right, callback); // {2}  
        callback(node.key); // {3}  
    }  
}
```

Nesse caso, o percurso pós-ordem visitará o nó à esquerda ({1}), depois o nó à direita ({2}) e, por último, o nó raiz ({3}).

Os algoritmos para as abordagens em-ordem, pré-ordem e pós-ordem são muito parecidos; a única diferença está na ordem em que as linhas {1}, {2} e {3} são executadas em cada método.

Esta será a saída exibida no console (cada número será exibido em uma

linha diferente):

3 6 5 8 10 9 7 12 14 13 18 25 20 15 11

O diagrama a seguir mostra o caminho percorrido pelo método `postOrderTraverse`:

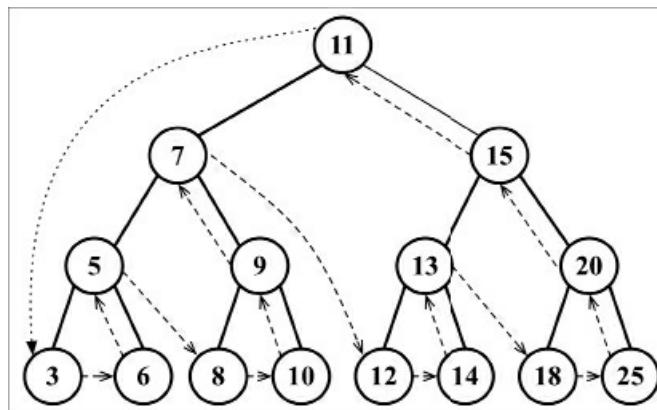


Figura 10.8

Pesquisando valores em uma árvore

Há três tipos de pesquisa geralmente executados em árvores:

- pesquisa de valores mínimos;
- pesquisa de valores máximos;
- pesquisa de um valor específico.

Vamos analisar cada uma dessas pesquisas nas próximas seções.

Pesquisando valores mínimos e máximos

Considere a árvore a seguir (Figura 10.9) em nossos exemplos.

Apenas olhando a Figura 10.9, você poderia encontrar facilmente os valores mínimo e máximo na árvore?

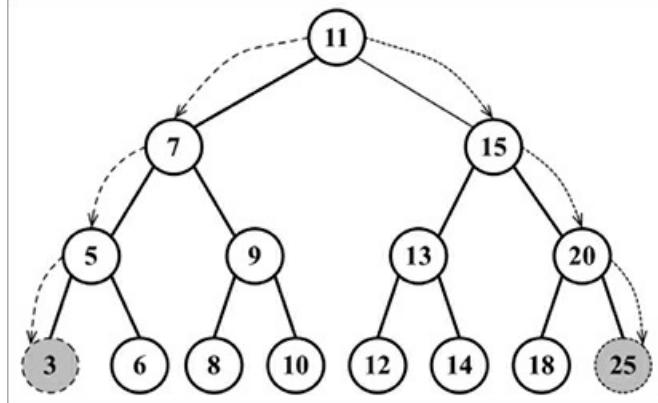


Figura 10.9

Se observar o nó mais à esquerda no último nível da árvore, você verá o valor **3**, que é a menor chave nessa árvore; se observar o nó mais distante à direita (também no último nível da árvore), encontrará o valor **25**, que é a maior chave nessa árvore. Essa informação nos ajuda muito na implementação de métodos que encontrarão os nós mínimo e máximo da árvore.

Inicialmente vamos analisar o método que encontrará a chave mínima da árvore:

```
min() {
    return this.minNode(this.root); // {1}
}
```

O método **min** será o método exposto ao usuário, o qual chama o método **minNode** ({1}), declarado a seguir:

```
minNode(node) {
    let current = node;
    while (current != null && current.left != null) { // {2}
        current = current.left; // {3}
    }
    return current; // {4}
}
```

O método **minNode** nos permite encontrar a chave mínima, a partir de qualquer nó da árvore. Podemos usá-lo para encontrar a chave mínima de uma subárvore ou da própria árvore. Por esse motivo, chamaremos o método **minNode** passando o nó **root** da árvore ({1}), pois queremos encontrar a chave mínima de toda a árvore.

No método **minNode**, percorreremos a aresta esquerda da árvore (linhas {2} e {3}) até encontrar o nó no nível mais alto dela (na extremidade

esquerda).

A lógica usada no método `minNode` é muito semelhante ao código que usamos para iterar até o último nó de uma lista ligada no Capítulo 6, *Listas ligadas*. A diferença, nesse caso, é que estamos iterando até encontrar o nó mais à esquerda da árvore.

De modo semelhante, também temos o método `max`, cujo código apresenta o aspecto a seguir:

```
max() {
    return this.maxNode(this.root);
}
maxNode(node) {
    let current = node;
    while (current != null && current.right != null) { // {5}
        current = current.right;
    }
    return current;
}
```

Para encontrar a chave máxima, percorremos a aresta direita da árvore (`{5}`) até encontrar o último nó na extremidade direita dela.

Assim, para o valor mínimo, sempre percorreremos o lado esquerdo da árvore; para o valor máximo, sempre navegaremos para o lado direito dela.

Pesquisando um valor específico

Nos capítulos anteriores, implementamos também os métodos `find`, `search` e `get` para encontrar um valor específico na estrutura de dados. Implementaremos o método `search` para a BST também. Vamos analisar a sua implementação:

```
search(key) {
    return this.searchNode(this.root, key); // {1}
}
searchNode(node, key) {
    if (node == null) { // {2}
        return false;
    }
    if (this.compareFn(key, node.key) === Compare.LESS_THAN) { // {3}
        return this.searchNode(node.left, key); // {4}
    } else if (
        this.compareFn(key, node.key) === Compare.BIGGER_THAN
    ) { // {5}
```

```

        return this.searchNode(node.right, key); // {6}
    } else {
        return true; // {7}
    }
}

```

Nossa primeira tarefa deve ser declarar o método **search**. Seguindo o padrão dos demais métodos declarados para a BST, usaremos um método auxiliar para nos ajudar na lógica de recursão ({1}).

O método **searchNode** pode ser usado para encontrar uma chave específica na árvore ou em qualquer uma de suas subárvores. Esse é o motivo pelo qual chamaremos esse método na linha {1} passando o nó **root** da árvore como parâmetro.

Antes de iniciar o algoritmo, validaremos se o **node** passado como parâmetro é válido (não é **null** ou **undefined**). Se for inválido, é sinal de que a chave não foi encontrada, e devolveremos **false**.

Se o nó não for **null**, devemos continuar a pesquisa. Se a **key** que estamos procurando for menor que o nó atual ({3}), continuaremos a pesquisa usando a subárvore do filho à esquerda ({4}). Se o valor que estamos procurando for maior que o nó atual ({5}), continuaremos a pesquisa a partir do filho à direita do nó atual ({6}). Caso contrário, significa que a chave que estamos procurando é igual à chave do nó atual, e devolveremos **true** para informar que ela foi encontrada ({7}).

Podemos testar esse método usando o código a seguir:

```

console.log(tree.search(1) ? 'Key 1 found.' : 'Key 1 not found.');
console.log(tree.search(8) ? 'Key 8 found.' : 'Key 8 not found.');

```

Eis a saída exibida:

```

Value 1 not found.
Value 8 found.

```

Vamos descrever mais detalhes sobre como o método foi executado para encontrar a chave 1:

1. Chamamos o método **searchNode** passando o **root** da árvore como parâmetro ({1}). **node[root[11]]** não é **null** ({2}), portanto passamos para a linha {3}.
2. **key[1] < node[11]** é **true** ({3}), portanto passamos para a linha {4} e chamamos o método **searchNode** novamente, passando **node[7]**, **key[1]** como parâmetros.

3. `node[7]` não é `null` ({2}), portanto passamos para a linha {3}.
4. `key[1] < node[7]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[5]`, `key[1]` como parâmetros.
5. `node[5]` não é `null` ({2}), portanto passamos para a linha {3}.
6. `key[1] < node[5]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[3]`, `key[1]` como parâmetros.
7. `node[3]` não é `null` ({2}), portanto passamos para a linha {3}.
8. `key[1] < node[3]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `null`, `key[1]` como parâmetros. `null` foi passado como parâmetro porque `node[3]` é uma folha (ele não tem filhos, portanto o filho à esquerda será `null`).
9. `node` é `null` (linha {2}, o `node` a ser pesquisado nesse caso é `null`), portanto devolvemos `false`.
10. Depois disso, as chamadas dos métodos serão desempilhadas e a execução terminará.

Vamos fazer o mesmo exercício para pesquisar o valor 8, da seguinte maneira:

1. Chamamos o método `searchNode` passando `root` como parâmetro ({1}). `node[root[11]]` não é `null` ({2}), portanto passamos para a linha {3}.
2. `key[8] < node[11]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[7]`, `key[8]` como parâmetros.
3. `node[7]` não é `null` ({2}), portanto passamos para a linha {3}.
4. `key[8] < node[7]` é `false` ({3}), portanto passamos para a linha {5}.
5. `key[8] > node[7]` é `true` ({5}), portanto passamos para a linha {6} e chamamos o método `searchNode` novamente, passando `node[9]`, `key[8]` como parâmetros.
6. `node[9]` não é `null` ({2}), portanto passamos para a linha {3}.
7. `key[8] < node[9]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[8]`,

`key[8]` como parâmetros.

8. `node[8]` não é `null` ({2}), portanto passamos para a linha {3}.
9. `key[8] < node[8]` é `false` ({3}), portanto passamos para a linha {5}.
10. `key[8] > node[8]` é `false` ({5}), portanto passamos para a linha {7} e devolvemos `true` porque `node[8]` é a chave que estamos procurando.
11. Depois disso, as chamadas dos métodos serão desempilhadas e a execução terminará.

Removendo um nó

O próximo e último método que implementaremos para a nossa BST é o método `remove`. Esse é o método mais complexo que implementaremos neste livro. Vamos começar pelo método que estará disponível para ser chamado a partir da instância de uma árvore, assim:

```
remove(key) {
    this.root = this.removeNode(this.root, key); // {1}
}
```

Esse método recebe a `key` que desejamos remover, e chama também `removeNode`, passando `root` e a `key` a ser removida como parâmetros ({1}). Um aspecto muito importante a ser observado é que `root` recebe o valor devolvido pelo método `removeNode`. Entenderemos o porquê em breve.

A complexidade do método `removeNode` se deve aos diferentes cenários com os quais devemos lidar, além do fato de o método ser recursivo.

Vamos observar a implementação de `removeNode`, exibida a seguir:

```
removeNode(node, key) {
    if (node == null) { // {2}
        return null;
    }
    if (this.compareFn(key, node.key) === Compare.LESS_THAN) { // {3}
        node.left = this.removeNode(node.left, key); // {4}
        return node; // {5}
    } else if (
        this.compareFn(key, node.key) === Compare.BIGGER_THAN
    ) { // {6}
        node.right = this.removeNode(node.right, key); // {7}
        return node; // {8}
    } else {
        // key é igual a node.item
    }
}
```

```

// caso 1
if (node.left == null && node.right == null) { // {9}
    node = null; // {10}
    return node; // {11}
}
// caso 2
if (node.left == null) { // {12}
    node = node.right; // {13}
    return node; // {14}
} else if (node.right == null) { // {15}
    node = node.left; // {16}
    return node; // {17}
}
// caso 3
const aux = this.minNode(node.right); // {18}
node.key = aux.key; // {19}
node.right = this.removeNode(node.right, aux.key); // {20}
return node; // {21}
}
}

```

Como ponto de parada, temos a linha {2}. Se o nó que estamos analisando for `null`, isso significa que `key` não está presente na árvore e, por esse motivo, devolveremos `null`.

Se o nó não for `null`, precisamos encontrar a `key` na árvore. Portanto, se a `key` que estamos procurando tiver um valor menor que o nó atual ({3}), passaremos para o próximo nó da aresta à esquerda da árvore ({4}). Se `key` for maior que o nó atual ({6}), passaremos para o próximo nó na aresta à direita da árvore ({7}), o que significa que analisaremos as subárvore.

Se encontrarmos a chave que estamos procurando (`key` é igual a `node.key`), teremos três cenários distintos para tratar.

Removendo uma folha

O primeiro cenário é aquele em que o nó é uma folha, sem filhos à esquerda nem à direita ({9}). Nesse caso, tudo que temos a fazer é remover o nó atribuindo-lhe o valor `null` ({9}). No entanto, como aprendemos na implementação das listas ligadas, sabemos que atribuir `null` ao nó não é suficiente, e devemos cuidar também das referências (ponteiros). Nesse caso, o nó não tem nenhum filho, mas tem um nó pai. Devemos atribuir `null` em seu nó pai, e isso pode ser feito devolvendo `null` ({11}).

Como o nó já tem o valor `null`, a referência do pai ao nó receberá `null` também, e esse é o motivo pelo qual estamos devolvendo o valor de `node` no método `removeNode`. O nó pai sempre receberá o valor devolvido pelo método. Uma alternativa a essa abordagem seria passar o pai e o `node` como parâmetros do método.

Se observarmos as primeiras linhas de código desse método, veremos que estamos atualizando as referências dos ponteiros da esquerda e da direita dos nós nas linhas `{4}` e `{7}`, e estamos também devolvendo o nó atualizado nas linhas `{5}` e `{8}`.

O diagrama a seguir exemplifica a remoção de um nó que é uma folha:

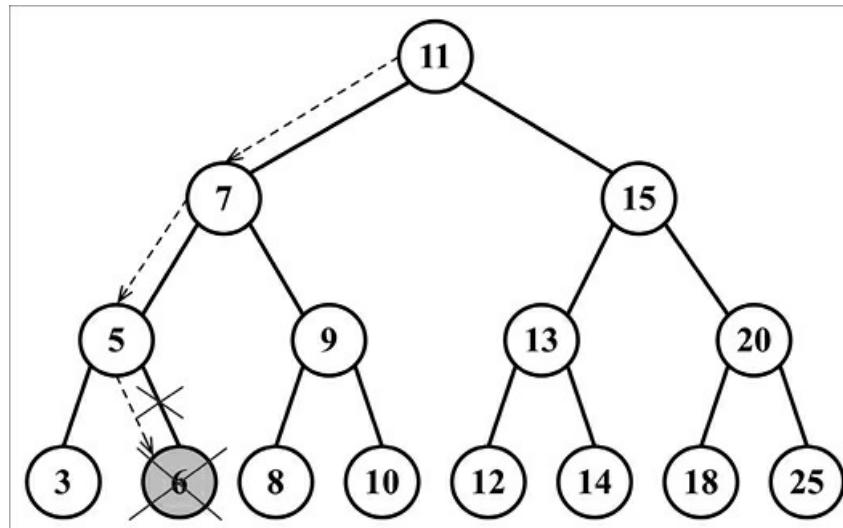


Figura 10.10

Removendo um nó com um filho à esquerda ou à direita

Vamos agora analisar o segundo cenário, aquele em que um nó tem um filho à esquerda ou à direita. Nesse caso, devemos pular esse nó e fazer o ponteiro do pai apontar para o nó filho.

Se o nó não tiver um filho à esquerda (`{12}`), significa que ele tem um filho à direita, portanto modificaremos a referência do nó para o seu filho à direita (`{13}`) e devolveremos o nó atualizado (`{14}`). Faremos o mesmo se o nó não tiver o filho à direita (`{15}`); atualizaremos a referência do nó para o seu filho à esquerda (`{16}`) e devolveremos o valor atualizado (`{17}`).

O diagrama a seguir exemplifica a remoção de um nó com apenas um filho à esquerda ou à direita:

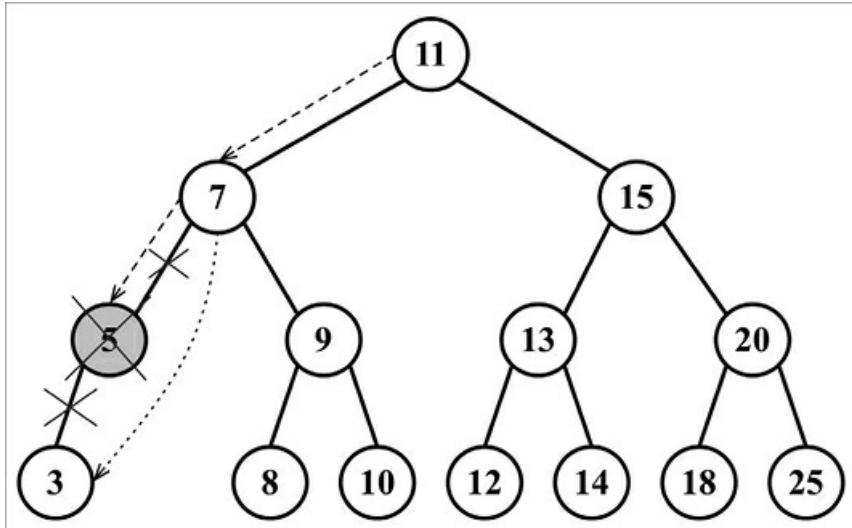


Figura 10.11

Removendo um nó com dois filhos

Agora chegamos ao terceiro cenário, o mais complexo: aquele em que o nó que estamos tentando remover tem dois filhos: um à direita e outro à esquerda. Para remover um nó com dois filhos, há quatro passos que devem ser executados, da seguinte maneira:

1. Depois que o nó que queremos remover for encontrado, precisamos encontrar o nó mínimo da subárvore da aresta à sua direita (o seu sucessor, {18}).
2. Em seguida, atualizamos o valor do nó com a chave do nó mínimo de sua subárvore à direita ({19}). Com essa ação, estamos substituindo a chave do nó, o que significa que ele foi removido.
3. No entanto, agora temos dois nós na árvore com a mesma chave, e isso não pode ocorrer. O que devemos fazer agora é remover o nó mínimo da subárvore à direita, pois ele foi transferido para o local em que estava o nó removido ({20}).
4. Por fim, devolvemos a referência ao nó atualizado para o seu pai ({21}).

A implementação do método `findMinNode` é praticamente igual à implementação do método `min`. A única diferença é que, no método `min`, devolvemos apenas a chave, enquanto, no método `findMinNode`, devolvemos o nó.

O diagrama a seguir exemplifica a remoção de um nó com um filho à esquerda e outro à direita:

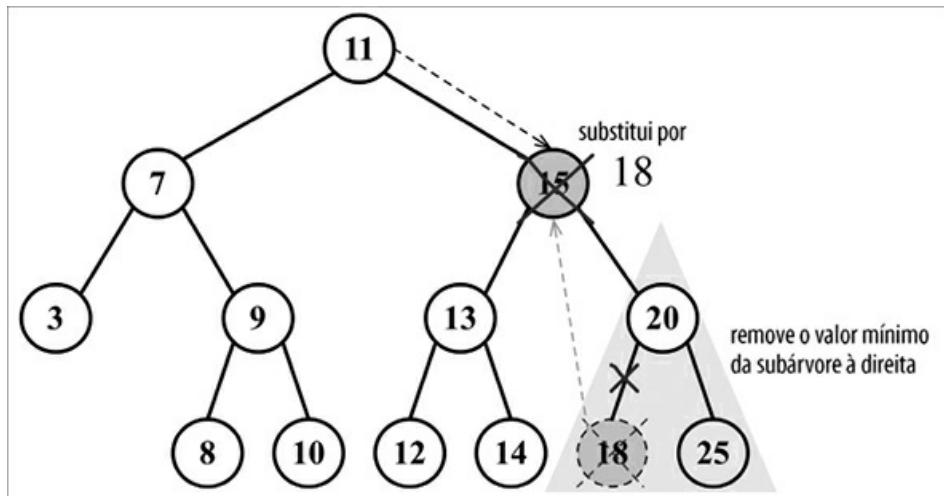


Figura 10.12

Árvores autobalanceadas

Agora que você já sabe como trabalhar com uma BST, poderá mergulhar no estudo das árvores, se quiser.

A BST tem um problema: conforme a quantidade de nós que você adicionar, uma das arestas da árvore poderá ser muito profunda, o que significa que um galho da árvore poderá ter um nível alto, enquanto outro galho poderá ter um nível baixo, como mostra o diagrama a seguir:

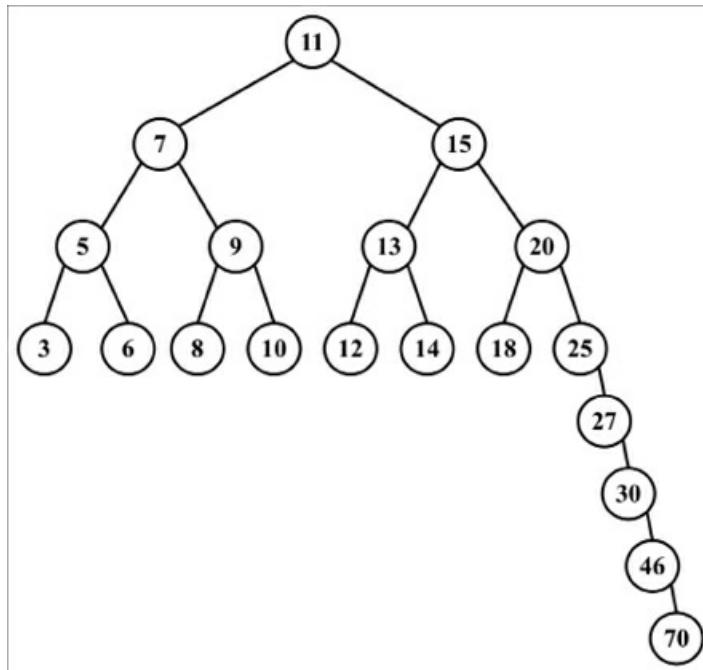


Figura 10.13

Isso pode causar problemas de desempenho na adição, na remoção e na pesquisa de um nó em uma aresta em particular da árvore. Por esse motivo, há uma árvore chamada **AVL** (**Árvore de Adelson-Velskii e Landi**). A árvore AVL é uma BST autobalanceada, o que significa que a altura das subárvore à esquerda e à direita de qualquer nó difere no máximo em 1. Conheceremos melhor a árvore AVL na próxima seção.

Árvore de Adelson-Velskii e Landi (árvore AVL)

A árvore AVL é uma árvore autobalanceada, isto é, uma árvore que tenta se autobalancear sempre que um nó é adicionado ou removido. As alturas das subárvore à esquerda ou à direita de qualquer nó (e em qualquer nível) diferem no máximo em 1. Isso quer dizer que a árvore tentará ser completa sempre que for possível quando adicionar ou remover um nó.

Vamos começar criando a nossa classe **AVLTree**, declarada assim:

```

class AVLTree extends BinarySearchTree {
    constructor(compareFn = defaultCompare) {
        super(compareFn);
        this.compareFn = compareFn;
        this.root = null;
    }
}

```

Como a árvore AVL é uma BST, podemos estender a classe BST que criamos e somente sobrescrever os métodos necessários para manter o balanceamento da árvore AVL, isto é, os métodos `insert`, `insertNode` e `removeNode`. Todos os outros métodos da BST serão herdados pela classe `AVLTree`.

Inserir e remover nós de uma árvore AVL funciona do mesmo modo que em uma BST. No entanto, a diferença na árvore AVL é que precisaremos verificar o seu **fator de balanceamento** (ou fator de balanço) e, se for necessário, aplicaremos a lógica para autobalancear a árvore.

Veremos como criar os métodos `remove` e `insert`; antes, porém, devemos conhecer a terminologia da árvore AVL e suas operações de rotação.

Altura de um nó e o fator de balanceamento

Conforme vimos no início deste capítulo, a altura de um nó é definida como o número máximo de arestas, do nó para qualquer uma de suas folhas. O diagrama a seguir exemplifica uma árvore com a altura de cada nó:

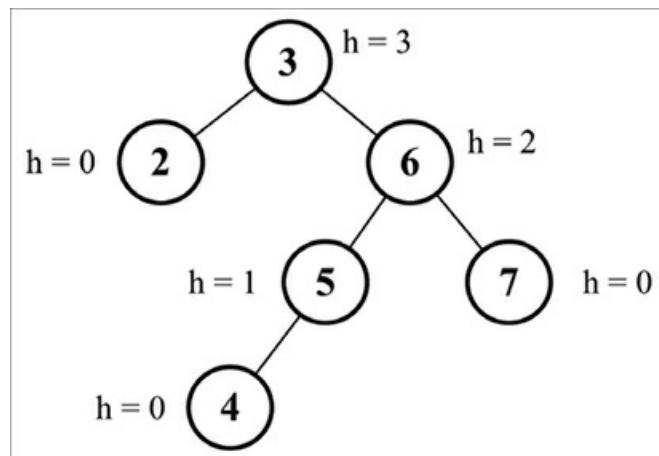


Figura 10.14

A seguir, vemos o código para calcular a altura de um nó:

```
getNodeHeight(node) {  
    if (node == null) {  
        return -1;  
    }  
    return Math.max(  
        this.getNodeHeight(node.left), this.getNodeHeight(node.right)  
    ) + 1;  
}
```

}

Em uma árvore AVL, sempre que um nó for inserido ou removido da árvore, devemos calcular a diferença entre a altura da subárvore do lado direito (hr) e da subárvore do lado esquerdo (hl). O resultado de $hr - hl$ deve ser **0**, **1** ou **-1**. Se o resultado for diferente desses valores, é sinal de que a árvore precisa ser balanceada. Esse conceito se chama **fator de balanceamento** (ou fator de balanço).

O diagrama a seguir exemplifica o fator de balanceamento de algumas árvores (todas as árvores estão平衡adas):

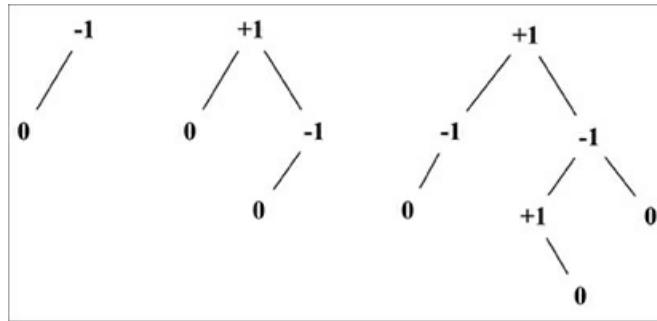


Figura 10.15

O código seguinte calcula o fator de balanceamento de um nó e devolve o seu estado:

```
getBalanceFactor(node) {
    const heightDifference = this.getNodeHeight(node.left) -
        this.getNodeHeight(node.right);
    switch (heightDifference) {
        case -2:
            return BalanceFactor.UNBALANCED_RIGHT;
        case -1:
            return BalanceFactor.SLIGHTLY_UNBALANCED_RIGHT;
        case 1:
            return BalanceFactor.SLIGHTLY_UNBALANCED_LEFT;
        case 2:
            return BalanceFactor.UNBALANCED_LEFT;
        default:
            return BalanceFactor.BALANCED;
    }
}
```

Para evitar ter de trabalhar diretamente com os números do fator de balanceamento no código, criaremos também uma constante em JavaScript que funcionará como um enumerado:

```
const BalanceFactor = {
```

```

UNBALANCED_RIGHT: 1,
SLIGHTLY_UNBALANCED_RIGHT: 2,
BALANCED: 3,
SLIGHTLY_UNBALANCED_LEFT: 4,
UNBALANCED_LEFT: 5
};

```

Veremos o que cada `heightDifference` significa na próxima seção.

Operações de balanceamento – rotações na árvore AVL

Depois de inserir ou remover nós de uma árvore AVL, calcularemos a altura dos nós e verificaremos se a árvore precisa ser balanceada. Há dois processos de平衡amento que podem ser usados: rotação simples ou rotação dupla. Entre a rotação simples e a rotação dupla, há quatro cenários:

- **LL (Left-Left, ou Esquerda-Esquerda)**: é uma rotação simples à direita;
- **RR (Right-Right, ou Direita-Direita)**: é uma rotação simples à esquerda;
- **LR (Left-Right, ou Esquerda-Direita)**: é uma rotação dupla à direita (rotação à esquerda e depois à direita);
- **RL (Right-Left, ou Direita-Esquerda)**: é uma rotação dupla à esquerda (rotação à direita e depois à esquerda).

Rotação Esquerda-Esquerda: rotação simples à direita

Esse caso ocorre quando a altura do filho à esquerda de um nó torna-se maior que a altura do filho à direita, e o filho à esquerda está balanceado ou mais pesado à esquerda, como mostra o diagrama a seguir:

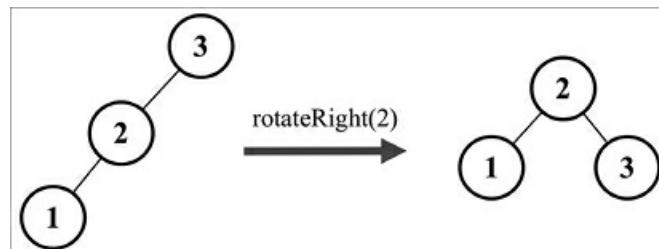


Figura 10.16

Vamos usar um exemplo prático. Considere o diagrama a seguir:

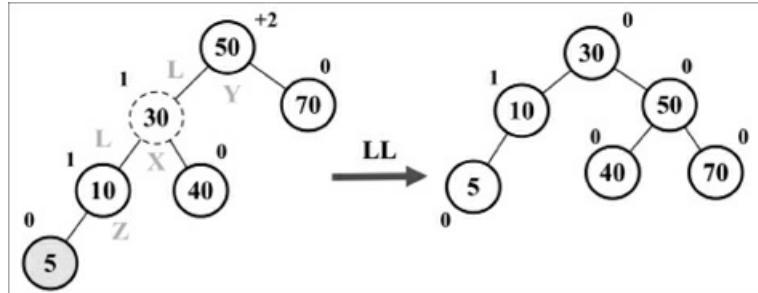


Figura 10.17

Suponha que o nó 5 tenha sido o último nó inserido na árvore AVL. Isso deixaria a árvore desbalanceada (o nó 50, isto é, Y, tem altura igual a +2), portanto precisamos balanceá-la.

Eis os passos que devemos executar para平衡ar a árvore:

- O nó X, que está no meio dos três nós envolvidos no balanceamento (X, Y e Z), ocupará o lugar do nó Y, que tem um fator de balanceamento igual a +2 ({1}).
- A subárvore do lado esquerdo do nó X (nó Z) não será alterada.
- A subárvore do lado direito do nó X será posicionada como a subárvore à esquerda do nó Y ({2}).
- O filho à direita do nó X referenciará o nó Y ({3}).

O código a seguir exemplifica esse processo:

```
rotationLL(node) {
    const tmp = node.left; // {1}
    node.left = tmp.right; // {2}
    tmp.right = node; // {3}
    return tmp;
}
```

Rotação Direita-Direita: rotação simples à esquerda

O caso da rotação direita-direita é o inverso da rotação esquerda-esquerda. Ela ocorre quando a altura do filho à direita de um nó torna-se maior que a altura do filho à esquerda, e o filho à direita está balanceado ou é mais pesado à direita, como mostra o diagrama a seguir:

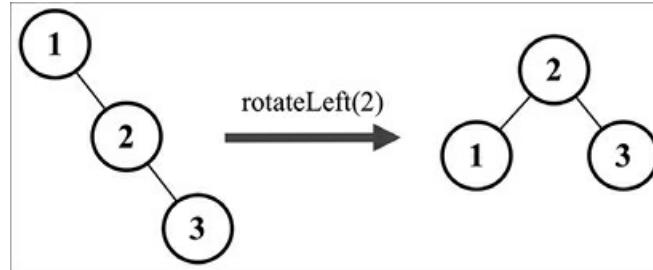


Figura 10.18

Vamos usar um exemplo prático. Considere o diagrama a seguir:

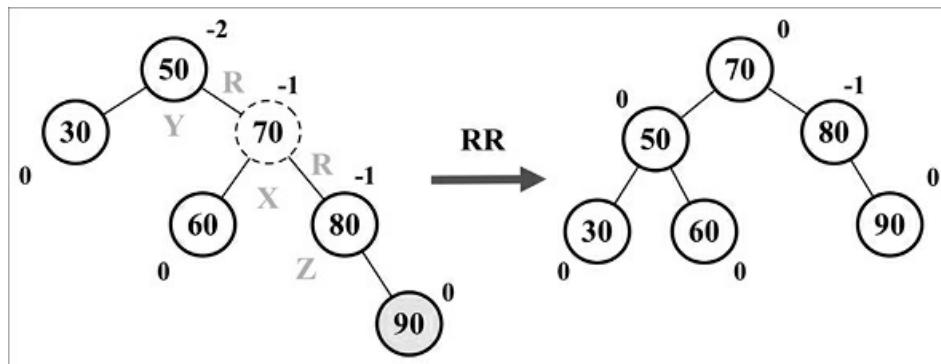


Figura 10.19

Suponha que o nó **90** tenha sido o último nó inserido na árvore AVL. Isso deixaria a árvore desbalanceada (o nó **50**, isto é, **Y**, tem altura igual a **-2**), portanto precisamos balanceá-la. Eis os passos que executaremos para balancear a árvore:

- O nó **X**, que está no meio dos três nós envolvidos no balanceamento (**X**, **Y** e **Z**), ocupará o lugar do nó **Y**, que tem um fator de balanceamento de **-2** (**{1}**).
- A subárvore do lado direito do nó **X** (nó **Z**) não será alterada.
- A subárvore do lado esquerdo do nó **X** será posicionada como a subárvore à direita do nó **Y** (**{2}**).
- O filho à esquerda do nó **X** referenciará o nó **Y** (**{3}**).

O código a seguir exemplifica esse processo:

```
rotationRR(node) {
    const tmp = node.right; // {1}
    node.right = tmp.left; // {2}
    tmp.left = node; // {3}
    return tmp;
}
```

Esquerda-Direita: rotação dupla à direita

Esse caso ocorre quando a altura do filho à esquerda de um nó torna-se maior que a altura do filho à direita, e o filho à esquerda é mais pesado à direita. Nesse cenário, podemos corrigir a árvore fazendo uma rotação à esquerda no filho à esquerda, o que resulta no caso esquerda-esquerda; em seguida, corrigimos a árvore novamente fazendo uma rotação à direita no nó desbalanceado, conforme mostra o diagrama a seguir:

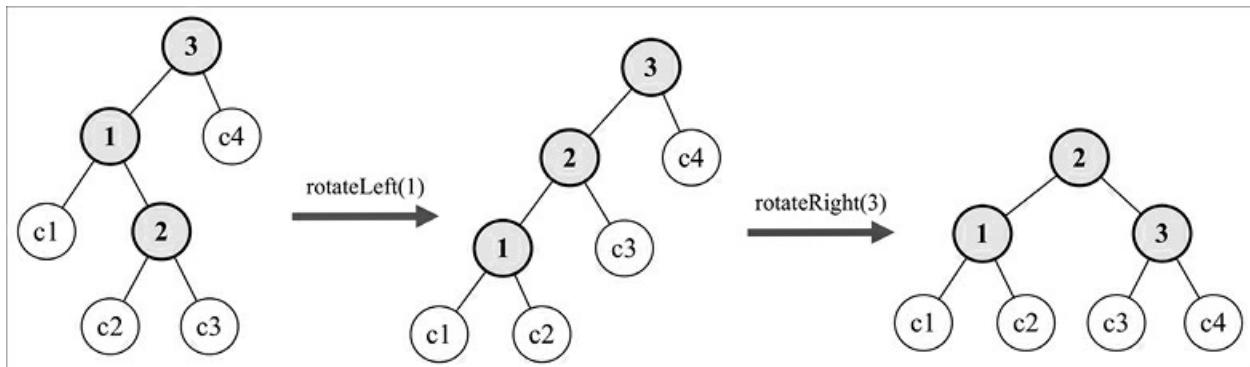


Figura 10.20

Vamos usar um exemplo prático. Considere o diagrama a seguir:

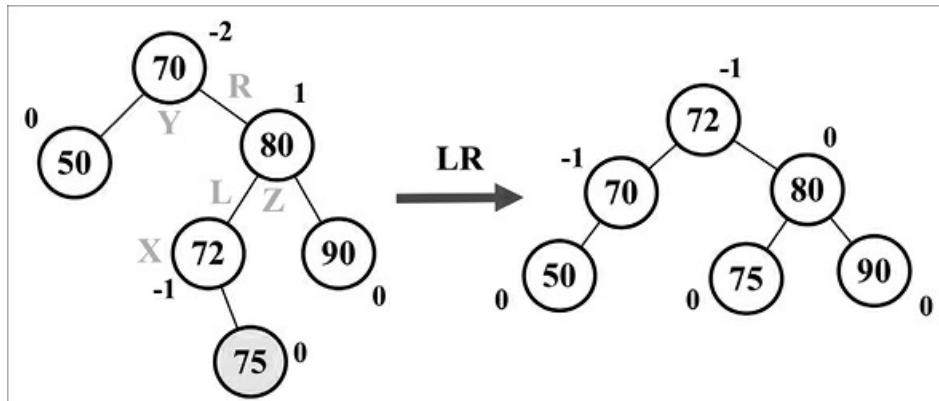


Figura 10.21

Suponha que o nó 75 tenha sido o último nó inserido na árvore AVL. Isso deixaria a árvore desbalanceada (nó 70, isto é, Y, tem altura igual a -2), portanto precisamos balanceá-la. Eis os passos que executaremos para平衡ar a árvore:

- O nó X ocupará o lugar do nó Y, que tem um fator de balanceamento igual a -2.
- A subárvore do lado direito do nó X será posicionada como a subárvore

à esquerda do nó Z.

- A subárvore do lado esquerdo do nó X será posicionada como a subárvore à direita do nó Y.
- O filho à esquerda do nó X referenciará o nó Y.
- O filho à direita do nó X referenciará o nó Z.

Portanto, basicamente, estamos fazendo uma rotação LL antes e, em seguida, uma rotação RR.

O código a seguir exemplifica esse processo:

```
rotationLR(node) {  
    node.left = this.rotationRR(node.left);  
    return this.rotationLL(node);  
}
```

Direita-Esquerda: rotação dupla à esquerda

O caso direita-esquerda é o inverso do caso esquerda-direita. Ele ocorre quando a altura do filho à direita de um nó torna-se maior que a altura do filho à esquerda, e o filho à direita é mais pesado à esquerda. Nesse cenário, podemos corrigir a árvore fazendo uma rotação à direita no filho à direita, o que resulta no caso direita-direita; em seguida, corrigimos a árvore novamente fazendo uma rotação à esquerda no nó desbalanceado, conforme mostra o diagrama a seguir:

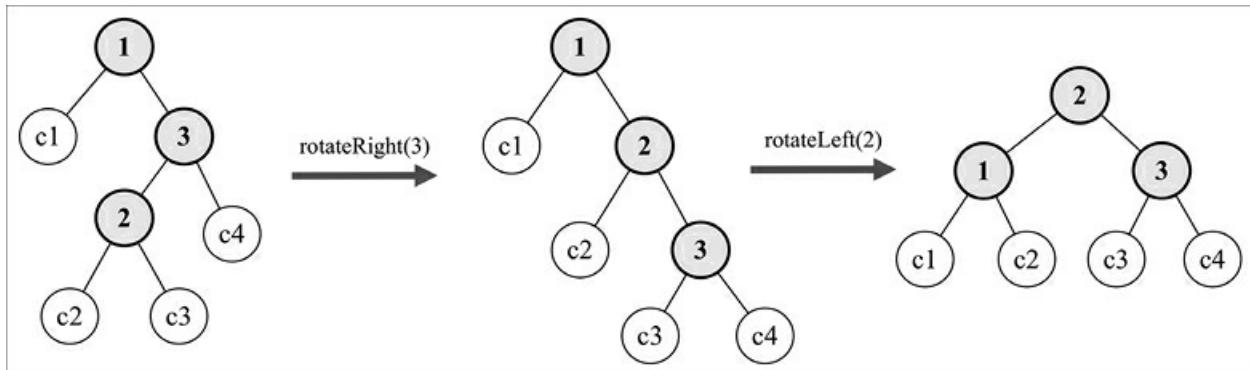


Figura 10.22

Vamos usar um exemplo prático. Considere o diagrama a seguir:

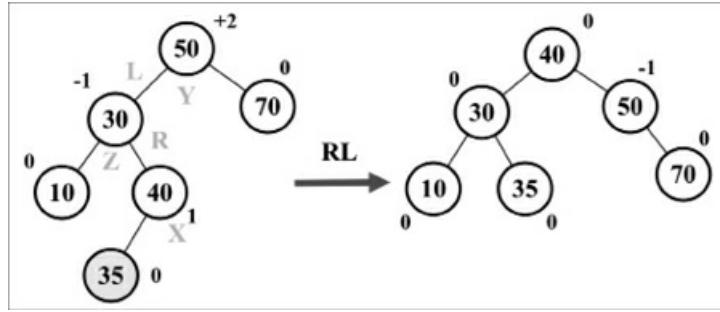


Figura 10.23

Suponha que o nó 35 tenha sido o último nó inserido na árvore AVL. Isso deixaria a árvore desbalanceada (nó 50, isto é, Y tem altura igual a +2), portanto precisamos balanceá-la. Eis os passos que executaremos para balancear a árvore:

- O nó X ocupará o lugar do nó Y, que tem um fator de平衡amento igual a +2.
- A subárvore do lado direito do nó X será posicionada como a subárvore à esquerda do nó Y.
- A subárvore do lado esquerdo do nó X será posicionada como a subárvore à direita do nó Z.
- O filho à direita do nó X referenciará o nó Y.
- O filho à esquerda do nó X referenciará o nó Z.

Portanto, basicamente, estamos fazendo uma rotação RR antes e, em seguida, uma rotação LL.

O código a seguir exemplifica esse processo:

```
rotationRL(node) {
    node.right = this.rotationLL(node.right);
    return this.rotationRR(node);
}
```

Com esses conceitos dominados, podemos agora nos concentrar no código para inserção e remoção de nós da árvore AVL.

Inserindo um nó na árvore AVL

Inserir um nó em uma árvore AVL funciona do mesmo modo que em uma BST. Além de inserir o nó, verificaremos também se a árvore continua balanceada após a inserção; se não estiver, aplicaremos as operações de rotação, conforme forem necessárias.

O código a seguir insere um novo nó em uma árvore AVL:

```
insert(key) {
    this.root = this.insertNode(this.root, key);
}
insertNode(node, key) {
    // insere o nó como em uma BST
    if (node == null) {
        return new Node(key);
    } else if (this.compareFn(key, node.key) === Compare.LESS_THAN) {
        node.left = this.insertNode(node.left, key);
    } else if (this.compareFn(key, node.key) === Compare.BIGGER_THAN) {
        node.right = this.insertNode(node.right, key);
    } else {
        return node; // chave duplicada
    }
    // balanceia a árvore, se for necessário
    const balanceFactor = this.getBalanceFactor(node); // {1}
    if (balanceFactor === BalanceFactor.UNBALANCED_LEFT) { // {2}
        if (this.compareFn(key, node.left.key) === Compare.LESS_THAN) { // {3}
            node = this.rotationLL(node); // {4}
        } else {
            return this.rotationLR(node); // {5}
        }
    }
    if (balanceFactor === BalanceFactor.UNBALANCED_RIGHT) { // {6}
        if (
            this.compareFn(key, node.right.key) === Compare.BIGGER_THAN
        ) { // {7}
            node = this.rotationRR(node); // {8}
        } else {
            return this.rotationRL(node); // {9}
        }
    }
    return node;
}
```

Depois de inserir o nó na árvore AVL, devemos verificar se ela precisa ser balanceada, portanto calcularemos o fator de平衡amento ({1}) de todos os nós, a partir do nó inserido até a raiz da árvore, de forma recursiva, e aplicaremos a rotação correta em cada caso.

Se, após inserir um nó na subárvore do lado esquerdo, a árvore estiver desbalanceada ({2}), teremos de comparar se a chave inserida é menor que a chave do filho à esquerda ({3}). Se for, fazemos uma rotação LL ({4}). Caso contrário, fazemos uma rotação LR ({5}).

Se, após inserir um nó na subárvore do lado direito, a árvore estiver desbalanceada ({6}), teremos de comparar se a chave inserida é maior que a chave do filho à direita ({7}). Se for, fazemos uma rotação RR ({8}). Caso contrário, fazemos uma rotação RL ({9}).

Removendo um nó da árvore AVL

Remover um nó de uma árvore AVL funciona do mesmo modo que em uma BST. Além de remover o nó, verificaremos também se a árvore continua balanceada após a remoção; se não estiver, aplicaremos as operações de rotação, conforme forem necessárias.

O código a seguir remove um nó de uma árvore AVL:

```
removeNode(node, key) {
    node = super.removeNode(node, key); // {1}
    if (node == null) {
        return node; // null, não é necessário balancear
    }
    // verifica se a árvore está balanceada
    const balanceFactor = this.getBalanceFactor(node); // {2}
    if (balanceFactor === BalanceFactor.UNBALANCED_LEFT) { // {3}
        const balanceFactorLeft = this.getBalanceFactor(node.left); // {4}
        if (
            balanceFactorLeft === BalanceFactor.BALANCED ||
            balanceFactorLeft === BalanceFactor.SLIGHTLY_UNBALANCED_LEFT
        ) { // {5}
            return this.rotationLL(node); // {6}
        }
        if (
            balanceFactorLeft === BalanceFactor.SLIGHTLY_UNBALANCED_RIGHT
        ) { // {7}
            return this.rotationLR(node.left); // {8}
        }
    }
    if (balanceFactor === BalanceFactor.UNBALANCED_RIGHT) { // {9}
        const balanceFactorRight = this.getBalanceFactor(node.right); // {10}
        if (
            balanceFactorRight === BalanceFactor.BALANCED ||
            balanceFactorRight === BalanceFactor.SLIGHTLY_UNBALANCED_RIGHT
        ) { // {11}
            return this.rotationRR(node); // {12}
        }
        if (
            balanceFactorRight === BalanceFactor.SLIGHTLY_UNBALANCED_LEFT
        )
    }
}
```

```

        ) { // {13}
    return this.rotationRL(node.right); // {14}
}
}
return node;
}

```

Como a **AVLTree** é uma classe-filha da classe **BinarySearchTree**, podemos usar o método `removeNode` da BST para remover o nó da árvore AVL também ({1}). Depois de remover o nó da árvore AVL, devemos verificar se ela precisa ser balanceada, portanto calcularemos o fator de balanceamento ({1}) de todos os nós, a partir do nó removido até a raiz da árvore, de forma recursiva, e aplicaremos a rotação correta em cada caso.

Se, após a remoção de um nó da subárvore do lado esquerdo, a árvore estiver desbalanceada ({3}), calculamos o fator de balanceamento da subárvore à esquerda ({4}). Se a subárvore à esquerda estiver desbalanceada para a esquerda ({5}), fazemos uma rotação LL ({6}); se a subárvore à esquerda estiver desbalanceada à direita ({7}), fazemos uma rotação LR ({8}).

O último caso é este: se, após a remoção de um nó da subárvore do lado direito, a árvore estiver desbalanceada ({9}), calculamos o fator de balanceamento da subárvore à direita ({10}). Se a subárvore à direita estiver desbalanceada para a direita ({11}), fazemos uma rotação RR ({12}); se a subárvore à direita estiver desbalanceada à esquerda ({13}), fazemos uma rotação LR ({14}).

Árvore rubro-negra

Assim como a árvore AVL, a **árvore rubro-negra** (red-black tree) é também uma árvore binária de busca autobalanceada. Vimos que inserir ou remover um nó da árvore AVL pode provocar rotações; assim, se precisarmos de uma árvore autobalanceada que envolva muitas inserções ou remoções frequentes, a árvore rubro-negra será preferível. Se as inserções e as remoções forem menos frequentes (estamos interessados em operações de busca frequentes), então a árvore AVL será preferível em relação à árvore rubro-negra.

Na árvore rubro-negra, todo nó segue as regras listadas a seguir:

1. Como o nome da árvore sugere, cada nó é vermelho ou preto.

2. A raiz da árvore é preta.
3. Todas as folhas são pretas (os nós representados com referência `NULL`).
4. Se um nó for vermelho, então seus dois filhos serão pretos.
5. Não pode haver dois nós vermelhos adjacentes. Um nó vermelho não pode ter um pai ou um filho vermelho.
6. Todo caminho (path) de um dado nó para qualquer um de seus descendentes (folhas `NULL`) contém o mesmo número de nós pretos.

Vamos começar criando a nossa classe `RedBlackTree`, declarada assim:

```
class RedBlackTree extends BinarySearchTree {
  constructor(compareFn = defaultCompare) {
    super(compareFn);
    this.compareFn = compareFn;
    this.root = null;
  }
}
```

Como a árvore rubro-negra também é uma árvore BST, podemos estender a classe BST que criamos e sobrescrever somente os métodos necessários para manter as propriedades da árvore rubro-negra. Começaremos pelos métodos `insert` e `insertNode`.

Inserindo um nó na árvore rubro-negra

Inserir um nó em uma árvore rubro-negra funciona do mesmo modo que em uma BST. Além de inserir o nó, aplicaremos também uma cor a ele e, após a inserção, verificaremos se a árvore continua obedecendo às regras da árvore rubro-negra e se está balanceada.

O código a seguir insere um novo nó em uma árvore rubro-negra:

```
insert(key: T) {
  if (this.root == null) { // {1}
    this.root = new RedBlackNode(key); // {2}
    this.root.color = Colors.BLACK; // {3}
  } else {
    const newNode = this.insertNode(this.root, key); // {4}
    this.fixTreeProperties(newNode); // {5}
  }
}
```

Se a árvore estiver vazia ({1}), criaremos um nó da árvore rubro-negra ({2}), e, para estarmos de acordo com a regra 2, definiremos a cor da raiz

(`color` de `root`) como preta ({3}). Por padrão, o nó será criado com a cor vermelha ({6}). Se a árvore não estiver vazia, inseriremos o nó em seu lugar correto usando a mesma lógica aplicada para inserir um nó em uma BST ({4}). O método `insertNode`, nesse caso, deve devolver o nó recém-inserido para que possamos verificar se, após a inserção, as regras da árvore rubro-negra continuam sendo satisfeitas ({5}).

Na árvore rubro-negra, o nó precisará de duas propriedades adicionais em comparação com a classe de nó que usamos antes: a cor do nó ({6}) e uma referência para o seu pai ({7}). Este código é mostrado a seguir:

```
class RedBlackNode extends Node {  
    constructor(key) {  
        super(key);  
        this.key = key;  
        this.color = Colors.RED; // {6}  
        this.parent = null; // {7}  
    }  
    isRed() {  
        return this.color === Colors.RED;  
    }  
}
```

O método `insertNode` sobrescrito também é apresentado a seguir:

```
insertNode(node, key) {  
    if (this.compareFn(key, node.key) === Compare.LESS_THAN) {  
        if (node.left == null) {  
            node.left = new RedBlackNode(key);  
            node.left.parent = node; // {8}  
            return node.left; // {9}  
        }  
        else {  
            return this.insertNode(node.left, key);  
        }  
    }  
    else if (node.right == null) {  
        node.right = new RedBlackNode(key);  
        node.right.parent = node; // {10}  
        return node.right; // {11}  
    }  
    else {  
        return this.insertNode(node.right, key);  
    }  
}
```

Como podemos ver, a lógica é a mesma que usamos em uma BST comum.

A diferença, nesse caso, é que estamos mantendo uma referência ao pai do nó inserido (`{8}` e `{10}`) e devolvendo também a referência ao nó (`{9}` e `{11}`) para que possamos verificar as propriedades da árvore em seguida.

Verificando as propriedades da árvore rubro-negra após a inserção

Para verificar se a árvore rubro-negra continua balanceada e ainda atende a todos os seus requisitos, usaremos dois conceitos: recoloração (recoloring) e rotação.

Depois de inserir um novo nó na árvore, esse nó será vermelho. Isso não afetará a regra do número de nós pretos (regra 6), mas poderá afetar a regra 5: dois nós vermelhos adjacentes não podem coexistir. Se o pai do nó inserido for preto, não haverá problemas. No entanto, se o pai do nó inserido for vermelho, teremos uma violação da regra 5. Para resolver essa violação, basta alterar a cor do **pai** do nó, do **avô** do nó e do **tio** do nó (porque estamos alterando também a cor do pai).

O diagrama a seguir exemplifica essa ação:

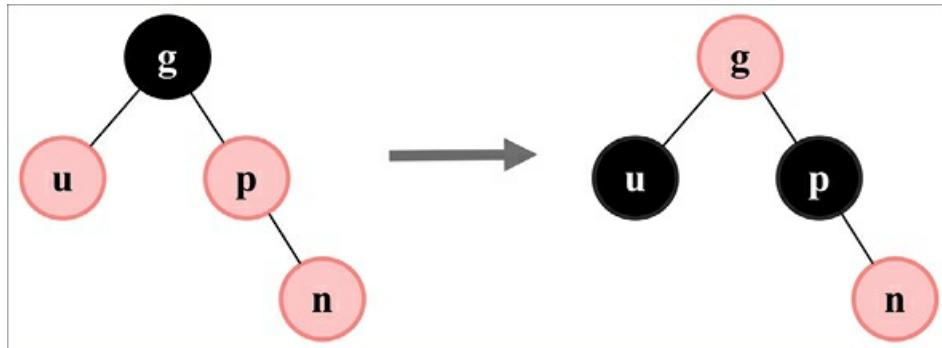


Figura 10.24

Eis o código inicial de `fixTreeProperties`:

```
fixTreeProperties(node) {
    while (node && node.parent && node.parent.color.isRed() // {1}
        && node.color !== Colors.BLACK) { // {2}
        let parent = node.parent; // {3}
        const grandParent = parent.parent; // {4}
        // caso A: o pai é o filho à esquerda
        if (grandParent && grandParent.left === parent) { // {5}
            const uncle = grandParent.right; // {6}
            // caso 1A: o tio do nó também é vermelho - basta recolorir
            if (uncle && uncle.color === Colors.RED) { // {7}

```

```

grandParent.color = Colors.RED;
parent.color = Colors.BLACK;
uncle.color = Colors.BLACK;
node = grandParent; // {8}
}
else {
    // caso 2A: o nó é o filho à direita - rotação à esquerda
    // caso 3A: o nó é o filho à esquerda - rotação à direita
}
}
else { // caso B: o pai é o filho à direita
const uncle = grandParent.left; // {9}
// caso 1B: o tio é vermelho - basta recolorir
if (uncle && uncle.color === Colors.RED) { // {10}
    grandParent.color = Colors.RED;
    parent.color = Colors.BLACK;
    uncle.color = Colors.BLACK;
    node = grandParent;
}
else {
    // caso 2B: o nó é o filho à esquerda - rotação à direita
    // caso 3B: o nó é o filho à direita - rotação à esquerda
}
}
}
this.root.color = Colors.BLACK; // {11}
}

```

Começando pelo nó inserido, verificaremos se o seu pai é vermelho ({1}) e se o nó também não é preto ({2}). Para deixar o nosso código mais fácil de ler, manteremos uma referência ao pai ({3}) e ao avô ({4}) do nó.

Em seguida, verificamos se o pai do nó é um filho à esquerda ({5} – caso A) ou à direita (caso B). No caso 1A, no qual só precisamos recolorir os nós, não fará diferença se o pai for um filho à esquerda ou à direita, mas isso não é verdade nos próximos casos que veremos.

Como teremos de alterar a cor do tio também, precisamos de uma referência para ele ({6} e {9}). Assim, se a cor do tio for vermelha ({7} e {10}), alteramos a cor do avô, do pai e do tio, mudamos também a referência do nó atual para o avô ({8}) e continuamos verificando a árvore para saber se há outras violações.

Para garantir que a cor da raiz é sempre preta (regra 2), atribuímos a cor para a raiz no final do código ({11}).

Caso o tio do nó seja preto, é sinal de que somente recolorir não será suficiente, pois a árvore não está balanceada; portanto, teremos de executar as rotações a seguir:

- **Esquerda-Esquerda (LL)**: o pai é o filho à esquerda do avô, e o nó é o filho à esquerda do pai (caso 3A).
- **Esquerda-Direita (LR)**: o pai é o filho à esquerda do avô, e o nó é o filho à direita do pai (caso 2A).
- **Direita-Direita (RR)**: o pai é o filho à direita do avô, e o nó é o filho à direita do pai (caso 3B).
- **Direita-Esquerda (RL)**: o pai é o filho à direita do avô, e o nó é o filho à esquerda do pai (caso 2B).

Vamos analisar os casos 2A e 3A:

```
// caso 2A: o nó é o filho à direita - rotação à esquerda
if (node === parent.right) {
    this.rotationRR(parent); // {12}
    node = parent; // {13}
    parent = node.parent; // {14}
}
// caso 3A: o nó é o filho à esquerda - rotação à direita
this.rotationLL(grandParent); // {15}
parent.color = Colors.BLACK; // {16}
grandParent.color = Colors.RED; // {17}
node = parent; // {18}
```

Se o pai for um filho à esquerda e o nó for um filho à direita, faremos uma rotação dupla: primeiro uma rotação direita-direita ({12}) e atualizaremos as referências para o nó ({13}) e para o pai ({10}). Depois da primeira rotação, faremos outra rotação novamente usando o avô como o nó de origem ({15}) e atualizaremos as cores do pai ({16}) e do avô ({17}) durante a rotação. Por fim, atualizamos a referência ao nó atual ({18}) para que continuemos verificando a árvore em busca de outras violações.

O caso 2A pode ser exemplificado pelo diagrama a seguir:

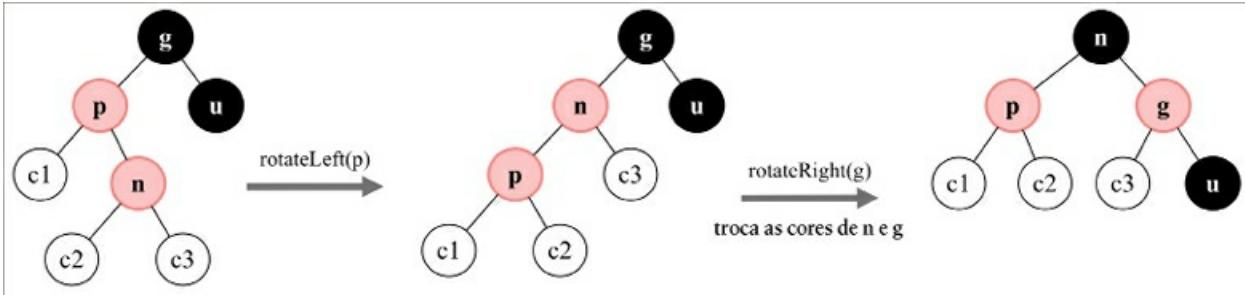


Figura 10.25

Caso o nó seja um filho à esquerda, iremos diretamente para a linha {15} para executar a rotação esquerda-esquerda. O caso 3A pode ser exemplificado pelo diagrama a seguir:

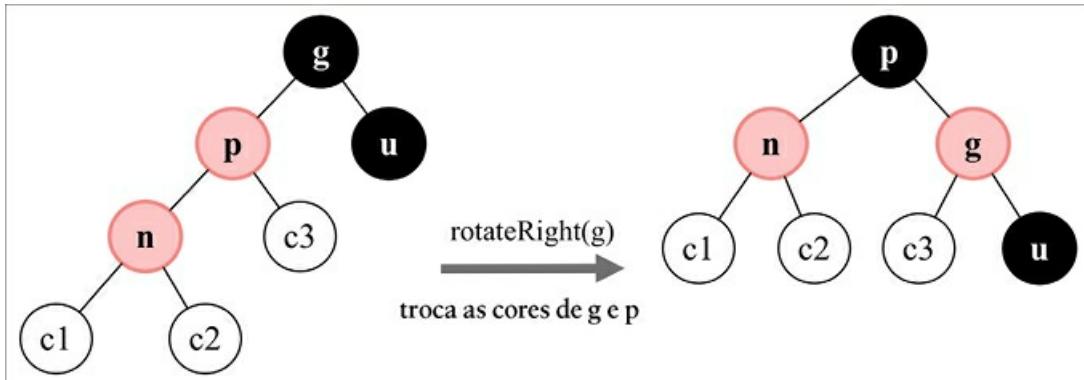


Figura 10.26

Agora vamos analisar os casos 2B e 3B:

```
// caso 2B: o nó é um filho à esquerda - rotação à esquerda
if (node === parent.left) {
    this.rotationLL(parent); // {19}
    node = parent;
    parent = node.parent;
}

// caso 3B: o nó é um filho à direita - rotação à esquerda
this.rotationRR(grandParent); // {20}
parent.color = Colors.BLACK;
grandParent.color = Colors.RED;
node = parent;
```

A lógica é a mesma, mas as diferenças estão nas rotações a serem executadas: primeiro a rotação esquerda-esquerda ({19}) e então a rotação direita-direita ({20}). O caso 2B pode ser exemplificado assim:

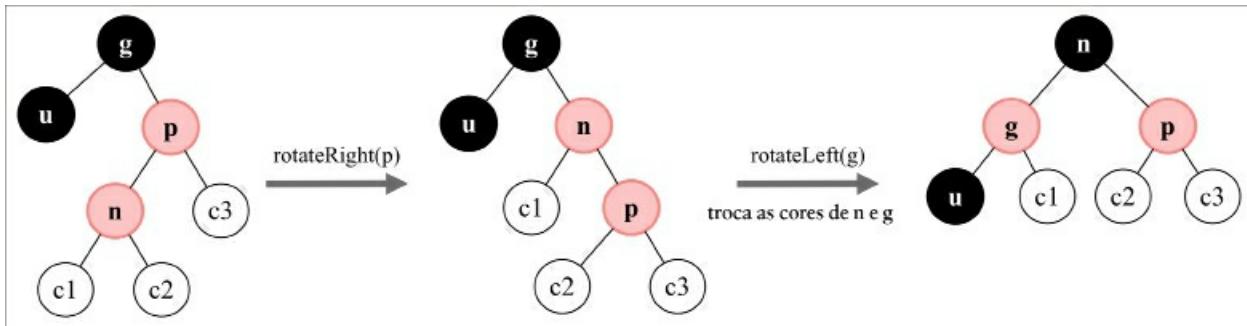


Figura 10.27

Por fim, o caso 3B pode ser exemplificado assim:

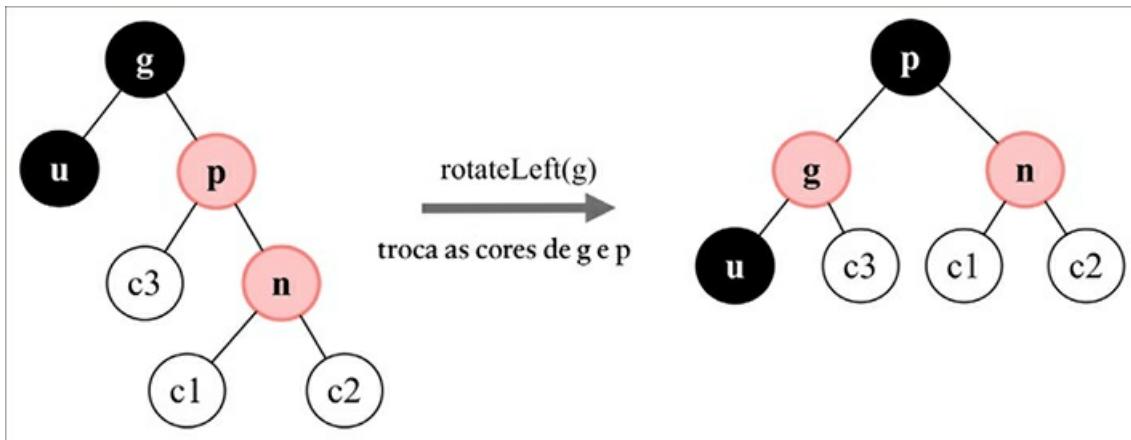


Figura 10.28

Rotações na árvore rubro-negra

No algoritmo de inserção, usamos somente as rotações direita-direita e esquerda-esquerda. A lógica é a mesma que a da árvore AVL; porém, como estamos mantendo uma referência ao pai do nó, temos de atualizar também a referência `node.parent` para o novo pai depois que o nó sofrer rotação.

Eis o código da rotação esquerda-esquerda (rotação à direita) – a atualização do pai está em destaque:

```
rotationLL(node) {
    const tmp = node.left;
    node.left = tmp.right;
    if (tmp.right && tmp.right.key) {
        tmp.right.parent = node;
    }
    tmp.parent = node.parent;
    if (!node.parent) {
```

```

        this.root = tmp;
    }
    else {
        if (node === node.parent.left) {
            node.parent.left = tmp;
        }
        else {
            node.parent.right = tmp;
        }
    }
    tmp.right = node;
    node.parent = tmp;
}

```

Este é o código da rotação direita-direita (rotação à esquerda) – a atualização do pai está em destaque:

```

rotationRR(node) {
    const tmp = node.right;
    node.right = tmp.left;
    if (tmp.left && tmp.left.key) {
        tmp.left.parent = node;
    }
    tmp.parent = node.parent;
    if (!node.parent) {
        this.root = tmp;
    }
    else {
        if (node === node.parent.left) {
            node.parent.left = tmp;
        }
        else {
            node.parent.right = tmp;
        }
    }
    tmp.left = node;
    node.parent = tmp;
}

```

Resumo

Neste capítulo, discutimos os algoritmos para adicionar, pesquisar e remover chaves de uma árvore binária de busca, que é a estrutura de dados básica de árvore, frequentemente usada em ciência da computação. Vimos três abordagens de percurso para visitar todos os nós de uma árvore. Também aprendemos a criar árvores autobalanceadas usando a árvore

AVL e a inserir e remover chaves dela; além disso, descrevemos a árvore rubro-negra.

No próximo capítulo, conheceremos uma estrutura de dados especial chamada heap (ou fila de prioridades).

CAPÍTULO 11

Heap binário e heap sort

No capítulo anterior, conhecemos a estrutura de dados de **árvore**. Neste capítulo, veremos um tipo especial de árvore binária, que é a estrutura de dados de **heap**, também conhecida como **heap binário**. O heap binário é uma estrutura de dados muito famosa em ciência da computação, comumente aplicada em **filas de prioridade** (priority queues) por causa de sua eficiência para extrair rapidamente os valores mínimo e máximo. É usada também pelo famoso algoritmo de **heap sort**.

Neste capítulo, abordaremos o seguinte:

- a estrutura de dados do heap binário;
- heap máximo (max heap) e heap mínimo (min heap);
- o algoritmo de heap sort.

Estrutura de dados do heap binário

O heap binário é uma árvore binária especial com as duas propriedades a seguir:

- É uma árvore binária completa, o que significa que todos os níveis da árvore têm filhos à esquerda e à direita (com exceção dos filhos no último nível), e o último nível tem todos os filhos o máximo possível à esquerda. É o que chamamos de **propriedade de forma** (shape property).
- Um heap binário é um heap mínimo (min heap) ou um heap máximo (max heap). O heap mínimo permite extrair rapidamente o valor mínimo da árvore, enquanto o heap máximo permite extrair rapidamente o valor máximo dela. Todos os nós são maiores ou iguais (heap máximo), ou menores ou iguais (heap mínimo), aos seus nós filhos. É o que chamamos de **propriedade de heap** (heap property).

O diagrama a seguir apresenta alguns exemplos de heaps inválidos e válidos:

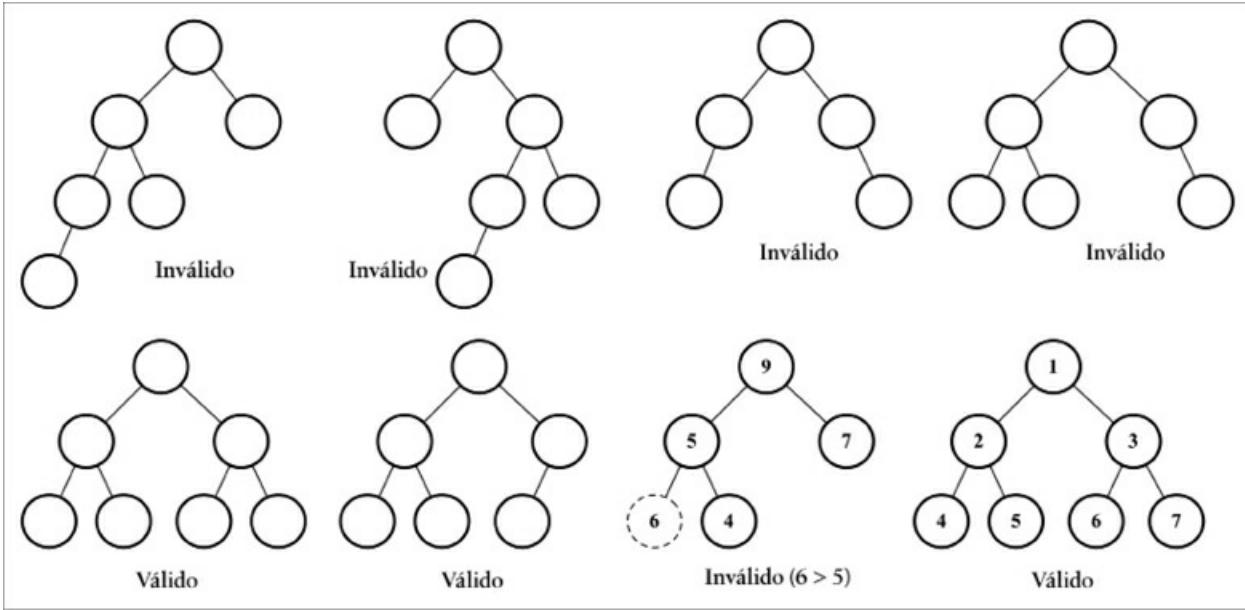


Figura 11.1

Embora o heap binário seja uma árvore binária, ele não é necessariamente uma **BST** (Binary Search Tree, ou Árvore Binária de Busca). No heap binário, todo nó filho deve ser maior ou igual ao seu nó pai (heap mínimo) ou menor ou igual a ele (heap máximo). Em uma BST, porém, o filho à esquerda é sempre menor que o seu pai e o filho à direita tem sempre uma chave maior.

Criando a classe MinHeap

Vamos começar criando a estrutura básica da classe **MinHeap**, assim:

```
import { defaultCompare } from '../util';
export class MinHeap {
  constructor(compareFn = defaultCompare) {
    this.compareFn = compareFn; // {1}
    this.heap = []; // {2}
  }
}
```

Para comparar os valores que serão armazenados nessa estrutura de dados, usaremos **compareFn** ({1}), que fará uma comparação básica caso nenhuma função personalizada seja passada para o construtor da classe, como fizemos em capítulos anteriores.

Para armazenar os valores, usaremos um array para representá-los ({2}).

Representação da árvore binária com um array

Há duas maneiras de representar uma árvore binária. A primeira é por meio de uma representação dinâmica usando ponteiros (representação em nós), como fizemos no capítulo anterior. A segunda é usar um array e acessar os índices corretos a fim de obter os valores do pai e dos filhos à esquerda e à direita. O diagrama seguinte mostra as diferentes representações de uma árvore binária:

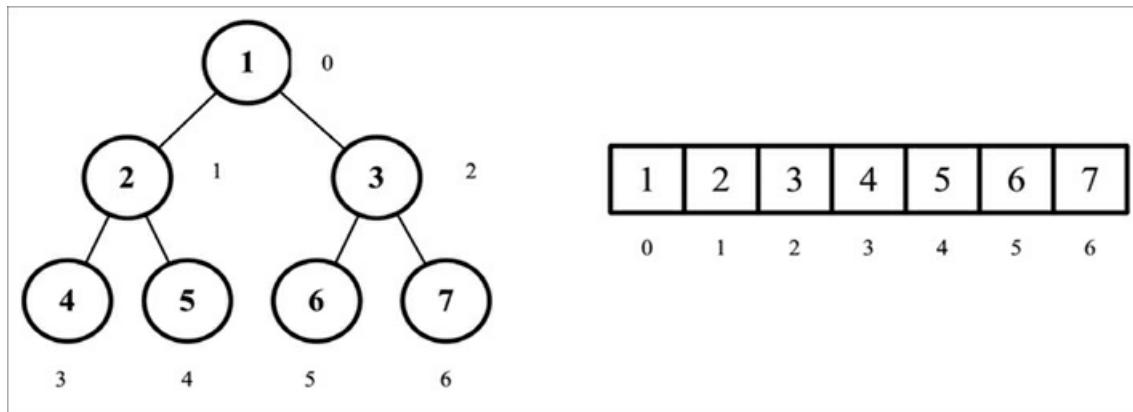


Figura 11.2

Para acessar os *nós* (nodes) de uma árvore binária usando um array comum, podemos manipular o índice com o comportamento a seguir:

Para qualquer dado nó na posição `index`:

- seu filho esquerdo estará localizado em `2 * index + 1` (se estiver disponível);
- seu filho direito estará localizado em `2 * index + 2` (se estiver disponível);
- seu nó pai estará localizado em `index / 2` (se estiver disponível).

Para acessar um nó específico seguindo as regras anteriores, podemos acrescentar os métodos a seguir na classe `MinHeap`:

```
getLeftIndex(index) {  
    return 2 * index + 1;  
}  
getRightIndex(index) {  
    return 2 * index + 2;  
}  
getParentIndex(index) {  
    if (index === 0) {  
        return undefined;  
    } else {  
        return index - 1 / 2;  
    }  
}
```

```

    }
    return Math.floor((index - 1) / 2);
}

```

É possível realizar três operações principais em uma estrutura de dados de heap:

- **insert(value)**: esse método insere um novo **value** no heap. Devolve **true** se **value** for inserido com sucesso, e **false** caso contrário.
- **extract()**: esse método remove o **value** mínimo (heap mínimo) ou máximo (heap máximo) e devolve esse valor.
- **findMinimum()**: esse método devolve o **value** mínimo (heap mínimo) ou máximo (heap máximo) sem removê-lo.

Vamos conhecer cada um desses métodos nas próximas seções.

Inserindo um valor no heap

A inserção de um valor no heap é feita adicionando **value** na folha que está na parte inferior do heap (a última posição do array – **{1}**), e então executando o método **siftUp** (**{2}**), o que significa que trocaremos **value** com o seu pai, até que esse seja menor que o **value** sendo inserido. A operação de **sift up** (literalmente, quer dizer peneirar para cima) também é chamada de **up head**, **percolate up**, **bubble up**, **heapify up** ou **cascade up**.

O código para inserir um novo **value** no heap é apresentado a seguir:

```

insert(value) {
  if (value != null) {
    this.heap.push(value); // {1}
    this.siftUp(this.heap.length - 1); // {2}
    return true;
  }
  return false;
}

```

Operação de sift up

Eis o código para a operação de **sift up**:

```

siftUp(index) {
  let parent = this.getParentIndex(index); // {1}
  while (
    index > 0 &&
    this.compareFn(this.heap[parent], this.heap[index]) > Compare.BIGGER_THAN
  )
    this.swap(index, parent);
    index = parent;
}

```

```

) { // {2}
swap(this.heap, parent, index); // {3}
index = parent;
parent = this.getParentIndex(index); // {4}
}
}

```

O método **siftUp** recebe o **index** do **value** inserido. Também precisamos obter o **index** de seu **parent** ({1}).

Se o **value** inserido for menor que o seu **parent** ({2}) — no caso do heap mínimo, ou maior que o seu **parent** no caso do heap máximo), trocamos o **element** com o seu **parent** ({3}). Repetiremos esse processo até que a raiz do heap também tenha sido processada, atualizando o **index** e o **parent** ({4}) depois de cada troca (swap).

A seguir, apresentamos a função **swap**:

```

function swap(array, a, b) {
  const temp = array[a]; // {5}
  array[a] = array[b]; // {6}
  array[b] = temp; // {7}
}

```

Para trocar dois valores em um array, precisamos de uma variável auxiliar que conterá uma cópia do primeiro elemento que queremos trocar ({5}). Em seguida, atribuímos o segundo elemento à posição do primeiro elemento ({6}). Por fim, sobrescrevemos o valor do segundo elemento com o valor do primeiro elemento ({7}) atribuindo-lhe a cópia feita na linha {5}.

A função **swap** será usada com frequência no Capítulo 13, *Algoritmos de ordenação e de busca*.

Também podemos reescrever a função **swap** usando a sintaxe da ECMAScript 2015 (ES6):

```
const swap = (array, a, b) => [array[a], array[b]] = [array[b], array[a]];
```

A ES2015 introduziu a funcionalidade de desestruturação de objetos e de array **[a, b] = [b, a]**, conforme vimos no Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*. No entanto, há atualmente (quando este livro foi escrito) uma questão aberta no que concerne ao fato de o desempenho da operação de desestruturação ser mais lento que uma atribuição usual. Para obter mais informações sobre o problema,

acesse https://bugzilla.mozilla.org/show_bug.cgi?id=1177319.

Vamos ver o método `insert` em ação. Considere a estrutura de dados de heap a seguir (Figura 11.3):

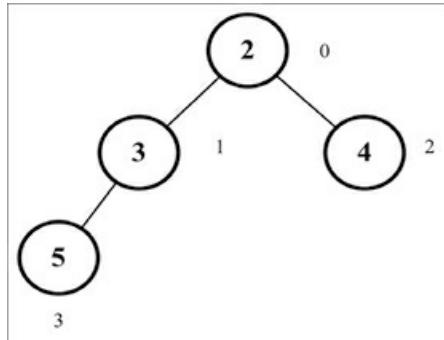


Figura 11.3

Suponha que queremos inserir o valor 1 em `heap`. O algoritmo fará algumas operações de sift up, conforme mostrado no diagrama a seguir:

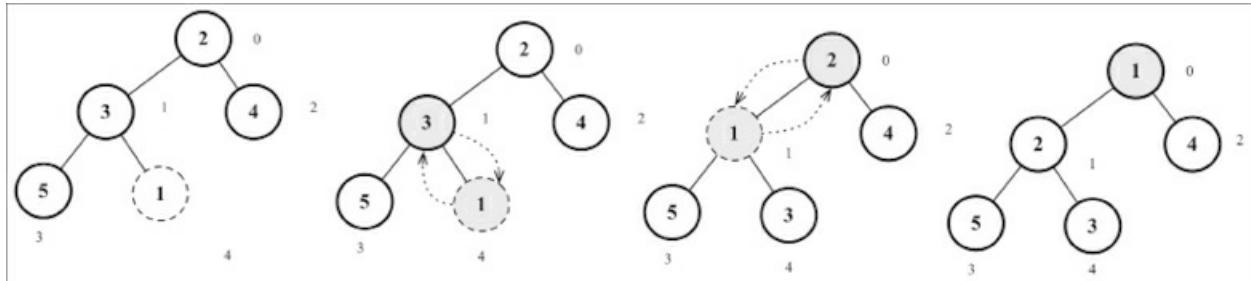


Figura 11.4

O código seguinte exemplifica a criação do `heap` e as ações exibidas no diagrama:

```
const heap = new MinHeap();
heap.insert(2);
heap.insert(3);
heap.insert(4);
heap.insert(5);
heap.insert(1);
```

Encontrando os valores mínimo e máximo no heap

No heap mínimo, o valor (`value`) mínimo estará sempre localizado no primeiro índice do array (a raiz do heap). Esse código é mostrado a seguir:

```
size() {
    return this.heap.length;
}
```

```

isEmpty() {
    return this.size() === 0;
}
findMinimum() {
    return this.isEmpty() ? undefined : this.heap[0]; // {1}
}

```

Assim, se o heap não estiver vazio, devolveremos o primeiro índice do array ({1}). Também podemos criar os métodos `size` e `empty` para a estrutura de dados `MinHeap`.

O código a seguir pode ser usado para testar esses três métodos:

```

console.log('Heap size: ', heap.size()); // 5
console.log('Heap is empty: ', heap.isEmpty()); // false
console.log('Heap min value: ', heap.findMinimum()); // 1

```

No heap máximo, o primeiro índice do array armazenará o valor (`value`) máximo, portanto podemos usar esse mesmo código.

Extraindo os valores mínimo e máximo do heap

Remover o valor (`value`) mínimo (heap mínimo) ou máximo (heap máximo) consiste em remover o elemento localizado no primeiro `index` do array (a raiz do heap). Após a remoção, movemos o último elemento do heap para a raiz e então executamos a função chamada `siftDown`, o que significa que faremos uma troca de elementos até que o heap esteja organizado novamente. A operação de `sift down` também é conhecida como `sink down`, `percolate down`, `bubble down`, `heapify down` ou `cascade down`.

Esse código se apresenta da seguinte maneira:

```

extract() {
    if (this.isEmpty()) {
        return undefined; // {1}
    }
    if (this.size() === 1) {
        return this.heap.shift(); // {2}
    }
    const removedValue = this.heap.shift(); // {3}
    this.siftDown(0); // {4}
    return removedValue; // {5}
}

```

Se o heap estiver vazio, não haverá valor para extrair, portanto podemos

devolver `undefined` ({1}). Se houver apenas um valor no heap, podemos simplesmente o remover e devolvê-lo ({2}). Contudo, se o heap tiver mais que um valor, removeremos o valor do primeiro índice ({3}) e o armazenaremos em uma variável temporária para que ele seja devolvido ({5}) após a execução da operação de sift down ({4}).

Operação de sift down

Eis o código da operação de sift down (heapify, ou “heapificação”):

```
siftDown(index) {
  let element = index;
  const left = this.getLeftIndex(index); // {1}
  const right = this.getRightIndex(index); // {2}
  const size = this.size();
  if (
    left < size &&
    this.compareFn(this.heap[element], this.heap[left]) > Compare.BIGGER_THAN
  ) { // {3}
    element = left; // {4}
  }
  if (
    right < size &&
    this.compareFn(this.heap[element], this.heap[right]) > Compare.BIGGER_THAN
  ) { // {5}
    element = right; // {6}
  }
  if (index !== element) { // {7}
    swap(this.heap, index, element); // {8}
    this.siftDown(element); // {9}
  }
}
```

O método `siftDown` recebe o `index` do valor removido. Fazemos uma cópia do `index` recebido na variável `element`. Também obtemos os índices dos filhos `left` ({1}) e `right` ({2}).

A operação de sift down consiste em trocar o `element` com o seu filho menor (heap mínimo) ou maior (heap máximo). Se `element` for menor que o seu filho `left` ({3} – e `index` também for válido), trocaremos `element` com seu filho `left` ({4}). Se `element` for menor que o seu filho `right` ({5} – e `index` também for válido), trocaremos `element` com seu filho `right` ({6}).

Depois de encontrar o `index` do filho menor, verificamos se o seu valor é igual ao índice em `element` (passado para o método `siftDown` – {7}); não faz sentido trocar o valor por ele mesmo! Se não for igual, nós o trocaremos com o `element` menor ({8}) e repetiremos o mesmo processo, começando pelo `element` menor ({9}), até que ele seja colocado em sua posição correta.

Suponha que queremos fazer uma extração no heap. O algoritmo fará algumas operações de sift down, conforme mostrado no diagrama a seguir:

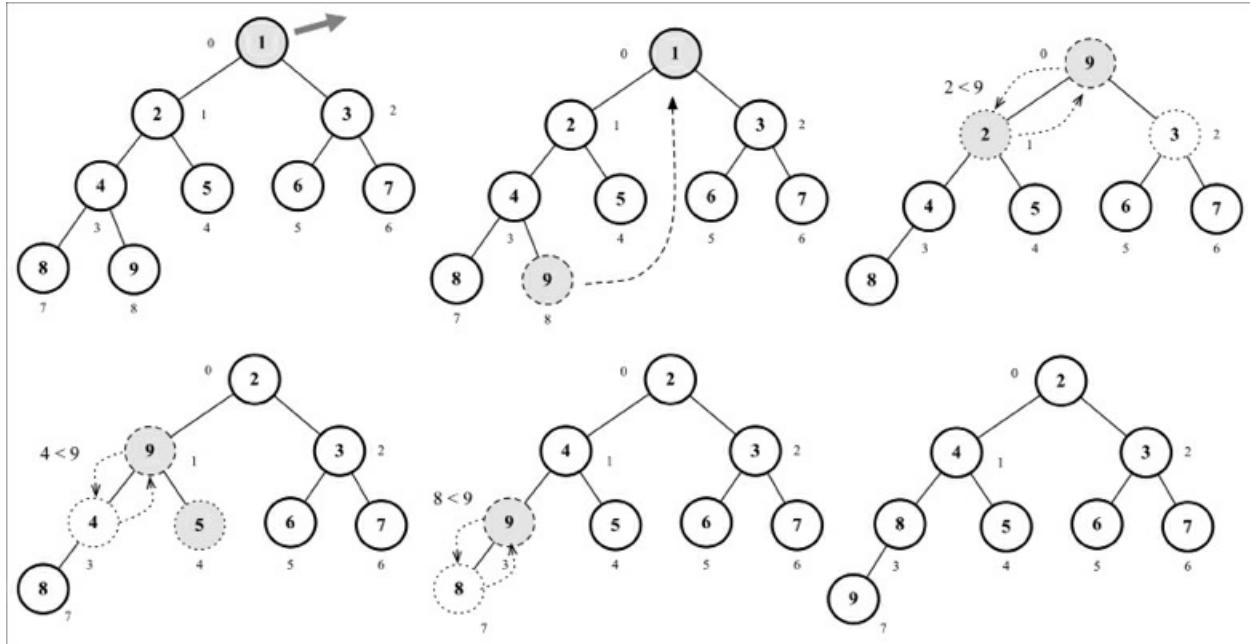


Figura 11.5

O código seguinte pode ser usado para testar as ações representadas no diagrama:

```
heap = new MinHeap();
for (let i = 1; i < 10; i++) {
  heap.insert(i);
}
console.log('Extract minimum: ', heap.extract()); // 1
```

Criando a classe MaxHeap

O algoritmo da classe `MaxHeap` será praticamente o mesmo da classe `MinHeap`. A diferença estará no fato de que, sempre que houver uma comparação de `>` (maior que), ela será trocada pela comparação de `<` (menor que).

Eis o código da classe `MaxHeap`:

```
export class MaxHeap extends MinHeap {  
  constructor(compareFn = defaultCompare) {  
    super(compareFn);  
    this.compareFn = reverseCompare(compareFn); // {1}  
  }  
}
```

Contudo, em vez de copiar o código e duplicá-lo, podemos estender a classe `MinHeap` para herdar todo o código que criamos neste capítulo e fazer uma comparação inversa sempre que necessário. Para inverter a comparação, em vez de comparar `a` com `b`, podemos comparar `b` com `a` (`{1}`), conforme vemos no código a seguir:

```
function reverseCompare(compareFn) {  
  return (a, b) => compareFn(b, a);  
}
```

Podemos usar o mesmo código utilizado com `MinHeap` para testar `MaxHeap`. A diferença é que o valor maior estará na raiz do heap, e não o valor menor.

```
const maxHeap = new MaxHeap();  
maxHeap.insert(2);  
maxHeap.insert(3);  
maxHeap.insert(4);  
maxHeap.insert(5);  
maxHeap.insert(1);  
console.log('Heap size: ', maxHeap.size()); // 5  
console.log('Heap min value: ', maxHeap.findMinimum()); // 5
```

Algoritmo de heap sort

Podemos usar a estrutura de dados do heap binário para nos ajudar a criar um algoritmo de ordenação muito famoso: o heap sort. O algoritmo de heap sort é composto de três passos:

1. Crie um heap máximo usando o array a ser ordenado como o array original.
2. Depois de criar o heap máximo, o maior valor estará armazenado no primeiro índice do heap. Substituiremos o primeiro valor pelo último do heap, decrementando o tamanho do heap de 1.
3. Por fim, executamos `heapify` (sift down) na raiz do heap e repetimos o passo 2 até que o tamanho do heap seja igual a 1.

Usamos o resultado do heap máximo em um array em ordem crescente (do menor para o maior). Se quisermos que o array seja ordenado em ordem decrescente, podemos usar o heap mínimo em seu lugar.

A seguir, apresentamos o código do algoritmo de heap sort:

```
function heapSort(array, compareFn = defaultCompare) {  
    let heapSize = array.length;  
    buildMaxHeap(array, compareFn); // step 1  
    while (heapSize > 1) {  
        swap(array, 0, --heapSize); // step 2  
        heapify(array, 0, heapSize, compareFn); // step 3  
    }  
    return array;  
}
```

Para implementar o heap máximo, podemos usar a função a seguir:

```
function buildMaxHeap(array, compareFn) {  
    for (let i = Math.floor(array.length / 2); i >= 0; i -= 1) {  
        heapify(array, i, array.length, compareFn);  
    }  
    return array;  
}
```

A função de heap máximo reorganizará o array. Por causa de todas as comparações que serão feitas, só precisaremos executar a função **heapify** (sift down) para a segunda metade das posições do array (a primeira metade se organizará automaticamente, portanto não será necessário executar a função nas posições que já sabemos que estarão ordenadas).

A função **heapify** tem o mesmo código que o método **siftDown** criado antes neste capítulo. A diferença é que passamos também como parâmetros o próprio heap, o seu tamanho e a função de comparação que queremos usar. Isso se deve ao fato de não estarmos utilizando diretamente a estrutura de dados de heap, mas usamos a sua lógica para desenvolver o algoritmo **heapSort**.

O diagrama a seguir exemplifica o algoritmo de heap sort:

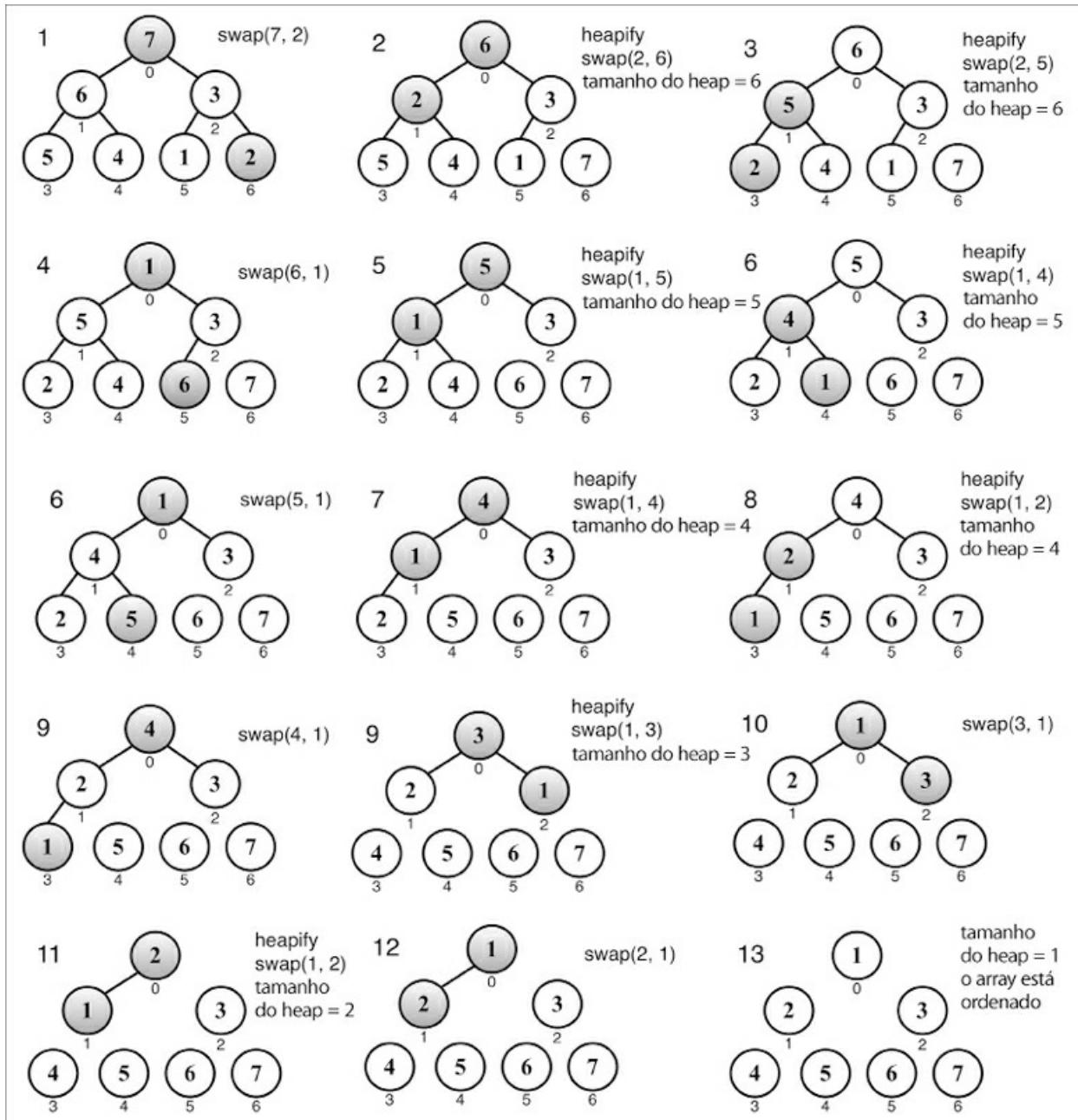


Figura 11.6

O seguinte código pode ser usado para testar a função `heapSort`:

```
const array = [7, 6, 3, 5, 4, 1, 2];
console.log('Before sorting: ', array);
console.log('After sorting: ', heapSort(array));
```

O algoritmo de heap sort não é um algoritmo de ordenação estável, isto é, se o array já estiver ordenado, é possível que os valores possam acabar em uma ordem diferente. Exploraremos melhor os algoritmos

de ordenação no Capítulo 13, *Algoritmos de ordenação e de busca*.

Resumo

Neste capítulo, conhecemos a estrutura de dados de heap binário e suas duas variantes: o heap mínimo (min heap) e o heap máximo (max heap). Vimos como inserir valores, como dar uma espiada ou encontrar os valores mínimo e máximo, e como extrair um valor do heap. Também descrevemos as operações de sift up e de sift down, que ajudam a manter o heap organizado.

Também aprendemos a usar a estrutura de dados de heap para criar o algoritmo de heap sort.

No próximo capítulo, estudaremos os conceitos básicos de grafos, que são estruturas de dados não lineares.

CAPÍTULO 12

Grafos

Neste capítulo, conhiceremos outra estrutura de dados não linear chamada grafo (graph). Será a última estrutura de dados que discutiremos antes de mergulhar de cabeça nos algoritmos de ordenação e de pesquisa.

Este capítulo incluirá uma parte considerável das maravilhosas aplicações dos grafos. Como esse é um assunto vasto, poderíamos escrever um livro como este simplesmente para explorar o incrível mundo dos grafos.

Neste capítulo, abordaremos o seguinte:

- a terminologia dos grafos;
- a representação de um grafo de três formas diferentes;
- a estrutura de dados dos grafos;
- algoritmos de busca em grafos;
- algoritmos de caminho mais curto;
- algoritmos de árvore de extensão mínima (minimum spanning tree).

Terminologia dos grafos

Um **grafo** é um modelo abstrato de uma estrutura de rede. Um grafo é um conjunto de **nós** (ou **vértices**) conectados por **arestas** (edges). Conhecer os grafos é importante porque qualquer relacionamento binário pode ser representado por um grafo.

Qualquer rede social, como Facebook, Twitter e Google+, pode ser representada por um grafo.

Também podemos usar grafos para representar estradas, voos e comunicações, como mostra a imagem a seguir.

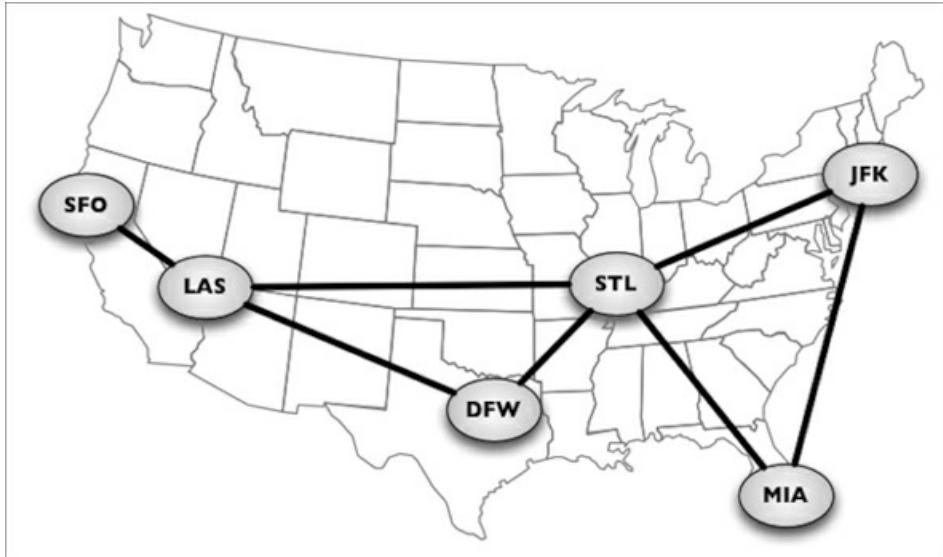


Figura 12.1

Vamos conhecer melhor os conceitos matemáticos e técnicos associados aos grafos.

Um grafo $G = (V, E)$ é composto de:

- V : um conjunto de vértices;
- E : um conjunto de arestas (edges) que conectam os vértices em V .

O diagrama a seguir representa um grafo:

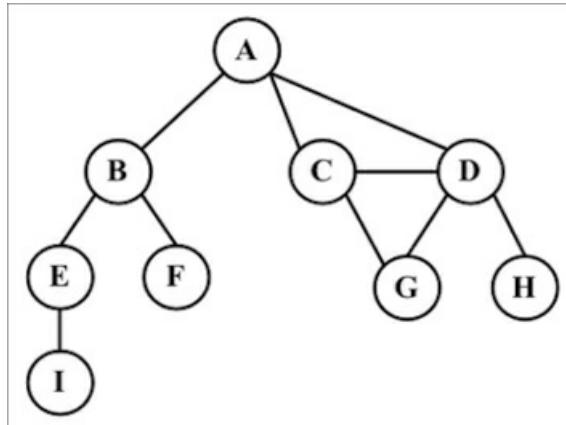


Figura 12.2

Apresentaremos a terminologia associada aos grafos antes de começar a implementar qualquer algoritmo.

Os vértices conectados por uma aresta são chamados de **vértices adjacentes**. Por exemplo, A e B são adjacentes, A e D são adjacentes, A e C são adjacentes, mas A e E não são adjacentes.

O **grau** de um vértice corresponde ao número de vértices adjacentes. Por exemplo, A está conectado a três vértices. Desse modo, A tem grau 3. E está conectado a dois vértices. Portanto E tem grau 2.

Um **caminho** (path) é uma sequência de vértices consecutivos, por exemplo, v_1, v_2, \dots, v_k , em que v_i e v_{i+1} são adjacentes. Usando o grafo do diagrama anterior como exemplo, temos os caminhos A B E I e A C D G, entre outros.

Um caminho simples não contém vértices repetidos. Como exemplo, temos o caminho A D G. Um **ciclo** é um caminho simples, exceto pelo último vértice, que é igual ao primeiro: A D C A (de volta para A).

Um grafo é **acíclico** se não tiver ciclos. Um grafo será **conectado** (ou conexo) se houver um caminho entre todos os pares de vértices.

Grafos direcionados e não direcionados

Os grafos podem ser **não direcionados** (as arestas não têm uma direção) ou **direcionados (digrafo)**, em que as arestas têm uma direção, como vemos a seguir.

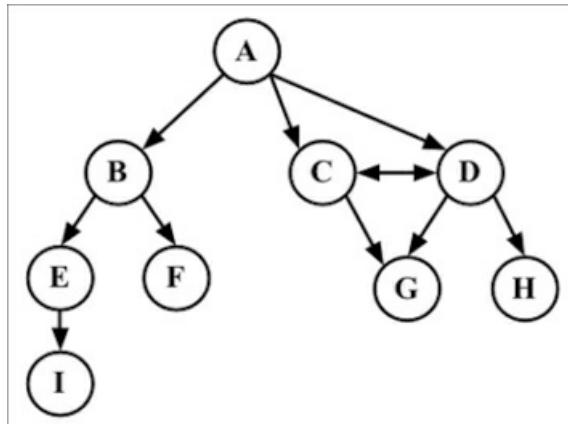


Figura 12.3

Um grafo será **fortemente conectado** se houver um caminho em ambas as direções entre todos os pares de vértices. Por exemplo, C e D são fortemente conectados, enquanto A e B não são fortemente conectados.

Os grafos também podem ser **sem peso** (como vimos até agora) ou **com peso** (nos quais as arestas têm pesos – ou valores), como mostra o diagrama a seguir.

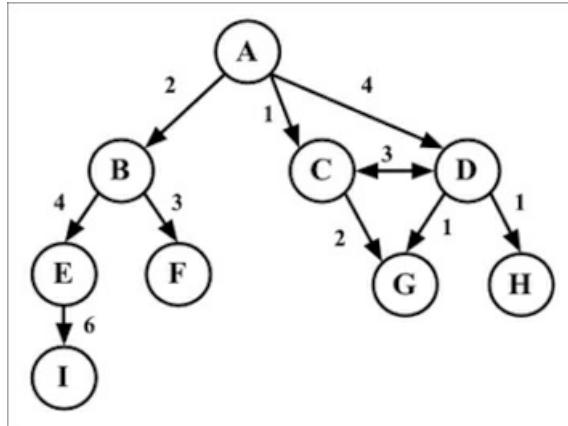


Figura 12.4

Podemos resolver muitos problemas no mundo da ciência da computação usando grafos, por exemplo, pesquisar um grafo em busca de um vértice ou de uma aresta específicos, encontrar um caminho no grafo (de um vértice a outro), descobrir o caminho mais curto entre dois vértices e detectar ciclos.

Representando um grafo

Há algumas maneiras com as quais podemos representar os grafos quando se trata de estruturas de dados. Não há uma única forma correta de representar um grafo entre as possibilidades existentes. Isso dependerá do tipo de problema que você terá de resolver e também do tipo de grafo.

A matriz de adjacências

A implementação mais comum usa uma **matriz de adjacências**. Cada nó é associado a um inteiro, que é o índice do array. Representaremos a conectividade entre os vértices usando um array bidimensional, como `array[i][j] == 1` se houver uma aresta do nó de índice *i* para o nó de índice *j*, ou como `array[i][j] == 0` caso contrário, conforme mostra o diagrama a seguir (Figura 12.5).

Os grafos que não são fortemente conectados (**grafos esparsos**) serão representados por uma matriz com muitas entradas iguais a zero na matriz de adjacências. Isso significa que desperdiçaríamos espaço na memória do computador para representar arestas inexistentes. Por exemplo, se precisássemos encontrar os vértices adjacentes de um dado vértice, teríamos de iterar por toda a linha, mesmo que esse vértice tivesse apenas

um vértice adjacente. Outro motivo pelo qual essa talvez não seja uma boa representação é que o número de vértices no grafo pode mudar, e um array bidimensional não é flexível.

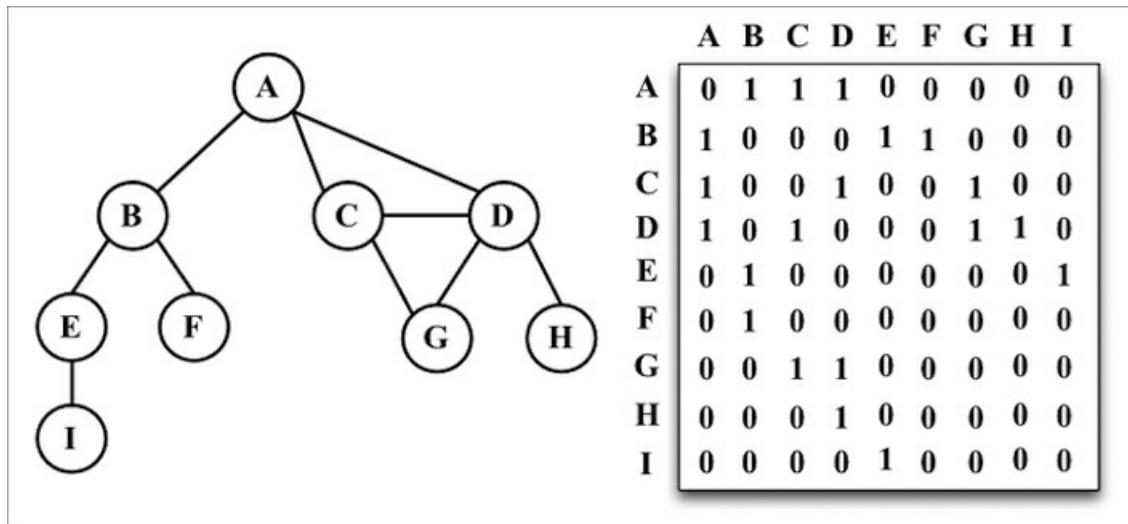


Figura 12.5

Lista de adjacências

Podemos usar também uma estrutura de dados dinâmica, chamada **lista de adjacências**, para representar os grafos. Essa estrutura é composta de uma lista de vértices adjacentes para cada vértice do grafo. Há algumas maneiras diferentes com as quais podemos representar essa estrutura de dados. Para representar a lista de vértices adjacentes, podemos usar uma lista (array), uma lista ligada ou até mesmo um mapa hash ou um dicionário. O diagrama a seguir exemplifica a estrutura de dados para uma lista de adjacências.

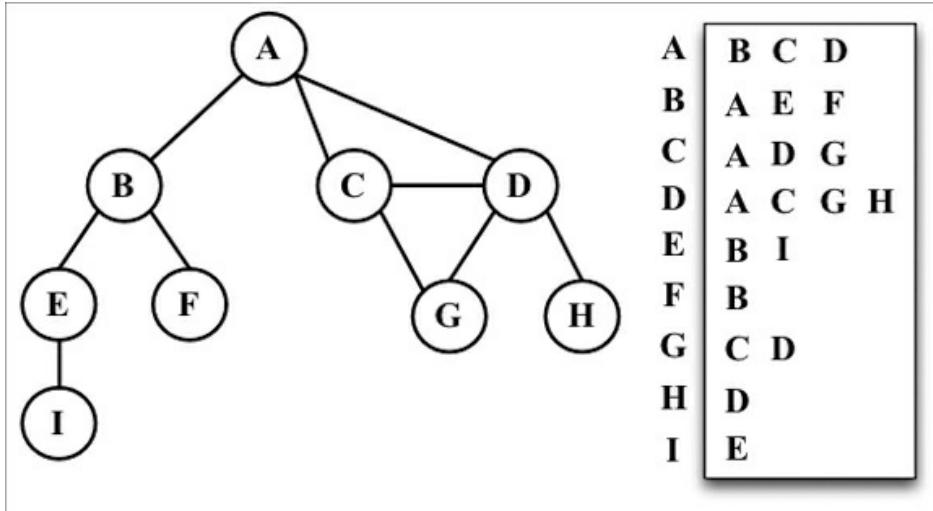


Figura 12.6

As duas representações são muito úteis e têm propriedades distintas (por exemplo, descobrir se os vértices v e w são adjacentes é mais rápido se usarmos uma matriz de adjacências), embora as listas de adjacências provavelmente sejam melhores para a maioria dos problemas. Usaremos a representação da lista de adjacências nos exemplos deste livro.

Matriz de incidências

Também podemos representar um grafo usando uma **matriz de incidências**. Em uma matriz de incidências, cada linha da matriz representa um vértice e cada coluna representa uma aresta. Representaremos a conectividade entre dois objetos usando um array bidimensional, como `array[v][e] == 1` se o vértice v for incidente sobre a aresta e , ou como `array[v][e] == 0` caso contrário, conforme mostra o diagrama a seguir.

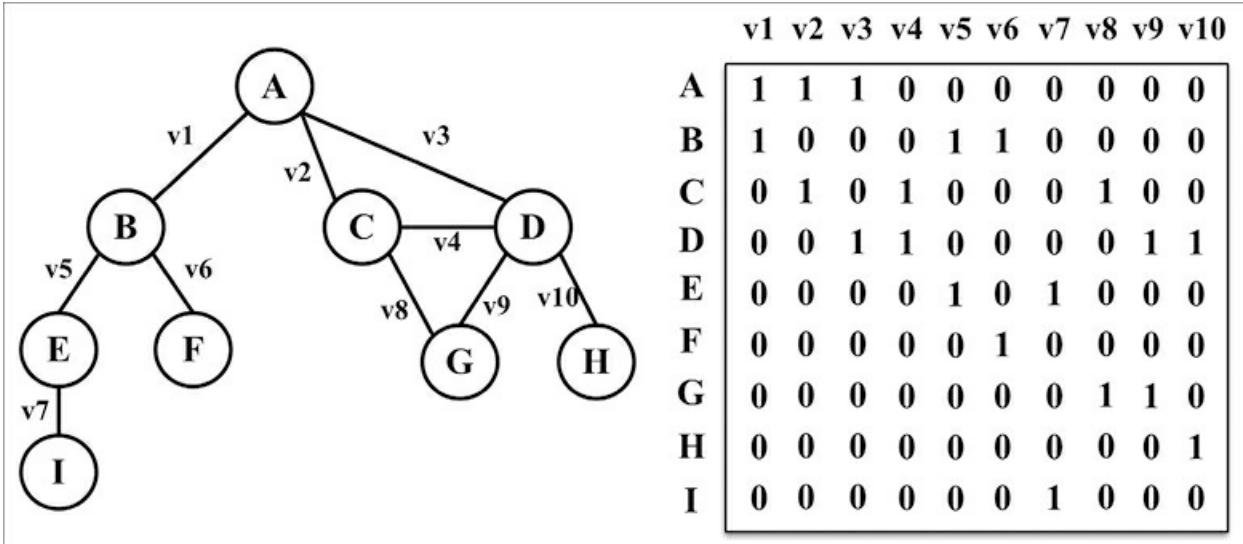


Figura 12.7

Uma matriz de incidências geralmente é usada para economizar espaço e memória quando temos mais arestas do que vértices.

Criando a classe Graph

Como sempre, vamos declarar a estrutura básica de nossa classe:

```
class Graph {
    constructor(isDirected = false) {
        this.isDirected = isDirected; // {1}
        this.vertices = []; // {2}
        this.adjList = new Dictionary(); // {3}
    }
}
```

O construtor de **Graph** pode receber um parâmetro para informar se o grafo é direcionado ou não ({1}) e, por padrão, ele não será direcionado. Usaremos um array para armazenar os nomes de todos os vértices do grafo ({2}), e um dicionário (implementado no Capítulo 8, *Dicionários e hashes*) para armazenar a lista de adjacências ({3}). O dicionário utilizará o nome do vértice como chave e a lista de vértices adjacentes como valor.

Em seguida, implementaremos dois métodos: um para adicionar um novo vértice no grafo (porque, quando instanciamos o grafo, um grafo vazio, sem vértices, será criado) e outro para adicionar arestas entre os vértices. Vamos implementar o método **addVertex** antes, assim:

```
addVertex(v) {
    if (!this.vertices.includes(v)) { // {5}
```

```

    this.vertices.push(v); // {6}
    this.adjList.set(v, []); // {7}
}
}

```

Esse método recebe um vértice `v` como parâmetro. Adicionaremos esse vértice à lista de vértices (`{6}`) somente se o vértice não estiver presente ainda no grafo, além de inicializarmos a lista de adjacências com um array vazio, definindo o valor da chave correspondente ao vértice `v` no dicionário com um array vazio (`{7}`).

Vamos agora implementar o método `addEdge` com o código a seguir.

```

addEdge(v, w) {
    if (!this.adjList.get(v)) {
        this.addVertex(v); // {8}
    }
    if (!this.adjList.get(w)) {
        this.addVertex(w); // {9}
    }
    this.adjList.get(v).push(w); // {10}
    if (!this.isDirected) {
        this.adjList.get(w).push(v); // {11}
    }
}

```

Esse método recebe dois vértices como parâmetros, que são os vértices que queremos ligar no grafo. Antes de ligar os vértices, verificamos se eles estão presentes no grafo. Se os vértices `v` ou `w` não existirem no grafo, eles serão adicionados à lista de vértices (`{8}` e `{9}`).

Então adicionamos uma aresta do vértice `v` para o vértice `w` (`{10}`) acrescentando `w` à lista de adjacências de `v`. Se quiséssemos implementar um grafo direcionado, a linha `{10}` seria suficiente. Como estamos trabalhando com grafos não direcionados na maior parte dos exemplos deste capítulo, devemos adicionar também uma aresta de `w` para `v` (`{11}`).

Observe que estamos apenas adicionando novos elementos no array, pois já o inicializamos na linha `{7}`.

Para concluir a criação de nossa classe `Graph`, declararemos também dois métodos `getter`: um que devolve a lista de vértices e outro que devolve a lista de adjacentes:

```

getVertices() {
    return this.vertices;
}

```

```

}
getAdjList() {
  return this.adjList;
}

```

Vamos testar esse código assim:

```

const graph = new Graph();
const myVertices = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']; // {12}
for (let i = 0; i < myVertices.length; i++) { // {13}
  graph.addVertex(myVertices[i]);
}
graph.addEdge('A', 'B'); // {14}
graph.addEdge('A', 'C');
graph.addEdge('A', 'D');
graph.addEdge('C', 'D');
graph.addEdge('C', 'G');
graph.addEdge('D', 'G');
graph.addEdge('D', 'H');
graph.addEdge('B', 'E');
graph.addEdge('B', 'F');
graph.addEdge('E', 'I');

```

Para facilitar nossa vida, vamos criar um array com todos os vértices que queremos adicionar em nosso grafo ({12}). Então só precisamos iterar pelo array **myVertices** e adicionar os valores, um a um, em nosso grafo ({13}). Por fim, adicionamos as arestas desejadas ({14}). Esse código criará o grafo que usamos nos diagramas apresentados até agora neste capítulo.

Para facilitar mais ainda nossas vidas, implementaremos também o método **toString** para a classe **Graph** a fim de que possamos exibir o grafo no console:

```

toString() {
  let s = '';
  for (let i = 0; i < this.vertices.length; i++) { // {15}
    s += `${this.vertices[i]} -> `;
    const neighbors = this.adjList.get(this.vertices[i]); // {16}
    for (let j = 0; j < neighbors.length; j++) { // {17}
      s += `${neighbors[j]} `;
    }
    s += '\n'; // {18}
  }
  return s;
}

```

Construiremos uma string com a representação da lista de adjacências. Inicialmente iteramos pela lista do array **vertices** ({15}) e acrescentamos

o nome do vértice em nossa string **s**. Em seguida, obtemos a lista de adjacências desse vértice (**{16}**) e iteramos por ela também (**{17}**) a fim de obter o nome do vértice adjacente e adicioná-lo em nossa string. Depois de iterar pela lista de adjacências, adicionamos uma quebra de linha em nossa string (**{18}**) para que possamos ver uma saída elegante no console. Vamos testar esse código:

```
console.log(graph.toString());
```

Eis a saída:

```
A -> B C D  
B -> A E F  
C -> A D G  
D -> A C G H  
E -> B I  
F -> B  
G -> C D  
H -> D  
I -> E
```

Uma bela lista de adjacências! A partir dessa saída, sabemos que o vértice **A** tem os seguintes vértices adjacentes: **B**, **C** e **D**.

Percorrendo grafos

De modo semelhante à estrutura de dados de árvores, podemos igualmente visitar todos os nós de um grafo. Há dois algoritmos que podem ser usados para percorrer um grafo, chamados **BFS** (Breadth-First Search, ou Busca em Largura) e **DFS** (Depth-First Search, ou Busca em Profundidade). Podemos percorrer um grafo para encontrar um vértice específico ou um caminho entre dois vértices, verificar se o grafo é conectado ou se contém ciclos, e assim por diante.

Antes de implementar os algoritmos, vamos tentar compreender melhor a ideia de percorrer um grafo.

A ideia dos **algoritmos para percorrer grafos** é que devemos registrar cada vértice quando o visitamos inicialmente e controlar quais vértices ainda não foram totalmente explorados. Nos dois algoritmos para percorrer grafos, devemos especificar qual será o primeiro vértice a ser visitado.

Para explorar totalmente um vértice, devemos ver todas as arestas desse vértice. Para cada aresta conectada a um vértice que não tenha sido visitado ainda, nós marcaremos esse vértice como descoberto e o

adicionaremos à lista de vértices a ser visitada.

Para ter algoritmos eficientes, devemos visitar cada vértice no máximo duas vezes, quando cada uma de suas extremidades for explorada. Todas as arestas e vértices no grafo conectado serão visitados.

Os algoritmos BFS e DFS são muito parecidos, mas têm uma diferença importante, que é a estrutura de dados usada para armazenar a lista de vértices a serem visitados. Observe a tabela a seguir.

| Algoritmo | Estrutura de dados | Descrição |
|-----------|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DFS | Pilha | Ao armazenar os vértices em uma pilha (nós vimos no Capítulo 4, <i>Pilhas</i>), os vértices serão explorados ao longo de um caminho, visitando um novo vértice adjacente se houver um disponível. |
| BFS | Fila | Ao armazenar os vértices em uma fila (nós vimos no Capítulo 5, <i>Filas e deque</i> s), os vértices não explorados mais antigos serão explorados antes. |

Para marcar os vértices que já visitamos, usaremos três cores para indicar o seu status:

- **branco** (white): indica que o vértice ainda não foi visitado;
- **cinza** (grey): indica que o vértice foi visitado, mas não foi explorado;
- **preto** (black): indica que o vértice foi totalmente explorado.

É por isso que devemos visitar cada vértice no máximo duas vezes, conforme mencionamos antes.

Para nos ajudar a marcar os vértices nos algoritmos BFS e DFS, usaremos a variável **Colors** (que funcionará como um enumerado), declarada assim:

```
const Colors = {
  WHITE: 0,
  GREY: 1,
  BLACK: 2
};
```

Os dois algoritmos precisarão também de um objeto auxiliar para ajudar a armazenar a informação sobre o vértice ter sido visitado ou não. No início de cada algoritmo, todos os vértices serão marcados como não visitados (cor branca). Usaremos a função a seguir para inicializar a cor dos vértices:

```
const initializeColor = vertices => {
  const color = {};
  for (let i = 0; i < vertices.length; i++) {
    color[vertices[i]] = Colors.WHITE;
  }
  return color;
};
```

Observe que estamos usando a sintaxe da ES2015 para declarar uma função usando um `const` e *funções de seta* que vimos no Capítulo 2, *Visão geral sobre ECMAScript e TypeScript*. Poderíamos também ter declarado a função `initializeColor` usando a sintaxe de função como `function initializeColor(vertices) {}`.

Busca em largura (BFS)

O algoritmo BFS (Breadth-First Search, ou Busca em Largura) começa percorrendo o grafo a partir do primeiro vértice especificado e visita todos os seus vizinhos (vértices adjacentes) antes, uma camada do grafo a cada vez. Em outras palavras, ele visita os vértices na largura antes, e depois em profundidade, como mostra o diagrama a seguir:

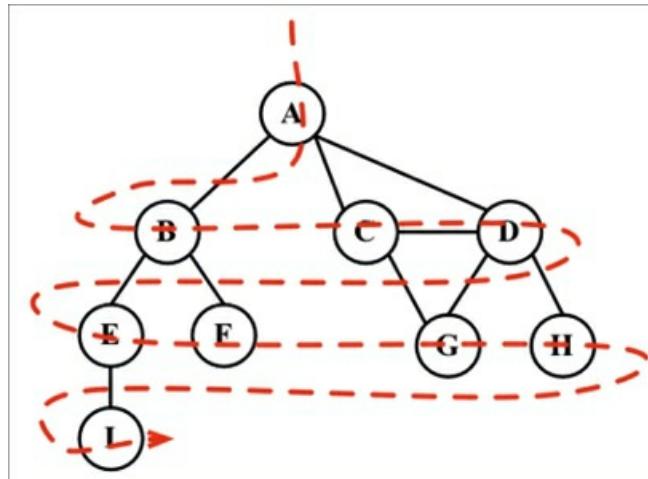


Figura 12.8

Eis os passos seguidos pelo algoritmo BFS, começando no vértice v :

1. Cria uma fila Q .
2. Marca v como descoberto (cinza) e insere v em Q com `enqueue`.
3. Enquanto Q não está vazia, executa os passos a seguir.
 1. Remove u de Q com `dequeue`.
 2. Marca u como descoberto (cinza).
 3. Insere todos os vizinhos não visitados (branco) w de u na fila com `enqueue`.
 4. Marca u como explorado (preto).

O algoritmo de BFS é declarado assim:

```
export const breadthFirstSearch = (graph, startVertex, callback) => {
  const vertices = graph.getVertices();
  const adjList = graph.getAdjList();
  const color = initializeColor(vertices); // {1}
  const queue = new Queue(); // {2}
  queue.enqueue(startVertex); // {3}
  while (!queue.isEmpty()) { // {4}
    const u = queue.dequeue(); // {5}
    const neighbors = adjList.get(u); // {6}
    color[u] = Colors.GREY; // {7}
    for (let i = 0; i < neighbors.length; i++) { // {8}
      const w = neighbors[i]; // {9}
      if (color[w] === Colors.WHITE) { // {10}
        color[w] = Colors.GREY; // {11}
        queue.enqueue(w); // {12}
      }
    }
    color[u] = Colors.BLACK; // {13}
    if (callback) { // {14}
      callback(u);
    }
  }
};
```

Vamos explorar a implementação do método BFS. Nossa primeira tarefa será usar a função `initializeColor` para inicializar o array `color` com branco ({1}). Também precisamos declarar e criar uma instância de `Queue` ({2}) que armazenará os vértices a serem visitados e explorados.

Seguindo os passos que explicamos no início deste capítulo, o método `breadthFirstSearch` recebe a instância `graph` e o vértice que será usado como ponto de origem para o nosso algoritmo. Como precisamos de um ponto de partida, vamos inserir esse vértice na fila usando `enqueue` ({3}).

Se a fila não estiver vazia ({4}), removemos um vértice da fila usando `dequeue` ({5}) e acessamos a sua lista de adjacências contendo todos os seus vizinhos ({6}). Também marcamos esse vértice como `grey` ({7}), o que significa que ele foi descoberto (mas ainda não acabamos de explorá-lo).

Para cada vizinho de `u` ({8}), obteremos o seu valor (o nome do vértice, em {9}) e, caso ele ainda não tenha sido visitado (a cor do vértice é `white`, em {10}), nós o marcaremos como descoberto (a cor é definida com `grey`, em

{11}) e adicionaremos esse vértice na fila ({12}) para que possamos acabar de explorá-lo quando ele for removido da fila com `dequeue`.

Quando acabarmos de explorar o vértice e os seus vértices adjacentes, nós o marcaremos como explorado (a cor é definida para `black`, {13}).

O método `breadthFirstSearch` que estamos implementando também recebe uma `callback` (usamos uma abordagem semelhante no Capítulo 10, Árvores, para percorrer as árvores). Esse parâmetro é opcional, e, se uma função `callback` for passada ({14}), ela será chamada.

Vamos testar esse algoritmo executando o código a seguir.

```
const printVertex = (value) => console.log('Visited vertex: ' + value); // {15}
breadthFirstSearch(graph, myVertices[0], printVertex);
```

Inicialmente declaramos uma função de callback ({15}) que simplesmente exibirá no console do navegador o nome do vértice que foi totalmente explorado pelo algoritmo. Em seguida, chamamos a função `breadthFirstSearch` passando `graph` (o mesmo grafo que usamos como exemplo para testar a classe `Graph` anteriormente neste capítulo), o primeiro vértice (o vértice A do array `myVertices`, que também declaramos no início deste capítulo) e a função de callback (`printVertex`). Quando esse código for executado, o algoritmo exibirá o resultado a seguir no console do navegador:

```
Visited vertex: A
Visited vertex: B
Visited vertex: C
Visited vertex: D
Visited vertex: E
Visited vertex: F
Visited vertex: G
Visited vertex: H
Visited vertex: I
```

A ordem dos vértices visitados é a mesma mostrada no diagrama no início desta seção.

Encontrando os caminhos mais curtos usando BFS

Até agora, apenas demonstramos como o algoritmo BFS funciona. Podemos usá-lo para outras tarefas que não apenas exibir a ordem dos vértices visitados. Por exemplo, como poderíamos resolver o problema a seguir?

Dado um grafo G e o vértice de origem v , encontre a distância (em número de arestas) de v até cada vértice $u \in G$ com o caminho mais curto entre v e u .

Dado um vértice v , o algoritmo BFS visita todos os vértices com distância 1, depois com distância 2, e assim sucessivamente. Portanto, podemos usar o algoritmo BFS para resolver esse problema. É possível modificar a função `breadthFirstSearch` para que ela devolva algumas informações para nós:

- `distances[u]` com as distâncias de v até u ;
- `predecessors[u]`, usados para obter o caminho mais curto de v até todos os demais vértices u .

Vamos observar a implementação de um método BFS melhorado:

```
const BFS = (graph, startVertex) => {
  const vertices = graph.getVertices();
  const adjList = graph.getAdjList();
  const color = initializeColor(vertices);
  const queue = new Queue();
  const distances = {} // {1}
  const predecessors = {} // {2}
  queue.enqueue(startVertex);
  for (let i = 0; i < vertices.length; i++) { // {3}
    distances[vertices[i]] = 0; // {4}
    predecessors[vertices[i]] = null; // {5}
  }
  while (!queue.isEmpty()) {
    const u = queue.dequeue();
    const neighbors = adjList.get(u);
    color[u] = Colors.GREY;
    for (let i = 0; i < neighbors.length; i++) {
      const w = neighbors[i];
      if (color[w] === Colors.WHITE) {
        color[w] = Colors.GREY;
        distances[w] = distances[u] + 1; // {6}
        predecessors[w] = u; // {7}
        queue.enqueue(w);
      }
    }
    color[u] = Colors.BLACK;
  }
  return { // {8}
    distances,
    predecessors
  };
}
```

```
};
```

O que mudou nessa versão do método **BFS**?

Temos de declarar também o array **distances** (**{1}**) e o array **predecessors** (**{2}**).

O próximo passo é inicializar o array **distances** com **0** (**{4}**) e o array **predecessors** com **null** (**{5}**) para todo vértice do grafo (**{3}**).

Quando descobrimos o vizinho **w** de um vértice **u**, definimos o valor do antecessor de **w** com **u** (**{7}**) e incrementamos a distância (**{6}**) entre **startVertex** e **w** somando 1 à distância de **u** (como **u** é um antecessor de **w**, já temos o valor de **distances[u]**).

No final do método, podemos devolver um objeto com **distances** e **predecessors** (**{8}**).

Agora podemos executar o método **BFS** novamente e armazenar o seu valor de retorno em uma variável, assim:

```
const shortestPathA = BFS(graph, myVertices[0]);
console.log(shortestPathA);
```

Como executamos o método **BFS** para o vértice **A**, eis a saída que será exibida no console:

```
distances: [A: 0, B: 1, C: 1, D: 1, E: 2, F: 2, G: 2, H: 2, I: 3],
predecessors: [A: null, B: "A", C: "A", D: "A", E: "B", F: "B", G: "C", H: "D",
I: "E"]
```

Isso significa que o vértice **A** está a uma distância de 1 aresta dos vértices **B**, **C** e **D**, a uma distância de 2 arestas dos vértices **E**, **F**, **G** e **H**, e a uma distância de 3 arestas do vértice **I**.

Com o array de antecessores, podemos construir o caminho do vértice **A** até os demais vértices usando o código a seguir.

```
const fromVertex = myVertices[0]; // {9}
for (i = 1; i < myVertices.length; i++) { // {10}
  const toVertex = myVertices[i]; // {11}
  const path = new Stack(); // {12}
  for (let v = toVertex;
    v !== fromVertex;
    v = shortestPathA.predecessors[v]) { // {13}
    path.push(v); // {14}
  }
  path.push(fromVertex); // {15}
  let s = path.pop(); // {16}
  while (!path.isEmpty()) { // {17}
```

```

    s += ' - ' + path.pop(); // {18}
}
console.log(s); // {19}
}

```

Usaremos o vértice A como o vértice de origem (**{9}**). Para todos os outros vértices (exceto o vértice A, na linha **{10}**), calcularemos o caminho do vértice A até o respectivo vértice. Para isso, obtemos o valor (nome) do vértice a partir do array `myVertices` (**{11}**) e criamos uma pilha para armazenar os valores dos caminhos (**{12}**).

Em seguida, seguimos o caminho de `toVertex` para `fromVertex` (**{13}**). A variável `v` recebe o valor de seu antecessor e poderemos fazer o mesmo caminho de trás para a frente. Adicionamos a variável `v` na pilha (**{14}**) e, por fim, acrescentamos também o vértice de origem na pilha (**{15}**) para termos o caminho completo.

Depois disso, criamos uma string `s` e lhe atribuímos o vértice de origem (será o último vértice adicionado na pilha, portanto será o primeiro item a ser removido, em **{16}**). Enquanto a pilha não estiver vazia (**{17}**), removemos um item dela e o concatenamos ao valor existente na string `s` (**{18}**). Por fim, simplesmente exibimos o caminho no console do navegador (**{19}**).

Após a execução do código anterior, veremos a seguinte saída:

```

A - B
A - C
A - D
A - B - E
A - B - F
A - C - G
A - D - H
A - B - E - I

```

Temos aqui o caminho mais curto (em número de arestas) de A até os demais vértices do grafo.

Estudos adicionais sobre algoritmos de caminhos mais curtos

O grafo que usamos nesse exemplo não é um grafo com pesos. Se quiséssemos calcular o caminho mais curto em grafos com pesos (por exemplo, qual é o caminho mais curto entre a cidade A e a cidade B – um algoritmo usado em GPS e no Google Maps), o BFS não seria um algoritmo

apropriado.

Temos, por exemplo, o **algoritmo de Dijkstra**, que resolve o problema do caminho mais curto com uma única origem. O **algoritmo de Bellman-Ford** resolve o problema da origem única se os pesos das arestas forem negativos. O **algoritmo de busca A*** apresenta o caminho mais curto para um único par de vértices usando métodos heurísticos para tentar agilizar a pesquisa. O **algoritmo de Floyd-Warshall** apresenta o caminho mais curto para todos os pares de vértices.

Exploraremos o algoritmo de Dijkstra e o algoritmo de Floyd-Warshall mais adiante neste capítulo.

Busca em profundidade (DFS)

O algoritmo DFS começará percorrendo o grafo a partir do primeiro vértice especificado, seguirá um caminho até que o seu último vértice tenha sido visitado. Em seguida, um backtracking (retrocesso) será feito no caminho e o próximo caminho será seguido. Em outras palavras, o algoritmo visita os vértices em profundidade antes, e depois em largura, como mostra o diagrama a seguir.

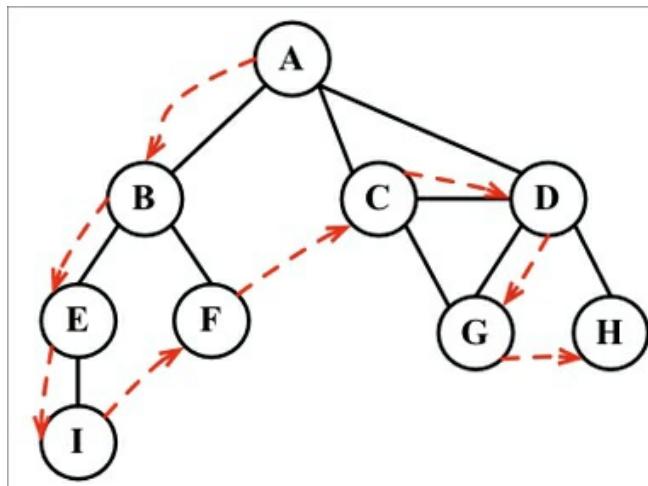


Figura 12.9

O algoritmo DFS não precisa de um vértice de origem. Nesse algoritmo, para cada vértice não visitado v no grafo G , você deverá visitar o vértice v .

Para visitar o vértice v , execute o seguinte:

1. Marque v como descoberto (cinza).

2. Para todos os vizinhos não visitados (branco) w de v , visite o vértice w .
3. Marca v como explorado (preto).

Como podemos observar, os passos do DFS são recursivos, o que significa que o algoritmo DFS utiliza uma pilha para armazenar as chamadas (uma pilha criada pelas chamadas recursivas).

Vamos implementar o algoritmo DFS assim:

```
const depthFirstSearch = (graph, callback) => { // {1}
  const vertices = graph.getVertices();
  const adjList = graph.getAdjList();
  const color = initializeColor(vertices);
  for (let i = 0; i < vertices.length; i++) { // {2}
    if (color[vertices[i]] === Colors.WHITE) { // {3}
      depthFirstSearchVisit(vertices[i], color, adjList, callback); // {4}
    }
  }
};

const depthFirstSearchVisit = (u, color, adjList, callback) => {
  color[u] = Colors.GREY; // {5}
  if (callback) { // {6}
    callback(u);
  }
  const neighbors = adjList.get(u); // {7}
  for (let i = 0; i < neighbors.length; i++) { // {8}
    const w = neighbors[i]; // {9}
    if (color[w] === Colors.WHITE) { // {10}
      depthFirstSearchVisit(w, color, adjList, callback); // {11}
    }
  }
  color[u] = Colors.BLACK; // {12}
};
```

A função `depthFirstSearch` recebeu uma instância da classe `Graph` e a função `callback` ({1}). Depois de inicializar as cores de cada vértice, para cada vértice não visitado ({2} e {3}) da instância de `Graph`, chamamos a função recursiva (`private`) `depthFirstSearchVisit`, passando o vértice `u` sendo visitado, o array `color` e a função `callback` ({4}).

Sempre que o vértice `u` for visitado, nós o marcaremos como descoberto (`grey`, em {5}). Se houver uma função `callback` ({6}), nós a chamaremos. Então, o próximo passo será obter a lista de vizinhos do vértice `u` ({7}). Para cada vizinho `w` não visitado (cor `white`, {10} e {8}) de `u`, chamamos a função `depthFirstSearchVisit`, passando `w` como o

vértice a ser visitado (`{11}`, adicionando `w` à pilha para que ele seja visitado em seguida). No final, depois que o vértice e seus vértices adjacentes forem visitados em profundidade, fazemos um backtrack (retrocesso), o que significa que o vértice foi totalmente explorado e será marcado com a cor **black** (`{12}`).

Vamos testar o método `depthFirstSearch` executando o código a seguir.

```
depthFirstSearch(graph, printVertex);
```

Eis a saída do código:

```
Visited vertex: A
Visited vertex: B
Visited vertex: E
Visited vertex: I
Visited vertex: F
Visited vertex: C
Visited vertex: D
Visited vertex: G
Visited vertex: H
```

A ordem é a mesma mostrada no diagrama apresentado no início desta seção. O diagrama a seguir (Figura 12.10) mostra o processo do algoritmo, passo a passo.

Nesse grafo que usamos como exemplo, a linha `{4}` será executada apenas uma vez, pois todos os demais vértices têm um caminho até o primeiro vértice que chamou a função `depthFirstSearchVisit` (vértice A). Se o vértice B fosse o primeiro a chamar a função, a linha `{4}` seria executada novamente para outro vértice (por exemplo, o vértice A).

O algoritmo usado pelo Angular (v2+) em sua lógica de detecção de mudança (verificar se o template HTML precisa ser atualizado) é muito semelhante ao algoritmo DFS. Para saber mais sobre ele, acesse <https://goo.gl/9kQj4i>. As estruturas de dados e os algoritmos também são importantes para entender como os frameworks de frontend famosos funcionam, e para levar o seu conhecimento a um novo patamar!

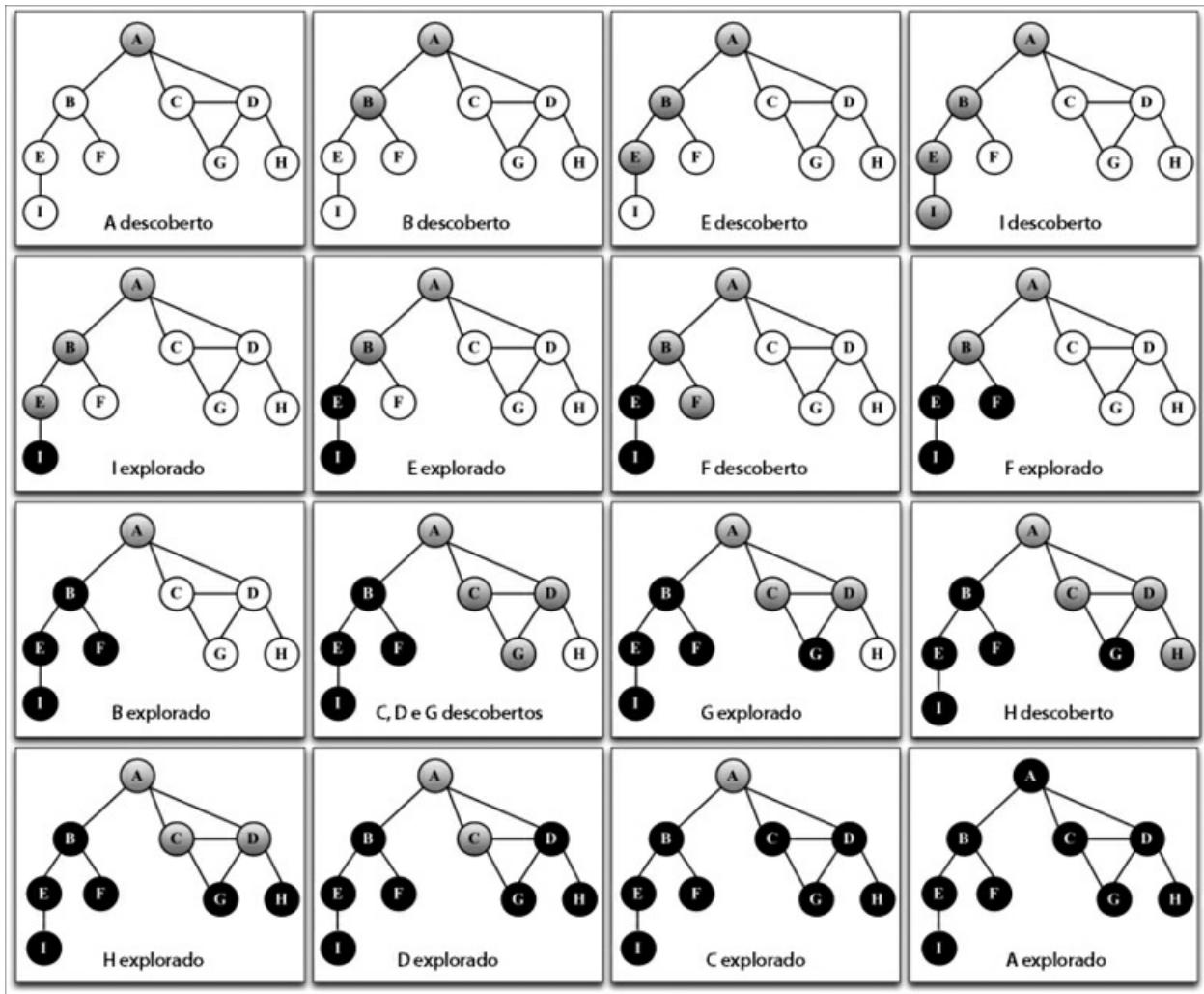


Figura 12.10

Explorando o algoritmo DFS

Até agora, simplesmente demonstramos como o algoritmo DFS funciona. Podemos usá-lo para outras tarefas que não sejam apenas exibir a ordem dos vértices visitados.

Dado um grafo G , o algoritmo DFS percorre todos os vértices de G e constrói uma floresta (uma coleção de **árvores com raiz**), junto com um conjunto de vértices de origem (raízes), e exibe dois arrays com: o instante da descoberta e o instante do término da exploração. Podemos modificar a função `depthFirstSearch` para que ela devolva algumas informações para nós, por exemplo:

- o instante da descoberta $d[u]$ de u ;

- o instante final $f[u]$ quando u é marcado com preto;
- os antecessores (**predecessors**) $p[u]$ de u .

Vamos observar a implementação do método **DFS**:

```
export const DFS = graph => {
  const vertices = graph.getVertices();
  const adjList = graph.getAdjList();
  const color = initializeColor(vertices);
  const d = {};
  const f = {};
  const p = {};
  const time = { count: 0 }; // {1}
  for (let i = 0; i < vertices.length; i++) { // {2}
    f[vertices[i]] = 0;
    d[vertices[i]] = 0;
    p[vertices[i]] = null;
  }
  for (let i = 0; i < vertices.length; i++) {
    if (color[vertices[i]] === Colors.WHITE) {
      DFSVisit(vertices[i], color, d, f, p, time, adjList);
    }
  }
  return { // {3}
    discovery: d,
    finished: f,
    predecessors: p
  };
};

const DFSVisit = (u, color, d, f, p, time, adjList) => {
  color[u] = Colors.GREY;
  d[u] = ++time.count; // {4}
  const neighbors = adjList.get(u);
  for (let i = 0; i < neighbors.length; i++) {
    const w = neighbors[i];
    if (color[w] === Colors.WHITE) {
      p[w] = u; // {5}
      DFSVisit(w, color, d, f, p, time, adjList);
    }
  }
  color[u] = Colors.BLACK;
  f[u] = ++time.count; // {6}
};
```

Como queremos monitorar o instante da descoberta e o instante em que terminamos a exploração, temos de declarar uma variável para isso ({1}).

Estamos declarando `time` como um objeto com a propriedade `count` por causa do sistema de parâmetros por valor e por referência em JavaScript. Em algumas linguagens, há uma diferença entre passar os parâmetros como valores ou como referências. Os valores primitivos são passados por valor, o que significa que o escopo do valor será válido somente durante a execução da função. Se modificarmos esse valor, o novo valor terá um escopo válido somente dentro da função. Quando os parâmetros são passados como referências (objetos), se modificamos qualquer propriedade do objeto, estaremos modificando o seu valor. Os objetos são passados como referência porque somente a referência à memória será passada para a função ou o método. Nesse caso em particular, queremos que o contador de tempo seja atualizado de modo a ser usado *globalmente* durante a execução desse algoritmo; assim, devemos passar o objeto como parâmetro, e não apenas o seu valor primitivo.

Em seguida, declaramos os arrays `d`, `f` e `p`; devemos também inicializar esses arrays para cada vértice do grafo `{2}`. No final do método, devolvemos esses valores `{3}` para que possamos trabalhar com eles depois.

Quando um vértice é inicialmente descoberto, registramos o instante de sua descoberta `{4}`. Quando ele for descoberto como uma aresta de `u`, registramos também o seu antecessor `{5}`. No final, quando o vértice estiver totalmente explorado, registramos o instante final `{6}`.

Qual é a ideia que está por trás do algoritmo DFS? As arestas são exploradas a partir do vértice `u` descoberto mais recentemente. Apenas as arestas para os vértices não visitados são exploradas. Quando todas as arestas de `u` forem exploradas, o algoritmo retrocederá (fará um backtracking) a fim de explorar outras arestas no ponto em que o vértice `u` foi descoberto. O processo continuará até descobrirmos todos os vértices que podem ser alcançados a partir do vértice de origem. Se restar algum vértice não descoberto, repetiremos o processo para um novo vértice de origem. Repetiremos o algoritmo até que todos os vértices do grafo tenham sido explorados.

Há dois pontos que devemos verificar no algoritmo DFS:

- A variável **time** só poderá ter valores de uma a duas vezes o número de vértices do grafo ($2|V|$).
- Para todos os vértices u , $d[u] < f[u]$ (isto é, o instante da descoberta deve ter um valor menor que o instante final).

Com essas duas suposições, temos a seguinte regra:

$$1 \leq d[u] < f[u] \leq 2|V|$$

Se executarmos o novo método DFS no mesmo grafo novamente, teremos os instantes de descoberta/fim a seguir para cada vértice do grafo:

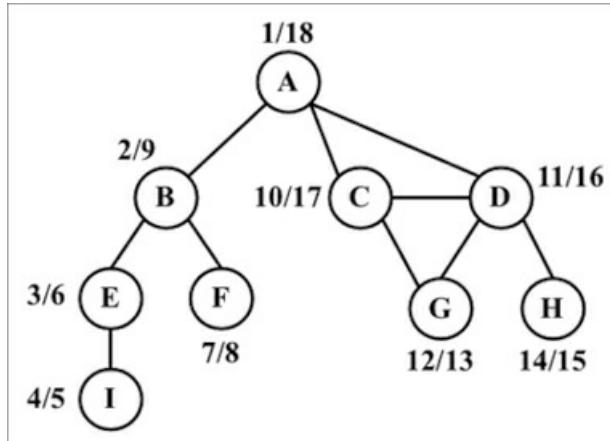


Figura 12.11

No entanto, o que podemos fazer com essas informações? Vamos descobrir na próxima seção.

Ordenação topológica usando DFS

Dado o grafo a seguir, suponha que cada vértice seja uma tarefa que você deve executar:

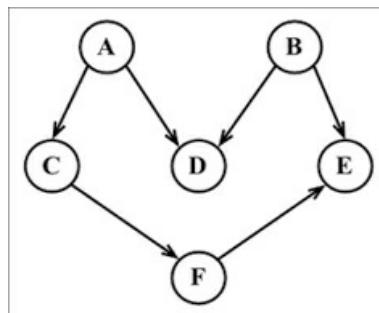


Figura 12.12

Esse é um grafo direcionado, o que significa que há uma ordem em que as tarefas devem ser executadas. Por exemplo, a tarefa F não pode ser executada antes da tarefa A. Observe que o grafo anterior também não tem nenhum ciclo, ou seja, é um grafo acíclico. Assim, podemos dizer que o grafo anterior é um DAG (Directed Acyclic Graph, ou Grafo Acíclico Direcionado).

Quando houver necessidade de especificar a ordem com que algumas tarefas ou passos devem ser executados, chamamos a isso de **ordenação topológica** (ou **topsort**, ou até mesmo **toposort**). Esse problema está presente em diferentes cenários de nossas vidas. Por exemplo, quando iniciamos um curso de ciência da computação, há uma sequência de disciplinas que devemos cursar antes de fazer qualquer outra disciplina (você não pode cursar Algoritmos II antes de fazer Algoritmos I). Quando trabalhamos em um projeto, há alguns passos que devem ser executados em sequência; por exemplo, devemos obter inicialmente os requisitos do cliente, depois desenvolver o que foi solicitado por ele e, por fim, entregar o projeto. Você não pode entregar o projeto e depois coletar os requisitos.

A ordenação topológica só pode ser aplicada em DAGs. Então, como podemos usar a ordenação topológica com DFS? Vamos executar o algoritmo DFS no diagrama apresentado no início desta seção:

```
graph = new Graph(true); // grafo direcionado
myVertices = ['A', 'B', 'C', 'D', 'E', 'F'];
for (i = 0; i < myVertices.length; i++) {
    graph.addVertex(myVertices[i]);
}
graph.addEdge('A', 'C');
graph.addEdge('A', 'D');
graph.addEdge('B', 'D');
graph.addEdge('B', 'E');
graph.addEdge('C', 'F');
graph.addEdge('F', 'E');
const result = DFS(graph);
```

Esse código criará o grafo e suas arestas, executará o algoritmo DFS melhorado e armazenará os resultados na variável **result**. O diagrama a seguir mostra os instantes de descoberta e de fim de exploração do grafo após o DFS ter sido executado.

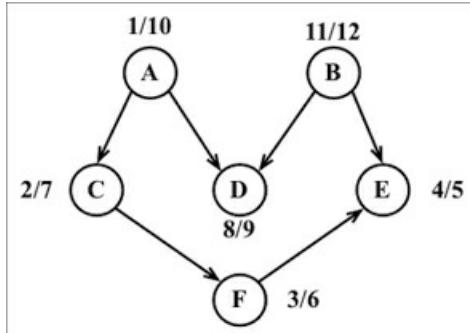


Figura 12.13

Tudo que temos a fazer agora é ordenar o array de instantes de finalização em ordem decrescente e teremos a ordenação topológica do grafo, assim:

```
const fTimes = result.finished;
s = '';
for (let count = 0; count < myVertices.length; count++) {
    let max = 0;
    let maxName = null;
    for (i = 0; i < myVertices.length; i++) {
        if (fTimes[myVertices[i]] > max) {
            max = fTimes[myVertices[i]];
            maxName = myVertices[i];
        }
    }
    s += ' - ' + maxName;
    delete fTimes[maxName];
}
console.log(s);
```

Após a execução do código anterior, veremos a seguinte saída:

B - A - D - C - F - E

Observe que o resultado do toposort anterior é apenas uma das possibilidades. Podemos ter resultados diferentes se modificarmos um pouco o algoritmo. Por exemplo, o resultado a seguir é uma das várias outras possibilidades.

A - B - C - D - F - E

Esse também poderia ser um resultado aceitável.

Algoritmos de caminho mais curto

Dado um mapa de ruas, suponha que você queira sair do ponto A e chegar ao ponto B usando o caminho mais curto possível. Como exemplo desse problema, podemos usar o caminho de **Santa Monica Blvd** até a

Hollywood Blvd em Los Angeles, como mostra a Figura 12.14.

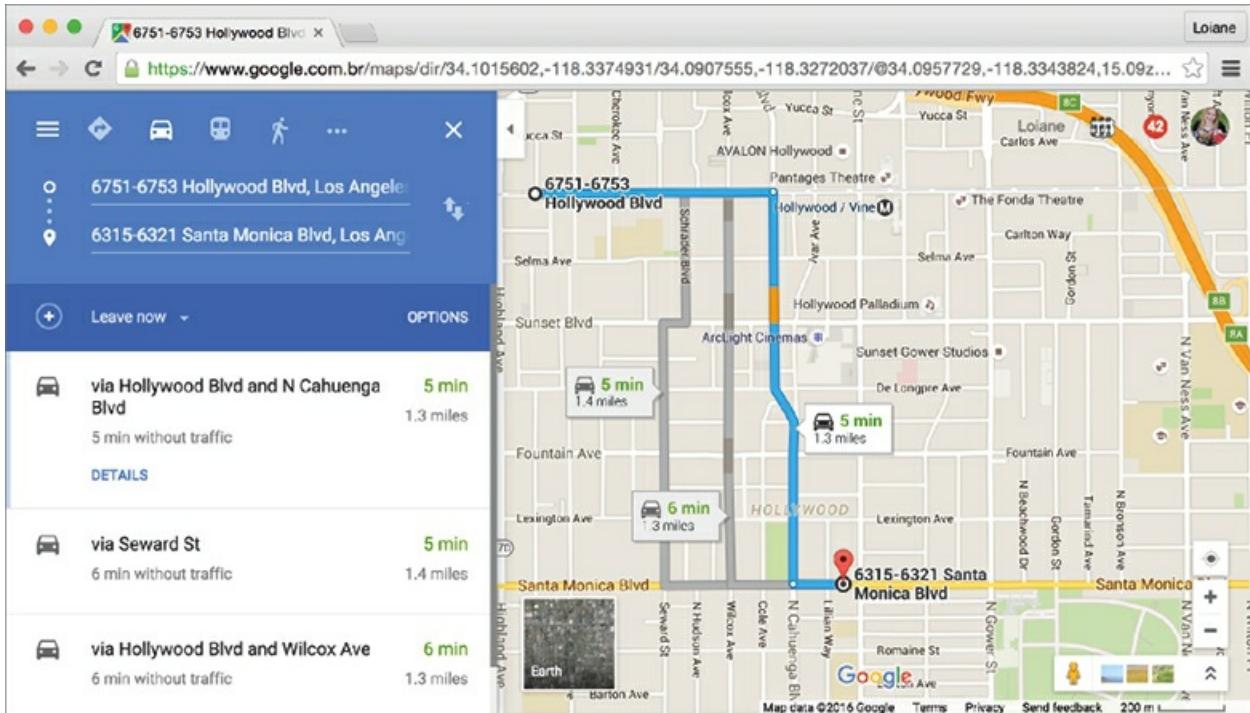


Figura 12.14

Esse é um problema muito comum em nossas vidas, e usaremos aplicações como *Apple* ou *Google Maps* ou *Waze* para tentar resolvê-lo, especialmente se você mora em uma cidade grande. É claro que temos outras restrições envolvidas também, como tempo ou tráfego, mas o problema original permanece o mesmo: como vamos de A até B usando o caminho mais curto?

Podemos usar grafos para resolver esse problema para nós, e o algoritmo é chamado de algoritmo do caminho mais curto. Há dois algoritmos muito famosos, o algoritmo de Dijkstra e o algoritmo de Floyd-Warshall, que serão discutidos nas próximas seções.

Algoritmo de Dijkstra

O algoritmo de Dijkstra é um **algoritmo guloso** (greedy algorithm – você conhecerá melhor esse tipo de algoritmo no Capítulo 14, *Design de algoritmos e técnicas*) para calcular o caminho mais curto entre uma única origem e todos os demais vértices, o que significa que podemos usá-lo para calcular o caminho mais curto do vértice de um grafo para todos os outros.

Considere o grafo a seguir.

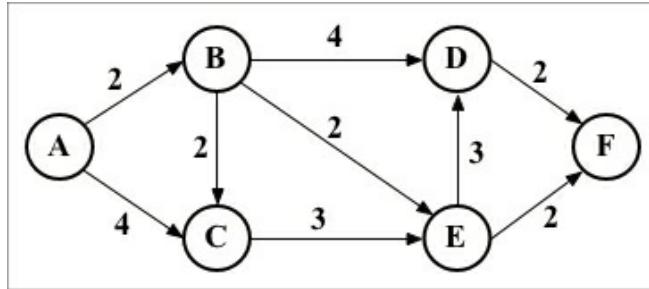


Figura 12.15

Vamos ver como podemos encontrar o caminho mais curto entre o vértice A e todos os demais vértices. Antes, porém, devemos declarar a matriz de adjacências que representa o grafo anterior, assim:

```

var graph = [[0, 2, 4, 0, 0, 0],
             [0, 0, 1, 4, 2, 0],
             [0, 0, 0, 0, 3, 0],
             [0, 0, 0, 0, 0, 2],
             [0, 0, 0, 3, 0, 2],
             [0, 0, 0, 0, 0, 0]];
  
```

O algoritmo de Dijkstra é apresentado a seguir.

```

const INF = Number.MAX_SAFE_INTEGER;
const dijkstra = (graph, src) => {
  const dist = [];
  const visited = [];
  const { length } = graph;
  for (let i = 0; i < length; i++) { // {1}
    dist[i] = INF;
    visited[i] = false;
  }
  dist[src] = 0; // {2}
  for (let i = 0; i < length - 1; i++) { // {3}
    const u = minDistance(dist, visited); // {4}
    visited[u] = true; // {5}
    for (let v = 0; v < length; v++) {
      if (!visited[v] &&
          graph[u][v] !== 0 &&
          dist[u] !== INF &&
          dist[u] + graph[u][v] < dist[v]) { // {6}
        dist[v] = dist[u] + graph[u][v]; // {7}
      }
    }
  }
  return dist; // {8}
}
  
```

};

Eis a descrição do funcionamento do algoritmo:

1. {1}: em primeiro lugar, devemos inicializar todas as distâncias (**dist**) como infinitas (número máximo `INF = Number.MAX_SAFE_INTEGER` em JavaScript) e **visited**[] como `false`.
2. {2}: em segundo lugar, definimos a distância do vértice de origem a partir de si mesmo como `0`.
3. {3}: então encontramos o caminho mais curto para todos os vértices.
4. {4}: para isso, devemos selecionar o vértice com distância mínima a partir do conjunto de vértices que ainda não foram processados.
5. {5}: devemos marcar o vértice selecionado como **visited** para que não façamos o cálculo duas vezes.
6. {6}: caso o caminho mais curto seja encontrado, definimos o novo valor como o caminho mais curto ({7}).
7. {8}: depois que todos os vértices forem processados, devolvemos o resultado contendo o valor do caminho mais curto a partir do vértice de origem (**src**) para todos os outros vértices do grafo.

Para calcular a **minDistance**, procuraremos o valor mínimo no array **dist**, como vemos a seguir, e devolveremos o índice do array que contém esse valor.

```
const minDistance = (dist, visited) => {
  let min = INF;
  let minIndex = -1;
  for (let v = 0; v < dist.length; v++) {
    if (visited[v] === false && dist[v] <= min) {
      min = dist[v];
      minIndex = v;
    }
  }
  return minIndex;
};
```

Se o algoritmo anterior for executado no grafo apresentado no início desta seção, veremos a saída a seguir.

```
0 0
1 2
2 4
3 6
4 4
```

Também é possível modificar o algoritmo para que ele devolva o valor do caminho mais curto, além do próprio caminho.

Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall é um **algoritmo de programação dinâmica** (você conhecerá melhor a programação dinâmica no Capítulo 14, *Design de algoritmos e técnicas*) para calcular todos os caminhos mais curtos em um grafo. Com esse algoritmo, podemos encontrar o caminho mais curto a partir de todas as origens para todos os vértices.

Eis o algoritmo de Floyd-Warshall:

```
const floydWarshall = graph => {
  const dist = [];
  const { length } = graph;
  for (let i = 0; i < length; i++) { // {1}
    dist[i] = [];
    for (let j = 0; j < length; j++) {
      if (i === j) {
        dist[i][j] = 0; // {2}
      } else if (!isFinite(graph[i][j])) {
        dist[i][j] = Infinity; // {3}
      } else {
        dist[i][j] = graph[i][j]; // {4}
      }
    }
  }
  for (let k = 0; k < length; k++) { // {5}
    for (let i = 0; i < length; i++) {
      for (let j = 0; j < length; j++) {
        if (dist[i][k] + dist[k][j] < dist[i][j]) { // {6}
          dist[i][j] = dist[i][k] + dist[k][j]; // {7}
        }
      }
    }
  }
  return dist;
};
```

A seguir, apresentamos a descrição do funcionamento do algoritmo.

Em primeiro lugar, inicializamos o array de distâncias com o valor do peso entre cada vértice ({1}), pois a distância mínima possível entre **i** e **j** é o

peso desses vértices ($\{4\}$). A distância do vértice para si mesmo é zero ($\{2\}$). Caso não haja nenhuma aresta entre dois vértices, isso será representado com **Infinity** ($\{3\}$). Usando os vértices $0 \dots k$ como pontos intermediários ($\{5\}$), o caminho mais curto entre i e j é dado por k . A fórmula usada para calcular o caminho mais curto entre i e j pelo vértice k é dada na linha $\{6\}$. Se um novo valor para o caminho mais curto for encontrado, nós o usaremos e ele será armazenado ($\{7\}$).

A fórmula na linha $\{6\}$ é o coração do algoritmo de Floyd-Warshall. Se o algoritmo anterior for executado no grafo que usamos como exemplo no início desta seção, veremos a saída a seguir.

```
0 2 4 6 4 6
INF 0 2 4 2 4
INF INF 0 6 3 5
INF INF INF 0 INF 2
INF INF INF 3 0 2
INF INF INF INF INF 0
```

Nesse caso, **INF** significa que não há nenhum caminho mais curto entre os vértices i e j .

Outra maneira de obter o mesmo resultado seria executar o algoritmo de Dijkstra para cada vértice do grafo.

Árvore de extensão mínima (MST)

O problema da **MST** (Minimum Spanning Tree, ou Árvore de Extensão Mínima/Árvore Geradora Mínima) é muito comum no design de redes. Suponha que você tenha um negócio com vários escritórios e queira conectar as linhas telefônicas dos escritórios, umas às outras, com um custo total mínimo para economizar. Qual é a melhor maneira de resolver esse problema?

Isso também pode ser aplicado ao problema das pontes entre ilhas. Suponha que você tenha n ilhas e queira construir pontes para conectar todas elas com um custo mínimo.

Os dois problemas anteriores podem ser resolvidos com um algoritmo de MST, em que cada escritório ou ilha pode ser representado como um vértice de um grafo e as arestas representam o custo. A seguir, temos um exemplo de um grafo em que as arestas mais espessas são uma solução para a MST.

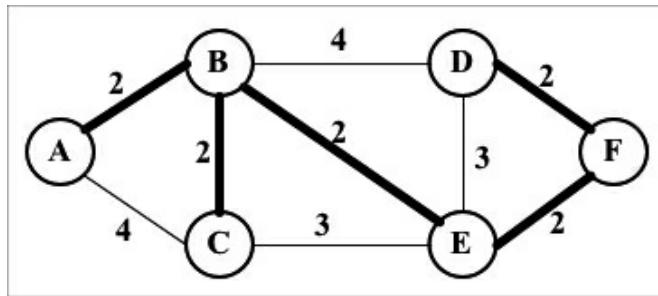


Figura 12.16

Há dois algoritmos principais para encontrar as MSTs: o **algoritmo de Prim** e o **algoritmo de Kruskal**, que veremos nas próximas seções.

Algoritmo de Prim

O algoritmo de Prim é um algoritmo guloso (greedy), que resolve um problema de MST para um grafo conectado não direcionado com peso. Ele encontra um subconjunto das arestas que formam uma árvore, incluindo todos os vértices, em que o peso total de todas as arestas da árvore é minimizado.

O algoritmo de Prim é apresentado a seguir.

```

const INF = Number.MAX_SAFE_INTEGER;
const prim = graph => {
  const parent = [];
  const key = [];
  const visited = [];
  const { length } = graph;
  for (let i = 0; i < length; i++) { // {1}
    key[i] = INF;
    visited[i] = false;
  }
  key[0] = 0; // {2}
  parent[0] = -1;
  for (let i = 0; i < length - 1; i++) { // {3}
    const u = minKey(graph, key, visited); // {4}
    visited[u] = true; // {5}
    for (let v = 0; v < length; v++) {
      if (graph[u][v] && !visited[v] && graph[u][v] < key[v]) { // {6}
        parent[v] = u; // {7}
        key[v] = graph[u][v]; // {8}
      }
    }
  }
  return parent; // {9}
}

```

};

Eis a descrição do funcionamento do algoritmo:

1. {1}: em primeiro lugar, devemos inicializar todos os vértices em **key** com infinito (número máximo `INF = Number.MAX_SAFE_INTEGER` em JavaScript) e em **visited[]** com `false`.
2. {2}: em segundo lugar, definiremos a primeira **key** com `0` para que esse vértice seja escolhido como o primeiro vértice e **parent[0] = -1** porque o primeiro nó é sempre a raiz da MST.
3. {3}: então encontraremos a MST para todos os vértices.
4. {4}: para isso, devemos selecionar o vértice com a **key** mínima a partir do conjunto de vértices que ainda não foi processado (é a mesma função que usamos no algoritmo de Dijkstra [`minDistance`], porém com um nome diferente).
5. {5}: devemos marcar o vértice selecionado como **visited** para que não façamos o cálculo duas vezes.
6. {6}: caso um peso mínimo seja encontrado, armazenaremos o valor do caminho da MST (**parent**, em {7}) e definiremos o novo custo para o valor da MST ({8}).
7. {9}: depois que todos os vértices forem processados, devolveremos o resultado contendo a MST.

Se compararmos o algoritmo de Prim com o algoritmo de Dijkstra, veremos que eles são muito parecidos, com exceção das linhas {7} e {8}. A linha {7} mantém o array de pais, que é o array que armazena a MST. A linha {8} armazena o valor mínimo da aresta, enquanto, no algoritmo de Dijkstra, o array de distâncias é usado no lugar do array **key** para armazenar a distância. Podemos modificar o algoritmo de Dijkstra de modo a acrescentar o array **parent** e, desse modo, poderemos monitorar o caminho, junto com o valor da distância.

Vamos agora executar o algoritmo anterior para o grafo a seguir.

```
var graph = [[0, 2, 4, 0, 0, 0],
             [2, 0, 2, 4, 2, 0],
             [4, 2, 0, 0, 3, 0],
             [0, 4, 0, 0, 3, 2],
             [0, 2, 3, 3, 0, 2],
             [0, 0, 0, 2, 2, 0]];
```

Veremos a seguinte saída:

Edge Weight

```
0 - 1 2
1 - 2 2
5 - 3 2
1 - 4 2
4 - 5 2
```

Algoritmo de Kruskal

De modo semelhante ao algoritmo de Prim, o algoritmo de Kruskal também é um algoritmo guloso, que encontra a MST para um grafo conectado, não direcionado com peso.

O algoritmo de Kruskal é apresentado a seguir.

```
const kruskal = graph => {
  const { length } = graph;
  const parent = [];
  let ne = 0;
  let a; let b; let u; let v;
  const cost = initializeCost(graph); // {1}
  while (ne < length - 1) { // {2}
    for (let i = 0, min = INF; i < length; i++) { // {3}
      for (let j = 0; j < length; j++) {
        if (cost[i][j] < min) {
          min = cost[i][j];
          a = u = i;
          b = v = j;
        }
      }
    }
    u = find(u, parent); // {4}
    v = find(v, parent); // {5}
    if (union(u, v, parent)) { // {6}
      ne++;
    }
    cost[a][b] = cost[b][a] = INF; // {7}
  }
  return parent;
};
```

Eis a descrição do funcionamento do algoritmo:

1. {1}: em primeiro lugar, copiamos os valores da matriz de adjacências para o array **cost** para que possamos modificá-lo sem perder os valores originais ({7}).

2. {2}: enquanto a MST tiver menos arestas que o total de arestas menos 1.
3. {3}: encontra a aresta com o custo mínimo.
4. {4} e {5}: para evitar ciclos, verifica se a aresta já está na MST.
5. {6}: se as arestas **u** e **v** não forem iguais, adiciona na MST.
6. {7}: remove as arestas da lista para que não façamos o cálculo duas vezes.
7. {8}: devolve a MST.

A seguir, apresentamos a função **find**. Ela evita ciclos em uma MST.

```
const find = (i, parent) => {
  while (parent[i]) {
    i = parent[i];
  }
  return i;
};
```

Apresentamos também a função **union** a seguir.

```
const union = (i, j, parent) => {
  if (i !== j) {
    parent[j] = i;
    return true;
  }
  return false;
};
```

Há algumas variações desse algoritmo que podem ser desenvolvidas. Isso dependerá da estrutura de dados usada para ordenar os pesos dos valores das arestas (como em uma fila de prioridades), e também de como o grafo é representado.

Resumo

Neste capítulo, vimos os conceitos básicos associados aos grafos. Conhecemos as diferentes maneiras com as quais podemos representar essa estrutura de dados e implementamos uma classe para representar um grafo usando uma lista de adjacências. Também aprendemos a percorrer um grafo usando as abordagens BFS e DFS. Este capítulo também incluiu duas aplicações de BFS e DFS para encontrar o caminho mais curto usando BFS e a ordenação topológica com DFS.

O capítulo também abordou alguns dos algoritmos famosos, como o de Dijkstra e o de Floyd-Warshall para calcular o caminho mais curto. Além disso, discutimos o algoritmo de Prim e o algoritmo de Kruskal para calcular a MST (Minimum Spanning Tree, ou Árvore de Extensão Mínima) de um grafo.

No próximo capítulo, veremos os algoritmos mais comuns de ordenação usados em ciência da computação.

CAPÍTULO 13

Algoritmos de ordenação e de busca

Suponha que temos uma agenda telefônica (ou um caderninho de anotações) que não esteja em nenhuma ordem. Quando houver necessidade de adicionar um contato com números de telefone, você simplesmente o anotará no próximo espaço disponível. Suponha também que você tenha uma grande quantidade de contatos em sua lista. Em um dia qualquer, você precisa localizar um contato em particular bem como o seu número de telefone. No entanto, como a lista de contatos não está organizada em nenhuma sequência, você deverá verificar cada um dos contatos até encontrar a informação desejada. Essa abordagem é horrível, não acha? Imagine se tivesse de localizar um contato nas *Páginas Amarelas* e elas não estivessem organizadas! Demoraria uma eternidade!

Por esse motivo, entre outros, é necessário organizar os conjuntos de informações, por exemplo, aqueles que armazenamos em estruturas de dados. Os algoritmos para ordenação e busca são amplamente utilizados nos problemas cotidianos que devemos resolver.

Neste capítulo, conheceremos os algoritmos de ordenação e busca mais comumente usados como bubble sort (ordenação por flutuação), selection sort (ordenação por seleção), insertion sort (ordenação por inserção), merge sort (ordenação por intercalação ou mistura), quick sort (ordenação rápida), counting sort (ordenação por contagem), bucket sort (ordenação por balde ou recipiente) e radix sort (ordenação por raízes), assim como os algoritmos de busca sequencial, por interpolação e binária.

Algoritmos de ordenação

Nesta seção, descreveremos alguns dos algoritmos mais conhecidos de ordenação em ciência da computação. Começaremos pelo algoritmo mais lento e então discutiremos alguns algoritmos melhores. Veremos que, inicialmente, devemos aprender a ordenar e então pesquisaremos uma dada informação qualquer.

Veja uma versão animada do funcionamento dos algoritmos mais famosos discutidos neste capítulo nos seguintes links: <https://visualgo.net/en/sorting> e <https://www.toptal.com/developers/sorting-algorithms>.

Vamos começar!

Bubble sort

Quando começam a estudar os algoritmos de ordenação, os desenvolvedores em geral conhecem antes o bubble sort, pois é o mais simples deles. No entanto, é um dos piores algoritmos de ordenação no que diz respeito ao tempo de execução, e você verá por quê.

O algoritmo de **bubble sort** (ordenação por flutuação) compara cada dois valores adjacentes e faz a sua troca (swap) se o primeiro valor for maior que o segundo. Ele tem esse nome porque os valores tendem a se mover para cima na ordem correta, como se fossem bolhas (bubbles) subindo para a superfície.

Vamos implementar o algoritmo de bubble sort assim:

```
function bubbleSort(array, compareFn = defaultCompare) {  
    const { length } = array; // {1}  
    for (let i = 0; i < length; i++) { // {2}  
        for (let j = 0; j < length - 1; j++) { // {3}  
            if (compareFn(array[j], array[j + 1]) === Compare.BIGGER_THAN) { // {4}  
                swap(array, j, j + 1); // {5}  
            }  
        }  
    }  
    return array;  
}
```

Todo algoritmo de ordenação **sem distribuição** que criaremos neste capítulo receberá o array a ser ordenado como parâmetro, além de uma função de comparação. Para facilitar a compreensão dos testes, trabalharemos com arrays de números em nossos exemplos. Contudo, caso seja necessário ordenar um array de objetos complexos (array de objetos **people** ordenado pela propriedade **age**), nosso algoritmo estará igualmente preparado. A função **compareFn**, a ser usada como default, é a função **defaultCompare** que usamos nos capítulos anteriores (**return a < b ? Compare.LESS_THAN : Compare.BIGGER_THAN**).

Inicialmente vamos declarar uma variável chamada `length`, que armazenará o tamanho do array (`{1}`). Esse passo nos ajudará a obter o tamanho do array, que será usado nas linhas `{2}` e `{3}`, mas ele é opcional. Em seguida, temos um laço externo (`{2}`) que fará uma iteração pelo array, a partir de sua primeira posição até a última, controlando quantas vezes passaremos por ele (deve haver uma passagem por item do array, pois o número de passagens é igual ao tamanho dele). Depois disso, temos um laço interno (`{3}`), que fará uma iteração pelo array, a partir de sua primeira posição até o penúltimo valor, no qual realmente compararemos o valor atual com o próximo (`{4}`). Se os valores estiverem fora de ordem (isto é, o valor atual é maior que o próximo), faremos a sua troca (`{5}`), o que significa que o valor da posição `j+1` será transferido para a posição `j`, e vice-versa.

Criamos a função `swap` no Capítulo 11, *Heap binário e heap sort*. Somente para recordar, o código dessa função é apresentado a seguir.

```
function swap(array, a, b) {
    /* const temp = array[a];
    array[a] = array[b];
    array[b] = temp; */ // modo clássico
    [array[a], array[b]] = [array[b], array[a]]; // modo ES2015
}
```

O diagrama a seguir mostra o bubble sort em ação.

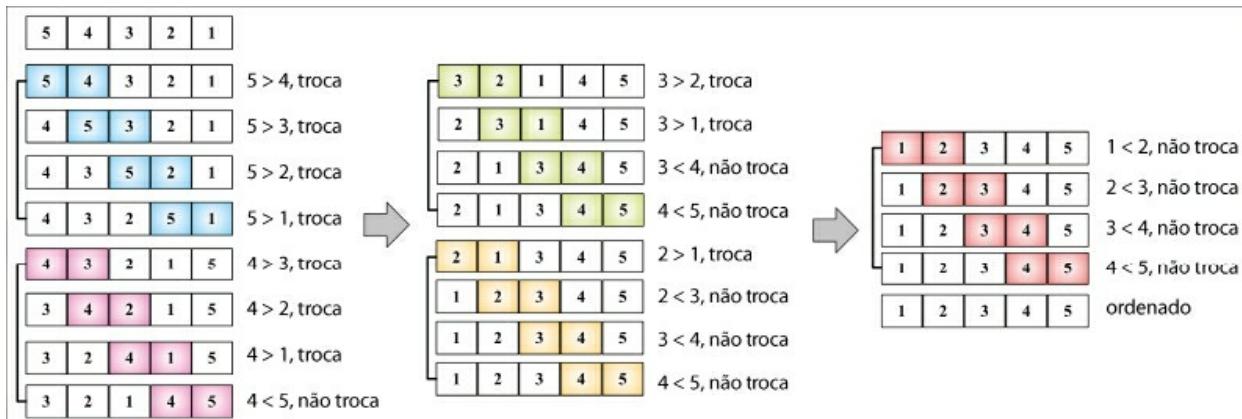


Figura 13.1

Cada seção diferente no diagrama anterior representa uma passagem pelo laço externo (`{2}`), e cada comparação entre dois valores adjacentes é feita pelo laço interno (`{3}`).

Para testar o algoritmo de bubble sort e obter o mesmo resultado mostrado

no diagrama, usaremos o código a seguir.

```
function createNonSortedArray(size) { // 6
  const array = [];
  for (let i = size; i > 0; i--) {
    array.push(i);
  }
  return array;
}
let array = createNonSortedArray(5); // {7}
console.log(array.join()); // {8}
array = bubbleSort(array); // {9}
console.log(array.join()); // {10}
```

Para nos ajudar a testar os algoritmos de ordenação que veremos neste capítulo, implementaremos uma função que criará automaticamente um array não ordenado, cujo tamanho será passado como parâmetro ({6}). Se passarmos 5 como parâmetro, a função criará o seguinte array para nós: [5, 4, 3, 2, 1]. Então, tudo que temos de fazer é chamar essa função e armazenar o seu valor de retorno em uma variável que contenha a instância do array inicializada com alguns números ({7}). Somente para garantir que temos um array não ordenado, exibiremos o seu conteúdo no console ({8}), chamaremos a função de bubble sort ({9}) e exibiremos o conteúdo do array ordenado no console novamente para que possamos verificar se o array foi ordenado ({10}).

Veja mais exemplos e casos de teste no código-fonte baixado a partir da página de suporte (ou do repositório do GitHub em <https://github.com/loiane/javascript-datastructures-algorithms>).

Observe que, quando o algoritmo executar a segunda passagem pelo laço externo (a segunda seção no diagrama anterior), os números 4 e 5 já estarão ordenados. Apesar disso, em comparações subsequentes, continuamos comparando esses números, mesmo que não seja necessário. Por esse motivo, faremos uma pequena melhoria no algoritmo de bubble sort.

Bubble sort melhorado

Se subtrairmos do laço interno o número de passagens pelo laço externo, evitaremos todas as comparações desnecessárias feitas pelo laço interno ({1}):

```

function modifiedBubbleSort(array, compareFn = defaultCompare) {
  const { length } = array;
  for (let i = 0; i < length; i++) {
    for (let j = 0; j < length - 1 - i; j++) { // {1}
      if (compareFn(array[j], array[j + 1]) === Compare.BIGGER_THAN) {
        swap(array, j, j + 1);
      }
    }
  }
  return array;
}

```

O diagrama a seguir exemplifica como o bubble sort melhorado funciona.



Figura 13.2

Observe que não comparamos os números que já estão nos lugares corretos. Apesar de termos feito essa pequena mudança para melhorar um pouco o algoritmo de bubble sort, esse não é um algoritmo recomendado. Sua complexidade é de $O(n^2)$.

Discutiremos melhor a **notação big O** (O grande) no Capítulo 15, *Complexidade de algoritmos*, para saber mais sobre os algoritmos.

Selection sort

O algoritmo de **selection sort** (ordenação por seleção) é um algoritmo de ordenação por comparação in-place. A ideia geral por trás do selection sort é encontrar o valor mínimo na estrutura de dados, colocá-lo na primeira posição e então encontrar o segundo valor mínimo, colocá-lo na segunda posição, e assim sucessivamente.

A seguir, apresentamos o código-fonte do algoritmo de selection sort:

```

function selectionSort(array, compareFn = defaultCompare) {
  const { length } = array; // {1}
  let indexMin;

```

```

for (let i = 0; i < length - 1; i++) { // {2}
    indexMin = i; // {3}
    for (let j = i; j < length; j++) { // {4}
        if (compareFn(array[indexMin], array[j]) === Compare.BIGGER_THAN) { // {5}
            indexMin = j; // {6}
        }
    }
    if (i !== indexMin) { // {7}
        swap(array, i, indexMin);
    }
}
return array;
};

```

Inicialmente declaramos algumas das variáveis que serão usadas no algoritmo ({1}). Em seguida, temos um laço externo ({2}) que fará uma iteração pelo array e controlará as passagens (isto é, qual é o *enésimo* valor do array que devemos encontrar a seguir, ou seja, o próximo valor mínimo). Partiremos do pressuposto de que o primeiro valor da iteração atual é o valor mínimo do array ({3}). Então, partindo do valor de **i** atual até o final do array ({4}), comparamos se o valor na posição **j** é menor que o valor mínimo atual ({5}); se for, mudamos o valor mínimo para o novo valor mínimo ({6}). Ao sair do laço interno ({4}), teremos o enésimo valor mínimo do array. Se o valor mínimo for diferente do valor mínimo original ({7}), faremos a troca desses valores.

Para testar o algoritmo de selection sort, podemos usar o código a seguir.

```

let array = createNonSortedArray(5);
console.log(array.join());
array = selectionSort(array);
console.log(array.join());

```

O diagrama seguinte (Figura 13.3) exemplifica o algoritmo de selection sort em ação com base em nosso array usado no código anterior, isto é, [5, 4, 3, 2, 1].

As setas na parte de baixo do array indicam as posições consideradas no momento para encontrar o valor mínimo (laço interno, em {4}), e cada passo do diagrama anterior representa o laço externo ({2}).

O selection sort também é um algoritmo de complexidade $O(n^2)$. De modo semelhante ao bubble sort, ele contém dois laços aninhados, responsáveis pela complexidade quadrática. No entanto, o desempenho do selection sort é pior que o desempenho do algoritmo de insertion sort, que veremos

a seguir.

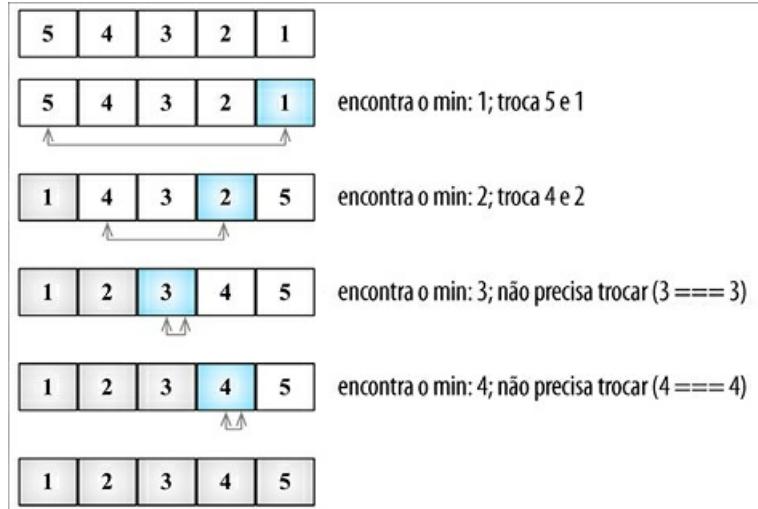


Figura 13.3

Insertion sort

O algoritmo de **insertion sort** (ordenação por inserção) constrói o array ordenado final, um valor de cada vez. Ele pressupõe que o primeiro elemento já está ordenado. Então, uma comparação com o segundo valor é realizada: o segundo valor deve permanecer em seu lugar ou deverá ser inserido antes do primeiro? Os dois primeiros valores serão ordenados; em seguida, a comparação será feita com o terceiro valor (isto é, ele deverá ser inserido na primeira, na segunda ou na terceira posição?), e assim sucessivamente.

O código a seguir representa o algoritmo de insertion sort.

```
function insertionSort(array, compareFn = defaultCompare) {  
    const { length } = array; // {1}  
    let temp;  
    for (let i = 1; i < length; i++) { // {2}  
        let j = i; // {3}  
        temp = array[i]; // {4}  
        while (j > 0 && compareFn(array[j - 1], temp) === Compare.BIGGER_THAN) { // {5}  
            array[j] = array[j - 1]; // {6}  
            j--;  
        }  
        array[j] = temp; // {7}  
    }  
    return array;
```

};

Como sempre, a primeira linha do algoritmo é usada para declarar as variáveis que usaremos no código-fonte (**{1}**). Em seguida, iteramos pelo array a fim de encontrar o lugar correto para o valor de **i** (**{2}**). Observe que começamos na segunda posição (índice **1**), em vez de começar pela posição **0** (pois consideramos que o primeiro valor já está ordenado). Então atribuímos o valor de **i** a uma variável auxiliar (**{3}**) e, além disso, armazenamos o valor da posição **i** do array em uma variável temporária (**{4}**) para que possamos inseri-lo na posição correta depois. O próximo passo é encontrar o local correto para inserir o valor. Enquanto a variável **j** for maior que **0** (porque o primeiro índice do array é **0** e não há índices negativos) e o valor anterior no array for maior que o valor com o qual estamos fazendo a comparação (**{5}**), deslocaremos o valor anterior para a posição atual (**{6}**) e decrementaremos o valor de **j**. No final, inserimos o valor em sua posição correta.

O diagrama a seguir exemplifica o insertion sort em ação.

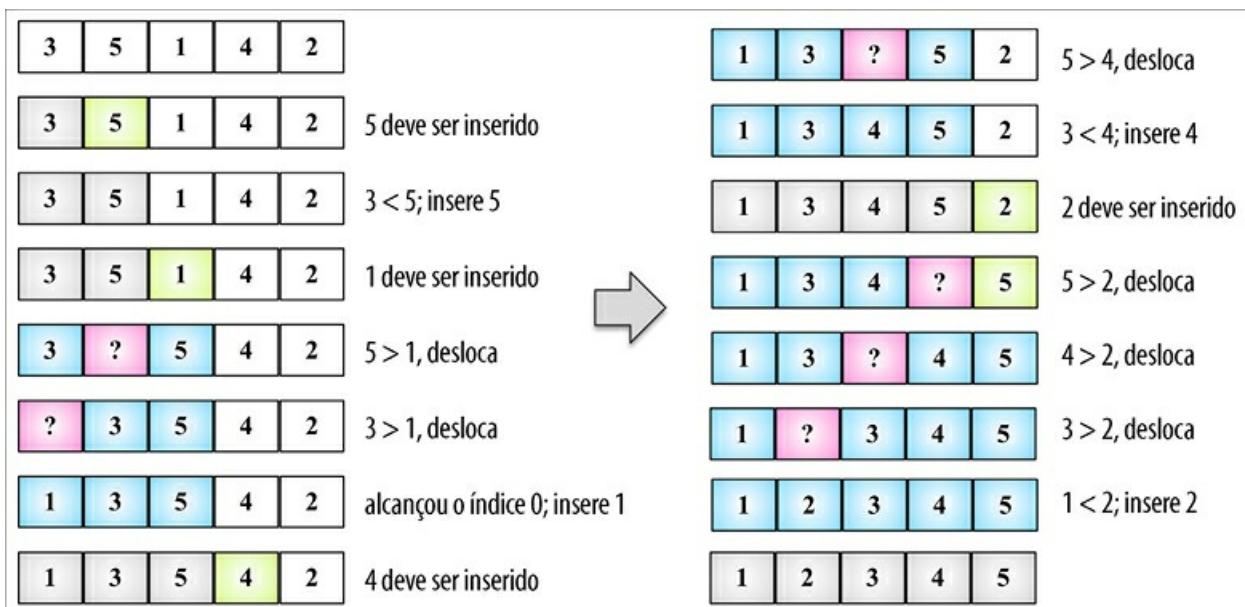


Figura 13.4

Por exemplo, suponha que o array que estamos tentando ordenar seja **[3, 5, 1, 4, 2]**. Esses valores serão tratados nos passos executados pelo algoritmo de insertion sort, conforme descritos a seguir.

1. O valor **3** já está ordenado, portanto começaremos ordenando o segundo valor do array, que é **5**. O valor **3** é menor que **5**, portanto **5**

permanecerá no mesmo lugar (isto é, na segunda posição do array). Os valores **3** e **5** já estão ordenados.

2. O próximo valor a ser ordenado e inserido no local correto é **1** (que, atualmente, está na terceira posição do array). O valor **5** é maior que **1**, portanto **5** é deslocado para a terceira posição. Devemos analisar se **1** deve ser inserido na segunda posição – **1** é maior que **3**? Não é, portanto o valor **3** é deslocado para a segunda posição. Em seguida, devemos verificar se **1** deve ser inserido na primeira posição do array. Como **0** é a primeira posição e não há posições negativas, **1** deve ser inserido na primeira posição. Os valores **1**, **3** e **5** estão ordenados.
3. Passamos para o próximo valor: **4**. O valor **4** deve permanecer na posição atual (índice **3**) ou deverá ser movido para uma posição inferior? O valor **4** é menor que **5**, portanto **5** será deslocado para o índice **3**. Devemos inserir **4** no índice **2**? O valor **4** é maior que **3**, portanto **4** é inserido na posição **3** do array.
4. O próximo valor a ser inserido é **2** (posição **4** do array). O valor **5** é maior que **2**, portanto **5** será deslocado para o índice **4**. O valor **4** é maior que **2**, portanto **4** será deslocado (posição **3**). O valor **3** também é maior que **2**, e **3** também será deslocado. O valor **1** é menor que **2**, portanto **2** será inserido na segunda posição do array. Desse modo, o array estará ordenado.

Esse algoritmo tem um desempenho melhor que os algoritmos de selection e bubble sort na ordenação de arrays pequenos.

Merge sort

O algoritmo de **merge sort** (ordenação por intercalação ou mistura) é o primeiro algoritmo de ordenação que pode ser usado em um cenário do mundo real. Os três primeiros algoritmos de ordenação que conhecemos neste livro não têm um bom desempenho, mas o merge sort tem, apresentando uma complexidade de $O(n \log n)$.

A classe `Array` de JavaScript define uma função `sort` (`Array.prototype.sort`) que pode ser usada para ordenar arrays usando JavaScript (sem que nós mesmos tenhamos de implementar o algoritmo). A ECMAScript não define o algoritmo de ordenação que deve ser usado, portanto

cada navegador pode implementar o seu próprio algoritmo. Por exemplo, o Mozilla Firefox usa o merge sort como a implementação de `Array.prototype.sort`, enquanto o Chrome (engine V8) utiliza uma variação do quick sort (que veremos em seguida).

O merge sort é um algoritmo do tipo “dividir e conquistar”. A ideia por trás dele é dividir o array original em arrays menores até que cada array menor tenha apenas uma posição e, em seguida, combinar esses arrays menores em arrays maiores até que tenhamos um único array grande e ordenado no final.

Por causa da abordagem de dividir e conquistar, o algoritmo de merge sort também é recursivo. Dividiremos o algoritmo em duas funções: a primeira será responsável por dividir o array em arrays menores e chamar a função auxiliar que fará a ordenação. Vamos observar a função principal declarada a seguir.

```
function mergeSort(array, compareFn = defaultCompare) {  
  if (array.length > 1) { // {1}  
    const { length } = array;  
    const middle = Math.floor(length / 2); // {2}  
    const left = mergeSort(array.slice(0, middle), compareFn); // {3}  
    const right = mergeSort(array.slice(middle, length), compareFn); // {4}  
    array = merge(left, right, compareFn); // {5}  
  }  
  return array;  
}
```

O merge sort transformará um array maior em arrays menores até que eles contenham apenas um valor. Como o algoritmo é recursivo, precisamos ter uma condição de parada – que será quando o array tiver um tamanho menor que 1 (**{1}**). Em caso afirmativo, devolveremos o array de tamanho 1 ou o array vazio porque ele já estará ordenado.

Se o tamanho do array for maior que 1, ele será dividido em arrays menores. Para isso, devemos achar primeiro o meio do array (**{2}**); depois de encontrá-lo, dividimos o array em dois arrays menores, que chamaremos de **left** (**{3}**) e **right** (**{4}**). O array **left** é composto de elementos do índice **0** até o índice do meio, e o array **right** é constituído dos elementos do índice do meio até o final do array original. As linhas **{3}** e **{4}** chamarão a própria função **mergeSort** até que os arrays **left** e **right** tenham um tamanho menor ou igual a 1.

Os próximos passos serão chamar a função `merge` ({5}), que será responsável por ordenar os arrays menores e combiná-los em arrays maiores até que tenhamos o array original ordenado e recomposto. A função `merge` é apresentada a seguir.

```
function merge(left, right, compareFn) {  
    let i = 0; // {6}  
    let j = 0;  
    const result = [];  
    while (i < left.length && j < right.length) { // {7}  
        result.push(  
            compareFn(left[i], right[j]) === Compare.LESS_THAN ? left[i++] : right[j++]  
        ); // {8}  
    }  
    return result.concat(i < left.length ? left.slice(i) : right.slice(j)); // {9}  
}
```

A função `merge` recebe dois arrays e os combina formando um array maior. A ordenação ocorre durante o `merge`. Inicialmente, devemos declarar um novo array que será criado para a combinação, e declaramos também duas variáveis ({6}) que serão usadas para iterar pelos dois arrays (os arrays `left` e `right`). Enquanto for possível iterar pelos dois arrays ({7}), comparamos se o valor do array `left` é menor que o valor do array `right`. Se for, adicionamos o valor do array `left` no array `result` combinado e incrementamos também a variável usada para iterar pelo array ({8}); caso contrário, adicionamos o valor do array `right` e incrementamos a variável usada para iterar pelo array.

Em seguida, adicionamos todos os valores restantes do array `left` ({9}) no array com os resultados combinados, e fazemos o mesmo com os valores restantes no array `right`. No final, devolveremos um array combinado.

Se executarmos a função `mergeSort`, ela será executada como na Figura 13.5.

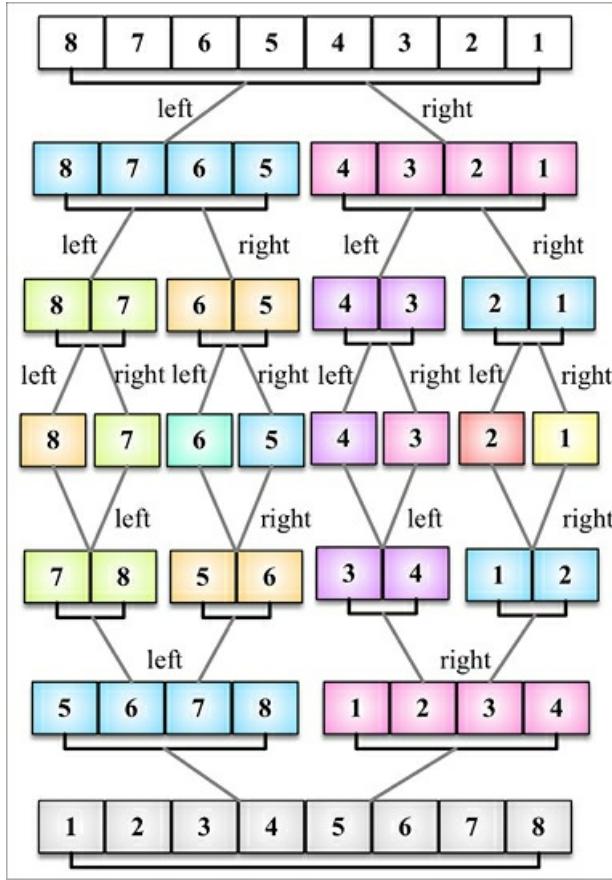


Figura 13.5

Observe que, inicialmente, o algoritmo divide o array original até que ele tenha arrays menores com um único elemento e, então, começa a combiná-los. Enquanto a combinação ocorre, a ordenação também é feita, até termos o array original totalmente recomposto e ordenado.

Quick sort

O **quick sort** (ordenação rápida) provavelmente é o algoritmo de ordenação mais usado. Tem complexidade igual a $O(n \log n)$, e geralmente apresenta um desempenho melhor que outros algoritmos de ordenação de mesma complexidade. De modo semelhante ao merge sort, esse algoritmo também utiliza a abordagem de dividir e conquistar, dividindo o array original em arrays menores (mas sem separá-los como faz o merge sort) para fazer a ordenação.

O algoritmo de quick sort é um pouco mais complexo que os outros algoritmos vistos até agora. Vamos analisá-lo, passo a passo, como mostrado a seguir.

1. Em primeiro lugar, devemos selecionar um valor do array chamado **pivô**, que será o valor no meio do array.
2. Criaremos dois ponteiros (referências) – o ponteiro da esquerda apontará para o primeiro valor do array, enquanto o ponteiro da direita apontará para o último valor do array. Moveremos o ponteiro da esquerda até encontrarmos um valor que seja maior que o pivô, e moveremos também o ponteiro da direita até encontrarmos um valor que seja menor que o pivô, e faremos a troca desses valores. Repetiremos esse processo até que o ponteiro da esquerda ultrapasse o ponteiro da direita. Esse processo contribui para que tenhamos valores menores que a referência do pivô antes dele e valores maiores que o pivô depois de sua referência. Essa operação é chamada de **partição**.
3. Em seguida, o algoritmo repete os dois passos anteriores para arrays menores (subarrays com valores menores e, então, subarrays com valores maiores), até que os arrays estejam totalmente ordenados.

Vamos iniciar a implementação do algoritmo de quick sort usando o código a seguir.

```
function quickSort(array, compareFn = defaultCompare) {
  return quick(array, 0, array.length - 1, compareFn);
};
```

De modo semelhante ao merge sort, começaremos declarando o método principal que chamará a função recursiva, passando o array que queremos ordenar, junto com o índice **0** e a última posição (porque queremos ter o array todo ordenado, e não apenas um subconjunto dele).

Em seguida, declararemos a função **quick**, da seguinte maneira:

```
function quick(array, left, right, compareFn) {
  let index; // {1}
  if (array.length > 1) { // {2}
    index = partition(array, left, right, compareFn); // {3}
    if (left < index - 1) { // {4}
      quick(array, left, index - 1, compareFn); // {5}
    }
    if (index < right) { // {6}
      quick(array, index, right, compareFn); // {7}
    }
  }
  return array;
};
```

Inicialmente declaramos a variável **index** ({1}), que nos ajudará a separar

o subarray em valores menores e maiores para que possamos chamar de novo a função **quick** recursivamente. O valor de **index** será obtido a partir do valor de retorno da função **partition** ({3}).

Se o tamanho do array for maior que 1 (porque um array com um único elemento já estará ordenado – {2}), executaremos a operação **partition** no subarray especificado (o array completo será passado na primeira chamada) a fim de obter **index** ({3}). Se houver um subarray com elementos menores ({4}), repetiremos o processo para esse subarray ({5}). Faremos o mesmo para o subarray com valores maiores. Se houver um subarray com valores maiores ({6}), repetiremos o processo do quick sort ({7}).

Vamos ver como o processo de partição funciona na próxima seção.

Processo de partição

Nossa primeira tarefa deve ser escolher o elemento pivô. Há algumas maneiras de fazer isso. O modo mais simples é selecionar o primeiro valor do array (o item mais à esquerda). No entanto, estudos mostram que essa não é uma boa opção caso o array esteja quase ordenado, resultando no pior comportamento possível do algoritmo. Outra abordagem é selecionar um valor aleatório ou o valor que estiver no meio do array.

Vamos analisar agora o método **partition**:

```
function partition(array, left, right, compareFn) {
  const pivot = array[Math.floor((right + left) / 2)]; // {8}
  let i = left; // {9}
  let j = right; // {10}
  while (i <= j) { // {11}
    while (compareFn(array[i], pivot) === Compare.LESS_THAN) { // {12}
      i++;
    }
    while (compareFn(array[j], pivot) === Compare.BIGGER_THAN) { // {13}
      j--;
    }
    if (i <= j) { // {14}
      swap(array, i, j); // {15}
      i++;
      j--;
    }
  }
  return i; // {16}
```

}

Nessa implementação, selecionamos o item do meio como **pivot** ({8}). Também inicializamos os dois ponteiros: **left** (inferior, na linha {9}) com o primeiro elemento do array e **right** (superior, na linha {10}) com o último elemento do array.

Enquanto os ponteiros **left** e **right** não se cruzarem ({11}), executaremos a operação de partição. Inicialmente, deslocaremos o ponteiro **left** até encontrarmos um elemento que seja maior que **pivot** ({12}). Faremos o mesmo com o ponteiro **right**, isto é, ele será deslocado até encontrarmos um elemento que seja menor que **pivot** ({13}).

Quando o ponteiro **left** for maior que **pivot** e o ponteiro **right** for menor que **pivot**, comparamos se o índice do ponteiro **left** é maior que o índice do ponteiro **right** ({14}), isto é, o valor à esquerda é maior que o valor à direita. Trocamos esses valores ({15}), fazemos os ponteiros serem deslocados e repetimos o processo (começando novamente na linha {11}).

No final da operação de partição, devolvemos o índice do ponteiro esquerdo, que será usado para criar os subarrays na linha {3}.

Quick sort em ação

Vamos ver o algoritmo de quick sort em ação, passo a passo (Figura 13.6).

Dado o array [3, 5, 1, 6, 4, 7, 2], o diagrama anterior representa a primeira execução da operação de partição.

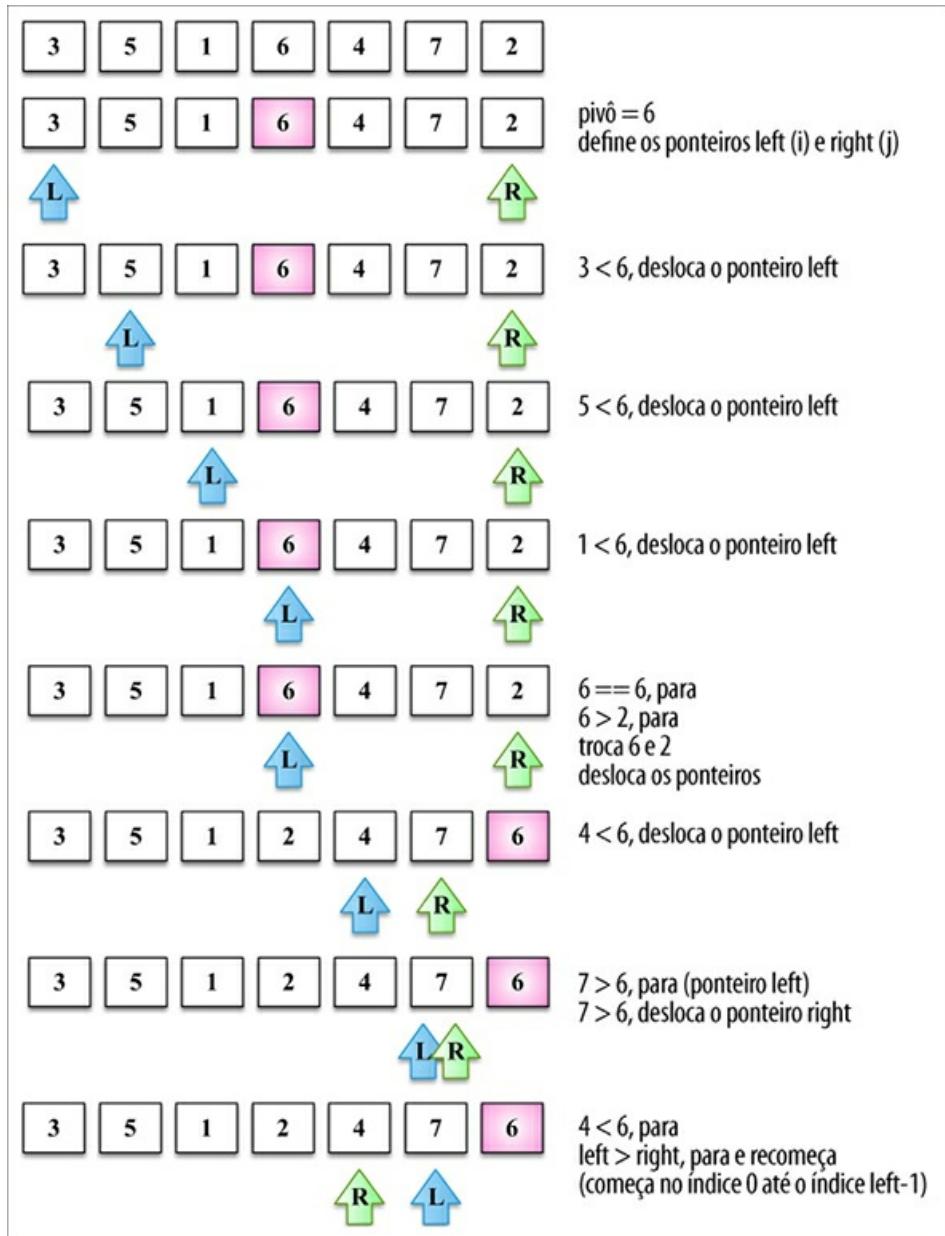


Figura 13.6

A Figura 13.7 exemplifica a execução da operação de partição para o primeiro subarray de valores menores (observe que 7 e 6 não fazem parte do subarray).

Em seguida, continuaremos criando subarrays, como vemos na Figura 13.8, mas agora com valores maiores que o subarray do diagrama anterior (o subarray inferior com valor 1 não precisa ser particionado, pois contém somente um valor).

O subarray inferior [2, 3] do subarray [2, 3, 5, 4] continua a ser

particionado (linha {5} do algoritmo) (Figura 13.9).

Em seguida, o subarray superior [5, 4] do subarray [2, 3, 5, 4] também continua a ser particionado (linha {7} do algoritmo), como vemos na Figura 13.10.

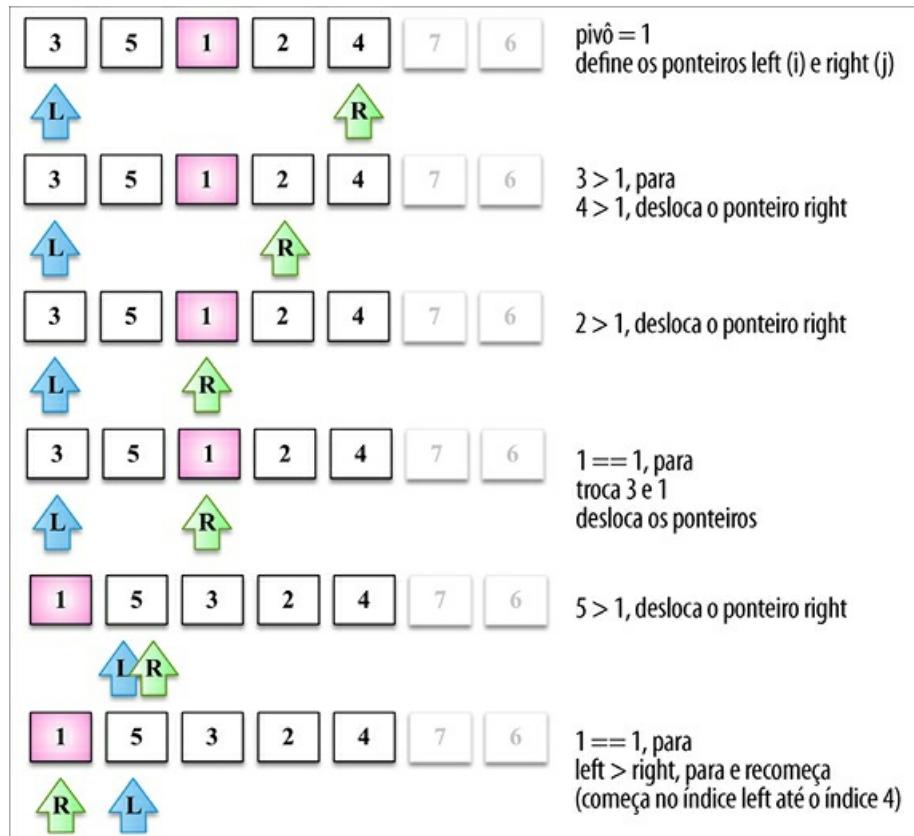


Figura 13.7

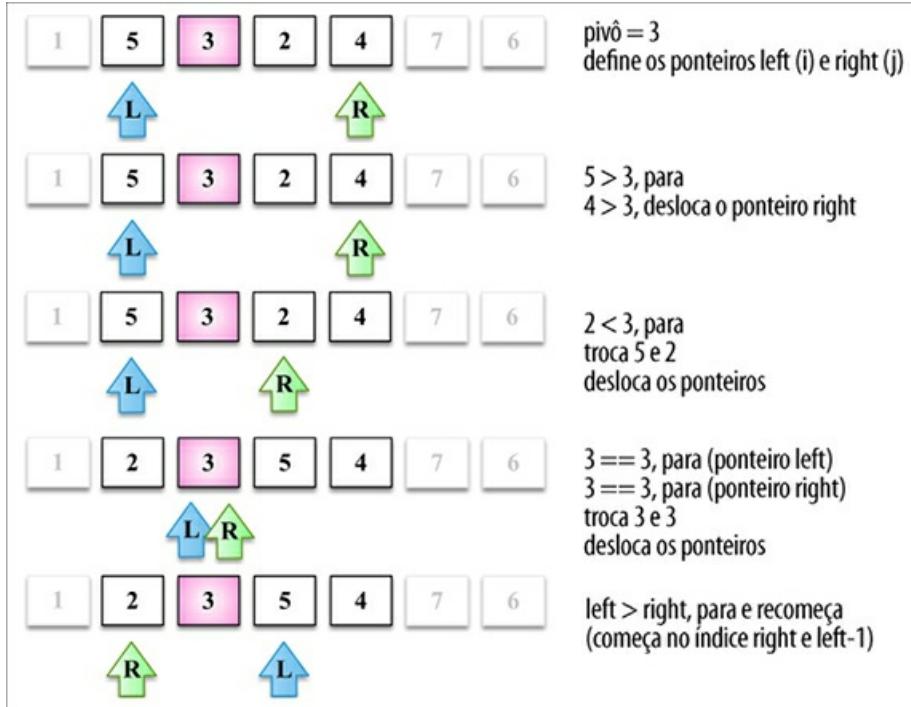


Figura 13.8

No final, o subarray superior [6, 7] também será afetado pela operação de partição, concludo a execução do algoritmo de quick sort.

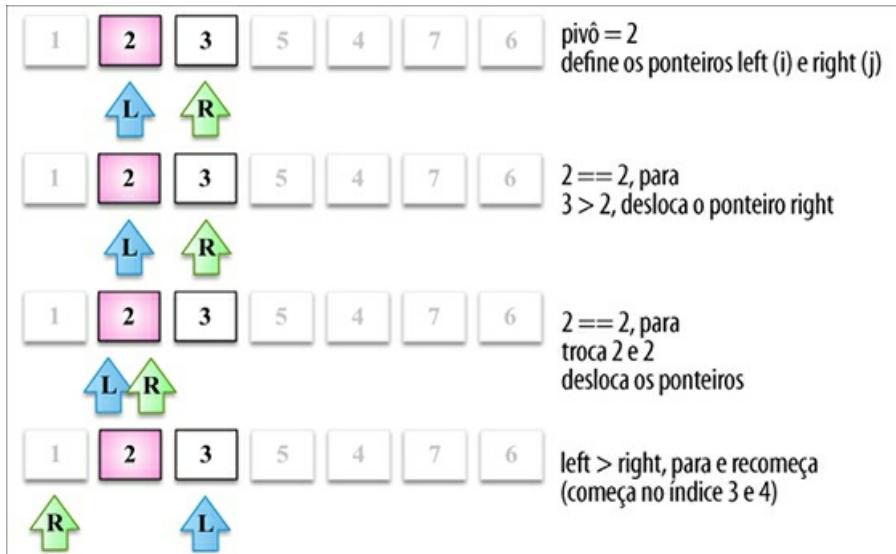


Figura 13.9

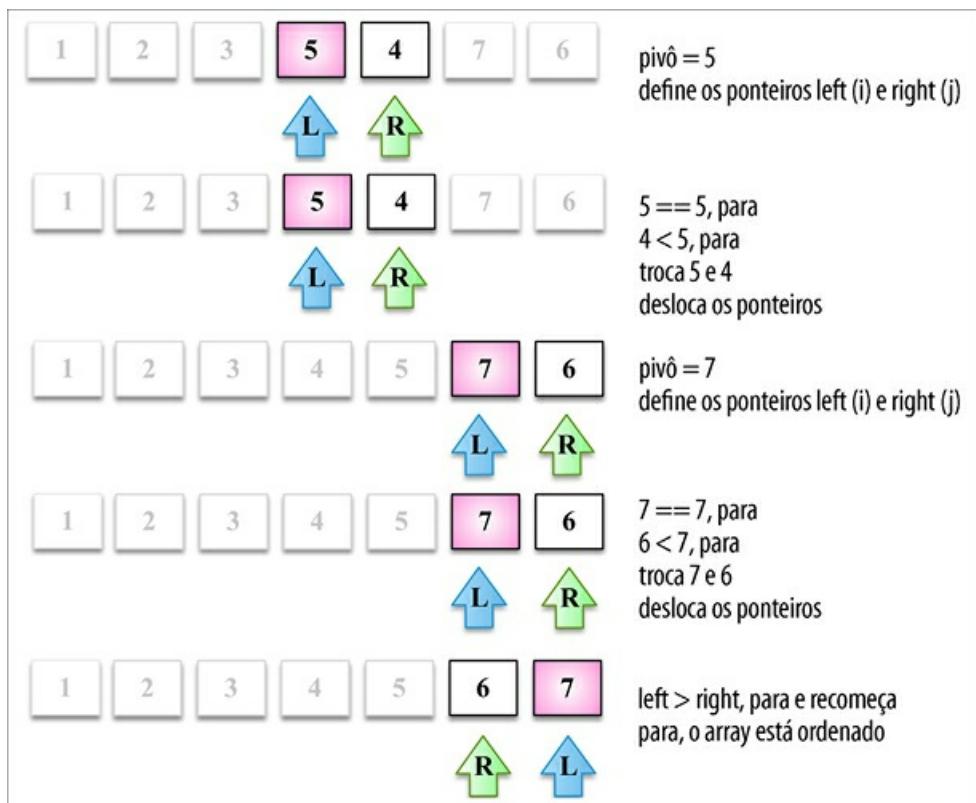


Figura 13.10

Counting sort

O **counting sort** (ordenação por contagem) é o primeiro algoritmo de ordenação com distribuição que conheceremos neste livro. Algoritmos de ordenação com distribuição usam estruturas de dados auxiliares (conhecidas como buckets), que são organizadas e então combinadas, resultando no array ordenado. O counting sort usa um array temporário que armazenará quantas vezes cada elemento aparece no array original. Depois que todos os elementos forem contabilizados, o array temporário será ordenado e uma iteração poderá ser feita nele para construir o array ordenado resultante.

É um bom algoritmo para ordenar inteiros (é um **algoritmo de ordenação de inteiros**) com complexidade $O(n + k)$, em que k é o tamanho do array temporário de contagem; no entanto, mais memória será necessária para o array temporário.

O código a seguir representa o algoritmo de counting sort.

```
function countingSort(array) {
    if (array.length < 2) { // {1}
```

```

        return array;
    }
    const maxValue = findMaxValue(array); // {2}
    const counts = new Array(maxValue + 1); // {3}
    array.forEach(element => {
        if (!counts[element]) { // {4}
            counts[element] = 0;
        }
        counts[element]++; // {5}
    });
    let sortedIndex = 0;
    counts.forEach((count, i) => {
        while (count > 0) { // {6}
            array[sortedIndex++] = i; // {7}
            count--; // {8}
        }
    });
    return array;
}

```

Se o array a ser ordenado não tiver nenhum elemento ou tiver apenas um ({1}), não haverá necessidade de executar o algoritmo de ordenação.

No algoritmo de counting sort, precisamos criar o array de contadores, começando do índice **0** (zero) até o índice do *valor máximo + 1* ({3}). Por esse motivo, temos também de encontrar o valor máximo armazenado no array ({2}). Para encontrar o valor máximo do array, basta iterar por ele e localizar o valor maior, assim:

```

function findMaxValue(array) {
    let max = array[0];
    for (let i = 1; i < array.length; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}

```

Em seguida, iteramos pelas posições do array e incrementamos o contador do elemento no array **counts** ({5}). Somente para garantir que o incremento funcionará, se o array **counts** não foi inicializado com **0** (zero), na primeira vez que contabilizamos o elemento, atribuímos o valor zero para ele também ({4}).

Nesse algoritmo, não estamos usando o laço **for** para iterar pelos arrays

do índice 0 até o seu tamanho. Isso serve para mostrar que, embora o modo clássico de iterar por arrays seja usando o laço `for`, também há outras possibilidades, como usar o método `forEach`, conforme vimos no Capítulo 3, *Arrays*.

Agora que todos os elementos foram contados, iteramos pelo array `counts` e construímos o array ordenado resultante. Como pode haver mais de um elemento com o mesmo valor, o elemento será adicionado tantas vezes quantas ele aparecer no array original. Para isso, decrementamos `count` (`{8}`) até que seu valor seja zero (`{6}`), adicionando o valor (`i`) no array resultante. Por esse motivo, precisamos também de um índice auxiliar (`sortedIndex`) para nos ajudar a atribuir os valores aos seus índices corretos no array ordenado resultante.

Vamos ver o algoritmo de counting sort em ação para nos ajudar a compreender o código anterior.

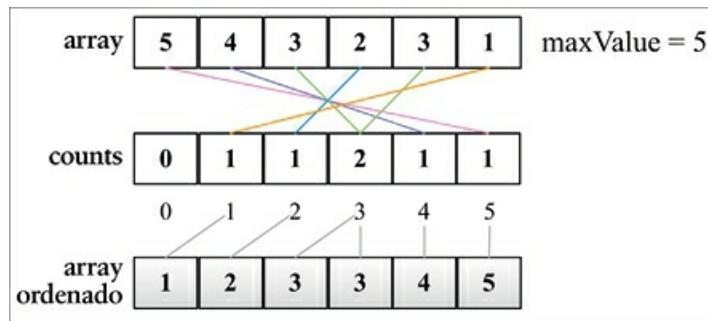


Figura 13.11

Bucket sort

O algoritmo de **bucket sort** (ordenação por balde ou recipiente, também conhecido como bin sort) também é um algoritmo de ordenação com distribuição, que separa os elementos em diferentes *buckets* (arrays menores) e então usa um algoritmo de ordenação mais simples, por exemplo o insertion sort (um bom algoritmo para arrays pequenos) para ordenar cada bucket. Então todos os buckets são combinados no array ordenado resultante.

O código a seguir representa o algoritmo de bucket sort.

```
function bucketSort(array, bucketSize = 5) { // {1}
  if (array.length < 2) {
    return array;
```

```

    }
    const buckets = createBuckets(array, bucketSize); // {2}
    return sortBuckets(buckets); // {3}
}

```

No algoritmo de bucket sort, precisamos especificar quantos buckets serão usados para ordenar os elementos ({1}). Por padrão, usaremos 5 buckets. O algoritmo de bucket sort executará em seu melhor cenário se os elementos puderem ser distribuídos uniformemente entre os buckets. Se os elementos estiverem muito esparsos, será melhor usar mais buckets. Se os elementos estiverem alocados de forma densa, usar menos buckets será melhor. Por esse motivo, permitimos que `bucketSize` seja passado como parâmetro.

Dividiremos o algoritmo em duas partes: a primeira parte consiste em criar os buckets e distribuir os elementos nos diferentes buckets ({2}). Na segunda parte do algoritmo, executamos o algoritmo de insertion sort em cada bucket e adicionamos todos os elementos do bucket no array ordenado resultante ({3}).

Vamos observar o código responsável pela criação dos buckets.

```

function createBuckets(array, bucketSize) {
  let minValue = array[0];
  let maxValue = array[0];
  for (let i = 1; i < array.length; i++) { // {4}
    if (array[i] < minValue) {
      minValue = array[i];
    } else if (array[i] > maxValue) {
      maxValue = array[i];
    }
  }
  const bucketCount = Math.floor((maxValue - minValue) / bucketSize) + 1; // {5}
  const buckets = [];
  for (let i = 0; i < bucketCount; i++) { // {6}
    buckets[i] = [];
  }
  for (let i = 0; i < array.length; i++) { // {7}
    const bucketIndex = Math.floor((array[i] - minValue) / bucketSize); // {8}
    buckets[bucketIndex].push(array[i]);
  }
  return buckets;
}

```

O primeiro passo importante no bucket sort é calcular quantos elementos serão distribuídos em cada bucket ({5}). Para calcular esse número,

usaremos uma fórmula que consiste em calcular a diferença entre os valores maior e menor do array, dividido pelo tamanho do bucket. Nesse caso, também temos de iterar pelo array original e encontrar os valores máximo e mínimo (**{4}**). Poderíamos ter usado a função **findMaxValue** que criamos para o algoritmo de counting sort, e ter criado também uma função **findMinValue**, mas isso significaria iterar pelo mesmo array duas vezes. Portanto, para otimizar essa busca, podemos localizar os dois valores iterando somente uma vez no array.

Depois de calcular o **bucketCount**, devemos inicializar cada bucket (**{6}**). A estrutura de dados de **buckets** é uma matriz (um array multidimensional). Cada posição da variável **buckets** armazenará outro array.

O último passo é distribuir os elementos nos buckets. Devemos iterar pelos elementos do array (**{7}**), calcular em qual bucket colocaremos o elemento (**{8}**) e inserir o elemento no bucket correto. Com esse passo, concluímos a primeira parte do algoritmo.

Vamos analisar a próxima parte do algoritmo de bucket sort, que ordenará cada bucket.

```
function sortBuckets(buckets) {
  const sortedArray = []; // {9}
  for (let i = 0; i < buckets.length; i++) { // {10}
    if (buckets[i] != null) {
      insertionSort(buckets[i]); // {11}
      sortedArray.push(...buckets[i]); // {12}
    }
  }
  return sortedArray;
}
```

Criamos um array que armazenará o array ordenado resultante (**{9}**) – isso significa que o array original não será modificado, pois devolveremos um novo array. Em seguida, iteramos em cada bucket válido e aplicamos o insertion sort (**{11}**) – conforme o cenário, poderíamos aplicar também outros algoritmos de ordenação, por exemplo, o quick sort. Por fim, adicionamos todos os elementos do bucket ordenado no array ordenado (**{12}**).

Observe que, na linha **{12}**, estamos usando o operador de desestruturação introduzido na ES2015, que vimos no Capítulo 2, *Visão*

geral sobre ECMAScript e TypeScript. Uma abordagem clássica seria iterar pelos elementos de `buckets[i]` (`buckets[i][j]`) e adicionar cada elemento separadamente no array ordenado.

O diagrama a seguir mostra o algoritmo de bucket sort em ação.

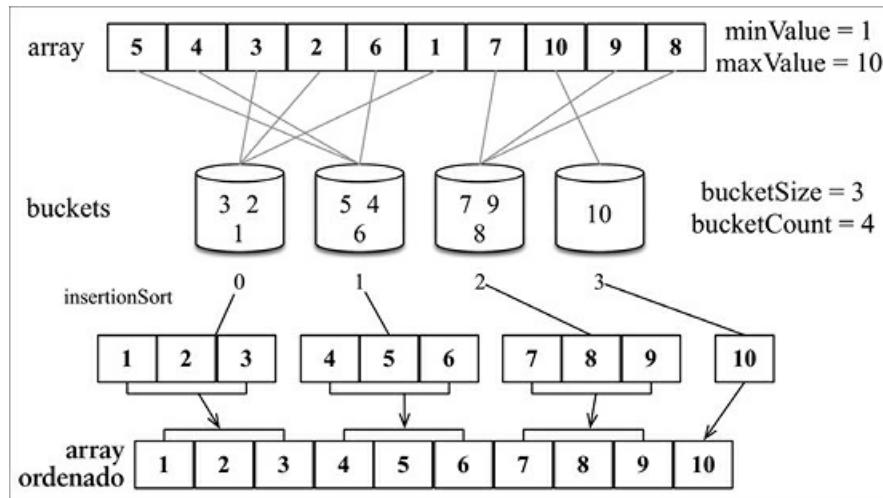


Figura 13.12

Radix sort

O **radix sort** (ordenação por raízes) é também um algoritmo de ordenação por distribuição, que distribui os inteiros em buckets com base no *dígito ou valor significativo* de um número (a raiz – daí o nome ordenação por raízes). A raiz é baseada no sistema numérico dos valores dos arrays.

Por exemplo, para os números do sistema decimal, a raiz (ou base) 10 é usada; assim, o algoritmo usará 10 buckets para distribuir os elementos, e ordenará inicialmente os números com base nos 1s, depois nos 10s, depois nos 100s, e assim por diante.

O código a seguir representa o algoritmo de radix sort.

```
function radixSort(array, radixBase = 10) {
  if (array.length < 2) {
    return array;
  }
  const minValue = findMinValue(array);
  const maxValue = findMaxValue(array);
  let significantDigit = 1; // {1}
  while ((maxValue - minValue) / significantDigit >= 1) { // {2}
    array = countingSortForRadix(array, radixBase, significantDigit, minValue); // {3}
```

```

    significantDigit *= radixBase; // {4}
}
return array;
}

```

Como o radix sort também é usado para ordenar inteiros, começaremos ordenando todos os números com base em seu último dígito ({1}). Também é possível modificar esse algoritmo para que ele funcione com caracteres alfabéticos. Ordenaremos os números com base somente em seu último dígito e, na próxima iteração, ordenaremos com base no segundo dígito significativo (os 10s), depois no terceiro dígito significativo (os 100s), e assim por diante ({4}). Faremos isso até que não haja mais dígitos significativos para ordenar ({2}), e é por isso que precisamos saber quais são os valores mínimo e máximo presentes no array.

Se o array a ser ordenado contiver somente valores entre **1** e **9**, o laço na linha {2} será executado somente uma vez. Se houver valores menores que **99**, o laço executará uma segunda vez, e assim sucessivamente.

Vamos observar o código responsável pela ordenação baseada no dígito significativo (a raiz).

```

function countingSortForRadix(array, radixBase, significantDigit, minValue) {
  let bucketsIndex;
  const buckets = [];
  const aux = [];
  for (let i = 0; i < radixBase; i++) { // {5}
    buckets[i] = 0;
  }
  for (let i = 0; i < array.length; i++) { // {6}
    bucketsIndex = Math.floor(((array[i] - minValue) / significantDigit) %
radixBase); // {7}
    buckets[bucketsIndex]++;
  }
  for (let i = 1; i < radixBase; i++) { // {9}
    buckets[i] += buckets[i - 1];
  }
  for (let i = array.length - 1; i >= 0; i--) { // {10}
    bucketsIndex = Math.floor(((array[i] - minValue) / significantDigit) %
radixBase); // {11}
    aux[-buckets[bucketsIndex]] = array[i]; // {12}
  }
  for (let i = 0; i < array.length; i++) { // {13}
    array[i] = aux[i];
  }
  return array;
}

```

}

Em primeiro lugar, inicializamos os buckets (**{5}**) de acordo com a base da raiz. Como estamos trabalhando com números de base 10, trabalharemos com 10 buckets. Em seguida, fazemos uma ordenação por contagem (**{8}**) com base no dígito significativo dos números (**{7}**) que estão no array (**{6}**). Como estamos executando um counting sort, também devemos calcular os contadores para que tenhamos a contagem correta (**{9}**).

Depois de contar os valores, começamos a movê-los de volta para o array original. Usaremos um array temporário (**aux**) que nos ajudará nessa tarefa. Para cada valor do array original (**{10}**), obteremos novamente o seu dígito significativo (**{11}**) e moveremos o seu valor para o array **aux** (subtraindo o contador do array **buckets** – **{12}**). O último passo é opcional (**{13}**), quando transferirmos todos os valores do array **aux** para o **array** original. Como estamos devolvendo o **array**, poderíamos devolver o array **aux** diretamente, em vez de copiar os seus valores.

Vamos ver o algoritmo de radix sort em ação.

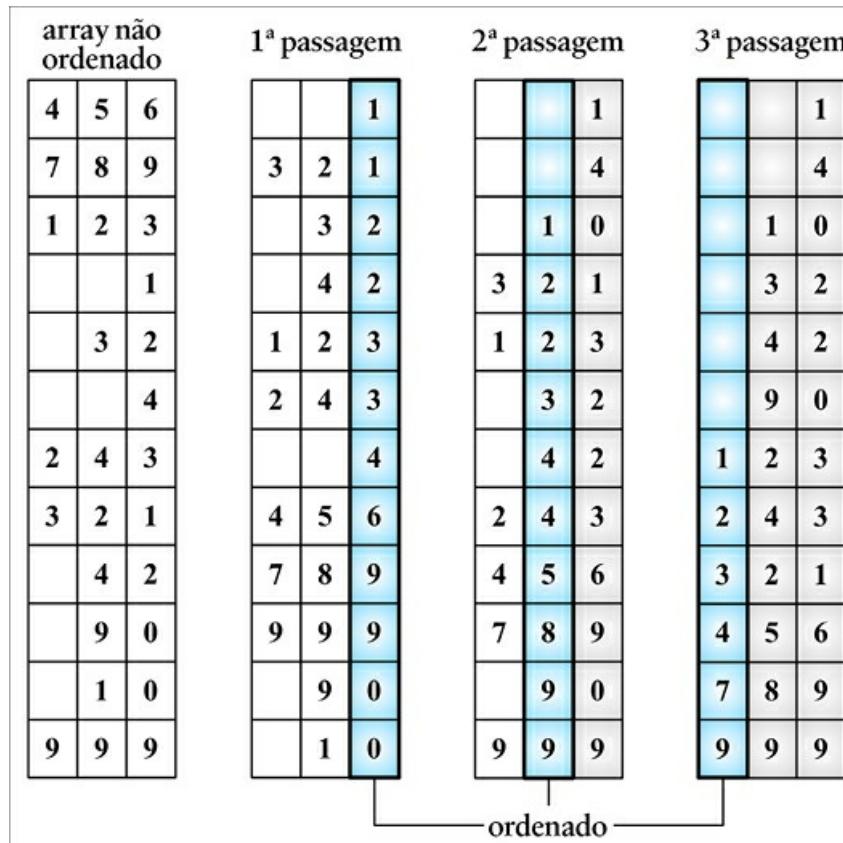


Figura 13.13

Algoritmos de busca

Vamos falar agora sobre os algoritmos de busca. Se observarmos os algoritmos que implementamos nos capítulos anteriores, por exemplo, o método `search` da classe `BinarySearchTree` (Capítulo 10, Árvores) ou o método `indexOf` da classe `LinkedList` (Capítulo 6, *Listas ligadas*), veremos que todos são algoritmos de busca e, é claro, cada um deles foi implementado de acordo com o comportamento de sua estrutura de dados. Portanto, já temos familiaridade com dois algoritmos de busca; só não conhecíamos ainda o seu nome “oficial”!

Busca sequencial

A **busca sequencial** (sequential search) ou **busca linear** (linear search) é o algoritmo de busca mais básico que existe. Ela consiste em comparar cada elemento da estrutura de dados com aquele que estamos procurando. É também o algoritmo mais ineficiente que há.

Vamos ver a sua implementação.

```
const DOES_NOT_EXIST = -1;
function sequentialSearch(array, value, equalsFn = defaultEquals) {
    for (let i = 0; i < array.length; i++) { // {1}
        if (equalsFn(value, array[i])) { // {2}
            return i; // {3}
        }
    }
    return DOES_NOT_EXIST; // {4}
}
```

A busca sequencial itera pelo array ({1}) e compara cada valor com o **value** que estamos procurando ({2}). Se ele for encontrado, podemos devolver alguma informação para sinalizar isso. Podemos devolver o próprio **value**, o valor `true` ou o seu `index` ({3}). Na implementação anterior, devolvemos o `index` de `value`. Se `value` não for encontrado, podemos devolver `-1` ({4}), informando que o `index` não existe; os valores `false` e `null` estão entre as demais opções.

Suponha que temos o array `[5, 4, 3, 2, 1]` e estamos procurando o valor `3`; nesse caso, o diagrama a seguir mostra os passos da busca sequencial.

| | | | | | |
|---|---|---|---|---|---------------------------|
| 5 | 4 | 3 | 2 | 1 | procurando 3 |
| 5 | 4 | 3 | 2 | 1 | 5 === 3? Não, próximo! |
| 5 | 4 | 3 | 2 | 1 | 4 === 3? Não, próximo! |
| 5 | 4 | 3 | 2 | 1 | 3 === 3? Sim, encontrado! |

Figura 13.14

Busca binária

O algoritmo de **busca binária** (binary search) funciona de modo semelhante ao jogo de adivinhação de números, no qual alguém diz: “Estou pensando em um número entre 1 e 100”. Começamos respondendo com um número, e a pessoa dirá “é maior”, “é menor”, ou dirá que acertamos.

Para fazer o algoritmo funcionar, a estrutura de dados deve estar ordenada antes. Eis os passos seguidos pelo algoritmo:

1. Um valor é selecionado no meio do array.
2. Se o valor for aquele que estamos procurando, é sinal de que terminamos (o valor foi encontrado).
3. Se o valor que estamos procurando for menor que o valor selecionado, retornaremos ao passo 1 usando o subarray à esquerda (inferior).
4. Se o valor que estamos procurando for maior que o valor selecionado, retornaremos ao passo 1 usando o subarray à direita (superior).

Vamos analisar a sua implementação.

```
function binarySearch(array, value, compareFn = defaultCompare) {
  const sortedArray = quickSort(array); // {1}
  let low = 0; // {2}
  let high = sortedArray.length - 1; // {3}
  while (lesserOrEquals(low, high, compareFn)) { // {4}
    const mid = Math.floor((low + high) / 2); // {5}
    const element = sortedArray[mid]; // {6}
    if (compareFn(element, value) === Compare.LESS_THAN) { // {7}
      low = mid + 1; // {8}
    } else if (compareFn(element, value) === Compare.BIGGER_THAN) { // {9}
      high = mid - 1; // {10}
    } else {
      return mid; // {11}
    }
  }
}
```

```

        }
    }
    return DOES_NOT_EXIST; // {12}
}

```

Para começar, nossa primeira tarefa deve ser ordenar o array. Podemos usar qualquer algoritmo que implementamos na seção *Algoritmos de ordenação*. O quick sort foi escolhido para essa implementação ({1}). Depois que o array estiver ordenado, definimos os ponteiros **low** ({2}) e **high** ({3}), que atuarão como fronteiras.

Enquanto **low** for menor que **high** ({4}) – nesse caso, se **low** for maior que **high**, é sinal de que o valor não existe, portanto, devolveremos -1 ({12}) –, encontramos o índice do meio ({5}) e, assim, teremos o seu valor ({6}). Então começamos a comparar se o valor selecionado é menor que o **value** que estamos procurando ({7}); nesse caso, devemos verificar os valores maiores ({8}) e recomeçar. Se o valor selecionado for maior que o **value** que estamos procurando ({9}), devemos verificar os valores menores ({10}) e recomeçar. Caso contrário, é sinal de que o valor é igual ao **value** que estamos procurando e, portanto, devolveremos o seu índice ({11}).

A função **lesserOrEquals** usada no código anterior é declarada assim:

```

function lesserOrEquals(a, b, compareFn) {
  const comp = compareFn(a, b);
  return comp === Compare.LESS_THAN || comp === Compare.EQUALS;
}

```

Dado o array do diagrama a seguir, vamos tentar procurar o valor 2. Eis os passos executados pelo algoritmo:

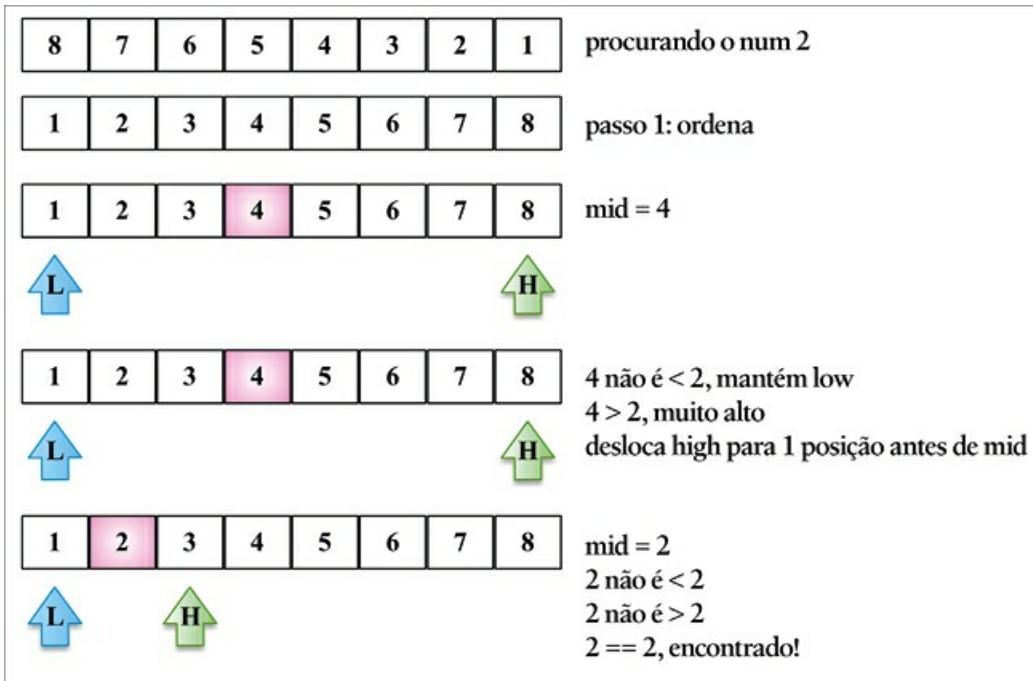


Figura 13.15

A classe `BinarySearchTree` que implementamos no Capítulo 10, Árvores, tem um método de busca exatamente igual à busca binária, porém aplicado às estruturas de dados de árvores.

Busca por interpolação

O algoritmo de **busca por interpolação** (interpolation search) é uma variação melhorada da busca binária. Enquanto a busca binária sempre verifica o valor da posição do meio (`mid`), a busca por interpolação pode verificar lugares diferentes do array, dependendo do valor procurado.

Para fazer o algoritmo funcionar, a estrutura de dados deve ser ordenada antes. Eis os passos seguidos pelo algoritmo:

1. Um valor é selecionado usando a fórmula de `position`.
2. Se o valor for aquele que estamos procurando, é sinal de que terminamos (`value` foi encontrado).
3. Se o valor (`value`) que estamos procurando for menor que o valor selecionado, retornaremos ao passo 1 usando o subarray à esquerda (inferior).
4. Se o valor (`value`) que estamos procurando for maior que o valor

selecionado, retornaremos ao passo 1 usando o subarray à direita (superior).

Vamos analisar a sua implementação.

```
function interpolationSearch(array, value,
  compareFn = defaultCompare,
  equalsFn = defaultEquals,
  diffFn = defaultDiff
) {
  const { length } = array;
  let low = 0;
  let high = length - 1;
  let position = -1;
  let delta = -1;
  while (
    low <= high &&
    biggerOrEquals(value, array[low], compareFn) &&
    lesserOrEquals(value, array[high], compareFn)
  ) {
    delta = diffFn(value, array[low]) / diffFn(array[high], array[low]); // {1}
    position = low + Math.floor((high - low) * delta); // {2}
    if (equalsFn(array[position], value)) { // {3}
      return position;
    }
    if (compareFn(array[position], value) === Compare.LESS_THAN) { // {4}
      low = position + 1;
    } else {
      high = position - 1;
    }
  }
  return DOES_NOT_EXIST;
}
```

Nossa primeira tarefa é calcular a **position** ({2}) com a qual o valor será comparado. A ideia da fórmula é encontrar o maior valor próximo a **position** se o valor procurado estiver mais próximo de **array[high]** e o menor valor próximo a **position** se o valor procurado estiver mais próximo de **array[low]**. Esse algoritmo funcionará melhor se as instâncias dos valores estiverem uniformemente distribuídas no array (**delta** será bem baixo se essas instâncias estiverem uniformemente distribuídas ({1})).

Se **value** for encontrado, devolveremos o seu índice ({3}). Caso contrário, se o valor na posição atual for menor que o **value** procurado, repetiremos a lógica usando o subarray à direita; caso contrário, usaremos o subarray à esquerda ({4}).

As funções `lesserEquals` e `biggerEquals` estão apresentadas a seguir.

```
function lesserEquals(a, b, compareFn) {  
    const comp = compareFn(a, b);  
    return comp === Compare.LESS_THAN || comp === Compare.EQUALS;  
}  
function biggerEquals(a, b, compareFn) {  
    const comp = compareFn(a, b);  
    return comp === Compare.BIGGER_THAN || comp === Compare.EQUALS;  
}
```

O diagrama seguinte mostra o algoritmo em ação – o array está uniformemente distribuído (delta/diferença entre os números é bem pequena).

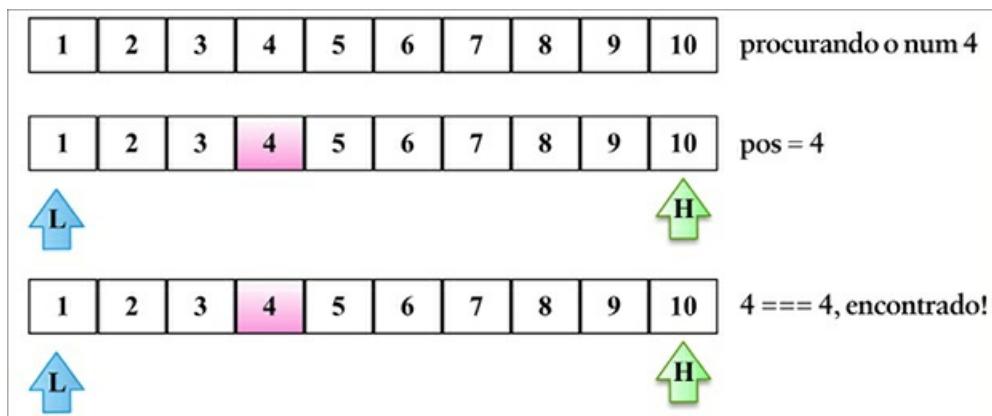


Figura 13.16

Algoritmos de embaralhamento

Neste capítulo, vimos como ordenar um array para organizar todos os seus elementos, e como procurar elementos depois que o array estiver ordenado. Contudo, há também cenários em que precisaremos embaralhar os valores de um array. Um cenário comum na vida real é aquele em que embaralhamos as cartas de um baralho.

Na próxima seção, conheceremos o mais famoso algoritmo usado para embaralhar arrays.

Algoritmo de embaralhamento de Fisher-Yates

Esse algoritmo foi criado por **Fisher** e **Yates** e popularizado por *Donald E. Knuth* na série de livros *The Art of Computer Programming*.

Consiste em iterar pelas posições do array, começando pela última posição e trocando a posição atual por uma posição aleatória. A posição aleatória é menor que a posição atual; desse modo, o algoritmo garante que as posições já embaralhadas não serão embaralhadas novamente (quanto mais embaralhamos um jogo de baralho, pior será o embaralhamento).

O código a seguir apresenta o algoritmo de embaralhamento de Fisher-Yates.

```
function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
    const randomIndex = Math.floor(Math.random() * (i + 1));
    swap(array, i, randomIndex);
  }
  return array;
}
```

Na Figura 13.17 podemos ver o algoritmo em ação.

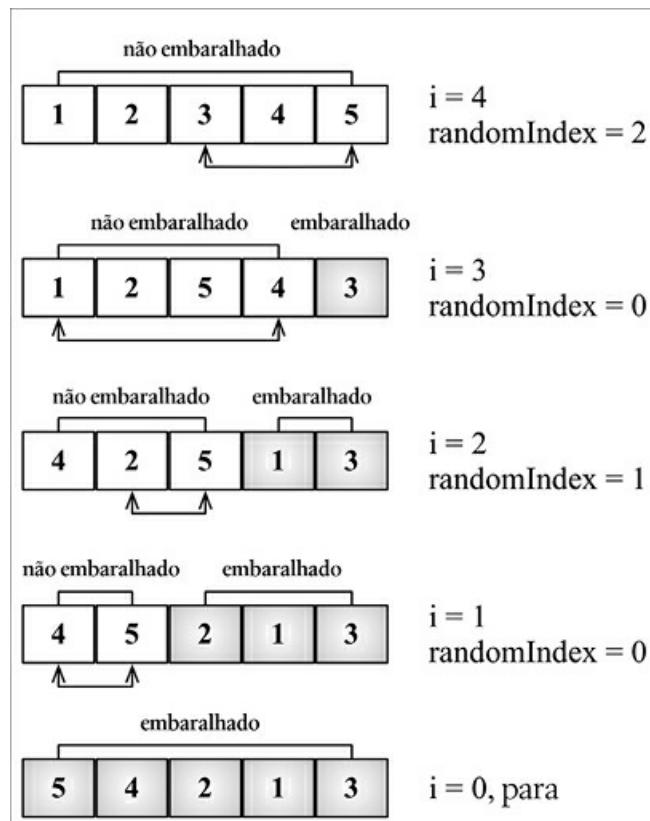


Figura 13.17

Resumo

Neste capítulo, conhecemos os algoritmos de ordenação, busca e embaralhamento.

Vimos os algoritmos de bubble, selection, insertion, merge, quick, counting, bucket e radix sort, usados para ordenar estruturas de dados. Também conhecemos a busca sequencial, a busca por interpolação e a busca binária (que exigem que a estrutura de dados já esteja ordenada). Além disso, descrevemos como embaralhar os valores em um array.

No próximo capítulo, conheceremos algumas técnicas avançadas usadas em algoritmos.

CAPÍTULO 14

Designs de algoritmos e técnicas

Até agora, nós nos divertimos discutindo como implementar várias estruturas de dados diferentes e os algoritmos mais usados de ordenação e de busca. No mundo da programação, os algoritmos são muito interessantes. O aspecto mais interessante acerca deles (e da lógica de programação) é o fato de haver mais de uma abordagem para resolver um problema. Conforme vimos em capítulos anteriores, podemos conceber uma solução usando a abordagem iterativa ou podemos deixar o código mais legível com a recursão. Há também outras técnicas que podemos usar para solucionar problemas com algoritmos. Neste capítulo, conhceremos algumas técnicas diferentes e discutiremos também os próximos passos caso você esteja interessado em explorar esse mundo com mais detalhes.

Neste capítulo, abordaremos o seguinte:

- algoritmos com a abordagem de dividir e conquistar;
- programação dinâmica;
- algoritmos gulosos (greedy);
- algoritmos de backtracking;
- problemas famosos com algoritmos.

Dividir e conquistar

No Capítulo 13, *Algoritmos de ordenação e de busca*, vimos como desenvolver os algoritmos de merge e quick sort. O que as duas soluções de ordenação têm em comum é o fato de serem algoritmos que usam a abordagem de dividir e conquistar, uma das abordagens para o design de algoritmos. O problema é dividido em subproblemas menores, semelhantes ao problema original; os subproblemas são resolvidos recursivamente e as soluções dos subproblemas são combinadas a fim de solucionar o problema original.

O algoritmo de dividir e conquistar pode ser dividido em três partes:

1. **Dividir** o problema original em subproblemas menores (instâncias menores do problema original).
2. **Conquistar** os subproblemas menores resolvendo-os com algoritmos recursivos que devolvem a solução dos subproblemas. O caso de base do algoritmo recursivo resolve e devolve a solução do subproblema menor.
3. **Combinar** as soluções dos subproblemas na solução do problema original.

Como já discutimos os dois algoritmos mais famosos que usam a abordagem de dividir e conquistar no Capítulo 13, *Algoritmos de ordenação e de busca*, veremos como implementar a **busca binária** usando essa abordagem.

Busca binária

No Capítulo 13, *Algoritmos de ordenação e de busca*, vimos como implementar a busca binária usando uma abordagem iterativa. Se observarmos o algoritmo novamente, veremos que é possível usar a abordagem de dividir e conquistar para implementar a busca binária também. Eis a sua lógica:

- **Dividir**: calcular `mid` e procurar o valor na metade inferior ou superior do array.
- **Conquistar**: procurar o valor na metade inferior ou superior do array.
- **Combinar**: não é aplicável, pois estamos devolvendo o índice diretamente.

O algoritmo de busca binária usando a abordagem de dividir e conquistar é apresentado a seguir.

```
function binarySearchRecursive(
  array, value, low, high, compareFn = defaultCompare
) {
  if (low <= high) {
    const mid = Math.floor((low + high) / 2);
    const element = array[mid];
    if (compareFn(element, value) === Compare.LESS_THAN) { // {1}
      return binarySearchRecursive(array, value, mid + 1, high, compareFn);
    } else if (compareFn(element, value) === Compare.BIGGER_THAN) { // {2}
      return binarySearchRecursive(array, value, low, mid - 1, compareFn);
    } else {
      return mid; // {3}
    }
  }
}
```

```

        }
    }
    return DOES_NOT_EXIST; // {4}
}

export function binarySearch(array, value, compareFn = defaultCompare) {
    const sortedArray = quickSort(array);
    const low = 0;
    const high = sortedArray.length - 1;
    return binarySearchRecursive(array, value, low, high, compareFn);
}

```

No algoritmo anterior, temos duas funções: `binarySearch` e `binarySearchRecursive`. A função `binarySearch` é exposta ao desenvolvedor para efetuar a busca. A função `binarySearchRecursive` implementa o algoritmo de dividir e conquistar. Começamos passando o parâmetro `low` como `0` e o parâmetro `high` como `sortedArray.length - 1` para fazer a busca no array ordenado completo. Depois de calcular qual é o índice do elemento `mid`, verificamos se esse elemento é menor que o valor que estamos procurando. Se for menor ({1}) ou maior ({2}), chamamos a função `binarySearchRecursive` novamente, mas, dessa vez, executamos a busca somente no subarray, adaptando os parâmetros `low` ou `high` (em vez de mover o ponteiro como no algoritmo clássico que vimos no Capítulo 13, *Algoritmos de ordenação e de busca*). Se não for menor nem maior, é sinal de que encontramos o valor ({3}), e esse é um dos casos de base. A situação em que `low` é maior que `high` também é outro caso de base, e significa que o algoritmo não encontrou o valor ({4}).

O diagrama a seguir exemplifica o algoritmo em ação.

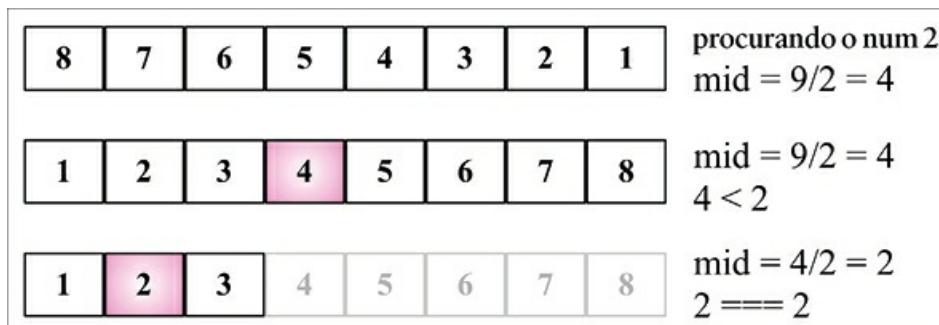


Figura 14.1

Programação dinâmica

A **programação dinâmica** (PD) é uma técnica de otimização usada para

resolver problemas complexos dividindo-os em subproblemas menores.

Observe que a abordagem da programação dinâmica é diferente da abordagem de dividir e conquistar. Enquanto a abordagem de dividir e conquistar divide o problema em subproblemas independentes e então combina as soluções, a programação dinâmica divide o problema em subproblemas dependentes.

Um exemplo de um algoritmo de programação dinâmica é o problema de Fibonacci que resolvemos no Capítulo 9, *Recursão*. Dividimos o problema de Fibonacci em problemas menores.

Há três passos importantes que devemos seguir ao resolver problemas usando a PD:

1. Definir os subproblemas.
2. Implementar a recorrência que resolve os subproblemas (nesse passo, devemos seguir os passos para a recursão, discutidos na seção anterior).
3. Reconhecer e resolver os casos de base.

Há alguns problemas famosos que podem ser solucionados com a programação dinâmica:

- **O problema da mochila:** nesse problema, dado um conjunto de itens, cada um com um valor e um volume, o objetivo é determinar a melhor coleção de itens do conjunto de modo a maximizar o valor total. A restrição para o problema é que o volume total deve ser o volume que a *mochila* é capaz de suportar, ou menor.
- **A maior subsequência comum:** esse problema consiste em encontrar a maior subsequência (uma sequência que pode ser derivada de outra sequência apagando alguns elementos, sem alterar a ordem dos elementos remanescentes) comum a todas as sequências em um conjunto de sequências.
- **Multiplicação de cadeia de matrizes:** nesse problema, dada uma sequência de matrizes, o objetivo é descobrir a maneira mais eficiente de multiplicá-las (com o menor número possível de operações). A multiplicação não é realizada; a solução consiste em encontrar as sequências em que as matrizes devem ser multiplicadas.
- **Troco em moedas:** esse problema consiste em descobrir de quantas maneiras diferentes podemos dar um troco para um dado número de

centavos usando uma determinada quantidade de denominações definidas ($d_1 \dots d_n$).

- **Caminhos mais curtos entre todos os pares em um grafo:** consiste em descobrir o caminho mais curto do vértice u ao vértice v para todos os pares de vértices (u, v). Vimos esse problema no Capítulo 12, *Grafos*, usando o algoritmo de **Floyd-Warshall**.

Discutiremos esses problemas nas próximas seções.

Esses problemas e suas soluções são muito comuns em entrevistas de emprego para programação em empresas de grande porte, como Google, Amazon, Microsoft e Oracle.

Problema do número mínimo de moedas para troco

O problema do número mínimo de moedas para troco é uma variação do **problema do troco em moedas**. O problema do troco em moedas consiste em descobrir de quantas maneiras podemos dar um troco para um dado número de centavos usando uma determinada quantidade de denominações definidas ($d_1 \dots d_n$). O problema do número mínimo de moedas para troco consiste em descobrir o número mínimo de moedas necessário para compor determinado número de centavos usando uma dada quantidade de denominações definidas ($d_1 \dots d_n$).

Por exemplo, os Estados Unidos têm as seguintes denominações (moedas): $d_1 = 1$, $d_2 = 5$, $d_3 = 10$ e $d_4 = 25$.

Se precisarmos de um troco de 36 centavos, podemos usar uma moeda de 25 centavos, uma moeda de 10 centavos e uma moeda de 1 centavo.

Como podemos transformar essa solução em um algoritmo?

A solução para o problema do número mínimo de moedas para troco consiste em encontrar o número mínimo de moedas para n . Porém, para isso, devemos inicialmente encontrar a solução para todo $x < n$. Então podemos compor a solução a partir das soluções aplicadas aos valores menores.

Vamos observar o algoritmo:

```
function minCoinChange(coins, amount) {  
    const cache = [];  
    const makeChange = (value) => {  
        // ...  
    }  
    // ...  
}
```

```

if (!value) { // {3}
    return [];
}
if (cache[value]) { // {4}
    return cache[value];
}
let min = [];
let newMin;
let newAmount;
for (let i = 0; i < coins.length; i++) { // {5}
    const coin = coins[i];
    newAmount = value - coin; // {6}
    if (newAmount >= 0) {
        newMin = makeChange(newAmount); // {7}
    }
    if (
        newAmount >= 0 && // {8}
        (newMin.length < min.length - 1 || !min.length) && // {9}
        (newMin.length || !newAmount) // {10}
    ) {
        min = [coin].concat(newMin); // {11}
        console.log('new Min ' + min + ' for ' + amount);
    }
}
return (cache[value] = min); // {12}
};
return makeChange(amount); // {13}
}

```

A função `minCoinChange` recebe o parâmetro `coins`, que representa as denominações para o nosso problema. No sistema norte-americano de moedas, será `[1, 5, 10, 25]`. Podemos passar a denominação que desejarmos. Além disso, para ser mais eficiente e evitar ter de recalcular os valores, manteremos um `cache` ({1}) – essa técnica se chama **memoização**).

Temos então a função `makeChange` dentro da função `minCoinChange` ({2}), que também é recursiva e é a função que resolverá o problema para nós. A função `makeChange` é chamada na linha {13} com o `amount` passado como parâmetro para a função `minCoinChange`. Como `makeChange` é uma função interna, ela também tem acesso à variável `cache`.

Vamos agora abordar a lógica principal do algoritmo. Inicialmente, se `amount` não for positivo (`< 0`), devolveremos um array vazio ({3}); no final da execução desse método, devolveremos um array com a quantidade de

cada moeda que poderá ser usada para compor o troco (a quantidade mínima de moedas). Em seguida, verificaremos o `cache`. Se o resultado já estiver no cache ({4}), simplesmente devolveremos o seu `value`; caso contrário, executaremos o algoritmo.

Para nos ajudar mais ainda, resolveremos o problema com base no parâmetro `coins` (denominações). Portanto, para cada `coin` ({5}), calcularemos `newAmount` ({6}), que decrementará `value` até alcançarmos a quantidade mínima de troco que podemos dar (lembre-se de que esse algoritmo calculará todos os resultados de `makeChange` para $x < amount$). Se `newAmount` for um valor válido (valor positivo), calcularemos também o resultado para ele ({7}).

No final, verificaremos se `newAmount` é válido, se `newMin` (a quantidade mínima de moedas) é o melhor resultado e se `newMin` e `newAmount` são valores válidos ({10}). Se todas as verificações forem positivas, é sinal de que temos um resultado melhor que o anterior ({11}; por exemplo, para 5 centavos, podemos dar 5 moedas de 1 centavo ou 1 moeda de 5 – 1 moeda de 5 é a melhor solução). Por fim, devolvemos o resultado definitivo ({12}).

Vamos testar esse algoritmo com o código a seguir.

```
console.log(minCoinChange([1, 5, 10, 25], 36));
```

Observe que, se inspecionarmos a variável `cache`, ela armazenará todos os resultados de 1 a 36 centavos. O resultado para o código anterior será [1, 10, 25].

No código-fonte deste livro, você verá algumas linhas de código extras que exibirão os passos desse algoritmo. Por exemplo, se usarmos as denominações 1, 3 e 4 e o algoritmo for executado para a quantidade 6, o resultado a seguir será gerado.

```
new Min 1 for 1
new Min 1,1 for 2
new Min 1,1,1 for 3
new Min 3 for 3
new Min 1,3 for 4
new Min 4 for 4
new Min 1,4 for 5
new Min 1,1,4 for 6
new Min 3,3 for 6
[3, 3]
```

Assim, para uma quantidade igual a 6, a melhor solução é dar duas moedas de valor 3.

Problema da mochila

O problema da mochila é um problema de otimização combinatória. Ele pode ser descrito assim: dada uma mochila de tamanho fixo com capacidade para carregar um peso W e um conjunto de itens que têm um valor e um peso, encontre a melhor solução de modo a encher a mochila com os itens mais valiosos e que o peso total seja menor ou igual a W .

Eis um exemplo:

| Num. item | Peso | Valor |
|-----------|------|-------|
| 1 | 2 | 3 |
| 2 | 3 | 4 |
| 3 | 4 | 5 |

Considere que a mochila tenha capacidade para carregar um peso máximo igual a 5. Nesse exemplo, podemos dizer que a melhor solução seria carregar a mochila com os itens 1 e 2 que, em conjunto, pesam 5 e têm um valor total igual a 7.

Há duas versões desse problema: a versão 0-1, em que podemos carregar a mochila apenas com itens inteiros, e o problema fracionário da mochila, em que podemos usar frações dos itens. Nesse exemplo, trabalharemos com a versão 0-1 do problema. A versão fracionária não pode ser resolvida com programação dinâmica, mas podemos usar um algoritmo guloso (greedy algorithm), que veremos mais adiante neste capítulo.

Vamos observar o algoritmo da mochila a seguir.

```
function knapSack(capacity, weights, values, n) {
    const kS = [];
    for (let i = 0; i <= n; i++) { // {1}
        kS[i] = [];
    }
    for (let i = 0; i <= n; i++) {
        for (let w = 0; w <= capacity; w++) {
            if (i === 0 || w === 0) { // {2}
                kS[i][w] = 0;
            } else if (weights[i - 1] <= w) { // {3}
                kS[i][w] = Math.max(kS[i - 1][w], kS[i - 1][w - weights[i - 1]] + values[i - 1]);
            } else {
                kS[i][w] = kS[i - 1][w];
            }
        }
    }
}
```

```

        const a = values[i - 1] + kS[i - 1][w - weights[i - 1]];
        const b = kS[i - 1][w];
        kS[i][w] = a > b ? a : b; // {4} max(a,b)
    } else {
        kS[i][w] = kS[i - 1][w]; // {5}
    }
}
}
findValues(n, capacity, kS, weights, values); // {6} código adicional
return kS[n][capacity]; // {7}
}

```

Vamos analisar o funcionamento desse algoritmo.

Em primeiro lugar, inicializamos a matriz que será usada para encontrar a solução ({1}). Essa matriz é `kS[n+1][capacity+1]`. Em seguida, ignoramos a primeira coluna e linha da matriz para que possamos trabalhar apenas com índices diferentes de 0 ({2}) e fazemos uma iteração pelos itens disponíveis no array. O item `i` só poderá fazer parte da solução se o seu peso for menor que a limitação (`capacity`, em {3}); caso contrário, o peso total será maior que `capacity`, e isso não pode acontecer. Se isso ocorrer, basta ignorar o seu valor e usar o valor anterior ({5}). Quando descobrimos que um item pode fazer parte da solução, escolhemos aquele que tem o valor máximo ({4}). A solução poderá ser encontrada na última célula da tabela bidimensional, que está no canto inferior direito da tabela ({7}).

Podemos testar o algoritmo a seguir usando o nosso exemplo inicial.

```

const values = [3,4,5],
weights = [2,3,4],
capacity = 5,
n = values.length;
console.log(knapSack(capacity, weights, values, n)); //exibe 7

```

O diagrama a seguir (Figura 14.2) exemplifica a construção da matriz `kS` de nosso exemplo.

Observe que esse algoritmo exibe apenas o valor máximo que pode ser carregado na mochila, mas não os itens propriamente ditos. Podemos adicionar a função extra a seguir para encontrar os itens que fazem parte da solução.

```

function findValues(n, capacity, kS, weights, values) {
    let i = n;
    let k = capacity;

```

```

console.log('Items that are part of the solution:');
while (i > 0 && k > 0) {
  if (kS[i][k] !== kS[i - 1][k]) {
    console.log(`item ${i} can be part of solution w,v: ${weights[i - 1]} , ${values[i - 1]}`);
    i--;
    k -= kS[i][k];
  } else {
    i--;
  }
}
}

```

Podemos chamar essa função imediatamente antes da linha {7} na função `knapSack` ({6}). Se o algoritmo completo for executado, veremos a seguinte saída:

```

Items that are part of the solution:
item 2 can be part of solution w,v: 3,4
item 1 can be part of solution w,v: 2,3
Total value that can be carried: 7

```

O problema da mochila também pode ser implementado recursivamente. A versão recursiva pode ser vista no pacote de código-fonte que acompanha este livro.

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|-----|---|---|---|---|---|---|--|----------|-----|---|---|---|---|---|---|--|----------|-----|---|---|---|---|---|---|---|----------|-----|---|---|---|---|---|---|--|
| 1 | i/w | 0 | 1 | 2 | 3 | 4 | 5 | | 2 | i/w | 0 | 1 | 2 | 3 | 4 | 5 | | 3 | i/w | 0 | 1 | 2 | 3 | 4 | 5 | | 4 | i/w | 0 | 1 | 2 | 3 | 4 | 5 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | |
| | 1 | 0 | | | | | | | 1 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | | 1 | 0 | 0 | 3 | 3 | 4 | 7 | | 1 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | | |
| | 2 | 0 | | | | | | | 2 | 0 | | | | | | | | 2 | 0 | | | | | | | | 2 | 0 | 0 | 3 | 4 | 4 | 7 | | |
| | 3 | 0 | | | | | | | 3 | 0 | | | | | | | | 3 | 0 | | | | | | | | 3 | 0 | 0 | 3 | 4 | 5 | 7 | | |

Itens:
1: (2,3)
2: (3,4)
3: (4,5)

Figura 14.2

Maior subsequência comum

Outro problema de PD usado com muita frequência em problemas de

desafios de programação é o **LCS** (**Longest Common Subsequence**, ou Maior Subsequência Comum). Esse problema consiste em encontrar o tamanho da maior subsequência em duas sequências de string. A maior subsequência é uma sequência que aparece na mesma ordem relativa, mas não precisa ser necessariamente contígua (não é uma substring) nas duas strings.

Considere o exemplo a seguir.

| | | | | | | |
|----------------------------------|----------|----------|----------|----------|----------|----------|
| string 1 | a | c | b | a | e | d |
| string 2 | a | b | c | a | d | f |
| LCS: "acad" com tamanho 4 | | | | | | |

Figura 14.3

Vamos agora observar o seguinte algoritmo:

```
function lcs(wordX, wordY) {
  const m = wordX.length;
  const n = wordY.length;
  const l = [];
  for (let i = 0; i <= m; i++) {
    l[i] = []; // {1}
    for (let j = 0; j <= n; j++) {
      l[i][j] = 0; // {2}
    }
  }
  for (let i = 0; i <= m; i++) {
    for (let j = 0; j <= n; j++) {
      if (i === 0 || j === 0) {
        l[i][j] = 0;
      } else if (wordX[i - 1] === wordY[j - 1]) {
        l[i][j] = l[i - 1][j - 1] + 1; // {3}
      } else {
        const a = l[i - 1][j];
        const b = l[i][j - 1];
        l[i][j] = a > b ? a : b; // {4} max(a,b)
      }
    }
  }
  return l[m][n]; // {5}
}
```

Se compararmos o problema da mochila com o algoritmo de LCS, perceberemos que ambos são muito semelhantes. Essa técnica se chama

memoização (memoization) e consiste em construir a solução de cima para baixo (top-down); a solução é dada no canto inferior direito da tabela/matriz.

Assim como o algoritmo do problema da mochila, essa abordagem exibe apenas o tamanho da LCS, mas não a LCS propriamente dita. Para extrair essa informação, devemos modificar um pouco o nosso algoritmo declarando uma nova matriz chamada **solution**. Observe que, em nosso código, há alguns comentários, e devemos substituí-los pelo código a seguir.

- Linha {1}: `solution[i] = [];`
- Linha {2}: `solution[i][j] = '0';`
- Linha {3}: `solution[i][j] = 'diagonal';`
- Linha {4}: `solution[i][j]=(l[i][j] == l[i-1][j]) ? 'top' : 'left';`
- Linha {5}: `printSolution(solution, wordX, m, n);`

A seguir, apresentamos a função **printSolution**:

```
function printSolution(solution, wordX, m, n) {  
    let a = m;  
    let b = n;  
    let x = solution[a][b];  
    let answer = '';  
    while (x !== '0') {  
        if (solution[a][b] === 'diagonal') {  
            answer = wordX[a - 1] + answer;  
            a--;  
            b--;  
        } else if (solution[a][b] === 'left') {  
            b--;  
        } else if (solution[a][b] === 'top') {  
            a--;  
        }  
        x = solution[a][b];  
    }  
    console.log('lcs: ' + answer);  
}
```

Podemos adicionar o **char** em **answer** sempre que a direção da matriz **solution** for **diagonal**.

Se o algoritmo anterior for executado com as strings '**acbaed**' e '**abcadf**',

veremos 4 na saída. A matriz \mathbf{l} usada para gerar o resultado terá uma aparência semelhante àquela mostrada a seguir. Podemos usar o algoritmo adicional para fazer um backtracking no valor da LCS também (está em destaque no diagrama seguinte).

| | a | b | c | a | d | f |
|---|---|---|---|---|---|---|
| a | 0 | 0 | 0 | 0 | 0 | 0 |
| c | 0 | 1 | 1 | 2 | 2 | 2 |
| b | 0 | 1 | 2 | 2 | 2 | 2 |
| a | 0 | 1 | 2 | 2 | 3 | 3 |
| e | 0 | 1 | 2 | 2 | 3 | 3 |
| d | 0 | 1 | 2 | 2 | 3 | 4 |

Figura 14.4

Com base na matriz anterior, sabemos que o algoritmo de LCS gera **acad** com tamanho 4.

O problema de LCS também pode ser implementado recursivamente. A versão recursiva pode ser vista no pacote de código-fonte que acompanha este livro.

Multiplicação de cadeia de matrizes

A **multiplicação de cadeia de matrizes** é outro problema famoso que pode ser resolvido com programação dinâmica. O problema consiste em encontrar a melhor maneira (ordem) de multiplicar um conjunto de matrizes.

Vamos tentar entender um pouco melhor o problema. Multiplicaremos duas matrizes, em que A é uma matriz $m \times n$ e B é uma matriz $m \times p$. O resultado é a matriz C , $n \times p$.

Suponha agora que queremos multiplicar $A * B * C * D$. Como a multiplicação é associativa, podemos multiplicar essas matrizes em qualquer ordem. Vamos, portanto, considerar o seguinte:

- A é uma matriz 10×100 ;

- B é uma matriz 100 x 5;
- C é uma matriz 5 x 50;
- D é uma matriz 50 x 1.
- O resultado é uma matriz $A^*B^*C^*D$, 10 x 1.

Nesse exemplo, há cinco maneiras de fazer essa multiplicação:

- $(A(B(CD)))$: o total de multiplicações é 1.750;
- $((AB)(CD))$: o total de multiplicações é 5.300;
- $((AB)C)D$: o total de multiplicações é 8.000;
- $((A(BC))D)$: o total de multiplicações é 75.500;
- $(A((BC)D))$: o total de multiplicações é 31.000.

A ordem das multiplicações pode fazer diferença no número total de multiplicações feitas. Então, como podemos criar um algoritmo para descobrir qual é o número mínimo de operações? O algoritmo de multiplicação de cadeia de matrizes será apresentado a seguir.

```
function matrixChainOrder(p) {
  const n = p.length;
  const m = [];
  const s = [];
  for (let i = 1; i <= n; i++) {
    m[i] = [];
    m[i][i] = 0;
  }
  for (let l = 2; l < n; l++) {
    for (let i = 1; i <= (n - l) + 1; i++) {
      const j = (i + l) - 1;
      m[i][j] = Number.MAX_SAFE_INTEGER;
      for (let k = i; k <= j - 1; k++) {
        const q = m[i][k] + m[k + 1][j] + ((p[i - 1] * p[k]) * p[j]); // {1}
        if (q < m[i][j]) {
          m[i][j] = q; // {2}
        }
      }
    }
  }
  return m[1][n - 1]; // {3}
}
```

A linha mais importante nesse código é a linha **{1}** porque é a que faz toda a mágica, isto é, calcula o número de multiplicações para uma dada ordem de parênteses e armazena o valor na matriz auxiliar **m**.

Se executarmos o algoritmo anterior em nosso exemplo inicial, veremos o resultado **1750**, que, conforme informamos antes, é o número mínimo de operações. Observe o código a seguir.

```
const p = [10, 100, 5, 50, 1];
console.log(matrixChainOrder(p));
```

No entanto, esse algoritmo também não apresenta a ordem dos parênteses para a solução ideal. Podemos fazer algumas alterações em nosso código para que essa informação seja obtida.

Inicialmente, devemos declarar e inicializar uma matriz auxiliar **s** usando o código a seguir.

```
const s = [];
for (let i = 0; i <= n; i++){
    s[i] = [];
    for (let j=0; j <= n; j++){
        s[i][j] = 0;
    }
}
```

Em seguida, na linha {2} da função **matrixChainOrder**, adicionaremos este código:

```
s[i][j] = k;
```

Na linha {3}, chamaremos a função que exibirá os parênteses para nós, assim:

```
printOptimalParenthesis(s, 1, n-1);
```

Por fim, temos a função **printOptimalParenthesis**, que deverá ter o seguinte aspecto:

```
function printOptimalParenthesis(s, i, j){
    if(i === j) {
        console.log("A[" + i + "]");
    } else {
        console.log("(");
        printOptimalParenthesis(s, i, s[i][j]);
        printOptimalParenthesis(s, s[i][j] + 1, j);
        console.log(")");
    }
}
```

Se o algoritmo modificado for executado, teremos também a ordem ideal dos parênteses, isto é, $(A[1](A[2](A[3]A[4])))$, que pode ser traduzida para $(A(B(CD)))$.

Algoritmos gulosos

Um algoritmo guloso (**greedy algorithm**) segue o método heurístico de resolução de problemas segundo o qual fazemos a escolha localmente ideal (a melhor solução na ocasião) em cada etapa, na esperança de encontrar uma solução ideal global (a melhor solução global). O quadro geral não é avaliado, como fazemos em um algoritmo de programação dinâmica.

Vamos observar como o problema do número mínimo de moedas para troco e o problema da mochila, que discutimos na seção sobre programação dinâmica, podem ser resolvidos usando a abordagem gulosa.

Abordamos outros algoritmos gulosos neste livro no Capítulo 12, *Grafos*, por exemplo, o algoritmo de Dijkstra, o algoritmo de Prim e o algoritmo de Kruskal.

Problema do número mínimo de moedas para troco

O problema do número mínimo de moedas para troco também pode ser resolvido com um algoritmo guloso. Na maioria das vezes, o resultado também será o ideal; contudo, para algumas denominações, não será.

Vamos observar o algoritmo:

```
function minCoinChange(coins, amount) {  
    const change = [];  
    let total = 0;  
    for (let i = coins.length; i >= 0; i--) { // {1}  
        const coin = coins[i];  
        while (total + coin <= amount) { // {2}  
            change.push(coin); // {3}  
            total += coin; // {4}  
        }  
    }  
    return change;  
}
```

Observe que a versão gulosa de `minCoinChange` é muito mais simples que a versão com PD. Para cada `coin` ({1}), começando da maior para a menor), somamos o valor de `coin` em `total`, e `total` deverá ser menor que `amount` ({2}). Adicionamos `coin` ao resultado ({3}) e somamos também em `total` ({4}).

A solução é simples e gulosa. Começaremos com a moeda (`coin`) de maior valor e daremos o troco possível com essa moeda. Quando não for mais

possível dar moedas com o valor da moeda no momento, começaremos a dar o troco com a moeda com o segundo maior valor, e assim por diante.

Para testar, usaremos o mesmo código que utilizamos na abordagem com PD, assim:

```
console.log(minCoinChange([1, 5, 10, 25], 36));
```

O resultado também será [25, 10, 1], que é o mesmo obtido com a PD. O diagrama a seguir (Figura 14.5) exemplifica como o algoritmo é executado.

No entanto, se usarmos a denominação [1, 3, 4] e executarmos o algoritmo guloso anterior, teremos [4, 1, 1] como resultado. Se usarmos a solução com a programação dinâmica, veremos [3, 3] como resultado, que é o ideal.

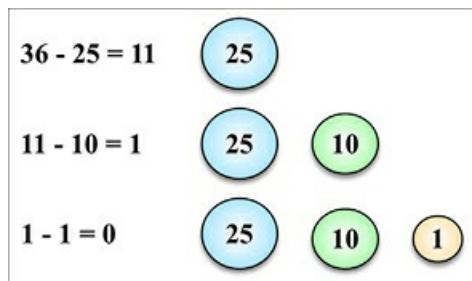


Figura 14.5

Os algoritmos gulosos são mais simples, além de serem mais rápidos que os algoritmos de programação dinâmica. No entanto, como podemos notar, eles nem sempre fornecem a resposta ideal. Entretanto, na média, uma solução aceitável será gerada, considerando o tempo consumido na execução.

Problema fracionário da mochila

O algoritmo para resolver o problema fracionário da mochila é um pouco diferente da versão com programação dinâmica. Enquanto na versão 0-1 do problema da mochila podíamos usar somente o item inteiro para encher a mochila, no problema fracionário, podemos usar frações dos itens. Usaremos o mesmo exemplo que vimos antes para comparar as diferenças, assim:

| Num. item | Peso | Valor |
|-----------|------|-------|
| 1 | 2 | 3 |
| 2 | 3 | 4 |
| 3 | 4 | 5 |

No exemplo com programação dinâmica, consideramos que a mochila era capaz de carregar um peso igual a 5. Nesse exemplo, podemos dizer que a melhor solução seria carregar a mochila com os itens 1 e 2 que, em conjunto, pesam 5 e têm um valor total igual a 7.

Se considerarmos a mesma capacidade no problema fracionário da mochila, teremos o mesmo resultado. Portanto, vamos considerar a capacidade de 6, em vez de 5.

Nesse caso, a solução seria usar os itens 1 e 2, e somente 25% do item 3. Isso nos daria um valor máximo de 8,25 com um peso total de 6.

Vamos observar o algoritmo a seguir.

```
function knapSack(capacity, weights, values) {  
    const n = values.length;  
    let load = 0;  
    let val = 0;  
    for (let i = 0; i < n && load < capacity; i++) { // {1}  
        if (weights[i] <= capacity - load) { // {2}  
            val += values[i];  
            load += weights[i];  
        } else {  
            const r = (capacity - load) / weights[i]; // {3}  
            val += r * values[i];  
            load += weights[i];  
        }  
    }  
    return val;  
}
```

Enquanto o total **load** for menor que **capacity** (não podemos carregar mais que **capacity**), iteramos pelos itens ({1}). Se pudermos usar o peso total do item ({2} – é menor ou igual a **capacity**), ele será adicionado ao valor total (**val**) e atualizaremos a carga atual da mochila em **load**. Se não for possível usar o peso total do item, calcularemos qual é a fração (**r**) que podemos usar ({3} – a fração que podemos carregar).

Se aplicarmos a capacidade de 6 na versão 0-1 do problema da mochila, veremos que os itens 1 e 3 serão selecionados como parte da solução. Nesse caso, teremos duas saídas diferentes para o mesmo problema, porém usando abordagens distintas para resolvê-lo.

Algoritmos de backtracking

O backtracking é uma estratégia usada para encontrar e compor uma solução de forma incremental. Começamos com um possível movimento e tentamos resolver o problema com o movimento selecionado. Se não funcionar, retrocedemos (fazemos um backtracking) e então selecionamos outro movimento, e assim por diante, até termos resolvido o problema. Em razão desse comportamento, os algoritmos de backtracking tentarão todos os movimentos possíveis (ou alguns movimentos, se a solução for encontrada antes) para solucionar um problema.

Há alguns problemas famosos que podem ser resolvidos com o backtracking:

- o problema do passeio do cavalo (Knight's tour problem);
- o problema das N rainhas (N Queen problem);
- rato em um labirinto;
- solucionador de sudoku.

Neste livro, veremos os problemas de Rato em um Labirinto e o Solucionador de Sudoku, pois são mais fáceis de compreender. No entanto, você poderá ver o código-fonte de outros problemas de backtracking no pacote de código-fonte que acompanha o livro.

Rato em um labirinto

Suponha que temos uma matriz de tamanho $N \times N$ e que cada posição da matriz seja um bloco. A posição (ou bloco) pode estar livre (valor 1) ou bloqueada (valor 0), conforme mostra o diagrama a seguir, em que S é a origem (source) e D é o destino (destination).

| | | | |
|----------|--|--|----------|
| S | | | |
| | | | |
| | | | |
| | | | |
| | | | D |

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |

Figura 14.6

A matriz é o labirinto, e o objetivo é que o “rato” comece na posição [0] [0] e vá para a posição [n-1][n-1] (o destino). O rato pode se mover em duas direções: verticalmente ou horizontalmente, para qualquer posição

que não esteja bloqueada.

Vamos começar declarando a estrutura básica de nosso algoritmo, assim:

```
export function ratInAMaze(maze) {  
  const solution = [];  
  for (let i = 0; i < maze.length; i++) { // {1}  
    solution[i] = [];  
    for (let j = 0; j < maze[i].length; j++) {  
      solution[i][j] = 0;  
    }  
  }  
  if (findPath(maze, 0, 0, solution) === true) { // {2}  
    return solution;  
  }  
  return 'NO PATH FOUND'; // {3}  
}
```

Começaremos criando a matriz que contém a solução. Inicializamos todas as posições com zero ({1}). Para cada movimento que o rato fizer, marcaremos o caminho com o valor 1. Se o algoritmo for capaz de encontrar uma solução ({2}), ele devolverá a matriz **solution**, ou devolverá uma mensagem de erro, caso contrário ({3}).

Em seguida, temos o método **findPath** que tentará encontrar a solução para uma dada matriz **maze**, começando nas posições **x** e **y**. Assim como no caso das outras técnicas apresentadas neste capítulo, a técnica de backtracking também utiliza recursão, e é isso que permite que o algoritmo faça retrocessos (backtracks), como vemos a seguir.

```
function findPath(maze, x, y, solution) {  
  const n = maze.length;  
  if (x === n - 1 && y === n - 1) { // {4}  
    solution[x][y] = 1;  
    return true;  
  }  
  if (isSafe(maze, x, y) === true) { // {5}  
    solution[x][y] = 1; // {6}  
    if (findPath(maze, x + 1, y, solution)) { // {7}  
      return true;  
    }  
    if (findPath(maze, x, y + 1, solution)) { // {8}  
      return true;  
    }  
    solution[x][y] = 0; // {9}  
    return false;  
  }
```

```
    return false; // {10}
}
```

O primeiro passo do algoritmo é verificar se o rato alcançou o seu destino ({4}). Em caso afirmativo, marcamos a última posição como parte do caminho e devolvemos `true`, o que significa que o movimento foi bem-sucedido. Se não for a última posição, verificamos se é seguro para o rato mover-se para a posição ({5}, o que significa que a posição está livre, conforme declarado no método `isSafe` apresentado adiante). Se for seguro, adicionamos o movimento no caminho ({6}) e tentaremos nos deslocar na matriz `maze` horizontalmente (para a direita), para a próxima posição ({7}). Se o movimento na horizontal não for possível, tentaremos nos deslocar verticalmente para a próxima posição, em direção à parte inferior da matriz ({8}). Se não for possível nos movermos horizontalmente nem verticalmente, removemos o movimento do caminho e retrocedemos ({9}), o que significa que o algoritmo tentará outra solução possível. Depois que o algoritmo tentar todos os movimentos possíveis e uma solução não for encontrada, ele devolverá `false` ({10}), sinalizando que não há solução para o problema.

```
function isSafe(maze, x, y) {
  const n = maze.length;
  if (x >= 0 && y >= 0 && x < n && y < n && maze[x][y] !== 0) {
    return true; // {11}
  }
  return false;
}
```

Podemos testar esse algoritmo com o código a seguir.

```
const maze = [
  [1, 0, 0, 0],
  [1, 1, 1, 1],
  [0, 0, 1, 0],
  [0, 1, 1, 1]
];
console.log(ratInAMaze(maze));
```

Eis o resultado:

```
[[1, 0, 0, 0],
 [1, 1, 1, 0],
 [0, 0, 1, 0],
 [0, 0, 1, 1]]
```

Solucionador de sudoku

O sudoku é um jogo de quebra-cabeça muito divertido, e um dos mais populares de todos os tempos. O objetivo é preencher uma matriz 9x9 com os dígitos de 1 a 9 de modo que cada linha e coluna seja composta de todos os nove dígitos. A matriz também contém quadrados menores (matrizes 3x3) que devem igualmente conter todos os nove dígitos. O jogo apresenta uma matriz inicial parcialmente preenchida, conforme vemos na Figura 14.7.

O algoritmo de backtracking para o Solucionador de Sudoku tentará colocar cada número nas linhas e nas colunas de modo a resolver o quebra-cabeça. Assim como no problema do Rato em um Labirinto, começaremos com o método principal do algoritmo:

```
function sudokuSolver(matrix) {
    if (solveSudoku(matrix) === true) {
        return matrix;
    }
    return 'NO SOLUTION EXISTS!';
}
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

Figura 14.7

O algoritmo devolverá a matriz preenchida com os dígitos que estavam faltando, caso uma solução seja encontrada, ou devolverá uma mensagem de erro. Vamos agora explorar a lógica principal do algoritmo:

```
const UNASSIGNED = 0;
function solveSudoku(matrix) {
    let row = 0;
    let col = 0;
```

```

let checkBlankSpaces = false;
for (row = 0; row < matrix.length; row++) { // {1}
    for (col = 0; col < matrix[row].length; col++) {
        if (matrix[row][col] === UNASSIGNED) {
            checkBlankSpaces = true; // {2}
            break;
        }
    }
    if (checkBlankSpaces === true) { // {3}
        break;
    }
}
if (checkBlankSpaces === false) {
    return true; // {4}
}
for (let num = 1; num <= 9; num++) { // {5}
    if (isSafe(matrix, row, col, num)) { // {6}
        matrix[row][col] = num; // {7}
        if (solveSudoku(matrix)) { // {8}
            return true;
        }
        matrix[row][col] = UNASSIGNED; // {9}
    }
}
return false; // {10}
}

```

O primeiro passo é verificar se o quebra-cabeça foi resolvido ({1}). Se não houver espaços em branco (posições com valor 0), é sinal de que o quebra-cabeça está completo ({4}). Contudo, se houver espaços em branco ({2}), sairemos dos dois laços ({3}) e as variáveis `row` e `col` terão a posição do espaço em branco que deve ser preenchido com um dígito de 1 a 9. Em seguida, o algoritmo tentará preencher os espaços em branco com os dígitos de 1 a 9, um de cada vez ({5}). Verificamos se é seguro adicionar o dígito ({6}), o que significa que ele não está presente na linha, na coluna ou no quadrado (a matriz 3x3). Se for seguro, adicionamos o dígito no quebra-cabeça ({7}) e executamos a função `solveSudoku` novamente para tentar o próximo espaço disponível ({8}). Caso um dígito seja colocado em uma posição incorreta, marcamos essa posição como vazia novamente ({9}), e o algoritmo fará o retrocesso ({10}) e tentará um dígito diferente.

A função `isSafe` é declarada com as verificações necessárias para inserir um dígito na matriz, assim:

```
function isSafe(matrix, row, col, num) {
```

```

        return (
            !usedInRow(matrix, row, num) &&
            !usedInCol(matrix, col, num) &&
            !usedInBox(matrix, row - (row % 3), col - (col % 3), num)
        );
    }
}

```

E as verificações específicas são declaradas da seguinte maneira:

```

function usedInRow(matrix, row, num) {
    for (let col = 0; col < matrix.length; col++) { // {11}
        if (matrix[row][col] === num) {
            return true;
        }
    }
    return false;
}

function usedInCol(matrix, col, num) {
    for (let row = 0; row < matrix.length; row++) { // {12}
        if (matrix[row][col] === num) {
            return true;
        }
    }
    return false;
}

function usedInBox(matrix, boxStartRow, boxStartCol, num) {
    for (let row = 0; row < 3; row++) {
        for (let col = 0; col < 3; col++) {
            if (matrix[row + boxStartRow][col + boxStartCol] === num) { // {13}
                return true;
            }
        }
    }
    return false;
}

```

Inicialmente, verificamos se o dígito já existe na linha (`row`), iterando por todas as posições da matriz na linha (`row`) especificada (`{11}`). Em seguida, iteramos por todas as colunas para verificar se o dígito já existe na dada coluna (`{12}`). A última verificação consiste em ver se o dígito existe no quadrado (`{13}`), iterando por todas as posições da matriz quadrada 3x3.

Podemos testar o algoritmo usando o exemplo a seguir.

```

const sudokuGrid = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
]

```

```

[4, 0, 0, 8, 0, 3, 0, 0, 1],
[7, 0, 0, 0, 2, 0, 0, 0, 6],
[0, 6, 0, 0, 0, 0, 2, 8, 0],
[0, 0, 0, 4, 1, 9, 0, 0, 5],
[0, 0, 0, 8, 0, 0, 7, 9]
];
console.log(sudokuSolver(sudokuGrid));

```

Eis o resultado:

```

[[5, 3, 4, 6, 7, 8, 9, 1, 2],
 [6, 7, 2, 1, 9, 5, 3, 4, 8],
 [1, 9, 8, 3, 4, 2, 5, 6, 7],
 [8, 5, 9, 7, 6, 1, 4, 2, 3],
 [4, 2, 6, 8, 5, 3, 7, 9, 1],
 [7, 1, 3, 9, 2, 4, 8, 5, 6],
 [9, 6, 1, 5, 3, 7, 2, 8, 4],
 [2, 8, 7, 4, 1, 9, 6, 3, 5],
 [3, 4, 5, 2, 8, 6, 1, 7, 9]]

```

Introdução à programação funcional

Até agora neste livro, usamos um paradigma chamado **programação imperativa**. Na programação imperativa, programamos cada passo descrevendo em detalhes o que deve ser feito e em que ordem isso deve ocorrer.

Nesta seção, introduziremos um novo paradigma chamado **programação funcional (PF)**. Já usamos alguns trechos de código com PF em alguns algoritmos no livro. A programação funcional é um paradigma usado especialmente em ambientes acadêmicos, e, graças às linguagens modernas como Python e Ruby, ela começou a se tornar popular também entre os desenvolvedores no mercado. Felizmente, podemos usar JavaScript na programação funcional, tirando proveito de suas funcionalidades de ES2015.

Programação funcional versus programação imperativa

Desenvolver no **paradigma funcional** não é difícil; é somente uma questão de se acostumar com o modo como o paradigma funciona. Vamos implementar um exemplo para perceber as diferenças.

Suponha que devemos exibir todos os elementos de um array. Podemos usar a programação imperativa para declarar a seguinte função:

```

const printArray = function(array) {
  for (var i = 0; i < array.length; i++){
    console.log(array[i]);
  }
};

printArray([1, 2, 3, 4, 5]);

```

No código anterior, iteramos pelo `array` e fizemos o log de cada um dos itens.

Vamos agora tentar converter o exemplo para usar a programação funcional. Na programação funcional, as funções são as estrelas. Devemos nos concentrar no que deve ser descrito, e não em como fazê-lo. Vamos retornar à frase “iteramos pelo array e fizemos o log de cada um dos itens”. Assim, a primeira tarefa na qual nos concentraremos é a iteração nos dados e, em seguida, executaremos alguma ação nesses dados, que significa fazer o logging dos itens. A função a seguir será responsável por iterar pelo array.

```

const forEach = function(array, action){
  for (var i = 0; i < array.length; i++){
    action(array[i]);
  }
};

```

Então criaremos outra função que será responsável pelo logging dos elementos do array no console (podemos considerá-la como uma **função de callback**), assim:

```

const logItem = function(item) {
  console.log(item);
};

```

Por fim, podemos usar as funções que declaramos, do seguinte modo:

```
forEach([1, 2, 3, 4, 5], logItem);
```

Observando atentamente o código anterior, podemos descrevê-lo dizendo que ele fará o log de cada item do array no console. E temos o nosso primeiro exemplo de programação funcional!

Alguns pontos que você deve ter em mente são:

- O principal objetivo é descrever os dados e a transformação que devemos aplicar nesses dados.
- A ordem da execução do programa tem pouca importância; por outro lado, os passos e sua sequência são muito importantes na programação imperativa.
- As funções e as coleções de dados são as estrelas na programação

funcional.

- Podemos usar e abusar de funções e da recursão na programação funcional, enquanto os laços, as atribuições, as condicionais e também as funções são usados na programação imperativa.
- Na programação funcional, devemos evitar efeitos colaterais e dados mutáveis, o que significa que não modificaremos os dados passados para a função. Se houver necessidade de devolver uma solução com base na entrada, podemos criar uma cópia e devolver a cópia modificada dos dados.

ES2015+ e a programação funcional

Com as funcionalidades das versões ES2015+, desenvolver programas funcionais em JavaScript se tornou mais fácil ainda. Vamos considerar um exemplo.

Suponha que queremos encontrar o valor mínimo de um array. Na programação imperativa, para executar essa tarefa, basta iterar pelo array e verificar se o valor mínimo atual é maior que o valor do array; em caso afirmativo, teremos um novo valor mínimo, assim:

```
var findMinArray = function(array){  
    var minValue = array[0];  
    for (var i=1; i<array.length; i++){  
        if (minValue > array[i]){  
            minValue = array[i];  
        }  
    }  
    return minValue;  
};  
console.log(findMinArray([8,6,4,5,9])); // exibe 4
```

Para executar a mesma tarefa com uma programação funcional, podemos usar a função `Math.min`, passando todos os elementos do array para serem comparados. Para transformar o array em elementos únicos, podemos usar o operador de espalhamento da ES2015 (...), como no exemplo a seguir.

```
const min_ = function(array){  
    return Math.min(...array)  
};  
console.log(min_([8,6,4,5,9])); //exibe 4
```

Com as **funções de seta** (arrow functions) da ES2015, podemos simplificar um pouco mais o código anterior:

```
const min = arr => Math.min(...arr);
console.log(min([8,6,4,5,9]));
```

Também podemos reescrever o primeiro exemplo usando a sintaxe da ES2015:

```
const forEach = (array, action) => array.forEach(item => action(item));
const logItem = (item) => console.log(item);
```

Caixa de ferramentas funcional de JavaScript – map, filter e reduce

As funções **map**, **filter** e **reduce** (que vimos no Capítulo 3, *Arrays*) são a base da programação funcional em JavaScript.

Com a função **map**, podemos transformar ou mapear uma coleção de dados para outra coleção de dados. Vamos observar um exemplo que usa a programação imperativa:

```
const daysOfWeek = [
  {name: 'Monday', value: 1},
  {name: 'Tuesday', value: 2},
  {name: 'Wednesday', value: 7}
];
let daysOfWeekValues_ = [];
for (let i = 0; i < daysOfWeek.length; i++) {
  daysOfWeekValues_.push(daysOfWeek[i].value);
}
```

Vamos agora considerar o mesmo exemplo usando a programação funcional com a sintaxe ES2015+, da seguinte maneira:

```
const daysOfWeekValues = daysOfWeek.map(day => day.value);
console.log(daysOfWeekValues);
```

Com a função **filter**, podemos filtrar valores de uma coleção. Vamos considerar um exemplo:

```
const positiveNumbers_ = function(array){
  let positive = [];
  for (let i = 0; i < array.length; i++) {
    if (array[i] >= 0){
      positive.push(array[i]);
    }
  }
  return positive;
}
console.log(positiveNumbers_([-1,1,2,-2]));
```

Podemos escrever o mesmo código usando o paradigma funcional, assim:

```
const positiveNumbers = (array) => array.filter(num => (num >= 0));
```

```
console.log(positiveNumbers([-1,1,2,-2]));
```

Além disso, com a função `reduce`, podemos reduzir uma coleção a um valor específico. Por exemplo, vamos observar como podemos somar os valores de um array:

```
const sumValues = function(array) {
  let total = array[0];
  for (let i = 1; i < array.length; i++) {
    total += array[i];
  }
  return total;
};
console.log(sumValues([1, 2, 3, 4, 5]));
```

Também podemos escrever o código anterior assim:

```
const sum_ = function(array){
  return array.reduce(function(a, b){
    return a + b;
  })
};
console.log(sum_([1, 2, 3, 4, 5]));
```

Essas funções também podem ser combinadas com as funcionalidades da ES2015, por exemplo, com o operador de desestruturação para atribuição e as funções de seta, como vemos no código a seguir.

```
const sum = arr => arr.reduce((a, b) => a + b);
console.log(sum([1, 2, 3, 4, 5]));
```

Vamos analisar outro exemplo. Suponha que tenhamos de escrever uma função para concatenar vários arrays. Para isso, podemos criar outro array que conterá todos os elementos dos demais arrays. O código a seguir pode ser executado usando o paradigma imperativo.

```
const mergeArrays_ = function(arrays){
  const count = arrays.length;
  let newArray = [];
  let k = 0;
  for (let i = 0; i < count; i++){
    for (var j = 0; j < arrays[i].length; j++){
      newArray[k++] = arrays[i][j];
    }
  }
  return newArray;
};
console.log(mergeArrays_([[1, 2, 3], [4, 5], [6]]));
```

Nesse exemplo, observe que declaramos variáveis e usamos laços. Vamos

agora executar o código implementado antes usando a programação funcional em JavaScript, assim:

```
const mergeArraysConcat = function(arrays){
  return arrays.reduce( function(p,n){
    return p.concat(n);
  });
};

console.log(mergeArraysConcat([[1, 2, 3], [4, 5], [6]]));
```

O código anterior faz exatamente a mesma tarefa, porém é orientado a função. Podemos também simplificar mais ainda o código com a ES2015, como vemos no código a seguir.

```
const mergeArrays = (...arrays) => [].concat(...arrays);
console.log(mergeArrays([1, 2, 3], [4, 5], [6]));
```

De 11 linhas de código para apenas uma (embora o código seja menos legível)!

Se quiser exercitar um pouco mais a programação funcional com JavaScript, tente fazer os exercícios em <http://reactivex.io/learnrx/> (são muito interessantes!).

Bibliotecas e estruturas de dados funcionais de JavaScript

Há algumas bibliotecas JavaScript excelentes que oferecem suporte para o paradigma funcional, contendo funções utilitárias e estruturas de dados funcionais. Na lista a seguir, há algumas das bibliotecas funcionais mais famosas de JavaScript.

- **Underscore.js**: <http://underscorejs.org/>
- **Bilby.js**: <http://bilby.brianmckenna.org/>
- **Lazy.js**: <http://danieltao.com/lazy.js/>
- **Bacon.js**: <https://baconjs.github.io/>
- **Fn.js**: <http://eliperelman.com/fn.js/>
- **Functional.js**: <http://functionaljs.com/>
- **Ramda.js**: <http://ramdajs.com/0.20.1/index.html>
- **Mori**: <http://swannodette.github.io/mori/>

Se estiver interessado em conhecer melhor a programação funcional em

JavaScript, dê uma olhada no livro da Packt, acessível em <https://www.packtpub.com/web-development/functional-programming-javascript>.

Resumo

Neste capítulo, discutimos os problemas mais famosos de programação dinâmica, como uma variação do problema do número mínimo de moedas para troco, o problema da mochila, a maior subsequência comum e a multiplicação de cadeia de matrizes. Conhecemos os algoritmos que usam a abordagem de dividir e conquistar, e vimos como eles diferem da programação dinâmica.

Vimos os algoritmos guloso (greedy) e como desenvolver uma solução gulosa para o problema do número mínimo de moedas para troco e o problema fracionário da mochila. Também discutimos o conceito de backtracking e vimos alguns problemas famosos, como o do Rato em um Labirinto e um Solucionador de Sudoku.

Além disso, conhecemos a programação funcional e discutimos alguns exemplos de como usar as funcionalidades de JavaScript nesse paradigma.

No próximo capítulo, abordaremos a notação *big-O* (O grande) e discutiremos o modo de calcular a complexidade de um algoritmo. Veremos também outros conceitos presentes no mundo dos algoritmos.

CAPÍTULO 15

Complexidade de algoritmos

Neste capítulo, discutiremos a famosa **notação big-O** (O grande) e a teoria de **NP-completo** (NP-completeness); veremos também como podemos nos divertir com alguns algoritmos e aperfeiçoar o nosso conhecimento a fim de melhorar nossas habilidades de programação e de resolução de problemas.

Notação big-O

No Capítulo 13, *Algoritmos de ordenação e de busca*, introduzimos o conceito de notação big-O. O que isso significa, exatamente? Essa notação é usada para descrever o desempenho ou a complexidade de um algoritmo. A notação big-O é usada para classificar algoritmos de acordo com o tempo que eles demorarão para executar, conforme os requisitos de espaço/memória à medida que o tamanho da entrada aumentar.

Ao analisar algoritmos, as classes de funções a seguir são as mais comumente encontradas.

| Notação | Nome |
|------------------|-----------------|
| $O(1)$ | Constante |
| $O(\log(n))$ | Logarítmica |
| $O((\log(n))^c)$ | Polilogarítmica |
| $O(n)$ | Linear |
| $O(n^2)$ | Quadrática |
| $O(n^c)$ | Polinomial |
| $O(c^n)$ | Exponencial |

Compreendendo a notação big-O

Como podemos medir a eficiência de um algoritmo? Em geral, usamos recursos como uso (tempo) de CPU, utilização de memória, de disco e de rede. Quando falamos da notação big-O, em geral, consideraremos o uso (tempo) de CPU.

Vamos tentar entender como a notação big-O funciona usando alguns exemplos.

O(1)

Considere a função a seguir.

```
function increment(num){  
    return ++num;  
}
```

Se tentarmos executar a função `increment(1)`, teremos um tempo de execução igual a x . Se tentarmos executar essa mesma função novamente com um parâmetro diferente (por exemplo, `num` igual a `2`), o tempo de execução também será x . O parâmetro não importa; o desempenho da função de incremento será o mesmo. Por esse motivo, podemos dizer que a função anterior tem complexidade igual a $O(1)$ (é constante).

O(n)

Vamos agora usar o algoritmo de busca sequencial que implementamos no Capítulo 13, *Algoritmos de ordenação e de busca*, como exemplo:

```
function sequentialSearch(array, value, equalsFn = defaultEquals) {  
    for (let i = 0; i < array.length; i++) {  
        if (equalsFn(value, array[i])) { // {1}  
            return i;  
        }  
    }  
    return -1;  
}
```

Se passarmos um array com 10 elementos (`[1, ..., 10]`) para essa função e procurarmos o elemento 1, na primeira tentativa, encontraremos o elemento que estávamos procurando. Vamos supor que o custo seja 1 para cada vez que a linha `{1}` é executada.

Vamos tentar outro exemplo. Suponha que estamos procurando o elemento 11. A linha `{1}` será executada 10 vezes (uma iteração será feita por todos os valores do array e o valor que estamos procurando não será encontrado; desse modo, `-1` será devolvido). Se a linha `{1}` tiver um custo igual a 1, executá-la 10 vezes terá um custo igual a 10, isto é, 10 vezes mais em comparação com o primeiro exemplo.

Suponha agora que o array tenha 1.000 elementos ($[1, \dots, 1.000]$). Procurar o elemento 1.001 resultará em a linha `{1}` ser executada 1.000 vezes (e então `-1` será devolvido).

Observe que o custo total da execução da função `sequentialSearch` depende do número de elementos do array (tamanho) e do valor que procuramos. Se o item que estivermos procurando estiver presente no array, quantas vezes a linha `{1}` será executada? Se o item procurado não existir, a linha `{1}` será executada o número de vezes correspondente ao tamanho do array, que chamaremos de cenário de pior caso.

Considerando o cenário de pior caso da função `sequentialSearch`, se tivermos um array de tamanho 10, o custo será igual a 10. Se tivermos um array de tamanho 1.000, o custo será igual a 1.000. Podemos concluir que a função `sequentialSearch` tem uma complexidade de $O(n)$ – em que n é o tamanho do array (entrada).

Para ver a explicação anterior na prática, vamos modificar o algoritmo a fim de calcular o custo, isto é, `cost` (cenário de pior caso), assim:

```
function sequentialSearch(array, value, equalsFn = defaultEquals) {
  let cost = 0;
  for (let i = 0; i < array.length; i++) {
    cost++;
    if (equalsFn(value, array[i])) {
      return i;
    }
  }
  console.log(`cost for sequentialSearch with input size ${array.length} is
${cost}`);
  return -1;
}
```

Experimente executar o algoritmo anterior usando tamanhos distintos de entrada para que você veja os diferentes resultados.

$O(n^2)$

Para o exemplo de $O(n^2)$, usaremos o algoritmo de *bubble sort*:

```
function bubbleSort(array, compareFn = defaultCompare) {
  const { length } = array;
  for (let i = 0; i < length; i++) { // {1}
    for (let j = 0; j < length - 1; j++) { // {2}
```

```

        if (compareFn(array[j], array[j + 1]) === Compare.BIGGER_THAN) {
            swap(array, j, j + 1);
        }
    }
    return array;
}

```

Considere que as linhas **{1}** e **{2}** tenham, cada uma, um custo igual a 1. Vamos modificar o algoritmo a fim de calcular o custo (**cost**) da seguinte maneira:

```

function bubbleSort(array, compareFn = defaultCompare) {
    const { length } = array;
    let cost = 0;
    for (let i = 0; i < length; i++) { // {1}
        cost++;
        for (let j = 0; j < length - 1; j++) { // {2}
            cost++;
            if (compareFn(array[j], array[j + 1]) === Compare.BIGGER_THAN) {
                swap(array, j, j + 1);
            }
        }
    }
    console.log(`cost for bubbleSort with input size ${length} is ${cost}`);
    return array;
}

```

Se executarmos **bubbleSort** para um array de tamanho 10, **cost** será igual a 10^2 . Se executarmos **bubbleSort** para um array de tamanho 100, **cost** será 10.000 (100^2). Observe que a execução demorará mais tempo ainda sempre que aumentarmos o tamanho da entrada.

Note que o código cuja complexidade é $O(n)$ tem apenas um laço **for**, enquanto, para $O(n^2)$, há dois laços **for** aninhados. Se o algoritmo tiver três laços **for** iterando pelo array, provavelmente ele terá uma complexidade de $O(n^3)$.

Comparando as complexidades

Podemos criar uma tabela com alguns valores para exemplificar o custo do algoritmo, dado o tamanho de sua entrada, da seguinte maneira:

| Tamanho da entrada (n) | $O(1)$ | $O(\log(n))$ | $O(n)$ | $O(n \log(n))$ | $O(n^2)$ | $O(2^n)$ |
|------------------------|--------|--------------|--------|----------------|----------|----------|
| | | | | | | |

| | | | | | | |
|--------|---|------|--------|----------|-------------|-------------------|
| 10 | 1 | 1 | 10 | 10 | 100 | 1.024 |
| 20 | 1 | 1,30 | 20 | 26,02 | 400 | 1.048.576 |
| 50 | 1 | 1,69 | 50 | 84,94 | 2.500 | Número bem grande |
| 100 | 1 | 2 | 100 | 200 | 10.000 | Número bem grande |
| 500 | 1 | 2,69 | 500 | 1.349,48 | 250.000 | Número bem grande |
| 1.000 | 1 | 3 | 1.000 | 3.000 | 1.000.000 | Número bem grande |
| 10.000 | 1 | 4 | 10.000 | 40.000 | 100.000.000 | Número bem grande |

Podemos desenhar um gráfico baseado nas informações apresentadas na tabela anterior para exibir o custo de diferentes complexidades em notação big-O, assim:

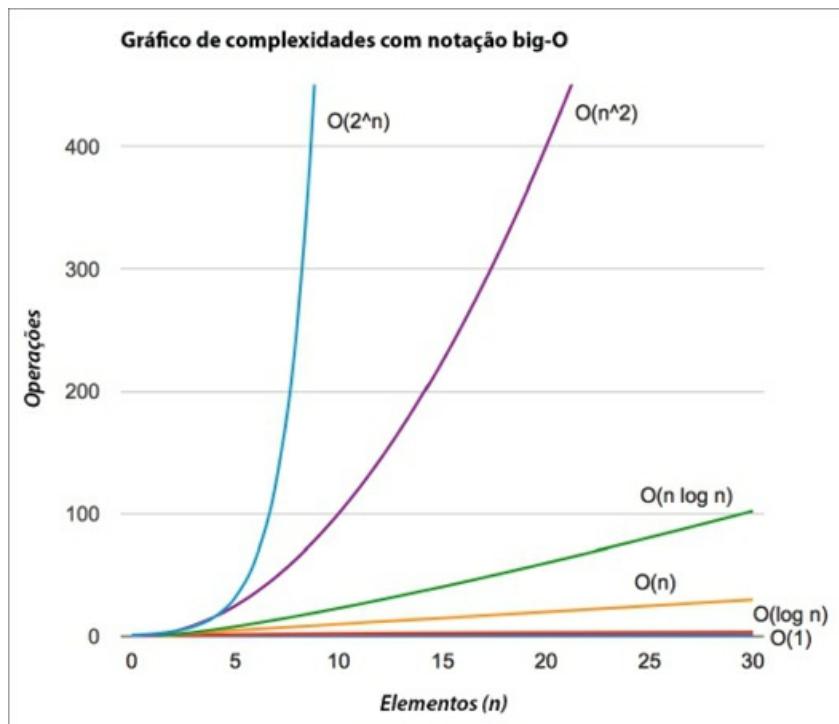


Figura 15.1

O gráfico anterior também foi gerado com JavaScript. Veja o código-fonte dele no diretório `examples/chapter15` no pacote de códigos-fontes.

Na próxima seção, você encontrará uma “colinha” que mostra as complexidades dos algoritmos implementados neste livro.

Se quiser uma versão impressa da folha de cola com a notação big-O, o link a seguir contém uma boa versão: <http://www.bigocheatsheet.com> (observe que, para algumas estruturas de dados como pilhas e filas,

implementamos neste livro uma versão melhorada da estrutura de dados e, desse modo, temos uma complexidade big-O menor do que aquela exibida no link citado.

Estruturas de dados

A tabela a seguir mostra as complexidades das estruturas de dados.

| Estrutura de dados | Casos médios | | | Piores casos | | |
|-------------------------|--------------|--------------|------------------------------|--------------|--------------|--------------|
| | Inserção | Remoção | Busca | Inserção | Remoção | Busca |
| Array/Pilha/Fila | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| Lista ligada | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| Lista duplamente ligada | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| Tabela hash | $O(1)$ | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| Árvore binária de busca | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(n)$ | $O(n)$ | $O(n)$ |
| Árvore AVL | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ |
| Árvore rubro-negra | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ |
| Heap binário | $O(\log(n))$ | $O(\log(n))$ | $O(1)$: encontrar máx./mín. | $O(\log(n))$ | $O(\log(n))$ | $O(1)$ |

Grafos

A tabela a seguir mostra as complexidades para os grafos.

| Gerenciamento de nós/arestas | Tamanho do repositório | Adição de vértice | Adição de aresta | Remoção de vértice | Remoção de aresta | Consulta |
|------------------------------|------------------------|-------------------|------------------|--------------------|-------------------|----------|
| Lista de adjacências | $O(V + E)$ | $O(1)$ | $O(1)$ | $O(V + E)$ | $O(E)$ | $O(V)$ |
| Matriz de adjacências | $O(V ^2)$ | $O(V ^2)$ | $O(1)$ | $O(V ^2)$ | $O(1)$ | $O(1)$ |

Algoritmos de ordenação

A tabela a seguir mostra as complexidades dos algoritmos de ordenação.

| Algoritmo (aplicado a um array) | Complexidade do tempo | | |
|---------------------------------|-----------------------|------------------|------------------|
| | Melhores casos | Casos médios | Piores casos |
| Bubble sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Shell sort | $O(n \log(n))$ | $O(n \log^2(n))$ | $O(n \log^2(n))$ |
| Merge sort | $O(n \log(n))$ | $O(n \log(n))$ | $O(n \log(n))$ |
| Quick sort | $O(n \log(n))$ | $O(n \log(n))$ | $O(n^2)$ |

| | | | |
|---------------|----------------|----------------|----------------|
| Heap sort | $O(n \log(n))$ | $O(n \log(n))$ | $O(n \log(n))$ |
| Counting sort | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ |
| Bucket sort | $O(n+k)$ | $O(n+k)$ | $O(n^2)$ |
| Radix sort | $O(nk)$ | $O(nk)$ | $O(nk)$ |

Algoritmos de busca

A tabela a seguir mostra as complexidades dos algoritmos de busca.

| Algoritmo | Estrutura de dados | Piores casos |
|-----------------------------|------------------------------------------|----------------|
| Busca sequencial | Array | $O(n)$ |
| Busca binária | Array ordenado | $O(\log(n))$ |
| Busca por interpolação | Array ordenado | $O(n)$ |
| Busca em profundidade (DFS) | Grafo com $ V $ vértices e $ E $ arestas | $O(V + E)$ |
| Busca em largura (BFS) | Grafo com $ V $ vértices e $ E $ arestas | $O(V + E)$ |

Introdução à teoria de NP-completo

Em geral, dizemos que um algoritmo é eficiente se tiver complexidade $O(n^k)$ para alguma constante k , e ele é chamado de algoritmo polinomial.

Dado um problema em que há um algoritmo polinomial mesmo para o pior caso, o algoritmo é representado por P (polinomial).

Há outro conjunto de algoritmos chamado **NP** (**Nondeterministic Polynomial**, ou Polinomial Não Determinístico). Um problema NP é um problema para o qual a solução pode ser verificada em um tempo polinomial.

Se um problema P tiver um algoritmo que execute em tempo polinomial, podemos também verificar a sua solução em tempo polinomial. Então, é possível concluir que P é um subconjunto de NP ou é igual a ele. No entanto, não sabemos se $P = NP$.

Problemas NP-completos são os mais difíceis de um conjunto NP. Um problema de decisão L será NP-completo se:

1. L está em NP (isto é, qualquer dada solução para problemas NP-completos pode ser verificada rapidamente, mas não há nenhuma solução eficiente conhecida).
2. Todo problema em NP pode ser reduzido a L em tempo polinomial.

Para compreender o que é a redução de um problema, considere L e M como dois problemas de decisão. Suponha que o algoritmo A resolva L . Isso quer dizer que, se y for uma entrada para M , o algoritmo B responderá *Sim* ou *Não* conforme y pertencer a M ou não. A ideia é encontrar uma transformação de L para M de modo que o algoritmo B faça parte de um algoritmo A para resolver A .

Também temos outro conjunto de problemas chamado **NP-difícil** (NP-hard). Um problema será NP-difícil se tiver a propriedade 2 (de NP-completo) e não precisar ter a propriedade 1. Assim, o conjunto NP-completo também é um subconjunto do conjunto NP-difícil.

Saber se $P = NP$ ou não é uma das principais perguntas em ciência da computação. Se alguém descobrir a resposta para essa pergunta, haverá um grande impacto em criptografia, pesquisa de algoritmos, inteligência artificial e muitas outras áreas.

A seguir, vemos o diagrama de Euler para os problemas **P**, **NP**, **NP-completo** e **NP-difícil**, considerando que $P < > NP$.

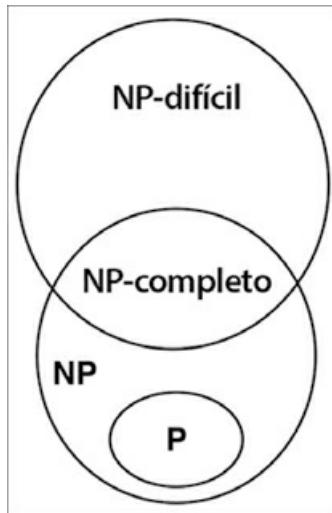


Figura 15.2

Como exemplos de problemas NP-difícil que não sejam problemas NP-completos, podemos mencionar o **problema da parada** (halting problem) e o **SAT (Boolean Satisfiability Problem)**, ou Problema de Satisfação Booleana).

Como exemplos de problemas NP-completos, podemos mencionar também o problema da soma de subconjuntos (subset sum problem), o

problema do caixeiro-viajante (traveling salesman problem) e o problema da cobertura de vértices (vertex cover problem).

Para mais informações sobre esses problemas, acesse <https://en.wikipedia.org/wiki/NP-completeness>.

Problemas impossíveis e algoritmos heurísticos

Alguns dos problemas mencionados são impossíveis de resolver. No entanto, algumas técnicas podem ser usadas para obter uma solução aproximada em um intervalo de tempo satisfatório. Uma técnica seria usar algoritmos heurísticos. Uma solução gerada por métodos heurísticos talvez não seja a melhor das soluções, mas será boa o suficiente para resolver o problema na ocasião.

Alguns exemplos de métodos heurísticos são: busca local, algoritmos genéticos, roteamento heurístico e aprendizado de máquina (machine learning). Para obter mais informações, acesse [https://en.wikipedia.org/wiki/Heuristic_\(computer_science\)](https://en.wikipedia.org/wiki/Heuristic_(computer_science)).

Métodos heurísticos são uma maneira ótima e divertida de tentar resolver um problema. Você pode tentar escolher um problema e desenvolver um método heurístico para o seu trabalho de conclusão de curso ou como tese de mestrado.

Divertindo-se com algoritmos

Não estudamos os algoritmos somente porque precisamos entendê-los na faculdade ou porque queremos nos tornar desenvolvedores. Você pode vir a ser um profissional mais bem-sucedido se aperfeiçoar suas habilidades para resolução de problemas usando os algoritmos que vimos neste livro como forma de solucionar problemas.

A melhor maneira de aperfeiçoar o seu conhecimento sobre resolução de problemas é praticando, e essa tarefa não precisa ser tediosa. Nesta seção, apresentaremos alguns sites que você poderá acessar para começar a se divertir com os algoritmos (e até mesmo ganhar um pouco de dinheiro enquanto faz isso!).

Eis uma lista de alguns sites úteis (alguns deles não aceitam uma solução

escrita em JavaScript, mas podemos aplicar a lógica discutida neste livro também em outras linguagens de programação).

- **UVa Online Judge** (<http://uva.onlinejudge.org/>): esse site contém um conjunto de problemas usados em vários concursos de programação pelo mundo, incluindo o **International Collegiate Programming Contest (ICPC)** da ACM, patrocinado pela IBM. (Se você ainda está na faculdade, experimente participar desse concurso; se sua equipe vencer, você poderá viajar pelo mundo com todas as despesas pagas!) O site contém centenas de problemas para os quais podemos usar os algoritmos que aprendemos neste livro.
- **Sphere Online Judge** (<http://www.spoj.com/>): esse site é semelhante ao UVa Online Judge, mas aceita mais linguagens (incluindo submissões em JavaScript).
- **Coderbyte** (<http://coderbyte.com/>): esse site contém problemas (níveis fácil, médio e difícil) que também podem ser resolvidos com JavaScript.
- **Project Euler** (<https://projecteuler.net/>): esse site contém uma série de problemas de matemática/programação. Tudo que você tem a fazer é fornecer a resposta para o problema, mas podemos usar algoritmos para descobrir a resposta para nós.
- **HackerRank** (<https://www.hackerrank.com>): esse site contém desafios divididos em 16 categorias (você pode usar os algoritmos que vimos neste livro e muitos outros). Também aceita JavaScript, entre outras linguagens.
- **CodeChef** (<http://www.codechef.com>): esse site também contém diversos problemas e organiza competições online.
- **Top Coder** (<http://www.topcoder.com>): esse site organiza torneios de programação, geralmente patrocinados por empresas como NASA, Google, Yahoo, Amazon e Facebook. Alguns concursos oferecem oportunidades para trabalhar na empresa patrocinadora, enquanto outros podem oferecer prêmios em dinheiro. O site também disponibiliza ótimos tutoriais para resolução de problemas e algoritmos.

Outro aspecto interessante sobre os sites anteriores é que, em geral, eles apresentam um problema do mundo real, e é necessário identificar o algoritmo que pode ser usado para resolvê-lo. É uma forma de saber que os

algoritmos que conhecemos neste livro não são apenas didáticos, mas também podem ser aplicados para resolver problemas do mundo real.

Se você está iniciando uma carreira em tecnologia, é extremamente recomendável criar uma conta gratuita no **GitHub** (<https://github.com>); assim poderá fazer commit dos códigos-fontes que escrever para resolver os problemas dos sites anteriores. Se você não tem nenhuma experiência profissional, o GitHub poderá ajudá-lo a desenvolver um portfólio e conseguir seu primeiro emprego!

Resumo

Neste capítulo, discutimos a notação big-O (O grande) e como podemos calcular a complexidade de um algoritmo aplicando esse conceito. Apresentamos a teoria de NP-completo – um conceito que você poderá explorar melhor se estiver interessado em aprender mais sobre resolução de problemas impossíveis e o uso de métodos heurísticos para obter uma solução aproximadamente satisfatória.

Também apresentamos alguns sites nos quais você poderá se registrar gratuitamente, aplicar todo o conhecimento adquirido com a leitura deste livro e até mesmo receber uma oferta para seu primeiro emprego em TI!

Boa programação!

Entendendo algoritmos

Um guia *ilustrado* para programadores
e outros curiosos

Aditya Y. Bhargava



novatec

MANNING

Entendendo Algoritmos

Bhargava, Aditya Y.

9788575226629

264 páginas

[Compre agora e leia](#)

Um guia ilustrado para programadores e outros curiosos. Um

algoritmo nada mais é do que um procedimento passo a passo para a resolução de um problema. Os algoritmos que você mais utilizará como um programador já foram descobertos, testados e provados. Se você quer entendê-los, mas se recusa a estudar páginas e mais páginas de provas, este é o livro certo. Este guia cativante e completamente ilustrado torna simples aprender como utilizar os principais algoritmos nos seus programas. O livro Entendendo Algoritmos apresenta uma abordagem agradável para esse tópico essencial da ciência da computação. Nele, você aprenderá como aplicar algoritmos comuns nos problemas de programação enfrentados diariamente. Você começará com tarefas básicas como a ordenação e a pesquisa. Com a prática, você enfrentará problemas mais complexos, como a compressão de dados e a inteligência artificial. Cada exemplo é apresentado em detalhes e inclui diagramas e códigos completos em Python. Ao final deste livro, você terá dominado algoritmos amplamente aplicáveis e saberá quando e onde utilizá-los. O que este livro inclui A abordagem de algoritmos de pesquisa, ordenação e algoritmos gráficos Mais de 400 imagens com descrições detalhadas Comparações de desempenho entre algoritmos Exemplos de código em Python Este livro de fácil leitura e repleto de imagens é destinado a programadores autodidatas, engenheiros ou pessoas que gostariam de recordar o assunto.

[Compre agora e leia](#)

O'REILLY®

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações
nativas de nuvem



novatec

Bilgin Ibryam
Roland Huß

Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

272 páginas

[Compre agora e leia](#)

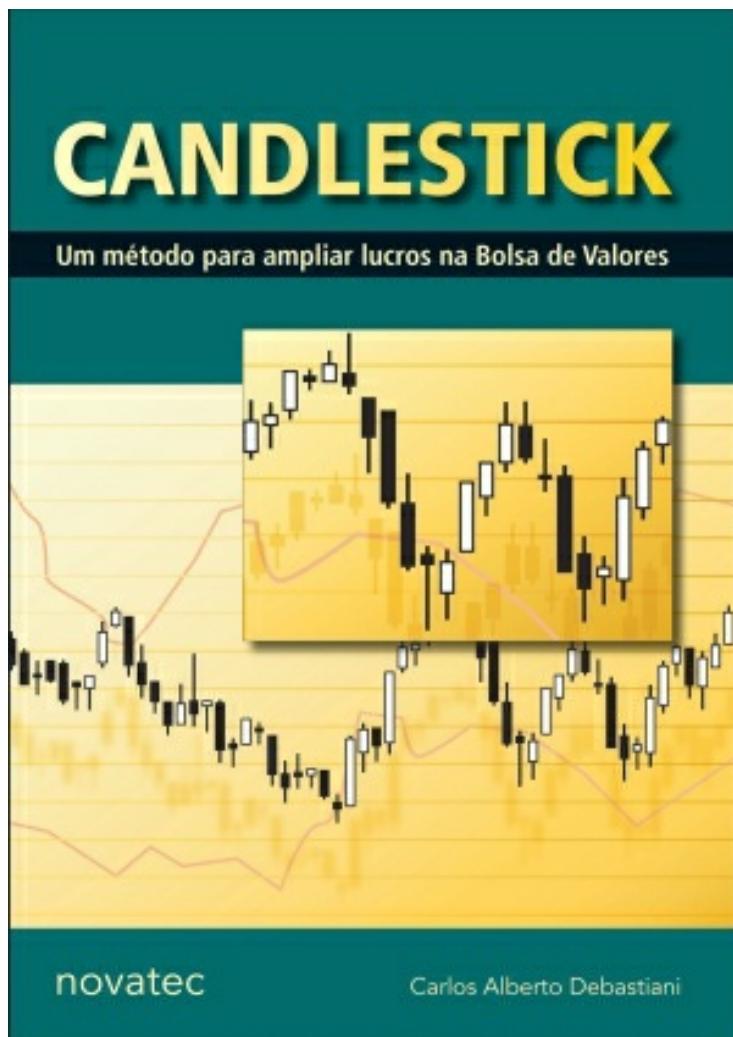
O modo como os desenvolvedores projetam, desenvolvem e

executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões:

- Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres.
- Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a plataforma.
- Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos.
- Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes.
- Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade

automática (autoscaling).

[Compre agora e leia](#)



Candlestick

Debastiani, Carlos Alberto
9788575225943
200 páginas

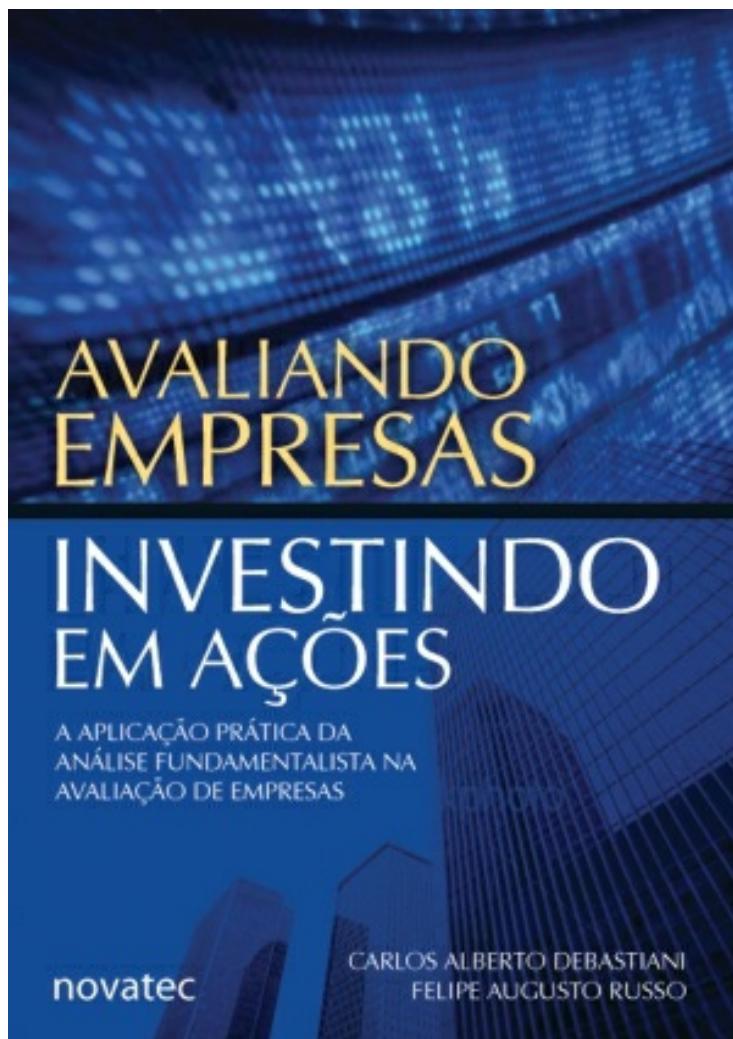
[Compre agora e leia](#)

A análise dos gráficos de Candlestick é uma técnica

amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos.

Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

[Compre agora e leia](#)



Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

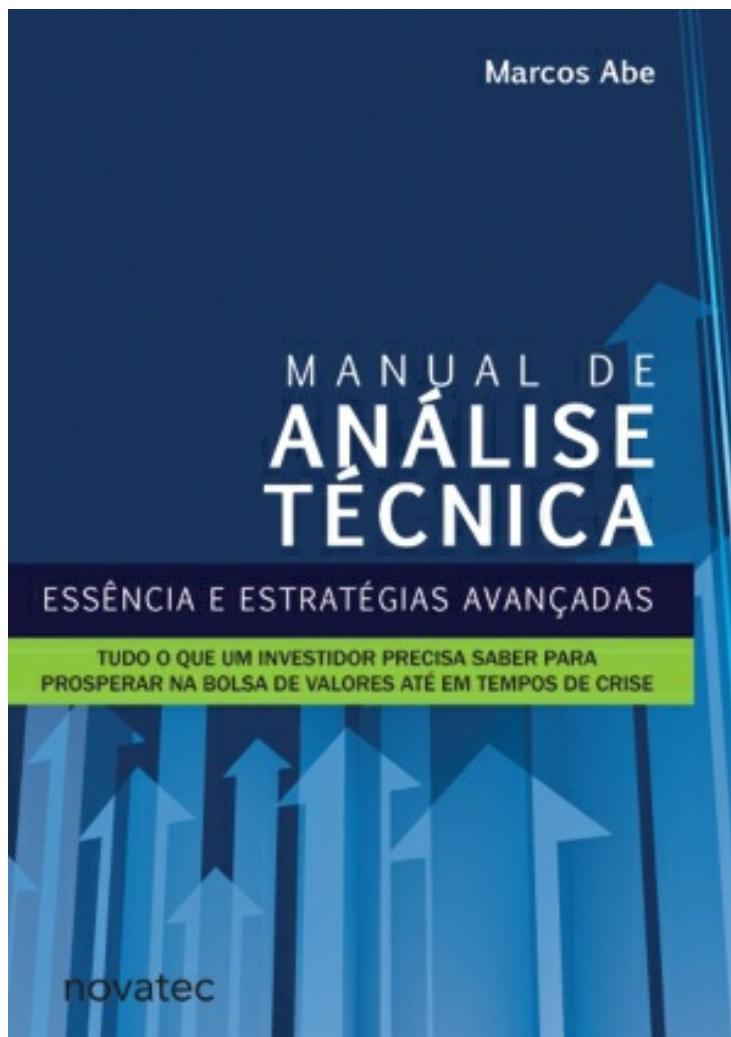
9788575225974

224 páginas

[Compre agora e leia](#)

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

[Compre agora e leia](#)



Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

[Compre agora e leia](#)

Este livro aborda o tema Investimento em Ações de maneira

inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

[Compre agora e leia](#)