Learning JavaScript Data Structures and Algorithms

Understand and implement classic data structures and algorithms using JavaScript

Loiane Groner

Learning JavaScript Data Structures and Algorithms

Understand and implement classic data structures and algorithms using JavaScript

Loiane Groner

Learning JavaScript Data Structures and Algorithms

Credits

Author
Loiane Groner

Reviewers
Yaroslav Bigus
Maxwell Dayvson Da Silva
Vincent Lark
Vishal Rajpal

Commissioning Editor
Kunal Parikh

Acquisition Editor
Subho Gupta

Content Development Editor
Akashdeep Kundu

Technical Editors
Madhuri Das
Nikhil Potdukhe

Copy Editors
Maria Gould
Ameesha Green
Paul Hindle

Project Coordinator
Neha Thakur

Proofreaders
Simran Bhogal
Ameesha Green

Indexers
Mariammal Chettiyar
Tejal Soni

Graphics
Ronak Dhruv
Valentina D'silva
Abhinash Sahu

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

Loiane Groner lives in São Paulo, Brazil, and has over 8 years of software development experience. While at university, she demonstrated a great interest in IT. She worked as an assistant teacher for two and a half years teaching Algorithms, Data Structures, and Computing Theory. She represented her university at the ACM International Collegiate Programming Contest — Brazilian finals (South America regionals), and also worked as student delegate of the Brazilian Computing Society (SBC) for 2 years. She won a merit award in her senior year for being one of the top three students in the Computer Science department and graduated with honors.

She has previously worked at multinational companies such as IBM. Her areas of expertise include Java SE and Java EE as well as Sencha technologies (Ext JS and Sencha Touch). Nowadays, she works as a software development manager at a financial institution, where she manages overseas solutions. She also works as an independent Sencha consultant and coach.

Loiane is also the author of *Ext JS 4 First Look*, *Mastering Ext JS*, and *Sencha Architect App Development*, all published by Packt Publishing.

She is passionate about Sencha and Java, and is the Campinas Java Users Group (CampinasJUG) leader and Espirito Santo Java Users Group (ESJUG) coordinator, both Brazilian JUGs.

Loiane also contributes to the software development community through her blogs at `http://loianegroner.com` (English) and `http://loiane.com` (Brazilian Portuguese), where she writes about her IT career, Ext JS, Sencha Touch, PhoneGap, Spring Framework, and general development notes as well as publishing screencasts.

If you want to get in touch, you can find Loiane on Facebook (`https://www.facebook.com/loianegroner`) and Twitter (`@loiane`).

Acknowledgments

I would like to thank my parents for giving me education, guidance, and advice over all these years and for helping me to be a better human being and professional. A very special thanks to my husband, for being patient and supportive and giving me encouragement.

I would also like to thank my professors from FAESA, who taught me about the algorithms and data structures presented in this book.

Also, a big thanks to my friends and readers for all the support. It is really nice when people get in touch at conferences and by any social network mentioning that they have read one of my books and give me feedback. Thank you very much!

About the Reviewers

Yaroslav Bigus is an expert in building cross-platform web and mobile applications. He has over 5 years of experience in development and has worked for companies in Leeds and New York. He has been using the .NET Framework stack to develop backend systems; JavaScript, AngularJS, jQuery, and Underscore for the frontend; and Xamarin for mobile devices.

He is currently working for an Israeli start-up named yRuler (Tangiblee). Previously, he has reviewed *Xamarin Mobile Application Development for iOS* and *iOS Development with Xamarin Cookbook*, both by Packt Publishing.

> I am thankful to my friends and family for their support and love.

Maxwell Dayvson Da Silva is a native Brazilian who works as a software architect for The New York Times. He has more than 11 years of experience working for two of Brazil's leading digital media companies. His work at Terra, a global digital media company, helped reach an audience of over 100 million people monthly with entertainment, sports, and news content. Later, he was part of Globo.com, the largest media conglomerate in Latin America. His contribution in the digital media field is only a portion of how he spends his time. Combining his passion for art and science, he creates games and interactive art installations. Inspired by his son, Arthur, he continually searches for new ways to spread science in a fun way to children both in NYC and Brazil. He can be reached at `https://github.com/dayvson`.

> I would like to say thank you to Juliane Inês do Nascimento for taking care of Arthur in such an amazing way while I am away. You are an amazing mother!

Vincent Lark is an experienced programmer who has worked as a backend and frontend web developer for web start-ups in Luxembourg and France. He now focuses on creating modern web UIs and develops games as a hobby. He has also reviewed the book *WebGL HOTSHOT*, *Packt Publishing*.

Vishal Rajpal is an experienced software engineer who started developing professional software applications in 2011. He has worked primarily on Java, JavaScript, and multiplatform mobile application development platforms including PhoneGap and Titanium.

At present, he is pursuing his Master's degree in Computer Science from Northeastern University, Seattle, and has been working on Scheme (Lisp), Objective-C, computer systems, and algorithms.

He lives in Seattle and can be reached at `vishalarajpal@gmail.com`. You can also find out more about his work at `https://github.com/vishalrajpal/` and `https://www.vishal-rajpal.blogspot.com`. He has also reviewed the book *PhoneGap 3.x Mobile Application Development HOTSHOT*, *Packt Publishing*.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

JavaScript is currently the most popular programming language. It is known as "the Internet language" due to the fact that Internet browsers understand JavaScript natively, without installing any plugins. JavaScript has grown so much that it is no longer just a frontend language; it is now also present on the server (Node.js) and database as well (MongoDB).

Understanding data structures is very important for any technology professional. Working as a developer means you have the ability to solve problems with the help of programming languages and data structures. They are an indispensable piece of the solutions we need to create to solve these problems. Choosing the wrong data structure can also impact the performance of the program we are writing. This is why it is important to get to know different data structures and how to apply them properly.

Algorithms play a major role in the art of Computer Science. There are so many ways of solving the same problem, and some approaches are better than others. That is why it is also very important to know the most famous algorithms.

Happy coding!

What this book covers

Chapter 1, *JavaScript – A Quick Overview*, covers the basics of JavaScript you need to know prior to learning about data structures and algorithms. It also covers setting up the development environment needed for this book.

Chapter 2, *Arrays*, explains how to use the most basic and most used data structure arrays. This chapter demonstrates how to declare, initialize, add, and remove elements from an array. It also covers how to use the native JavaScript array methods.

Chapter 3, *Stacks*, introduces the stack data structure, demonstrating how to create a stack and how to add and remove elements. It also demonstrates how to use stacks to solve some Computer Science problems.

Chapter 4, *Queues*, covers the queue data structure, demonstrating how to create a queue and add and remove elements. It also demonstrates how to use queues to solve some Computer Science problems and the major differences between queues and stacks.

Chapter 5, *Linked Lists*, explains how to create the linked list data structure from scratch using objects and the pointer concept. Besides covering how to declare, create, add, and remove elements, it also covers the various types of linked lists such as doubly linked lists and circular linked lists.

Chapter 6, *Sets*, introduces the set data structure and how it can be used to store non-repeated elements. It also explains the different types of set operations and how to implement and use them.

Chapter 7, *Dictionaries and Hashes*, explains the dictionary and hash data structures and the differences between them. This chapter covers how to declare, create, and use both data structures. It also explains how to handle collisions in hashes and techniques to create better hash functions.

Chapter 8, *Trees*, covers the tree data structure and its terminologies, focusing on binary search tree data as well as the methods trees use to search, traverse, add, and remove nodes. It also introduces the next steps you can take to delve deeper into the world of trees, covering what tree algorithms should be learned next.

Chapter 9, *Graphs*, introduces the amazing world of the graphs data structure and its application in real-world problems. This chapter covers the most common graph terminologies, the different ways of representing a graph, how to traverse graphs using the breadth-first search and depth-first search algorithms, and its applications.

Chapter 10, *Sorting and Searching Algorithms*, explores the most used sorting algorithms such as bubble sort (and its improved version), selection sort, insertion sort, merge sort, and quick sort. It also covers searching algorithms such as sequential and binary search.

Chapter 11, *More About Algorithms*, introduces some algorithm techniques and the famous big-O notation. It covers the recursion concept and some advanced algorithm techniques such as dynamic programming and greedy algorithms. This chapter introduces big-O notation and its concepts. Finally, it explains how to take your algorithm knowledge to the next level. This is an online chapter available on the Packt Publishing website. You can download it from `https://www.packtpub.com/sites/default/files/downloads/4874OS_Chapter11_More_About_Algorithms.pdf`.

Appendix, *Big-O Cheat Sheet*, lists the complexities of the algorithms (using big-O notation) implemented in this book. This is also an online chapter, which can be downloaded from `https://www.packtpub.com/sites/default/files/downloads/4874OS_Appendix_Big_O_Cheat_Sheet.pdf`.

What you need for this book

You can set up three different development environments for this book. You do not need to have all three environments; you can select one or give all of them a try!

- For the first option, you need a browser. It is recommended to use one of the following browsers:
    - Chrome (`https://www.google.com/chrome/browser/`)
    - Firefox (`https://www.mozilla.org/en-US/firefox/new/`)

- For the second option, you will need:
    - One of the browsers listed in the first option
    - A web server; if you do not have any web server installed in your computer, you can install XAMPP (`https://www.apachefriends.org`)

- The third option is an environment 100 percent JavaScript! For this, you will need the following elements:
    - One of the browsers listed in the first option
    - Node.js (`http://nodejs.org/`)
    - After installing Node.js, install `http-server` (package) as follows:

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

You can find more detailed instructions in *Chapter 1*, *JavaScript – A Quick Overview*, as well.

Who this book is for

This book is intended for students of Computer Science, people who are just starting their career in technology, and those who want to learn about data structures and algorithms with JavaScript. Some knowledge of programming logic is the only thing you need to know to start having fun with algorithms and JavaScript!

This book is written for beginners who want to learn about data structures and algorithms, and also for those who are already familiar with data structures and algorithms but who want to learn how to use them with JavaScript.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Inside the `script` tag, we have the JavaScript code."

A block of code is set as follows:

```
console.log("num: "+ num);
console.log("name: "+ name);
console.log("trueValue: "+ trueValue);
console.log("price: "+ price);
console.log("nullVar: "+ nullVar);
console.log("und: "+ und);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

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

Any command-line input or output is written as follows:

```
npm install http-server –g
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The number of **Node Packages Modules** (`https://www.npmjs.org/`) also grows exponentially."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

For this book, the code files can be downloaded or forked from the following GitHub repository as well: `https://github.com/loiane/javascript-datastructures-algorithms`.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/4874OS_ColoredImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
JavaScript – A Quick Overview

JavaScript is a very powerful language. It is the most popular language in the world and is one of the most prominent languages on the Internet. For example, GitHub (the world's largest code host, available at `https://github.com`) hosts over 400,000 JavaScript repositories (the largest number of projects is in JavaScript; refer to `http://goo.gl/ZFx6mg`). The number of projects in JavaScript in GitHub grows every year.

JavaScript is not a language that can only be used in the frontend. It can also be used in the backend as well, and Node.js is the technology responsible for this. The number of **Node Packages Modules** (`https://www.npmjs.org/`) also grows exponentially.

JavaScript is a must-have on your résumé if you are or going to become a web developer.

In this book, you are going to learn about the most used data structures and algorithms. But why use JavaScript to learn about data structures and algorithms? We have already answered this question. JavaScript is very popular, and JavaScript is appropriate to learn about data structures because it is a functional language. Also, this can be a very fun way of learning something new, as it is very different (and easier) than learning about data structures with a standard language such as C or Java. And who said data structures and algorithms were only made for languages such as C and Java? You might need to implement some of these languages while developing for the frontend as well.

Learning about data structures and algorithms is very important. The first reason is because data structures and algorithms can solve the most common problems efficiently. This will make a difference on the quality of the source code you write in the future (including performance—if you choose the incorrect data structure or algorithm depending on the scenario, you can have some performance issues). Secondly, algorithms are studied in college together with introductory concepts of Computer Science. And thirdly, if you are planning to get a job in the greatest **IT** (**Information Technology**) companies (such as Google, Amazon, Ebay, and so on), data structures and algorithms are subjects of interview questions.

Setting up the environment

One of the pros of the JavaScript language compared to other languages is that you do not need to install or configure a complicated environment to get started with it. Every computer has the required environment already, even though the user may never write a single line of source code. All we need is a browser!

To execute the examples in this book, it is recommended that you have Google Chrome or Firefox installed (you can use the one you like the most), an editor of your preference (such as Sublime Text), and a web server (XAMPP or any other of your preference—but this step is optional). Chrome, Firefox, Sublime Text, and XAMPP are available for Windows, Linux, and Mac OS.

If you use Firefox, it is also recommended to install the **Firebug** add-on (`https://getfirebug.com/`).

We are going to present you with three options to set up your environment.

The browser is enough

The simplest environment that you can use is a browser.

You can use Firefox + Firebug. When you have Firebug installed, you will see the following icon in the upper-right corner:
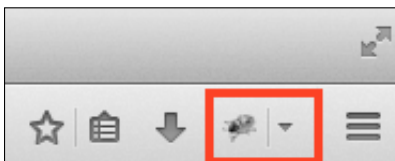
When you open Firebug (simply click on its icon), you will see the **Console** tab and you will be able to write all your JavaScript code on its command-line area as demonstrated in the following screenshot (to execute the source code you need to click on the **Run** button):
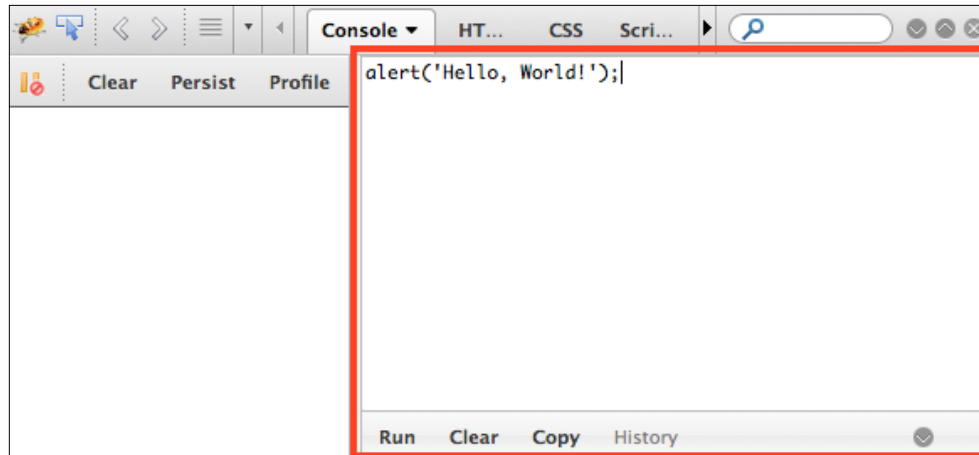
You can also expand the command line to fit the entire available area of the Firebug add-on.

You can also use Google Chrome. Chrome already comes with **Google Developer Tools**. To open it, locate the setting and control icon and navigate to **Tools | Developer Tools**, as shown in the following screenshot:
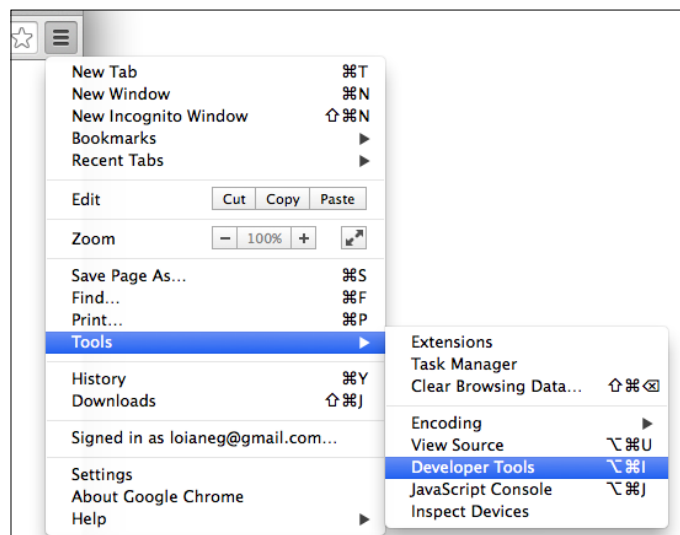
Then, in the **Console** tab, you can write your own JavaScript code for testing, as follows:
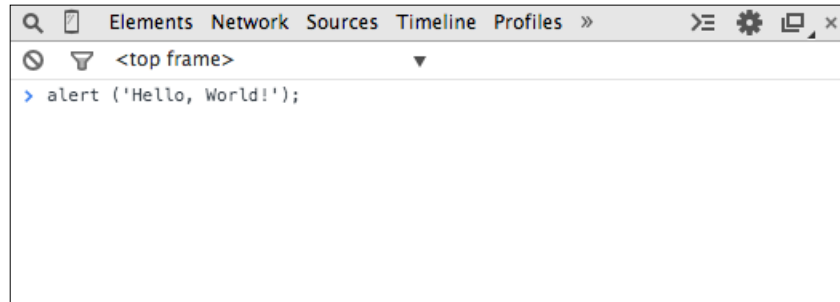
Using web servers (XAMPP)

The second environment you might want to install on your computer is also simple, but a little bit more complex than just using a browser.

You will need to install XAMPP (`https://www.apachefriends.org`) or any web server of your preference. Then, inside the XAMPP installation folder, you will find the `htdocs` directory. You can create a new folder where you can execute the source code we will implement in this book, or you can download the source code from this book and extract it to the `htdocs` directory, as follows:
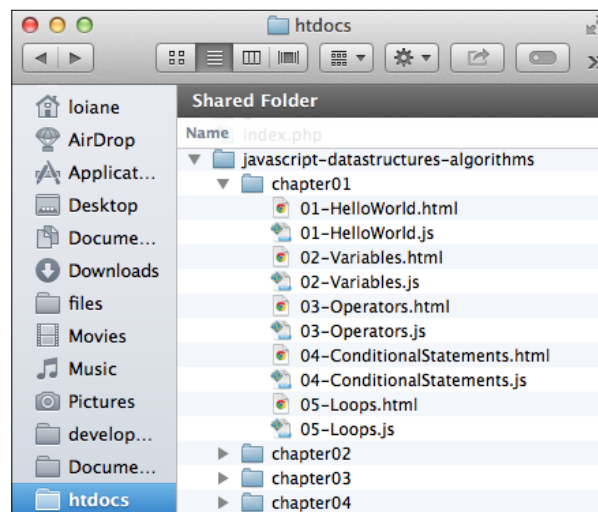
Then, you can access the source code from your browser using your localhost URL (after starting the XAMPP server) as shown in the following screenshot (do not forget to enable Firebug or Google Developer Tools to see the output):

> When executing the examples, always remember to have Google Developer Tools or Firebug open to see the output.

It's all about JavaScript (Node.js)

The third option is having an environment that is 100 percent JavaScript! Instead of using XAMPP, which is an Apache server, we can use a JavaScript server.

To do so, we need to have Node.js installed. Go to `http://nodejs.org/` and download and install Node.js. After that, open the terminal application (if you are using Windows, open the command prompt with Node.js that was installed with Node.js) and run the following command:
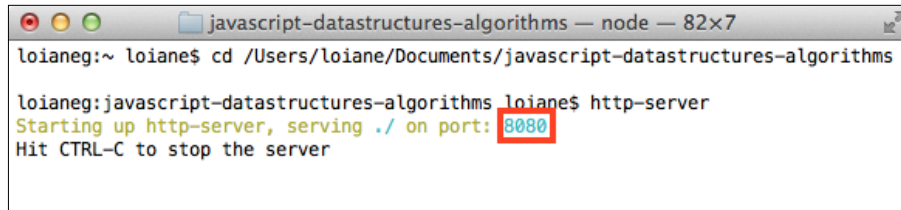
```
npm install http-server –g
```

Make sure you type the command and don't copy and paste it. Copying the command might give you some errors.

You can also execute the command as an administrator. For Linux and Mac systems, use the following command:

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

This command will install `http-server`, which is a JavaScript server. To start a server and run the examples from this book in the terminal application, change the directory to the folder that contains the book's source code and type `http-server`, as displayed in the following screenshot:

```
● ○ ○            javascript-datastructures-algorithms — node — 82×7
loianeg:~ loiane$ cd /Users/loiane/Documents/javascript-datastructures-algorithms

loianeg:javascript-datastructures-algorithms loiane$ http-server
Starting up http-server, serving ./ on port: 8080
Hit CTRL-C to stop the server
```

To execute the examples, open the browser and access the localhost on the port specified by the `http-server` command:
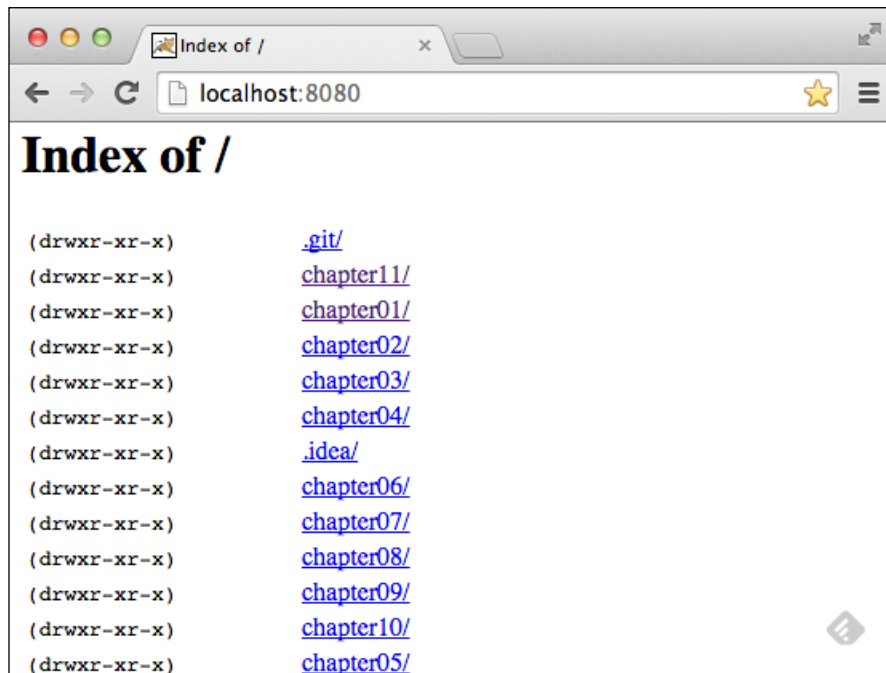
```
● ○ ○     Index of /                    ×

←  →  C      localhost:8080                        ☆   ≡
```

Index of /

(drwxr-xr-x)	.git/
(drwxr-xr-x)	chapter11/
(drwxr-xr-x)	chapter01/
(drwxr-xr-x)	chapter02/
(drwxr-xr-x)	chapter03/
(drwxr-xr-x)	chapter04/
(drwxr-xr-x)	.idea/
(drwxr-xr-x)	chapter06/
(drwxr-xr-x)	chapter07/
(drwxr-xr-x)	chapter08/
(drwxr-xr-x)	chapter09/
(drwxr-xr-x)	chapter10/
(drwxr-xr-x)	chapter05/

JavaScript basics

Before we start diving into the various data structures and algorithms, let's have a quick overview of the JavaScript language. This section will present the JavaScript basics required to implement the algorithms we will create in the subsequent chapters.

To start, let's see the two different ways we can use JavaScript code in an HTML page:

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

The first way is demonstrated by the previous code. We need to create an HTML file and write this code on it. In this example, we are declaring the `script` tag inside the HTML file, and inside the `script` tag, we have the JavaScript code.

For the second example, we need to create a JavaScript file (we can save it as `01-HelloWorld.js`), and inside this file, we will insert the following code:

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

Then, our HTML file will look like this:

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

The second example demonstrates how to include a JavaScript file inside an HTML file.

By executing any of these two examples, the output will be the same. However, the second example is the best practice.

> You may find JavaScript `include` statements or JavaScript code inside the `head` tag in some examples on the Internet. As a best practice, we will include any JavaScript code at the end of the `body` tag. This way, the HTML will be parsed by the browser and displayed before the scripts are loaded. This boosts the performance of the page.

Variables

Variables store data that can be set, updated, and retrieved whenever needed. Values that are assigned to a variable belong to a type. In JavaScript, the available types are **numbers**, **strings**, **Booleans**, **functions**, and **objects**. We also have **undefined** and **null**, along with **arrays**, **dates**, and **regular expressions**. The following is an example of how to use variables in JavaScript:

```javascript
var num = 1; //{1}
num = 3; //{2}

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

On line {1}, we have an example of how to declare a variable in JavaScript (we are declaring a number). Although it is not necessary to use the `var` keyword declaration, it is a good practice to always specify when we are declaring a new variable.

On line {2}, we are updating an existing variable. JavaScript is not a *strongly-typed* language. This means you can declare a variable and initialize it with a number, and then update it with a string or any other data type. Assigning a value to a variable that is different from its original type is also not a good practice.

On line {3}, we are also declaring a number, but this time it is a *decimal floating point*. On line {4}, we are declaring a string; on line {5}, we are declaring a Boolean. On line {6}, we are declaring a `null` value, and on line {7}, we are declaring an *undefined* variable. A `null` value means no value and `undefined` means a variable that has been declared but not yet assigned a value:

```
console.log("num: "+ num);
console.log("name: "+ name);
console.log("trueValue: "+ trueValue);
console.log("price: "+ price);
console.log("nullVar: "+ nullVar);
console.log("und: "+ und);
```

If we want to see the value of each variable we have declared, we can use `console.log` to do so, as listed in the previous code snippet.

> We have three ways of outputting values in JavaScript that we can use with the examples of this book. The first one is `alert('My text here')`, which will output an alert window on the browser; the second one is `console.log('My text here')`, which will output text on the **Console** tab of the debug tool (Google Developer Tools or Firebug, depending on the browser you are using). Finally, the third way is outputting the value directly on the HTML page that is being rendered by the browser by using `document.write('My text here')`. You can use the option that you feel most comfortable with.

The `console.log` method also accepts more than just arguments. Instead of `console.log("num: "+ num)`, we can also use `console.log("num: ", num)`.

We will discuss functions and objects later in this chapter.

Variable scope

Scope refers to where in the algorithm we can access the variable (it can also be a function when we are working with function scopes). There are local and global variables.

Let's look at an example:

```
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}
```

Line {1} will output `global` because we are referring to a global variable. Line {2} will output `local` because we declared the `myVariable` variable inside the `myFunction` function as a local variable, so the scope will be inside `myFunction` only.

Line {3} will output `global` because we are referencing the global variable named `myOtherVariable` that was initialized in the second line of the example. Line {4} will output `local`. Inside the `myOtherFunction` function, we are referencing the `myOtherVariable` global variable and assigning the value `local` to it because we are not declaring the variable using the `var` keyword. For this reason, line {5} will output `local` (because we changed the value of the variable inside `myOtherFunction`).

You may hear that global variables in JavaScript are evil, and this is true. Usually, the quality of JavaScript source code is measured by the number of global variables and functions (a large number is bad). So, whenever possible, try avoiding global variables.

Operators

We need operators when performing any operation in a programming language. JavaScript also has arithmetic, assignment, comparison, logical, bitwise, and unary operators, among others. Let's take a look at them:

```javascript
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));
```

On line {1}, we have the arithmetic operators. In the following table, we have the operators and their descriptions:

Arithmetic operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder of a division operation)
++	Increment
--	Decrement

On line {2}, we have the assignment operators. In the following table, we have the operators and their descriptions:

Assignment operator	Description
`=`	Assignment
`+=`	Addition assignment (x += y) == (x = x + y)
`-=`	Subtraction assignment (x -= y) == (x = x - y)
`*=`	Multiplication assignment (x *= y) == (x = x * y)
`/=`	Division assignment (x /= y) == (x = x / y)
`%=`	Remainder assignment (x %= y) == (x = x % y)

On line {3}, we have the comparison operators. In the following table, we have the operators and their descriptions:

Comparison operator	Description
`==`	Equal to
`===`	Equal to (value and object type both)
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to

And on line {4}, we have the logical operators. In the following table, we have the operators and their descriptions:

Logical operator	Description
`&&`	And
`||`	Or
`!`	Not

JavaScript also supports bitwise operators, shown as follows:

```
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));
```

The following table contains more detailed descriptions of the bitwise operators:

Bitwise operator	Description
&	And
|	Or
~	Not
^	Xor
<<	Left shift
>>	Right shift

The `typeof` operator returns the type of the variable or expression. For example, have a look at the following code:

```
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'});
```

The output will be as follows:

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

JavaScript also supports the `delete` operator, which deletes a property from an object:

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

In this book's algorithms, we will be using some of these operators.

Truthy and falsy

In JavaScript, true and false are a little bit tricky. In most languages, the Boolean values `true` and `false` represent the true/false results. In JavaScript, a string such as "Packt" has the value `true`, for example.

The following table can help us better understand how `true` and `false` work in JavaScript:

Value type	Result
`undefined`	`false`.
`null`	`false`.
`Boolean`	`true` is `true` and `false` is `false`!
`Number`	The result is `false` for +0, -0, or `NaN`; otherwise, the result is `true`.
`String`	The result is `false` if the string is empty (length is 0); otherwise, the result is `true` (length > 1).
`Object`	`true`.

Let's see some examples and verify their output:

```
function testTruthy(val){
    return val ? console.log('truthy') : console.log('falsy');
}

testTruthy(true); //true
testTruthy(false); //false
testTruthy(new Boolean(false)); //true (object is always true)

testTruthy(''); //false
testTruthy('Packt'); //true
testTruthy(new String('')); //true (object is always true)

testTruthy(1); //true
testTruthy(-1); //true
testTruthy(NaN); //false
testTruthy(new Number(NaN)); //true (object is always true)

testTruthy({}); //true (object is always true)

var obj = {name:'John'};
testTruthy(obj); //true
testTruthy(obj.name); //true
testTruthy(obj.age); //false (age does not exist)
```

The equals operators (== and ===)

The two equals operators supported by JavaScript can cause a little bit of confusion when working with them.

When using `==`, values can be considered equal even when they are of different types. This can be confusing even for a senior JavaScript developer. Let's analyze how `==` works using the following table:

Type(x)	Type(y)	Result
`null`	`undefined`	`true`
`undefined`	`null`	true
`Number`	`String`	`x == toNumber(y)`
`String`	`Number`	`toNumber(x) == y`
`Boolean`	`Any`	`toNumber(x) == y`
`Any`	`Boolean`	`x == toNumber(y)`
`String` or `Number`	`Object`	`x == toPrimitive(y)`
`Object`	`String` or `Number`	`toPrimitive(x) == y`

If *x* and *y* are the same type, then JavaScript will use the equals method to compare the two values or objects. Any other combination that is not listed in the table gives a false result.

The `toNumber` and `toPrimitive` methods are internal and evaluate the values according to the tables that follow.

The `toNumber` method is presented here:

Value type	Result
`undefined`	`NaN`.
`null`	+0.
`Boolean`	If the value is `true`, the result is 1; if the value is `false`, the result is +0.
`Number`	The value of the number.
`String`	This parses the string into a number. If the string consists of alphabetical characters, the result is NaN; if the string consists of numbers, it is transformed into a number.
`Object`	`toNumber(toPrimitive(value))`.

And `toPrimitive` is presented here:

Value type	Result
`Object`	If `valueOf` returns a primitive value, this returns the primitive value; otherwise, if `toString` returns a primitive value, this returns the primitive value; otherwise returns an error.

Let's verify the results of some examples. First, we know that the output of the following code is true (string length > 1):

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

Now, what about the following code? Let's see:

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

The output is false! Let's understand why:

1. First, it converts the Boolean value using `toNumber`, so we have `packt == 1`.
2. Then, it converts the string value using `toNumber`. As the string consists of alphabetical characters, it returns NaN, so we have `NaN == 1`, which is false.

And what about the following code? Let's see:

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

The output is also false! The following are the steps:

1. First, it converts the Boolean value using `toNumber`, so we have `packt == 0`.
2. Then, it converts the string value using `toNumber`. As the string consists of alphabetical characters, it returns NaN, so we have `NaN == 0`, which is false.

And what about the operator `===`? It is much easier. If we are comparing two values of different types, the result is always false. If they have the same type, they are compared according to the following table:

Type(x)	Values	Result
`Number`	x has the same value as y (but not NaN)	true
`String`	x and y are identical characters	true
`Boolean`	x and y are both true or both false	true
`Object`	x and y reference the same object	true

If x and y are different types, then the result is false.

Let's see some examples:

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

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

var person1 = {name:'John'};
var person2 = {name:'John'};
console.log(person1 === person2); //false, different objects
```

Control structures

JavaScript has a similar set of control structures as the C and Java languages. Conditional statements are supported by if...else and switch. Loops are supported by while, do...while, and for constructs.

Conditional statements

The first conditional statement we will take a look at is the if...else construct. There are a few ways we can use the if...else construct.

We can use the if statement if we want to execute a script only if the condition is true:

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

We can use the if...else statement if we want to execute a script if the condition is true or another script just in case the condition is false (else):

```
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);
}
```

The `if...else` statement can also be represented by a ternary operator. For example, take a look at the following `if...else` statement:

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

It can also be represented as follows:

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

And if we have several scripts, we can use `if...else` several times to execute different scripts based on different conditions:

```
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");
}
```

Finally, we have the `switch` statement. If the condition we are evaluating is the same as the previous one (however, it is being compared to different values), we can use the `switch` statement:

```
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");
}
```

One thing that is very important in a `switch` statement is the usage of `case` and `break` keywords. The `case` clause determines whether the value of `switch` is equal to the value of the `case` clause. The `break` statement stops the `switch` statement from executing the rest of the statement (otherwise, it will execute all the scripts from all `case` clauses below the matched case until a `break` statement is found in one of the `case` clauses). And finally, we have the `default` statement, which is executed by default if none of the case statements are `true` (or if the executed `case` statement does not have the `break` statement).

Loops

Loops are very often used when we work with arrays (which is the subject of the next chapter). Specifically, we will be using the `for` loop in our algorithms.

The `for` loop is exactly the same as in C and Java. It consists of a loop counter that is usually assigned a numeric value, then the variable is compared against another value (the script inside the `for` loop is executed while this condition is true), and then the numeric value is increased or decreased.

In the following example, we have a `for` loop. It outputs the value of `i` on the console while `i` is less than 10; `i` is initiated with `0`, so the following code will output the values 0 to 9:

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

The next loop construct we will look at is the `while` loop. The script inside the `while` loop is executed while the condition is true. In the following code, we have a variable, `i`, initiated with the value `0`, and we want the value of `i` to be outputted while `i` is less than 10 (or less than or equal to 9). The output will be the values from 0 to 9:

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

The `do...while` loop is very similar to the `while` loop. The only difference is that in the `while` loop, the condition is evaluated before executing the script, and in the `do...while` loop, the condition is evaluated after the script is executed. The `do...while` loop ensures that the script is executed at least once. The following code also outputs the values 0 to 9:

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

Functions

Functions are very important when working with JavaScript. We will also use functions a lot in our examples.

The following code demonstrates the basic syntax of a function. It does not have *arguments* or the `return` statement:

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

To call this code, we simply use the following call:

```
sayHello();
```

We can also pass arguments to a function. **Arguments** are variables with which a function is supposed to do something. The following code demonstrates how to use arguments with functions:

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

To use this function, we can use the following code:

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

You can use as many arguments as you like, as follows:

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

In this case, only the first argument is used by the function and the second one is ignored.

A function can also return a value, as follows:

```
function sum(num1, num2) {
    return num1 + num2;
}
```

This function calculates the sum of two given numbers and returns its result. We can use it as follows:

```
var result = sum(1,2);
output(result);
```

Object-oriented programming

JavaScript objects are very simple collections of name-value pairs. There are two ways of creating a simple object in JavaScript. The first way is as follows:

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

And the second way is as follows:

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

We can also create an object entirely as follows:

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

In **object-oriented programming** (**OOP**), an object is an instance of a class. A class defines the characteristics of the object. For our algorithms and data structures, we will create some classes that will represent them. This is how we can declare a class that represents a book:

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

To instantiate this class, we can use the following code:

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

Then, we can access its attributes and update them as follows:

```
console.log(book.title); //outputs the book title
book.title = 'new title'; //updates the value of the book title
console.log(book.title); //outputs the updated value
```

A class can also contain functions. We can declare and use a function as the following code demonstrates:

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

We can declare functions directly inside the class definition as well:

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

In the prototype example, the `printTitle` function is going to be shared between all instances, and only one copy is going to be created. When we use class-based definition, as in the previous example, each instance will have its own copy of the functions. Using the prototype method saves memory and processing cost in regards to assigning the functions to the instance. However, you can only declare public functions and properties using the prototype method. With a class-based definition, you can declare private functions and properties and the other methods inside the class can also access them. You will notice in the examples of this book that we use a class-based definition (because we want to keep some properties and functions private). But, whenever possible, we should use the prototype method.

Now we have covered all the basic JavaScript concepts that are needed for us to start having some fun with data structures and algorithms!

Debugging and tools

Knowing how to program with JavaScript is important, but so is knowing how to debug your code. Debugging is very useful to help find bugs in your code, but it can also help you execute your code at a lower speed so you can see everything that is happening (the stack of methods called, variable assignment, and so on). It is highly recommended that you spend some time debugging the source code of this book to see every step of the algorithm (it might help you understand it better as well).

Both Firefox and Chrome support debugging. A great tutorial from Google that shows you how to use Google Developer Tools to debug JavaScript can be found at `https://developer.chrome.com/devtools/docs/javascript-debugging`.

You can use any text editor of your preference. But there are other great tools that can help you be more productive when working with JavaScript as well:

- **Aptana**: This is a free and open source IDE that supports JavaScript, CSS3, and HTML5, among other languages (`http://www.aptana.com/`).
- **WebStorm**: This is a very powerful JavaScript IDE with support for the latest web technologies and frameworks. It is a paid IDE, but you can download a 30-day trial version (`http://www.jetbrains.com/webstorm/`).
- **Sublime Text**: This is a lightweight text editor, and you can customize it by installing plugins. You can buy the license to support the development team, but you can also use it for free (the trial version does not expire) at `http://www.sublimetext.com/`.

Summary

In this chapter, we learned how to set up the development environment to be able to create or execute the examples in this book.

We also covered the basics of the JavaScript language that are needed prior to getting started with constructing the algorithms and data structures covered in this book.

In the next chapter, we will look at our first data structure, which is array, the most basic data structure that many languages support natively, including JavaScript.

2
Arrays

An **array** is the simplest memory data structure. For this reason, all programming languages have a built-in array data type. JavaScript also supports arrays natively, even though its first version was released without array support. In this chapter, we will dive into the array data structure and its capabilities.

An array stores a sequence of values that are all of the same data type. Although JavaScript allows us to create arrays with values from different data types, we will follow the best practices and consider that we cannot do that (most languages do not have this capability).

Why should we use arrays?

Let's consider that we need to store the average temperature of each month of the year of the city that we live in. We could use something like the following to store this information:

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

However, this is not the best approach. If we store the temperature for only 1 year, we could manage 12 variables. However, what if we need to store the average temperature for more than 1 year? Fortunately, that is why arrays were created, and we can easily represent the same information mentioned earlier as follows:

```
averageTemp[0] = 31.9;
averageTemp[1] = 35.3;
averageTemp[2] = 42.4;
averageTemp[3] = 52;
averageTemp[4] = 60.8;
```

We can also represent the `averageTemp` array graphically:
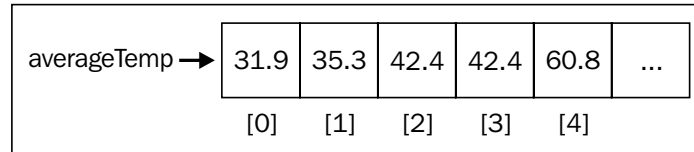
Creating and initializing arrays

Declaring, creating, and initializing an array in JavaScript is as simple, as follows:

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

We can simply declare and instantiate a new array by using the keyword `new` (line {1}). Also, using the keyword `new`, we can create a new array specifying the length of the array (line {2}). And a third option would be passing the array elements directly to its constructor (line {3}).

However, using the `new` keyword is not a best practice. If you want to create an array in JavaScript, simply use brackets (`[]`) like in the following example:

```
var daysOfWeek = [];
```

We can also initialize the array with some elements, as follows:

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

If we would like to know how many elements are in the array, we can use the `length` property. The following code will give an output of 7:

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

To access a particular position of the array, we also use brackets, passing the numeric position we would like to know the value of or assign a new value to. For example, let's say we would like to output all elements from the `daysOfWeek` array. To do so, we need to loop the array and print the elements, as follows:

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

Let's take a look at another example. Let's say that we want to find out the first 20 numbers of the Fibonacci sequence. The first two numbers of the Fibonacci sequence are 1 and 2, and each subsequent number is the sum of the previous two numbers:

```
var fibonacci = []; //{1}
fibonacci[1] = 1; //{2}
fibonacci[2] = 1; //{3}

for(var i = 3; i < 20; i++){
    fibonacci[i] = fibonacci[i-1] + fibonacci[i-2]; ////{4}
}

for(var i = 1; i<fibonacci.length; i++){ //{5}
    console.log(fibonacci[i]);              //{6}
}
```

So, in line {1}, we are declaring and creating an array. In lines {2} and {3}, we assign the first two numbers of the Fibonacci sequence to the second and third positions of the array (in JavaScript, the first position of the array is always referenced by 0, and as there is no 0 in the Fibonacci sequence, we skip it).

Then, all we have to do is create the third to the twentieth number of the sequence (as we know the first two numbers already). To do so, we can use a loop and assign the sum of the previous two positions of the array to the current position (line {4} — starting from index 3 of the array to the 19th index).

Then, to see the output (line {6}), we just need to loop the array from its first position to its length (line {5}).

> We can use `console.log` to output each index of the array (lines {5} and {6}) or we can also use `console.log(fibonacci)` to output the array itself. Most browsers have nice array representation in `console.log`.

If you would like to generate more than 20 numbers of the Fibonacci sequence, just change the number 20 to whatever number you like.

Adding and removing elements

Adding and removing elements from an array is not that difficult; however, it can be tricky. For the examples we will use in this section, let's consider we have the following `numbers` array initialized with numbers from 0 to 9:

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

If we want to add a new element to this array (for example, the number 10), all we have to do is reference the latest free position of the array and assign a value to it:

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

In JavaScript, an array is a mutable object. We can easily add new elements to it. The object will grow dynamically as we add new elements to it. In many other languages, such as C and Java, we need to determine the size of the array, and if we need to add more elements to the array, we need to create a completely new array; we cannot simply add new elements to it as we need them.

However, there is also a method called `push` that allows us to add new elements to the end of the array. We can add as many elements as we want as arguments to the `push` method:

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

As well if want it, can be done by using a for loop:
```
for (let i = 0; i <= numbers.length; i++) {
    if (i == numbers.length) { // when we reach the end
        numbers[i++] = 10;
    }
}
```
It does and Iteration in al the array once it reach the last position the condition is true and adds the value to the next iteration by doing i++

The output of the `numbers` array will be the numbers from 0 to 13.

Now, let's say we need to add a new element to the array, and we would like to insert it in the first position, not the last one. To do so, first we need to free the first position by shifting all the elements to the right. We can loop all the elements of the array starting from the last position + 1 (length) and shifting the previous element to the new position to finally assign the new value we want to the first position (-1):

```
for (var i=numbers.length; i>=0; i--){
    numbers[i] = numbers[i-1];
}
numbers[0] = -1;
```

We can represent this action with the following diagram:

0	1	2	3	...	11	12	13	...
[0]	[1]	[2]	[3]	[...]	[11]	[12]	[13]	[14]

[length = 14]

-1	0	1	2	...	10	11	12	13
[0]	[1]	[2]	[3]	[...]	[11]	[12]	[13]	[14]

[length = 15]
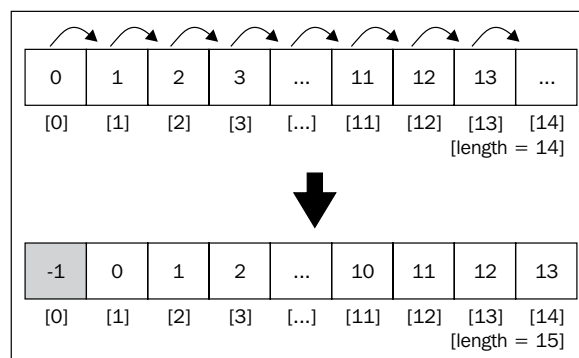
The JavaScript `array` class also has a method called `unshift`, which inserts the values passed in the method's arguments at the start of the array:

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

So, using the `unshift` method, we can add the value `-2` and then `-3` and `-4` to the beginning of the `numbers` array. The output of this array will be the numbers from -4 to 13.

So far, we have learned how to add values to the end and at the beginning of an array. Let's see how we can remove a value from an array.

To remove a value from the end of an array, we can use the `pop` method:

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

> The `push` and `pop` methods allow an array to emulate a basic **stack** data structure, which is the subject of the next chapter.

The output of our array will be the numbers from -4 to 12. The length of our array is 17.

To remove a value from the beginning of the array, we can use the following code:

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

We can represent the previous code using the following diagram:

We shifted all the elements one position to the left. However, the **length** of the array is still the same (**17**), meaning we still have an extra element in our arr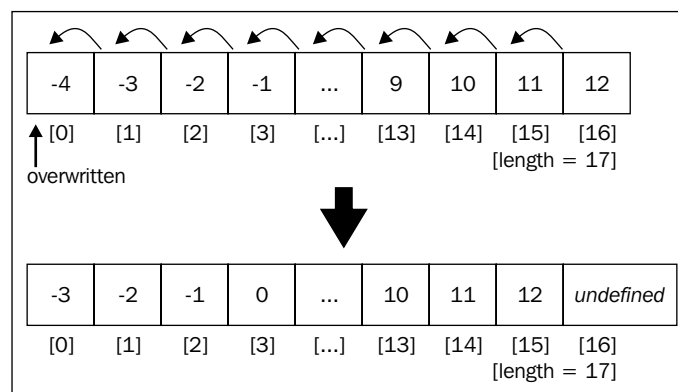ay (with an **undefined** value).The last time the code inside the loop is executed, `i+1` is a reference to a position that does not exist (in some languages, the code would throw an exception and we would have to end our loop at `numbers.length -1`).

As we can see, we have only overwritten the array's original values, and we did not remove the value for real (as the length of the array is still the same and we have this extra undefined element).

To actually remove an element from the beginning of the array, we can use the `shift` method as follows:

```
numbers.shift()
```

So, if we consider our array has the values -4 to 12 and a length of 17, after we execute the previous code, the array will contain the values -3 to 12 and have a length of 16.

This can be achive as well with a for loop:
for (var i=0; i<numbers.length - 1; i++){
 numbers[i] = numbers[i+1];
}
numbers.length = numbers.length - 1;
The loop itetares every single position and shift then by one, and to eleminated the last position it use length minus one

> The `shift` and `unshift` methods allow an array to emulate a basic **queue** data structure, which is the subject of *Chapter 4*, *Queues*.

So far, we have learned how to add elements at the end and at the beginning of an array, and we have also learned how to remove elements from the beginning and end of an array. What if we also want to add or remove elements from any particular position of our array? How can we do that?

We can use the `splice` method to remove an element from an array by simply specifying the position/index we would like to delete from and how many elements we would like to remove:

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

This code will remove three elements starting from index 5 of our array. This means `numbers[5]`, `numbers[6]`, and `numbers[7]` will be removed from the `numbers` array. The content of our array will be -3, -2, -1, 0, 1, 5, 6, 7, 8, 9, 10, 11, and 12 (as numbers 2, 3, and 4 have been removed).

Now, let's say we want to insert numbers 2 to 4 back into the array starting from position 5. We can again use the `splice` method to do this:

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

The first argument of the method is the index we want to remove or insert elements from. The second argument is the number of elements we want to remove (in this case, we do not want to remove any, so we pass the value 0 (zero)). And the third argument (and onwards) are the values we would like to insert into the array (elements 2, 3, and 4). The output will be the values from -3 to 12 again.

Finally, let's execute the following code:

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

The output will be the values from -3 to 12. This is because we are removing three elements starting from index 5 and we are also adding the elements 2, 3, and 4 starting at index 5.

Two-dimensional and multi-dimensional arrays

At the beginning of this chapter, we used the temperature measurement example. We will now use this example one more time. Let's consider that we need to measure the temperature hourly for a few days. Now that we already know that we can use an array to store the temperatures, we can easily write the following code to store the temperatures over two days:

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

However, this is not the best approach; we can write better code! We can use a matrix (two-dimensional array) to store this information, where each row will represent the day and each column will represent every hourly measurement of the temperature:

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

JavaScript only supports one-dimensional arrays; it does not support matrices. However, we can implement matrices or any multidimensional array by using an array of arrays, as in the previous code. The same code can also be written as follows:

```
//day 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;
//day 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] = 72;
```

In the previous code, we are specifying the value of each day and each hour separately. We can also represent this example in a diagram like the following:

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

Each row represents a day and each column represents an hour of the day (temperature).

If we would like to see the output of the matrix, we can create a generic function to log its output:

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

We need to loop through all the rows and all the columns. To do this, we need to use a nested `for` loop, where the variable `i` represents the rows and `j` represents the columns.

We can call the following code to see the output of the `averageTemp` matrix:

```
printMatrix(averageTemp);
```

We can also work with multidimensional arrays in JavaScript. For example, let's create a 3 x 3 matrix. Each cell will contain the sum of i (row) + j (column) + z (depth) of the matrix:

```
var matrix3x3x3 = [];
for (var i=0; i<3; i++){
    matrix3x3x3[i] = [];
    for (var j=0; j<3; j++){
        matrix3x3x3[i][j] = [];
        for (var z=0; z<3; z++){
            matrix3x3x3[i][j][z] = i+j+z;
        }
    }
}
```

It does not matter how many dimensions we have in the data structure; we need to loop each dimension to access the cell. We can represent a 3 x 3 x 3 matrix with a cube diagram, as follows:
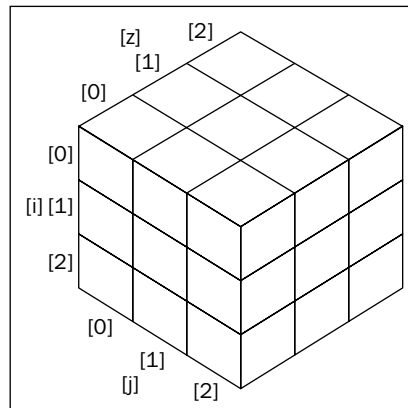
To output the content of this matrix, we can use the following code:

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

If we had a 3 x 3 x 3 x 3 matrix, we would have four nested `for` statements in our code, and so on.

References for JavaScript array methods

Arrays in JavaScript are modified objects, meaning that every array that we create has a few methods available to be used. JavaScript arrays are very interesting because they are very powerful and have more capabilities available than the primitive arrays of other languages. This means that we do not need to write basic capabilities ourselves, such as adding and removing elements in/from the middle of the data structure.

The following is a list of the core available methods in an array object. We have covered some methods already.

Method	Description
`concat`	Joins multiple arrays and returns a copy of the joined arrays
`every`	Calls a function for every element of the array until `false` is returned
`filter`	Creates an array with each element that evaluates to `true` in the function provided
`forEach`	Executes a specific function on each element of the array
`join`	Joins all the array elements into a string
`indexOf`	Searches the array for specific elements and returns its position
`lastIndexOf`	Returns the last item in the array that matches the search criteria and returns its position
`map`	Creates a new array with the result of calling the specified function on each element of the array
`reverse`	Reverses the array so the last items become the first and vice versa
`slice`	Returns a new array from the specified index
`some`	Passes each element through the supplied function until `true` is returned
`sort`	Sorts the array alphabetically or by the supplied function
`toString`	Returns the array as a string
`valueOf`	Like the method `toString`, this returns the array as a string

We have already covered the `push`, `pop`, `shift`, `unshift`, and `splice` methods. Let's take a look at these new ones. These methods will be very useful in the subsequent chapters of this book where we will code our own data structure and algorithms.

Joining multiple arrays

Consider a scenario where you have different arrays and you need to join all of them into a single array. We could iterate each array and add each element to the final array. Fortunately, JavaScript already has a method that can do that for us, named the `concat` method:

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

We can pass as many arrays and objects/elements to this array as we desire. The arrays will be concatenated to the specified array in the order the arguments are passed to the method. In this example, `zero` will be concatenated to `negativeNumbers`, and then `positiveNumbers` will be concatenated to the resulting array. The output of the `numbers` array will be the values -3, -2, -1, 0, 1, 2, and 3.

Iterator functions

Sometimes we need to iterate the elements of an array. We have learned that we can use a loop construct to do this, such as the `for` statement, as we saw in some previous examples.

JavaScript also has some built-in iterator methods that we can use with arrays. For the examples of this section, we will need an array and also a function. We will use an array with values from 1 to 15, and also a function that returns `true` if the number is a multiple of 2 (even), and `false` otherwise:

```
var isEven = function (x) {
    // returns true if x is a multiple of 2.
    console.log(x);
    return (x % 2 == 0) ? true : false;
};
var numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15];
```

`return (x % 2 == 0) ? true : false` can also be represented as `return (x % 2 == 0)`.

The first method we will take a look at is the `every` method. The `every` method will iterate each element of the array until the return of the function is `false`:

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

In this case, our first element of the `numbers` array is the number 1. 1 is not a multiple of 2 (it is an odd number), so the `isEven` function will return `false` and this will be the only time the function will be executed.

Next, we have the `some` method. It has the same behavior as the `every` method, however, the `some` method will iterate each element of the array until the return of the function is `true`:

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

In our case, the first even number of our `numbers` array is 2 (the second element). The first element that will be iterated is number 1; it will return `false`. Then, the second element that will be iterated is number 2, and it will return `true`—and the iteration will stop.

If we need the array to be completely iterated no matter what, we can use the `forEach` function. It has the same result as using a `for` loop with the function's code inside it:

```
numbers.forEach(function(x){
    console.log((x % 2 == 0));
});
```

Difference with a for loop:
for (let i = 0; i < numbers.length; i++) {
    console.log("for loop: ", (numbers[i] % 2 == 0));
    return (numbers[i] % 2 == 0) ? true : false;
}
In this case it breaks with the return evaluating the result

JavaScript also has two other iterator methods that return a new array with a result. The first one is the `map` method:

USING MAP() IN REACT:
.map((data, idx) => (
    <li key={idx} className="...">
        <p className="text-sm">Facts: {data.trim()}</p>

))

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

The `myMap` array will have the following values: `[false, true, false, true, false, true, false, true, false, true, false, true, false, true, false]`. It stores the result of the `isEven` function that was passed to the `map` method. This way, we can easily know whether a number is even or not. For example, `myMap[0]` returns `false` because 1 is not even and `myMap[1]` returns `true` because 2 is even.

We also have the `filter` method. It returns a new array with the elements that the function returned `true`:

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

In our case, the `evenNumbers` array will contain the elements that are multiples of 2: `[2, 4, 6, 8, 10, 12, 14]`.

Finally, we have the `reduce` method. The `reduce` method also receives a function with the following parameters: `previousValue`, `currentValue`, `index`, and `array`. We can use this function to return a value that will be added to an accumulator, which is going to be returned after the `reduce` method stops being executed. It can be very useful if we want to sum up all the values in an array, for example:

```
numbers.reduce(function(previous, current, index){
    return previous + current;
});
```

The output is going to be 120.

Searching and sorting

Throughout this book, we will learn how to write the most used searching and sorting algorithms. However, JavaScript also has a sorting method and a couple of searching methods available. Let's take a look at them.

First, let's take our `numbers` array and put the elements out of order (1, 2, 3, … 15 is already sorted). To do that, we can apply the `reverse` method, where the last item will be the first and vice versa:

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

So now, the output for the `numbers` array will be the following: `[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`. Then, we can apply the `sort` method:

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

However, if we output the array, the result will be `[1, 10, 11, 12, 13, 14, 15, 2, 3, 4, 5, 6, 7, 8, 9]`. This is not ordered correctly. This is because the `sort` method sorts the elements lexicographically and it assumes all the elements are strings.

We can also write our own comparison function. As our array has numeric elements, we can write the following code:

```
numbers.sort(function(a,b){
    return a-b;
});
```

This code will return a negative number if `b` is bigger than `a`, a positive number if `a` is bigger than `b`, and zero if they are equal. This means that if a negative value is returned, it implies that `a` is smaller than `b`, which is further used by the `sort` function to arrange the elements.

The previous code can be represented by the following code as well:

```
function compare(a, b) {
    if (a < b) {
        return -1;
    }
    if (a > b) {
        return 1;
    }
    // a must be equal to b
    return 0;
}

numbers.sort(compare);
```

This is because the sort function from the JavaScript `Array` class can receive a parameter called `compareFunction`, which will be responsible for sorting the array. In our example, we declared a function that will be responsible for comparing the elements of the array, resulting in an array sorted in ascending order.

Custom sorting

We can sort an array with any type of object in it, and we can also create a `compareFunction` to compare the elements as we need to. For example, suppose we have an object, `Person`, with name and age and we want to sort the array based on the age of the person; we can use the following code:

```
var friends = [
    {name: 'John', age: 30},
    {name: 'Ana', age: 20},
    {name: 'Chris', age: 25}
];

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));
```

In this case, the output from the previous code will be Ana (20), Chris (25), and John (30).

Sorting strings

Suppose we have the following array:

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

What do you think is going to be the output? The answer is as follows:

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

Why does `ana` come after `John`, since "a" comes first in the alphabet? The answer is because JavaScript compares each character according to its ASCII value. For example, A, J, a, and j have the decimal ASCII values of A: 65, J: 74, a: 97, and j: 106.

Therefore, J has a lower value than a, and because of this, it comes first in the alphabet.

> For more information about the ASCII table, please visit http://www.asciitable.com/.

Now, if we pass a `compareFunction` that contains the code to ignore the case of the letter, we will have the output `["Ana", "ana", "John", "john"]`, as follows:

```
names.sort(function(a, b){
    if (a.toLowerCase() < b.toLowerCase()){
        return -1
    }
    if (a.toLowerCase() > b.toLowerCase()){
        return 1
    }
    return 0;
});
```

For accented characters, we can use the `localeCompare` method as well:

```
var names2 = ['Maève', 'Maeve'];
console.log(names2.sort(function(a, b){
    return a.localeCompare(b);
}));
```

And the output will be `["Maeve", "Maève"]`.

Searching

We have two options for searching: the `indexOf` method, which returns the index of the first element that matches the argument passed, and `lastIndexOf`, which returns the index of the last element found that matches the argument passed. Let's go back to the `numbers` array we were using before:

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

In the previous example, the output in the console will be `9` for the first line and `-1` (because it does not exist in our array) for the second line.

We get the same result with the following code:

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

We added a new element with a value of 10, so the second line will output `15` (our array now has values from 1 to 15 + 10), and the third line outputs `-1` (because the element 100 does not exist in our array).

Outputting the array into a string

Finally, we come to the final two methods: `toString` and `join`.

If we would like to output all the elements of the array into a single string, we can use the `toString` method as follows:

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

This will output the values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, and 10 to the console.

If we would like to separate the elements by a different separator, such as `-`, we can use the `join` method to do just that:

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

The output will be as follows:

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

This can be useful if we need to send the array's content to a server or to be decoded (and then, knowing the separator, it is easy to decode).

> There are some great resources that you can use to boost your knowledge about arrays and their methods:
>
> - The first one is the arrays page from w3schools at `http://www.w3schools.com/js/js_arrays.asp`
> - The second one is the array methods page from w3schools at `http://www.w3schools.com/js/js_array_methods.asp`
> - Mozilla also has a great page about arrays and their methods with great examples at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array` (`http://goo.gl/vu1diT`)
> - There are also great libraries that are very useful when working with arrays in JavaScript projects:
>   - The Underscore library: `http://underscorejs.org/`
>   - The Lo-Dash library: `http://lodash.com/`

Summary

In this chapter, we covered the most used data structure: arrays. We learned how to declare, initialize, and assign values as well as add and remove elements. We also learned about two-dimensional and multi-dimensional arrays as well as the main methods of an array, which will be very useful when we start creating our own algorithms in later chapters.

In the next chapter, we will learn about stacks, an array with a special behavior.

3
Stacks

You learned in the previous chapter how to create and use arrays, which are the most common type of data structure in Computer Science. As you learned, we can add and remove elements from an array at any index desired. However, sometimes we need some form of data structure where we have more control over adding and removing items. There are two data structures that have some similarities to arrays but give us more control over the addition and removal of elements. These data structures are stacks and queues. In this chapter, we will cover stacks.

A **stack** is an ordered collection of items that follows the **LIFO** (short for **Last In First Out**) principle. The addition of new items or the removal of existing items takes place at the same end. The end of the stack is known as the top and the opposite is known as the base. The newest elements are near the top, and the oldest elements are near the base.

We have several examples of stacks in real life, for example, a pile of books, as we can see in the following image, or a stack of trays from a cafeteria or food court:

A stack is also used by compilers in programming languages and by computer memory to store variables and method calls.

Creating a stack

We are going to create our own class to represent a stack. Let's start from the basics and declare our class:

```
function Stack() {
    //properties and methods go here
}
```

First, we need a data structure that will store the elements of the stack. We can use an array to do this:

```
var items = [];
```

Next, we need to declare the methods available for our stack:

- `push(element(s))`: This adds a new item (or several items) to the top of the stack.
- `pop()`: This removes the top item from the stack. It also returns the removed element.
- `peek()`: This returns the top element from the stack. The stack is not modified (it does not remove the element; it only returns the element for information purposes).
- `isEmpty()`: This returns `true` if the stack does not contain any elements and `false` if the size of the stack is bigger than 0.
- `clear()`: This removes all the elements of the stack.
- `size()`: This returns how many elements the stack contains. It is similar to the `length` property of an array.

The first method we will implement is the `push` method. This method will be responsible for adding new elements to the stack with one very important detail: we can only add new items to the top of the stack, meaning at the end of the stack. The `push` method is represented as follows:

```
this.push = function(element){
    items.push(element);
};
```

As we are using an array to store the elements of the stack, we can use the `push` method from the JavaScript `array` class that we covered in the previous chapter.

Next, we are going to implement the `pop` method. This method will be responsible for removing the items from the stack. As the stack uses the LIFO principle, the last item that we added is the one that is removed. For this reason, we can use the `pop` method from the JavaScript `array` class that we also covered in the previous chapter. The `pop` method is represented as follows:

```
this.pop = function(){
    return items.pop();
};
```

With the `push` and `pop` methods being the only methods available for adding and removing items from the stack, the LIFO principle will apply to our own `Stack` class.

Now, let's implement some additional helper methods for our class. If we would like to know what the last item added to our stack was, we can use the `peek` method. This method will return the item from the top of the stack:

```
this.peek = function(){
    return items[items.length-1];
};
```

As we are using an array to store the items internally, we can obtain the last item from an array using `length - 1` as follows:

	top	
[2]	100	
[0]	72	
[1]	81	base
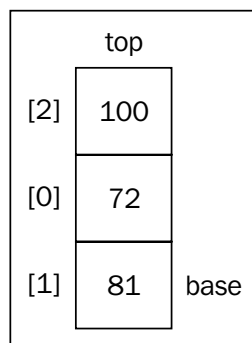
For example, in the previous diagram, we have a stack with three items; therefore, the `length` of the internal array is 3. The last position used in the internal array is 2. As a result, the `length - 1` (3 - 1) is 2!

The next method is the `isEmpty` method, which returns `true` if the stack is empty (no item has been added) and `false` otherwise:

```
this.isEmpty = function(){
    return items.length == 0;
};
```

Using the `isEmpty` method, we can simply verify whether the length of the internal array is 0.

Similar to the `length` property from the `array` class, we can also implement `length` for our `Stack` class. For collections, we usually use the term "size" instead of "length." And again, as we are using an array to store the items internally, we can simply return its length:

```
this.size = function(){
    return items.length;
};
```

Finally, we are going to implement the `clear` method. The `clear` method simply empties the stack, removing all its elements. The simplest way of implementing this method is as follows:

```
this.clear = function(){
    items = [];
};
```

An alternative implementation would be calling the `pop` method until the stack is empty.

And we are done! Our `Stack` class is implemented. Just to make our lives easier during the examples, to help us inspect the contents of our stack, let's implement a helper method called `print` that is going to output the content of the stack on the console:

```
this.print = function(){
    console.log(items.toString());
};
```

And now we are really done!

The complete Stack class

Let's take a look at how our `Stack` class looks after its full implementation:

```
function Stack() {

    var items = [];

    this.push = function(element){
        items.push(element);
    };

    this.pop = function(){
        return items.pop();
    };

    this.peek = function(){
        return items[items.length-1];
    };

    this.isEmpty = function(){
        return items.length == 0;
    };

    this.size = function(){
        return items.length;
    };

    this.clear = function(){
        items = [];
    };

    this.print = function(){
        console.log(items.toString());
    };
}
```

Using the Stack class

Before we dive into some examples, we need to learn how to use the `Stack` class.

The first thing we need to do is instantiate the `Stack` class we just created. Next, we can verify whether it is empty (the output is `true` because we have not added any elements to our stack yet):

```
var stack = new Stack();
console.log(stack.isEmpty()); //outputs true
```

Next, let's add some elements to it (let's push the numbers 5 and 8; you can add any element type to the stack):

```
stack.push(5);
stack.push(8);
```

If we call the `peek` method, the output will be the number 8 because it was the last element that was added to the stack:

```
console.log(stack.peek()); // outputs 8
```

Let's also add another element:

```
stack.push(11);
console.log(stack.size()); // outputs 3
console.log(stack.isEmpty()); //outputs false
```

We added the element 11. If we call the `size` method, it will give the output as 3, because we have three elements in our stack (5, 8, and 11). Also, if we call the `isEmpty` method, the output will be `false` (we have three elements in our stack). Finally, let's add another element:

```
stack.push(15);
```

The following diagram shows all the push operations we have executed so far and the current status of our stack:
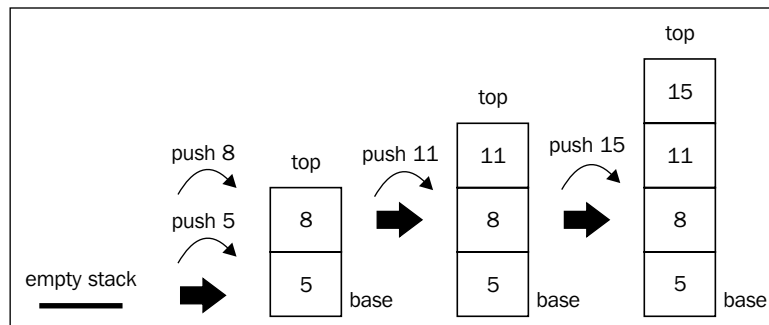
Next, let's remove two elements from the stack by calling the `pop` method twice:

```
stack.pop();
stack.pop();
console.log(stack.size()); // outputs 2
stack.print(); // outputs [5, 8]
```

Before we called the `pop` method twice, our stack had four elements in it. After the execution of the `pop` method two times, the stack now has only two elements: 5 and 8. The following diagram exemplifies the execution of the `pop` method:
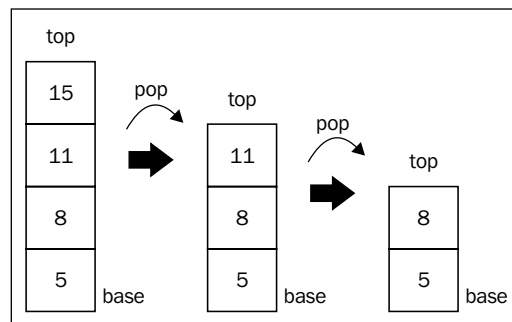

Decimal to binary

Now that we know how to use the `Stack` class, let's use it to solve some Computer Science problems.

You are probably already aware of the decimal base. However, binary representation is very important in Computer Science as everything in a computer is represented by binary digits (0 and 1). Without the ability to convert back and forth between decimal and binary numbers, it would be a little bit difficult to communicate with a computer.

To convert a decimal number to a binary representation, we can divide the number by 2 (binary is base 2 number system) until the division result is 0. As an example, we will convert the number 10 into binary digits:
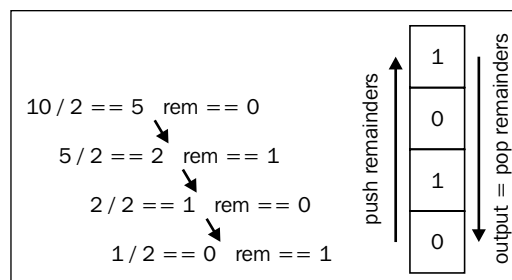
This conversion is one of the first things you learn in college (Computer Science classes). The following is our algorithm:

```
function divideBy2(decNumber){

    var remStack = new Stack(),
        rem,
        binaryString = '';

    while (decNumber > 0){ //{1}
        rem = Math.floor(decNumber % 2); //{2}
        remStack.push(rem); //{3}
        decNumber = Math.floor(decNumber / 2); //{4}
    }

    while (!remStack.isEmpty()){ //{5}
        binaryString += remStack.pop().toString();
    }

    return binaryString;
}
```

In this code, while the division result is not zero (line {1}), we get the remainder of the division (mod) and push it to the stack (lines {2} and {3}), and finally, we update the number that will be divided by 2 (line {4}). An important observation: JavaScript has a numeric data type, but it does not distinguish integers from floating points. For this reason, we need to use the Math.floor function to obtain only the integer value from the division operations. And finally, we pop the elements from the stack until it is empty, concatenating the elements that were removed from the stack into a string (line {5}).

We can try the previous algorithm and output its result on the console using the following code:

```
console.log(divideBy2(233));
console.log(divideBy2(10));
console.log(divideBy2(1000));
```

We can easily modify the previous algorithm to make it work as a converter from decimal to any base. Instead of dividing the decimal number by 2, we can pass the desired base as an argument to the method and use it in the divisions, as shown in the following algorithm:

```
function baseConverter(decNumber, base){

    var remStack = new Stack(),
        rem,
        baseString = '',
        digits = '0123456789ABCDEF'; //{6}

    while (decNumber > 0){
        rem = Math.floor(decNumber % base);
        remStack.push(rem);
        decNumber = Math.floor(decNumber / base);
    }

    while (!remStack.isEmpty()){
        baseString += digits[remStack.pop()]; //{7}
    }

    return baseString;
}
```

There is one more thing we need to change. In the conversion from decimal to binary, the remainders will be 0 or 1; in the conversion from decimal to octagonal, the remainders will be from 0 to 8, but in the conversion from decimal to hexadecimal, the remainders can be 0 to 8 plus the letters A to F (values 10 to 15). For this reason, we need to convert these values as well (lines {6} and {7}).

We can use the previous algorithm and output its result on the console as follows:

```
console.log(baseConverter(100345, 2));
console.log(baseConverter(100345, 8));
console.log(baseConverter(100345, 16));
```

> You will also find the balanced parentheses and the **Hanoi tower** examples when you download the source code of this book.

Summary

In this chapter, you learned about the stack data structure. We implemented our own algorithm that represents a stack and you learned how to add and remove elements from it using the `push` and `pop` methods. We also covered a very famous example of how to use a stack.

In the next chapter, you will learn about queues, which are very similar to stacks but use a different principle than LIFO.

4
Queues

We have already learned how stacks work. Queues are very similar, but instead of LIFO, they use a different principle that you will learn about in this chapter.

A **queue** is an ordered collection of items that follows the FIFO (which stands for **First In First Out**, also known as *first-come first-served*) principle. The addition of new elements in a queue is at the tail and the removal is from the front. The newest element added to the queue must wait at the end of the queue.

The most popular example of a queue in real life is the typical line that we form from time to time:
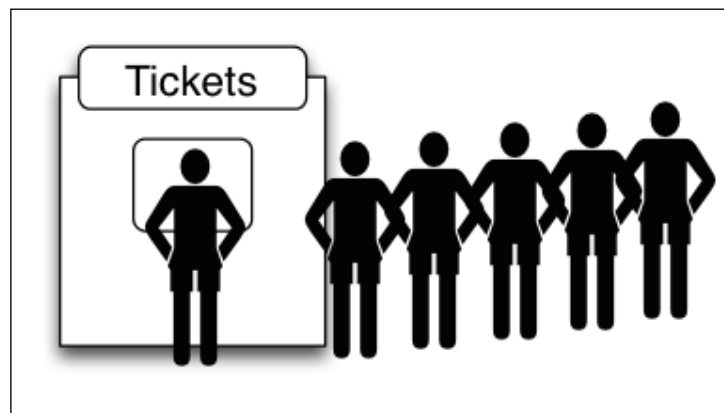
We have lines for movies, the cafeteria, and a checkout line at a grocery store, among other examples. The first person that is in the line is the first one that will be attended.

A very popular example in Computer Science is the printing line. Let's say we need to print five documents. We open each document and click on the print button. Each document will be sent to the print line. The first document that we asked to be printed is going to be printed first and so on, until all documents are printed.

Creating a queue

We are going to create our own class to represent a queue. Let's start from the basics and declare our class:

```
function Queue() {
    //properties and methods go here
}
```

First, we need a data structure that will store the elements of the queue. We can use an array to do it, just like we used for the `Stack` class in the previous chapter (you will notice the `Queue` and `Stack` class are very similar, just the principles for adding and removing the elements are different):

```
var items = [];
```

Next, we need to declare the methods available for a queue:

- `enqueue(element(s))`: This adds a new item (or several items) at the back of the queue.
- `dequeue()`: This removes the first item from the queue (the item that is in the front of the queue). It also returns the removed element.
- `front()`: This returns the first element from the queue, the first one added, and the first one that will be removed from the queue. The queue is not modified (it does not remove the element; it only returns the element for information purposes—very similar to the `peek` method from the `Stack` class).
- `isEmpty()`: This returns `true` if the queue does not contain any elements and `false` if the queue is bigger than 0.
- `size()`: This returns how many elements the queue contains. It is similar to the `length` property of the array.

The first method we will implement is the `enqueue` method. This method will be responsible for adding new elements to the queue with one very important detail; we can only add new items to the end of the queue:

```
this.enqueue = function(element){
    items.push(element);
};
```

As we are using an array to store the elements for the stack, we can use the `push` method from the JavaScript `array` class that we covered in *Chapter 2*, *Arrays*, and also in *Chapter 3*, *Stacks*.

Next, we are going to implement the `dequeue` method. This method will be responsible for removing the items from the queue. As the queue uses the FIFO principle, the first item that we added is the one that is removed. For this reason, we can use the `shift` method from the JavaScript `array` class that we also covered in *Chapter 2*, *Arrays*. If you do not remember, the `shift` method will remove the element that is stored at the index 0 (first position) of the array:

```
this.dequeue = function(){
    return items.shift();
};
```

With the `enqueue` and `dequeue` methods being the only methods available for adding and removing items from the queue, we assured the FIFO principle for our own `Queue` class.

Now, let's implement some additional helper methods for our class. If we want to know what the front item of our queue is, we can use the `front` method. This method will return the item from the front of the queue (index 0 of the array):

```
this.front = function(){
    return items[0];
};
```

The next method is the `isEmpty` method, which returns `true` if the queue is empty and `false` otherwise (note that this method is the same as the one in the `Stack` class):

```
this.isEmpty = function(){
    return items.length == 0;
};
```

For the `isEmpty` method, we can simply verify that the length of the internal array is 0.

Like the `length` property of the `array` class, we can also implement the same for our `Queue` class. The `size` method is also the same for the `Stack` class:

```
this.size = function(){
    return items.length;
};
```

And we are done! Our `Queue` class is implemented. Just like we did for the `Stack` class, we can also add the `print` method:

```
this.print = function(){
    console.log(items.toString());
};
```

And now we are really done!

The complete Queue class

Let's take a look how our Queue class looks like after its full implementation:

```
function Queue() {

    var items = [];

    this.enqueue = function(element){
        items.push(element);
    };

    this.dequeue = function(){
        return items.shift();
    };

    this.front = function(){
        return items[0];
    };

    this.isEmpty = function(){
        return items.length == 0;
    };

    this.clear = function(){
        items = [];
    };

    this.size = function(){
        return items.length;
    };

    this.print = function(){
        console.log(items.toString());
    };
}
```

> The Queue and Stack class are very similar. The only difference is the dequeue and front methods because of the difference between the FIFO and LIFO principles.

Using the Queue class

The first thing we need to do is instantiate the `Queue` class we just created. Next, we can verify that it is empty (the output is `true` because we have not added any elements to our queue yet):

```
var queue = new Queue();
console.log(queue.isEmpty()); //outputs true
```

Next, let's add some elements to it (let's enqueue the elements "John" and "Jack"—you can add any element type to the queue):

```
queue.enqueue("John");
queue.enqueue("Jack");
```

Let's add another element:

```
queue.enqueue("Camila");
```

Let's also execute some other commands:

```
queue.print();
console.log(queue.size()); //outputs 3
console.log(queue.isEmpty()); //outputs false
queue.dequeue();
queue.dequeue();
queue.print();
```

If we ask to print the contents of the queue, we will get `John`, `Jack`, and `Camila`. The size of the queue will be 3 because we have three elements queued on it (and it is also not going to be empty).

The following diagram exemplifies all the enqueue operations we executed so far and the current status of our queue:
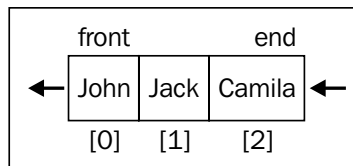
Next, we asked to dequeue two elements (the `dequeue` method is executed twice). The following diagram exemplifies the `dequeue` method execution:
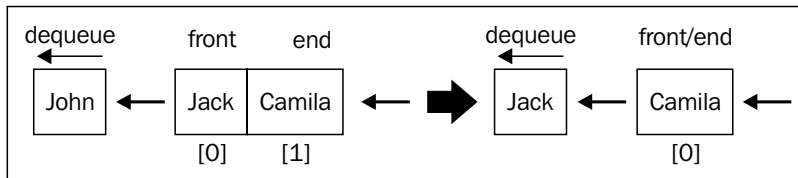
And when we finally ask to print the content of the queue again, we only have the element `Camila`. The first two elements queued, were dequeued; the final element queued to the data structure is the last one that will be dequeued from it. That being said, we are following the FIFO principle.

The priority queue

As queues are largely applied in Computer Science and also in our lives, there are some modified versions of the default queue we implemented in the previous topic.

One modified version is the **priority queue**. Elements are added and removed based on a priority. An example from real life is the boarding line at the airport. First class and business class passengers have priority over coach class passengers. In some countries, elderly people and pregnant women (or women with newborn children) also have priority for boarding over other passengers.

Another example from real life is the patient's waiting room from a hospital (emergency department). Patients that have a severe condition are seen by a doctor prior to patients with a less severe condition. Usually, a nurse will do the triage and assign a code to the patient depending on the condition severity.

There are two options when implementing a priority queue: you can set the priority and add the element at the correct position, or you can queue the elements as they are added to the queue and remove them according to the priority. For this example, we will add the elements at their correct position, so we can dequeue them by default:

```
function PriorityQueue() {

    var items = [];

    function QueueElement (element, priority){  // {1}
        this.element = element;
        this.priority = priority;
    }
```

```
    this.enqueue = function(element, priority){
        var queueElement = new QueueElement(element, priority);

        if (this.isEmpty()){
            items.push(queueElement); // {2}
        } else {
            var added = false;
            for (var i=0; i<items.length; i++){
                if (queueElement.priority <
items[i].priority){
                    items.splice(i,0,queueElement); // {3}
                    added = true;
                    break; // {4}
                }
            }
            if (!added){ //{5}
                items.push(queueElement);
            }
        }
    };

    //other methods - same as default Queue implementation
}
```

The difference between the implementation of the default `Queue` and `PriorityQueue` classes is that we need to create a special element (line {1}) to be added to `PriorityQueue`. This element contains the element we want to add to the queue (it can be any type) plus the priority on the queue.

If the queue is empty, we can simple enqueue the element (line {2}). If the queue is not empty, we need to compare its priority to the rest of the elements. When we find an item that has a higher priority than the element we are trying to add, then we insert the new element one position before (with this logic, we also respect the other elements with the same priority, but were added to the queue first). To do this, we can use the `splice` method from the JavaScript `array` class that you learned about in *Chapter 2, Arrays*. Once we find the element with bigger priority, we insert the new element (line {3}) and we stop looping the queue (line {4}). This way, our queue will also be sorted and organized by priority.

Also, if the priority we are adding is greater than any priority already added, we simply add to the end of the queue (line {5}):

```
var priorityQueue = new PriorityQueue();
priorityQueue.enqueue("John", 2);
priorityQueue.enqueue("Jack", 1);
priorityQueue.enqueue("Camila", 1);
priorityQueue.print();
```

In the previous code, we can see an example of how to use the `PriorityQueue` class. We can see each command result in the following diagram (a result of the previous code):
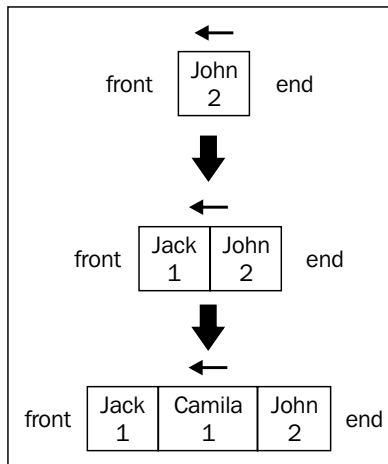
The first element that was added was `John` with priority 2. Because the queue was empty, this is the only element on it. Then, we added `Jack` with priority 1. As `Jack` has higher priority than `John`, it is the first element in the queue. Then, we added `Camila` also with priority 1. As `Camila` has the same priority as `Jack`, it will be inserted after `Jack` (as it was inserted first); and as `Camila` has a higher priority than `John`, it will be inserted before this element.

The priority queue we implemented is called a min priority queue, because we are adding the element with the lower value (1 has higher priority) to the front of the queue. There is also the max priority queue, which instead of adding the element with the lower value to front of the queue, it adds the element with greater value to the front of the queue.

The circular queue – Hot Potato

We also have another modified version of the queue implementation, which is the **circular queue**. An example of a circular queue is the *Hot Potato* game. In this game, children are organized in a circle, and they pass the hot potato to the neighbor as fast as they can. At a certain point of the game, the hot potato stops being passed around the circle of children and the child that has the hot potato is removed from the circle. This action is repeated until there is only one child left (the winner).

For this example, we will implement a simulation of the *Hot Potato* game:

```
function hotPotato (nameList, num){

    var queue = new Queue(); // {1}

    for (var i=0; i<nameList.length; i++){
        queue.enqueue(nameList[i]); // {2}
    }

    var eliminated = '';
    while (queue.size() > 1){
        for (var i=0; i<num; i++){
            queue.enqueue(queue.dequeue()); // {3}
        }
        eliminated = queue.dequeue();// {4}
        console.log(eliminated + ' was eliminated from the Hot Potato
game.');
    }

    return queue.dequeue();// {5}
}

var names = ['John','Jack','Camila','Ingrid','Carl'];
var winner = hotPotato(names, 7);
console.log('The winner is: ' + winner);
```

To implement a simulation of this game, we will use the Queue class we
implemented at the beginning of this chapter (line {1}). We will get a list of names
and queue all of them (line {2}). Given a number, we need to iterate the queue. We
will remove an item from the beginning of the queue and add it to the end of it (line
{3}) to simulate the hot potato (if you passed the hot potato to your neighbor, you
are not threatened to be eliminated right away). Once we reach the number, the
person that has the hot potato is eliminated (removed from the queue—line {4}).
When there is only one person left, this person is declared the winner (line {5}).

The output from the previous algorithm is:

```
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
```

This output is simulated in the following diagram:
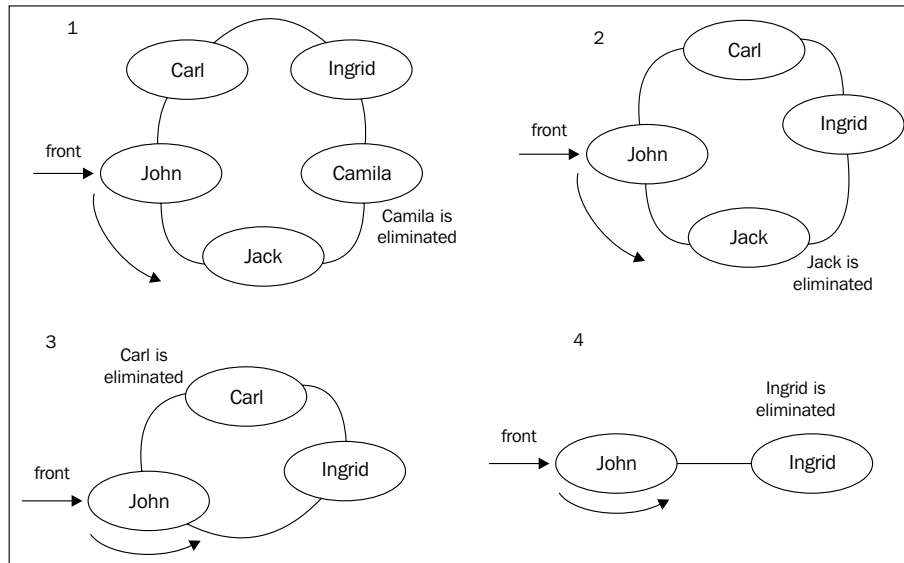
You can change the number passed to the `hotPotato` function to simulate different scenarios.

Summary

In this chapter, you learned about the queue data structure. We implemented our own algorithm that represents a queue; you learned how to add and remove elements from it using the `enqueue` and `dequeue` methods. We also covered two very famous special implementations of the queue: the priority queue and the circular queue (using the *Hot Potato* game implementation).

In the next chapter, you will learn about linked lists, a more complex data structure than the array.

5
Linked Lists

In *Chapter 2*, *Arrays*, we learned about the array data structure. An array (or we can also call it a list) is a very simple data structure that stores a sequence of data. In this chapter, you will learn how to implement and use a linked list, which is a dynamic data structure, meaning we can add or remove items from it at will and it will grow as needed.

Arrays (or lists) are probably the most common data structure used to store a collection of elements. As we mentioned before in this book, each language has its own implementation of arrays. This data structure is very convenient and provides a handy [] syntax to access its elements. However, this data structure has a disadvantage: the size of the array is fixed (in most languages) and inserting or removing items from the beginning or from the middle of the array is expensive because the elements need to be shifted over (even though we learned that JavaScript has methods from the array class that will do that for us, this is what happens behind the scenes as well).

Linked lists store a sequential collection of elements; but unlike arrays, in linked lists the elements are not placed contiguously in memory. Each element consists of a node that stores the element itself and also a reference (also known as a pointer or link) that points to the next element. The following diagram exemplifies the structure of a linked list:
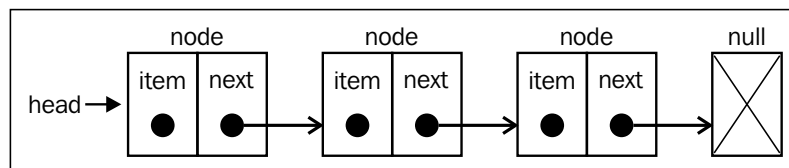
One of the benefits of a linked list over a conventional array is that we do not need to shift elements over when adding or removing elements. However, we need to use pointers when working with a linked list, and because of it, we need to pay some extra attention when implementing a linked list. Another detail in the array is that we can directly access any element at any position; with the linked list, if we want to access an element from the middle, we need to start from the beginning (**head**) and iterate the list until we find the desired element.

We have some real-world examples that can be exemplified as a linked list. The first example is a conga line. Each person is an element, and the hands would be the pointer that links to the next person in the conga line. You can add people to the line—you just need to find the spot where you want to add this person, decouple the connection, and then insert the new person and make the connection again.

Another example would be a scavenger hunt. You have a clue, and this clue is the pointer to the place where you can find the next clue. With this link, you go to the next place and get another clue that will lead to the next one. The only way to get a clue from the middle of the list is to follow the list from the beginning (from the first clue).

We have another example, which might be the most popular one used to exemplify linked lists, which is a train. A train consists of a series of vehicles (also known as wagons). Each vehicle or wagon is linked to each other. You can easily decouple a wagon, change its place, or add or remove it. The following figure demonstrates a train. Each wagon is the element of the list and the link between the wagons is the pointer:
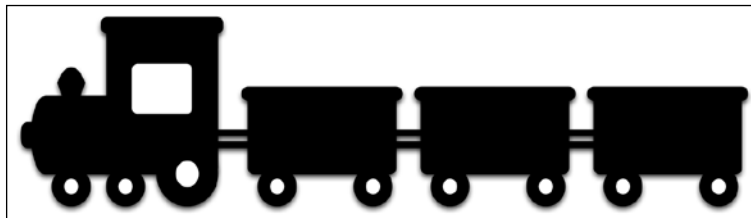
In this chapter, we will cover the linked list and also the doubly linked list. But let's start with the easiest data structure first.

Creating a linked list

Now that we understand what a linked list is, let's start implementing our data structure. This is the skeleton of our `LinkedList` class:

```
function LinkedList() {

    var Node = function(element){ // {1}
        this.element = element;
        this.next = null;
    };

    var length = 0; // {2}
    var head = null; // {3}

    this.append = function(element){};
    this.insert = function(position, element){};
    this.removeAt = function(position){};
    this.remove = function(element){};
    this.indexOf = function(element){};
    this.isEmpty = function() {};
    this.size = function() {};
    this.toString = function(){};
    this.print = function(){};
}
```

For the `LinkedList` data structure, we need a helper class called `Node` (line {1}). The `Node` class represents the item that we want to add to the list. It contains an `element` attribute, which is the value we want to add to the list, and a `next` attribute, which is the pointer that contains the link to the next node item of the list.

We also have the `length` property (line {2}) in the `LinkedList` class (internal/ private variable) that will store how many items we have on the list.

Another important note is that we need to store a reference for the first node as well. To do this, we can store this reference inside a variable that we will call `head` (line {3}).

Then, we have the methods of the `LinkedList` class. Let's see what each method will be responsible for before we implement each one:

* `append(element)`: This adds a new item to the end of the list.
* `insert(position, element)`: This inserts a new item at a specified position in the list.
* `remove(element)`: This removes an item from the list.
* `indexOf(element)`: This returns the index of the element in the list. If the element is not in the list, it returns -1.
* `removeAt(position)`: This removes an item from a specified position in the list.
* `isEmpty()`: This returns `true` if the linked list does not contain any elements and `false` if the size of the linked list is bigger than 0.
* `size()`: This returns how many elements the linked list contains. It is similar to the `length` property of the array.
* `toString()`: As the list uses a `Node` class as an item, we need to overwrite the default `toString` method inherited from the JavaScript object to output only the element values.

Appending elements to the end of the linked list

When adding an element to the end of a `LinkedList` object, there are two scenarios that can happen: the list is empty and we are adding its first element, or the list is not empty and we are appending elements to it.

Here, we have the implementation of the `append` method:

```
this.append = function(element){

    var node = new Node(element), //{1}
        current; //{2}

    if (head === null){ //first node on list //{3}
        head = node;

    } else {
```

```
        current = head; //{4}

        //loop the list until find last item
        while(current.next){
            current = current.next;
        }

        //get last item and assign next to node to make the link
        current.next = node; //{5}
    }

    length++; //update size of list //{6}
};
```
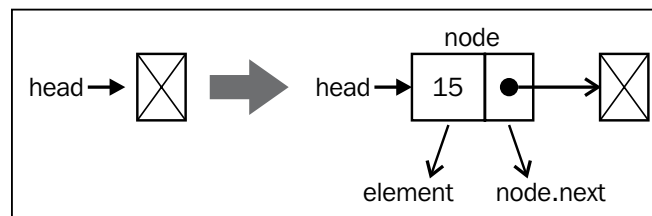
The first thing we need to do is to create the `Node` item passing `element` as its value (line `{1}`).

Let's implement the first scenario first: adding an element when the list is empty. When we create a `LinkedList` object, the `head` will be pointing to `null`:


If the `head` element is `null` (the list is empty — line `{3}`), it means we are adding the first element to the list. So, all we have to do is point the `head` element to the `node` element. The next `node` element will be `null` automatically (check the source code from the previous topic).

> The last node from the list will always have `null` as the next element.

So, we have covered the first scenario. Let's go to the second one, which is adding an element to the end of the list when it is not empty.

To add an element to the end of the list, we first need to find the last element. Remember that we only have a reference to the first element (line {4}), so we need to iterate through the list until we find the last item. To do so, we will need a variable that will point to the `current` item of the list (line {2}). When looping through the list, we know we reached its end when the `current.next` element is `null`. Then, all we have to do is link the current element's (which is the last one) next pointer to the node we want to add to the list (line {5}). The following diagram exemplifies this action:
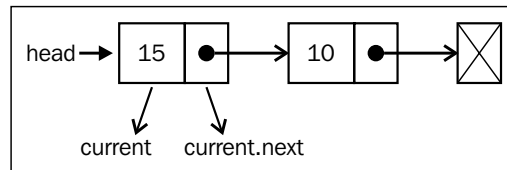
And as and when we create a `Node` element, its next pointer will always be `null`. We are OK with this because we know that it is going to be the last item of the list.

And of course, we cannot forget to increment the size of the list so that we can control it and easily get the list size (line {6}).

We can use and test the data structure we've created so far with the following code:

```
var list = new LinkedList();
list.append(15);
list.append(10);
```

Removing elements from the linked list

Now, let's see how we can remove elements from the `LinkedList` object. There are also two scenarios when removing elements: the first one is removing the first element, and the second one is removing any element but the first one. We are going to implement two `remove` methods: the first one is removing an element from a specified position and the second one is based on the element value (we will present the second `remove` method later).

Here is the implementation of the method that will remove an element based on a given position:

```
this.removeAt = function(position){

    //check for out-of-bounds values
    if (position > -1 && position < length){ // {1}
```

```
        var current = head, // {2}
            previous, // {3}
            index = 0; // {4}

        //removing first item
        if (position === 0){ // {5}
            head = current.next;
        } else {

            while (index++ < position){ // {6}

                previous = current;      // {7}
                current = current.next; // {8}
            }

            //link previous with current's next: skip it to remove
            previous.next = current.next; // {9}
        }

        length--; // {10}

        return current.element;

    } else {
        return null; // {11}
    }
};
```

We will dive into this code step by step. As the method is going to receive the position of the element that needs to be removed, we need to verify that the position value is a valid one (line {1}). A valid position would be from 0 (included) to the size of the list (size - 1, as the index starts from zero). If it is not a valid position, we return null (meaning no element was removed from the list).

Let's write the code for the first scenario: we want to remove the first element from the list (position === 0—line {5}). The following diagram exemplifies this:
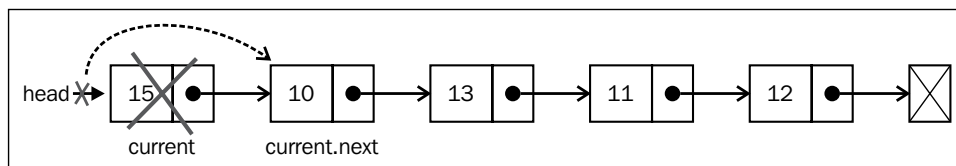
So, if we want to remove the first element, all we have to do is point `head` to the second element of the list. We will make a reference to the first element of the list using the `current` variable (line {2} — we will also use this to iterate the list, but we will get there in a minute). So, the `current` variable is a reference to the first element of the list. If we assign `head` to `current.next`, we will be removing the first element.

Now, let's say we want to remove the last item of the list or an item from the middle of the list. To do so, we need to iterate the list until the desired position (line {6} — we will use an `index` variable for internal control and increment) with one detail: the `current` variable will always make a reference to the current element of the list that we are looping through (line {8}). And we also need to make a reference to the element that comes before the current element (line {7}); we will name it `previous` (line {3}).

So, to remove the current element from the list, all we have to do is link `previous.next` with `current.next` (line {9}). This way, the current element will be lost in the computer memory and will be available to be cleaned by the garbage collector.

> To understand better how the JavaScript garbage collector works, please read `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management`.

Let's try to understand this better with some diagrams. First, let's consider that we want to remove the last element:
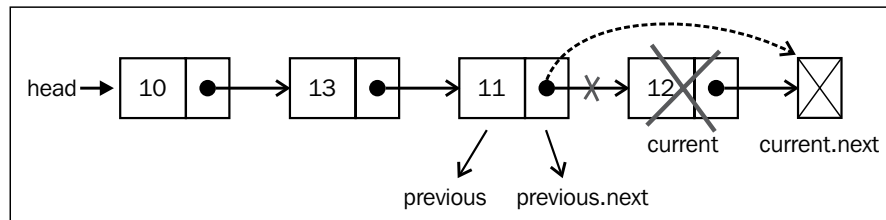
In the case of the last element, when we get off the loop in line {6}, the **current** variable will be a reference to the last element of the list (the one we want to remove). The **current.next** value will be `null` (because it is the last element). As we also keep a reference of the **previous** element (one element before the current one), **previous.next** will be pointing to **current**. So to remove **current**, all we have to do is change the value of **previous.next** to **current.next**.

Now let's see whether the same logic applies to an element from the middle of the list:
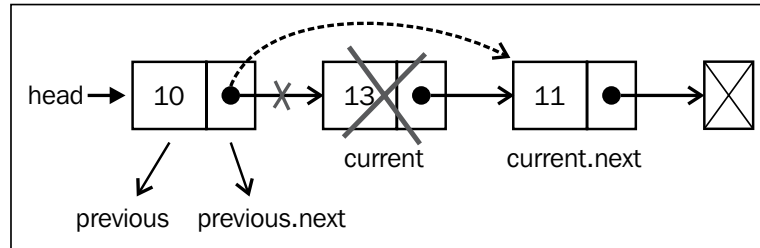
The **current** variable is a reference to the element we want to remove. The **previous** variable is a reference to the element that comes before the element we want to remove. So to remove the **current** element, all we need to do is link **previous.next** to **current.next**. So, our logic works for both cases.

Inserting an element at any position

Next, we are going to implement the insert method. This method provides you with the capability to insert an element at any position. Let's take a look at its implementation:

```
this.insert = function(position, element){

    //check for out-of-bounds values
    if (position >= 0 && position <= length){ //{1}

        var node = new Node(element),
            current = head,
            previous,
            index = 0;

        if (position === 0){ //add on first position

            node.next = current; //{2}
            head = node;

        } else {
            while (index++ < position){ //{3}
                previous = current;
                current = current.next;
            }
            node.next = current; //{4}
            previous.next = node; //{5}
        }
```

```
        length++; //update size of list

        return true;

    } else {
        return false; //{6}
    }
};
```

As we are handling positions, we need to check the out-of-bound values (line {1}, just like we did in the `remove` method). If it is out of bounds, we return the value `false` to indicate no item was added to the list (line {6}).

Now, we are going to handle the different scenarios. The first scenario is if in case we need to add an element at the beginning of the list, meaning the *first position*. The following diagram exemplifies this scenario:
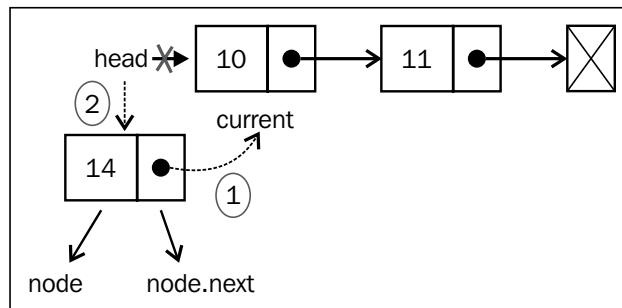
We have the **current** variable doing a reference to the first element of the list. What we need to is set the **node.next** value to **current** (the first element of the list). Now we have **head** and also **node.next** pointing to **current**. Next, all we have to do is change the **head** reference to **node** (line {2}) and we have a new element in the list.

Now let's handle the second scenario: adding an element in the middle or at the end of the list. First, we need to loop through the list until we reach the desired position (line {3}). When we get out of the loop, the `current` variable will be a reference to an element present after the position we would like to insert the new item, and `previous` will be a reference to an element present before the position we would like to insert the new item. In this case, we want to add the new item between `previous` and `current`. So, first we need to make a link between the new item (`node`) and current item (line {4}), and then we need to change the link between `previous` and `current`. We need `previous.next` to point to `node` (line {5}).

Let's see the code in action using a diagram to exemplify it:
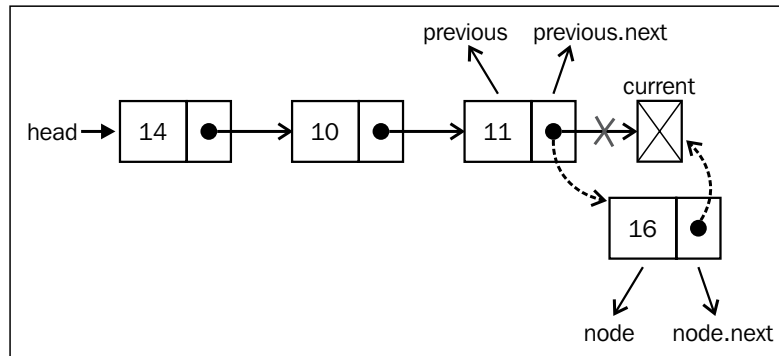
If we try to add a new element to the last position, **previous** will be a reference to the last item of the list and **current** will be null. In this case, **node.next** will point to **current** and **previous.next** will point to **node**, and we have a new item in the list.

Now, let's see how to add a new element in the middle of the list:
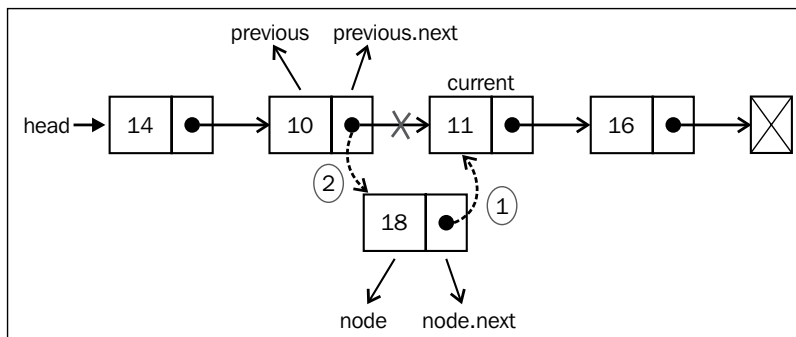
In this case, we are trying to insert the new item (**node**) between the **previous** and **current** elements. First, we need to set the value of the **node.next** pointer to **current**. Then, set the value of **previous.next** to **node**. And then we have a new item in the list!

> It is very important to have variables referencing the nodes we need to control so that we do not lose the link between the nodes. We could work with only one variable (`previous`), but it would be harder to control the links between the nodes. For this reason, it is better to declare an extra variable to help us with these references.

Implementing other methods

In this section we will learn how to implement the other methods from the
LinkedList class, such as toString, indexOf, isEmpty, and size.

The toString method

The toString method converts the LinkedList object into a string. The following
is the implementation of the toString method:

```
this.toString = function(){

    var current = head, //{1}
        string = '';    //{2}

    while (current) {    //{3}
        string = current.element; //{4}
        current = current.next;   //{5}
    }
    return string;              //{6}
};
```

First, to iterate through all elements of the list, we need a starting point, which is
head. We will use the current variable as our index (line {1}) and navigate through
the list. We also need to initialize the variable that we will be using to concatenate
the elements' values (line {2}).

Next, we iterate each element of the list (line {3}). We are going to use current to
check whether there is an element (if the list is empty or we reach the next of last
element of the list (null), the code inside the while loop will not be executed).
Then, we get the element's content and concatenate it to our string (line {4}).
And finally, we iterate the next element (line {5}).

And at last, we return the string with the list's content (list {6}).

The indexOf method

The next method we will implement is the indexOf method. The indexOf method
receives the value of an element and returns the position of this element if it is
found. Otherwise, it returns -1.

Let's take a look at its implementation:

```
this.indexOf = function(element){

    var current = head,  //{1}
        index = -1;

    while (current) {    //{2}
        if (element === current.element) {
            return index;        //{3}
        }
        index++;                 //{4}
        current = current.next; //{5}
    }

    return -1;
};
```

As always, we need a variable that will help us iterate through the list; this variable is `current` and its first value is the head (the first element of the list — we also need a variable to increment to count the position number, `index` (line {1})). Then, we iterate through the elements (line {2}) and check if the element we are looking for is the current one. If positive, we return its position (line {3}). If not, we continue counting (line {4}) and go to the next node of the list (line {5}).

The loop will not be executed if the list is empty or we reach the end of the list (`current = current.next` will be `null`). If we do not find the value, we return `-1`.

With this method implemented, we can implement other methods, such as the `remove` method:

```
this.remove = function(element){
    var index = this.indexOf(element);
    return this.removeAt(index);
};
```

We already have a method that removes an element at a given position (`removeAt`). Now that we have the `indexOf` method, if we pass the element's value, we can find its position and call the `removeAt` method passing the position that we found. It is very simple and it is also easier if we need to change the code from the `removeAt` method — it will be changed for both methods (this is what is nice about reusing code). This way, we do not need to maintain two methods to remove an item from the list, we need only one! Also, the bounds constraints will be checked by the `removeAt` method.

The isEmpty, size, and getHead methods

The `isEmpty` and `size` methods are the same as the ones we implemented for the classes implemented in previous chapter. But let's take a look at them anyway:

```
this.isEmpty = function() {
    return length === 0;
};
```

The `isEmpty` method returns `true` if there is no element in the list and `false` otherwise:

```
this.size = function() {
    return length;
};
```

The `size` method returns the `length` of the list. As a difference from the classes we implemented in earlier chapters, the `length` of the list is controlled internally as `LinkedList` is a class built from scratch.

And finally, we have the `getHead` method:

```
this.getHead = function(){
    return head;
};
```

The `head` variable is a private variable from the `LinkedList` class (meaning it can be accessed and changed only by the `LinkedList` instance, not outside of the instance). But if we need to iterate the list outside the class implementation, we need to provide a way to get the first element of the class.

Doubly linked lists

There are some different types of linked lists. In this section, we are going to cover the **doubly linked list**. The difference between a doubly linked list and a normal linked list is that in the linked list we make the link from one node to the next one only. In the doubly linked list, we have a double link: one for the next element and one for the previous element, as shown in the following diagram:
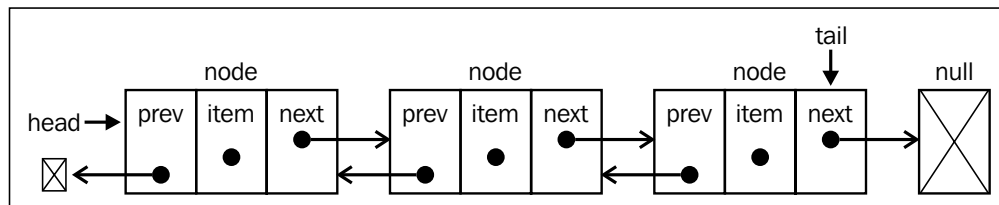
Let's get started with the changes that are needed to implement the
`DoublyLinkedList` class:

```
function DoublyLinkedList() {

    var Node = function(element){

        this.element = element;
        this.next = null;
        this.prev = null; //NEW
    };

    var length = 0;
    var head = null;
    var tail = null; //NEW

    //methods here
}
```

As we can see in this code, the differences between the `LinkedList` class and the
`DoublyLinkedList` class are marked by `NEW`. Inside the `Node` class we have the `prev`
attribute (a new pointer) and inside the `DoublyLinkedList` class we also have the
`tail` attribute to keep the reference of the last item of the list.

The doubly linked list provides us with two ways to iterate the list: from the
beginning to its end or vice versa. We can also go to the next element or the previous
element of a particular node. In the singly linked list, when you are iterating the list
and you miss the desired element, you need to go back to the beginning of the list
and start iterating it again. This is one of the advantages of the doubly linked list.

Inserting a new element at any position

Inserting a new item in the doubly linked list is very similar to the linked list.
The difference is that in the linked list we only control one pointer (`next`), and
in the doubly linked list we have to control both `next` and `prev` (previous).

Here we have the algorithm to insert a new element at any position:

```
this.insert = function(position, element){

        //check for out-of-bounds values
        if (position >= 0 && position <= length){

            var node = new Node(element),
```

```
                current = head,
                previous,
                index = 0;

        if (position === 0){ //add on first position

            if (!head){                //NEW {1}
                head = node;
                tail = node;
            } else {
                node.next = current;
                current.prev = node; //NEW {2}
                head = node;
            }
        } else  if (position === length) { //last item //NEW

            current = tail;     // {3}
            current.next = node;
            node.prev = current;
            tail = node;

        } else {
            while (index++ < position){ //{4}
                previous = current;
                current = current.next;
            }
            node.next = current; //{5}
            previous.next = node;

            current.prev = node; //NEW
            node.prev = previous; //NEW
        }

        length++; //update size of list

        return true;

    } else {
        return false;
    }
};
```

Let's analyze the first scenario: insert a new element at the first position of the list (beginning of the list). If the list is empty (line {1}), we simply need to point `head` and `tail` to the new node. If not, the `current` variable will be a reference to the first element of the list. As we did for the linked list, we set `node.next` to `current` and `head` will point to the `node` (it will be the first element of the list). The difference now is that we also need to set a value for the previous pointer of the elements. The `current.prev` pointer will be pointing to the new element (`node` — line {2}) instead of `null`. And the `node.prev` pointer is already `null`, so we do not need to update anything.

The following diagram demonstrates this process:
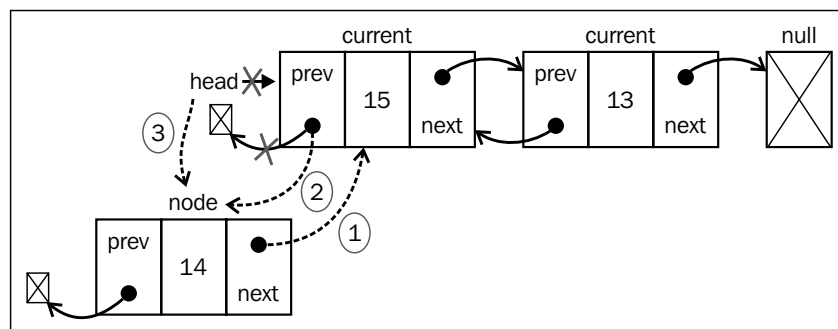
Now let's analyze this, just in case we want to add a new element as the last element of the list. As we are also controlling the pointer to the last element, this is a special case. The `current` variable will reference the last element (line {3}). Then, we start making the first link: `node.prev` will reference `current`. The `current.next` pointer (which is pointing to `null`) will point to `node` (`node.next` will be pointing to `null` already because of the constructor). Then, there is only one thing left to be done, which is updating `tail`, which will point to `node` instead of `current`. The following diagram demonstrates all these actions:
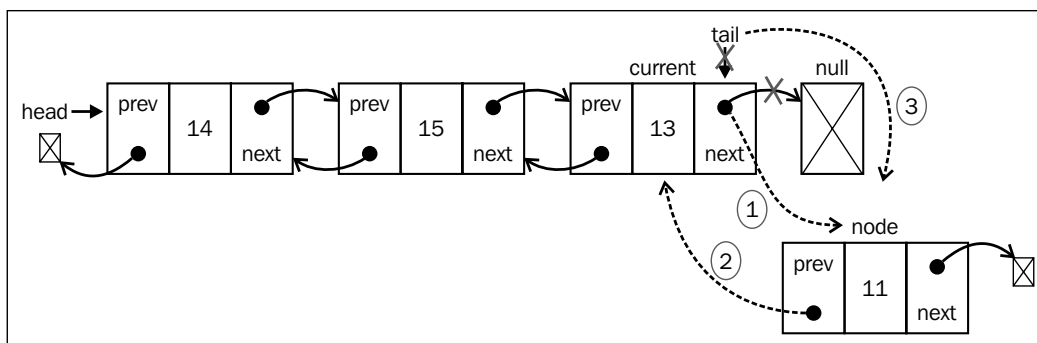
Then we have the third scenario: inserting a new element in the middle of the list. As we did for the previous methods, we will iterate the list until we get to the desired position (line {4}). We will be inserting the new element between the `current` and `previous` elements. First, `node.next` will point to `current` (line {5}) and `previous.next` will point to `node`, so that we do not lose the link between the nodes. Then, we need to fix all the links: `current.prev` will point to `node` and `node.prev` will point to `previous`. The following diagram exemplifies this:
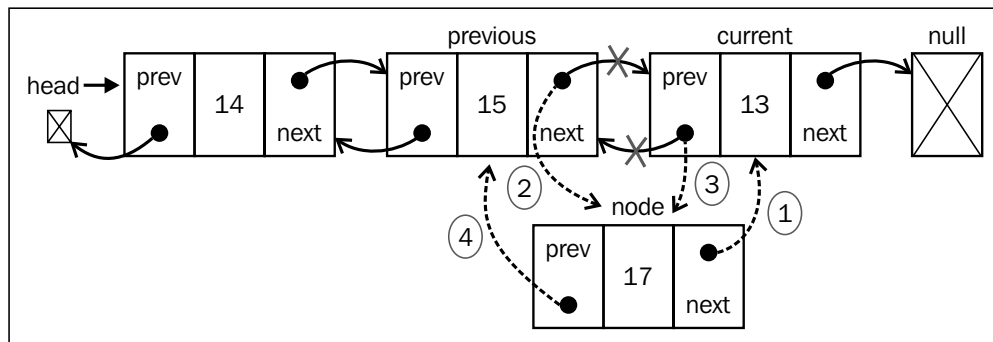
> We can do some improvements in both methods we implemented: `insert` and `remove`. In the case of a negative result, we could insert elements at the end of the list. There is also a performance improvement; for example, if `position` is greater than `length/2`, it is best to iterate from the end than start from the beginning (we will have to iterate fewer elements from the list).

Removing elements from any position

Removing elements from a doubly linked list is also very similar to a linked list. The only difference is that we need to set the previous pointer as well. Let's take a look at the implementation:

```
this.removeAt = function(position){

    //look for out-of-bounds values
    if (position > -1 && position < length){

        var current = head,
            previous,
            index = 0;
```

```
        //removing first item
        if (position === 0){

            head = current.next; // {1}

            //if there is only one item, update tail //NEW
            if (length === 1){ // {2}
                tail = null;
            } else {
                head.prev = null; // {3}
            }

        } else if (position === length-1){ //last item //NEW

            current = tail; // {4}
            tail = current.prev;
            tail.next = null;

        } else {

            while (index++ < position){ // {5}

                previous = current;
                current = current.next;
            }

            //link previous with current's next - skip it
            previous.next = current.next; // {6}
            current.next.prev = previous; //NEW
        }

        length--;

        return current.element;

    } else {
        return null;
    }
};
```

We need to handle three scenarios: removing an element from the beginning, from the middle, and the last element.

Let's take a look how to remove the first element. The `current` variable is a reference to the first element of the list, the one we want to remove. All we need to do is change the reference from `head`; instead of `current`, it will be the next element (`current.next`—line {1}). But we also need to update the `current.next` previous pointer (as the first element `prev` pointer is a reference to `null`). So, we change the reference of `head.prev` to `null` (line {3}—as `head` is also pointing to the new first element of the list or we can also use `current.next.prev`). As we also need to control the `tail` reference, we can check whether the element we are trying to remove is the first one and if positive, all we need to do is set `tail` to `null` as well (line {2}).

The following diagram illustrates the removal of the first element from a doubly linked list:
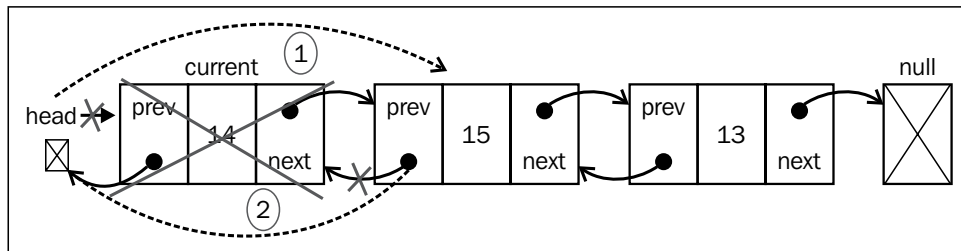
The next scenario removes an element from the last position. As we have the reference to the last element already (`tail`), we do not need to iterate the list to get to it. So, we can assign the `tail` reference to the `current` variable as well (line {4}). Next, we need to update the `tail` reference to the second-last element of the list (`current.prev` or `tail.prev` works as well). And now that `tail` is pointing to the second-last element, all we need to do is update the `next` pointer to `null` (`tail.next` = `null`). The following diagram demonstrates this action:
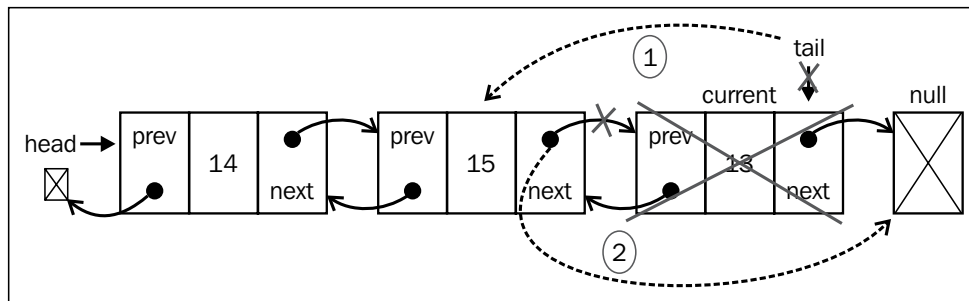
And the third and final scenario: removing an element from the middle of the list. First, we need to iterate until we get to the desired position (line {5}). The element we want to remove would be referenced by the `current` variable. So, to remove it, we can skip it in the list by updating the references of `previous.next` and `current.next.prev`. So, `previous.next` will point to `current.next` and `current.next.prev` will point to `previous`, as demonstrated by the following diagram:
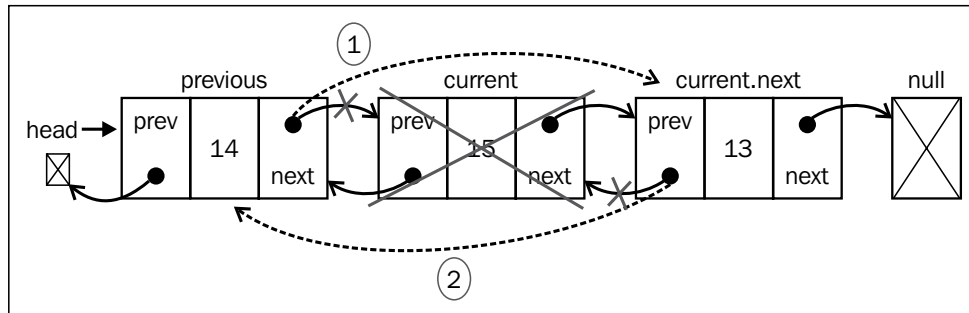

> To know the implementation of other methods of doubly linked lists, refer to the source code of the book. The download link of the source code is mentioned in the preface of the book.

Circular linked lists

A **circular linked list** can have only one reference direction (as the linked list) or a double reference as the doubly linked list. The only difference between the circular linked list and a linked list is that the last element's next (`tail.next`) pointer does not make a reference to `null`, but to the first element (`head`), as we can see in the following diagram:
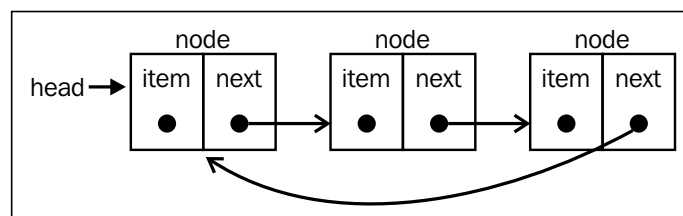
And a **doubly circular linked list** has `tail.next` pointing to the `head` element and `head.prev` pointing to the `tail` element:
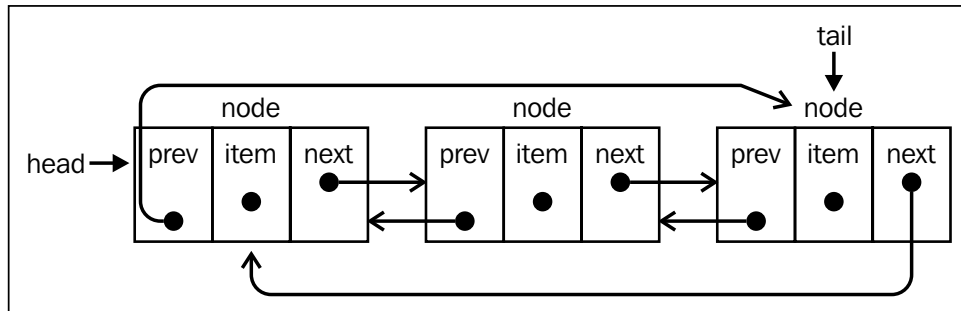
> We don't have the scope to cover the `CircularLinkedList` algorithm in this book (the source code is very similar to `LinkedList` and `DoublyLinkedList`). However, you can access the code by downloading this book's source code.

Summary

In this chapter, you learned about the linked list data structure and its variants, the doubly linked list and the circular linked list. You learned how to remove and add elements at any position and how to iterate through a linked list. You also learned that the most important advantage of a linked list over an array is that you can easily add and remove elements from a linked list without shifting over its elements. So, whenever you need to add and remove lots of elements, the best option would be a linked list instead of an array.

In the next chapter, you will learn about sets, the last sequential data structure that we will cover in this book.

6

Sets

So far, we have learned about sequential data structures such as arrays (lists), stacks, queues, and linked lists (and their variants). In this chapter, we will cover the data structure called a set.

A **set** is a collection of items that are unordered and consists of unique elements (meaning they cannot be repeated). This data structure uses the same math concept as of finite sets, but applied to a Computer Science data structure.

Let's take a look at the math concept of sets before we dive into the Computer Science implementation of it. In Mathematics, a set is a collection of distinct objects.

For example, we have a set of natural numbers, which consists of integer numbers greater than or equal to 0: *N = {0, 1, 2, 3, 4, 5, 6, …}*. The list of the objects within the set is surrounded by {} (curly braces).

There is also the null set concept. A set with no element is called a **null set** or **empty set**. For example, a set of prime numbers between 24 and 29. As there is no prime number (a natural number greater than 1 that has no positive divisors other than 1 and itself) between 24 and 29, the set will be empty. We represent an empty set as *{ }*.

You can also imagine a set as an array with no repeated elements and with no concept or order.

In Mathematics, a set also has some basic operations such as union, intersection, and difference. We will also cover these operations in this chapter.

Creating a set

The current implementation of JavaScript is based on **ECMAScript 5.1** (supported by modern browsers) published on June 2011. It contains the `Array` class implementation that we covered in earlier chapters. ECMAScript 6 (a work in progress, expected to be released in March 2015 at the time of writing this book) contains an implementation of the `Set` class.
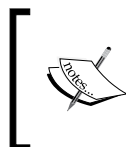
> You can see the details of the ECMAScript 6 `Set` class implementation at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set` (or `http://goo.gl/2li2a5`).

The class we are going to implement in this chapter is based on the `Set` implementation of ECMAScript 6.

This is the skeleton of our `Set` class:

```
function Set() {
    var items = {};
}
```

A very important detail is that *we are using an object to represent our set (*`items`*) instead of an array*. But we could also use an array to do this implementation. Let's use an object to implement things a little bit differently and learn new ways of implementing data structures that are similar. And also, objects in JavaScript do not allow you to have two different properties on the same key, which guarantees unique elements in our set.

Next, we need to declare the methods available for a set (we will try to simulate the same `Set` class implemented in ECMAScript 6):

- `add(value)`: This adds a new item to the set.
- `remove(value)`: This removes the value from the set.
- `has(value)`: This returns `true` if the value exists in the set and `false` otherwise.
- `clear()`: This removes all the items from the set.
- `size()`: This returns how many elements the set contains. It is similar to the `length` property of the array.
- `values()`: This returns an array of all the values of the set.

The has (value) method

The first method we will implement is the `has(value)` method. We will implement this method first because it will be used in other methods such as `add` and `remove`. We can see its implementation here:

```
this.has = function(value){
    return value in items;
};
```

As we are using an object to store all the values of the set, we can use JavaScript's `in` operator to verify that the given value is a property of the `items` object.

But there is a better way of implementing this method, which is as follows:

```
this.has = function(value){
    return items.hasOwnProperty(value);
};
```

All JavaScript objects have the `hasOwnProperty` method. This method returns a Boolean indicating whether the object has the specified property or not.

The add method

The next method we will implement is the `add` method:

```
this.add = function(value){
    if (!this.has(value)){
        items[value] = value; //{1}
        return true;
    }
    return false;
};
```

Given a `value`, we can check whether the `value` already exists in the set. If not, we add the `value` to the set (line {1}) and return `true` to indicate that the value was added. If the value already exists in the set, we simply return `false` to indicate that the value was not added.

> We are adding the value as key and value because it will help us search for the value if we store it as the key as well.

The remove and clear methods

Next, we will implement the `remove` method:

```
this.remove = function(value){
    if (this.has(value)){
        delete items[value]; //{2}
        return true;
    }
    return false;
};
```

In the `remove` method, we will verify that the given `value` exists in the set. If this is positive, we remove the `value` from the set (line {2}) and return `true` to indicate the value was removed; otherwise, we return `false`.

As we are using an object to store the `items` object of the set, we can simply use the `delete` operator to remove the property from the `items` object (line {2}).

To use the `Set` class, we can use the following code as an example:

```
var set = new Set();
set.add(1);
set.add(2);
```

> Just out of curiosity, if we output the `items` variable on the console (`console.log`) after executing the previous code, this will be the output in Google Chrome:
>
> `Object {1: 1, 2: 2}`
>
> As we can see, it is an object with two properties. The property name is the value we added to the set and its value as well.

If we want to remove all the values from the set, we can use the `clear` method:

```
this.clear = function(){
    items = {}; // {3}
};
```

All we need to do to reset the `items` object is assign it to an empty object again (line {3}). We could also iterate the set and remove all the values one by one using the `remove` method, but that is too much work as we have an easier way of doing it.

The size method

The next method we will implement is the `size` method (which returns how many items are in the set). There are three ways of implementing this method.

The first method is to use a `length` variable and control it whenever we use the `add` or `remove` method, as we used in the `LinkedList` class in the previous chapter.

In the second method, we use a built-in function from the built-in `Object` class in JavaScript (ECMAScript 5+):

```
this.size = function(){
    return Object.keys(items).length; //{4}
};
```

The `Object` class in JavaScript contains a method called `keys` that returns an array of all properties of a given object. In this case, we can use the `length` property of this array (line {4}) to return how many properties we have in the `items` object. This code will work only in modern browsers (such as IE9+, FF4+, Chrome5+, Opera12+, Safari5+, and so on).

The third method is to extract each property of the `items` object manually and count how many properties there are and return this number. This method will work in any browser and is the equivalent of the previous code:

```
this.sizeLegacy = function(){
    var count = 0;
    for(var prop in items) { //{5}
        if(items.hasOwnProperty(prop)) //{6}
            ++count; //{7}
    }
    return count;
};
```

So, first we iterate through all the properties of the `items` object (line {5}) and check whether that property is really a property (so we do not count it more than once—line {6}). If positive, we increment the `count` variable (line {7}) and at the end of the method we return this number.

> We cannot simply use the `for-in` statement and iterate through the properties of the `items` object and increment the `count` variable value. We also need to use the `has` method (to verify that the `items` object has that property) because the object's prototype contains additional properties for the object as well (properties are inherited from the base JavaScript `Object` class, but it still has properties of the object, which are not used in this data structure).

The values method

The same logic applies to the `values` method, using which we want to extract all the properties of the `items` object and return it as an array:

```
this.values = function(){
    return Object.keys(items);
};
```

This code will only work in modern browsers. As we are using Google Chrome and Firefox as testing browsers in this book, the code will work.

If we want code that can be executed in any browser, we can use the following code, which is equivalent to the previous code:

```
this.valuesLegacy = function(){
    var keys = [];
    for(var key in items){ //{7}
        keys.push(key); //{8}
    }
    return keys;
};
```

So, first we iterate through all the properties of the `items` object (line {7}), add them to an array (line {8}), and return this array.

Using the Set class

Now that we have finished implementing our data structure, let's see how we can use it. Let's give it a try and execute some commands to test our `Set` class:

```
var set = new Set();

set.add(1);
console.log(set.values()); //outputs ["1"]
console.log(set.has(1));   //outputs true
console.log(set.size());   //outputs 1

set.add(2);
console.log(set.values()); //outputs ["1", "2"]
console.log(set.has(2));   //true
console.log(set.size());   //2

set.remove(1);
console.log(set.values()); //outputs ["2"]

set.remove(2);
console.log(set.values()); //outputs []
```

So, now we have a very similar implementation of the `Set` class as in ECMAScript 6. As mentioned before, we could also have used an array instead of an object to store the elements. As we used arrays in *Chapter 2*, *Arrays*, *Chapter 3*, *Stacks*, and *Chapter 4*, *Queues*, it is nice to know there are different ways of implementing the same thing.

# Set operations

We can perform the following operations on sets:

- **Union**: Given two sets, this returns a new set with the elements from both given sets
- **Intersection**: Given two sets, this returns a new set with the elements that exist in both sets
- **Difference**: Given two sets, this returns a new set with all elements that exist in the first set and do not exist in the second set
- **Subset**: This confirms whether a given set is a subset of another set

Set union

The mathematic concept of **union** is that the union of sets *A* and *B*, denoted by $A \cup B$, is the set defined as:

$$A \cup B = \{x \mid x \in A \lor x \in B\}$$

This means that *x* (the element) exists in *A* or *x* exists in *B*. The following diagram exemplifies the union operation:
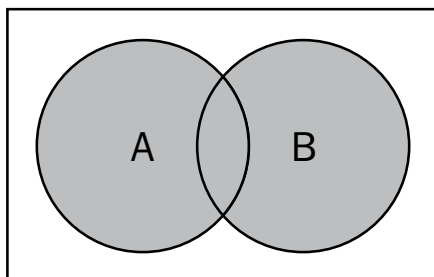
Now let's implement the `union` method in our `Set` class:

```
this.union = function(otherSet){
    var unionSet = new Set(); //{1}

    var values = this.values(); //{2}
    for (var i=0; i<values.length; i++){
        unionSet.add(values[i]);
    }

    values = otherSet.values(); //{3}
    for (var i=0; i<values.length; i++){
        unionSet.add(values[i]);
    }

    return unionSet;
};
```

First, we need to create a new set to represent the union of two sets (line {1}). Next, we get all the `values` from the first set (the current instance of the `Set` class), iterate through them, and add all the values to the set that represents the union (line {2}). Then, we do the exact same thing, but with the second set (line {3}). And at last, we return the result.

Let's test the previous code:

```
var setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);

var setB = new Set();
setB.add(3);
setB.add(4);
setB.add(5);
setB.add(6);

var unionAB = setA.union(setB);
console.log(unionAB.values());
```

The output will be `["1", "2", "3", "4", "5", "6"]`. Note that the element `3` is present in both *A* and *B*, and it appears only once in the result set.

Set intersection

The mathematic concept of **intersection** is that the intersection of sets *A* and *B*, denoted by $A \cap B$, is the set defined as:

$$A \cap B = \{x \mid x \in A \land x \in B\}$$

This means that *x* (the element) exists in *A* and *x* exists in *B*. The following diagram exemplifies the intersection operation:
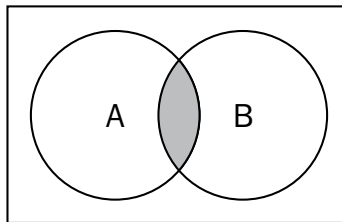
Now, let's implement the `intersection` method in our `Set` class:

```
this.intersection = function(otherSet){
    var intersectionSet = new Set(); //{1}

    var values = this.values();
    for (var i=0; i<values.length; i++){ //{2}
        if (otherSet.has(values[i])){     //{3}
            intersectionSet.add(values[i]); //{4}
        }
    }

    return intersectionSet;
}
```

For the `intersection` method, we need to find all elements from the current instance of the `Set` class that also exist in the given `Set` instance. So first, we create a new `Set` instance so that we can return it with the common elements (line {1}). Next, we iterate through all the `values` of the current instance of the `Set` class (line {2}) and we verify that the value exists in the `otherSet` instance as well (line {3}). We can use the `has` method that we implemented earlier in this chapter to verify that the element exists in the `Set` instance. Then, if the value exists in the other `Set` instance also, we add it to the created `intersectionSet` variable (line {4}) and return it.

Let's do some testing:

```
var setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);

var setB = new Set();
setB.add(2);
setB.add(3);
setB.add(4);

var intersectionAB = setA.intersection(setB);
console.log(intersectionAB.values());
```
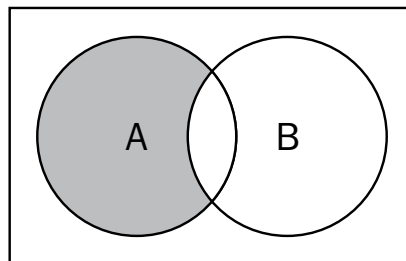
The output will be `["2", "3"]`, as the values 2 and 3 exist in both sets.

Set difference

The mathematic concept of **difference** is that the difference of set *A* from *B*, denoted by *A - B*, is the set defined as:

$$A\text{-}B = \{x \mid x \in A \wedge x \notin B\}$$

This means that *x* (the element) exists in *A* and *x* does not exist in *B*. The following diagram exemplifies the difference operation between sets *A* and *B*:

Now let's implement the `difference` method in our `Set` class:

```
this.difference = function(otherSet){
    var differenceSet = new Set(); //{1}

    var values = this.values();
    for (var i=0; i<values.length; i++){ //{2}
        if (!otherSet.has(values[i])){    //{3}
            differenceSet.add(values[i]); //{4}
        }
    }

    return differenceSet;
};
```

The `intersection` method will get all the `values` that exist in both sets. The `difference` method will get all the values that exist in *A* but not in *B*. So, the only difference in the implementation of the method is in line {3}. Instead of getting the values that also exist in the `otherSet` instance, we will get only the values that *do not exist*. Lines {1}, {2}, and {4} are exactly the same.

Let's do some testing (with the same sets we used in the `intersection` section):

```
var setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);

var setB = new Set();
setB.add(2);
setB.add(3);
setB.add(4);

var differenceAB = setA.difference(setB);
console.log(differenceAB.values());
```

The output will be `["1"]` because `1` is the only element that exists only in `setA`.

Subset

The last set operation we will cover is the subset. The mathematic concept of **subset** is that A is a subset of (or is included in) B, denoted by $A \subseteq B$, and is defined as:

$$\forall x \{x \in A \Rightarrow x \in B\}$$

This means that for every x (element) that exists in A it also *needs to exist* in B. The following diagram exemplifies when A is a subset of B and when it is not:
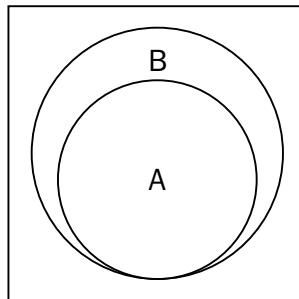
Now let's implement the `subset` method in our `Set` class:

```
this.subset = function(otherSet){

    if (this.size() > otherSet.size()){ //{1}
        return false;
    } else {
        var values = this.values();
        for (var i=0; i<values.length; i++){ //{2}
            if (!otherSet.has(values[i])){    //{3}
                return false; //{4}
            }
        }
        return true; //{5}
    }
};
```

The first verification that we need to do is check the size of the current instance of the `Set` class. If the current instance has more elements than the `otherSet` instance, it is not a subset (line {3}). A subset needs to have fewer or the same number of elements than the compared set.

Next, we will iterate through all the set elements (line {2}) and we will verify that the element also exists in `otherSet` (line {3}). If any element does not exist in `otherSet`, it means that it is not a subset, so we return `false` (line {4}). If all elements also exist in `otherSet`, line {4} will not be executed and then we will return `true` (line {5}).

Let's try the previous code:

```
var setA = new Set();
setA.add(1);
setA.add(2);

var setB = new Set();
setB.add(1);
setB.add(2);
setB.add(3);

var setC = new Set();
setC.add(2);
setC.add(3);
setC.add(4);

console.log(setA.subset(setB));
console.log(setA.subset(setC));
```

We have three sets: `setA` is a subset of `setB` (so the output is `true`), however, `setA` is not a subset of `setC` (`setC` only contains value `2` from `setA`, and not values `1` and `2`), so the output will be `false`.

Summary

In this chapter, we learned how to implement a `Set` class from scratch, which is similar to the `Set` class defined in the definition of ECMAScript 6. We also covered some methods that are not usually in other programming language implementations of the set data structure, such as union, intersection, difference, and subset. So, we implemented a very complete `Set` class compared to the current implementation of `Set` in other programming languages.

In the next chapter, we will cover hashes and dictionaries, which are non sequential data structures.

7
Dictionaries and Hashes

In the previous chapter, we learned about sets. In this chapter, we will continue our discussion about data structures that store unique values (non-repeated values) using dictionaries and hashes.

Sets, dictionaries, and hashes store unique values. In a set, we are interested in the value itself as the primary element. In a dictionary (or map), we store values as pairs as *[key, value]*. The same goes for hashes (they store values as pairs as *[key, value]*); however, the way that we implement these data structures is a little bit different, as we will see in this chapter.

Dictionaries

As you have learned, a set is a collection of distinct elements (non-repeated elements). A **dictionary** is used to store *[key, value]* pairs, where the key is used to find a particular element. The dictionary is very similar to a set; a set stores a *[key, key]* collection of elements, and a dictionary stores a *[key, value]* collection of elements. A dictionary is also known as a **map**.

In this chapter, we will cover some examples of the use of the dictionary data structure in the real world: a dictionary itself (the words and their definitions), and an address book.

Creating a dictionary

Similar to the Set class, ECMAScript 6 also contains an implementation of the Map class—also known as a dictionary.
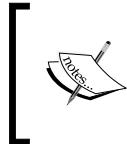
> You can check out the details of the ECMAScript 6 Map class implementation at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map` (or `http://goo.gl/dm8VP6`).

The class we are going to implement in this chapter is based on the Map implementation of ECMAScript 6. You will notice that it is very similar to the Set class (but instead of storing *[key, key]* pair, we will store *[key, value]* pair).

This is the skeleton of our Dictionary class:

```
function Dictionary(){
    var items = {};
}
```

Similar to the Set class, we will also store the elements in an Object instance instead of an array.

Next, we need to declare the methods available for a map/dictionary:

- set(key,value): This adds a new item to the dictionary.
- remove(key): This removes the value from the dictionary using the key.
- has(key): This returns true if the key exists in the dictionary and false otherwise.
- get(key): This returns a specific value searched by the key.
- clear(): This removes all the items from the dictionary.
- size(): This returns how many elements the dictionary contains. It is similar to the length property of the array.
- keys(): This returns an array of all the keys the dictionary contains.
- values(): This returns an array of all the values of the dictionary.

The has and set methods

The first method we will implement is the `has(key)` method. We will implement this method first because it will be used in other methods such as `set` and `remove`. We can see its implementation in the following code:

```
this.has = function(key){
    return key in items;
};
```

The implementation is exactly the same as what we did for the `Set` class. We are using the JavaScript `in` operator to verify that the `key` is a property of the `items` object.

The next method is the `set` method:

```
this.set = function(key, value){
    items[key] = value; //{1}
};
```

This receives a `key` and a `value` parameter. We simply set the value to the `key` property of the `items` object. This method can be used to add a new value or update an existing one.

The remove method

Next, we will implement the `remove` method. It is very similar to the `remove` method from the `Set` class; the only difference is that we first search for `key` (instead of `value`):

```
this.remove = function(key){
    if (this.has(key)){
        delete items[key];
        return true;
    }
    return false;
};
```

Then we use the JavaScript `remove` operator to remove the `key` attribute from the `items` object.

The get and values methods

If we want to search for a particular item from the dictionary and retrieve its value, we can use the following method:

```
this.get = function(key) {
    return this.has(key) ? items[key] : undefined;
};
```

The `get` method will first verify that the value that we would like to retrieve exists (by searching for `key`), and if the result is positive, its value is returned, if not, an `undefined` value is returned (remember that `undefined` is different from `null`—we covered this concept in *Chapter 1, Javascript – A Quick Overview*).

The next method is the `values` method. This method will be used to retrieve an array of all `values` instances present in the dictionary:

```
this.values = function(){
    var values = [];
    for (var k in items) { //{1}
        if (this.has(k)) {
            values.push(items[k]); //{2}
        }
    }
    return values;
};
```

First, we will iterate through all attributes from the `items` object (line {1}). Just to make sure the value exists, we will use the `has` function to verify that `key` really exists, and then we add its value to the `values` array (line {2}). At the end, we simply return all the values found.

> We cannot simply use the `for-in` statement and iterate through the properties of the `items` object. We also need to use the `has` method (to verify if the `items` object has that property) because the object's prototype contains additional properties of the object as well (properties are inherited from the base JavaScript `Object` class, but it still has properties of the object—which we are not interested in for this data structure).

The clear, size, keys, and getItems methods

The `clear`, `size`, and `keys` methods are exactly the same from the `Set` class. For this reason, we will not go through them again in this chapter.

Finally, just so that we can verify the output of the `items` property, let's implement a method called `getItems` that will return the `items` variable:

```
this.getItems = function(){
    return items;
}
```

Using the Dictionary class

First, we create an instance of the `Dictionary` class, and then we add three e-mails to it. We are going to use this `dictionary` instance to exemplify an e-mail address book.

Let's execute some code using the class we created:

```
var dictionary = new Dictionary();
dictionary.set('Gandalf', 'gandalf@email.com');
dictionary.set('John', 'johnsnow@email.com');
dictionary.set('Tyrion', 'tyrion@email.com');
```

If we execute the following code, we will get the output as `true`:

```
console.log(dictionary.has('Gandalf'));
```

The following code will output `3` because we added three elements to our dictionary instance:

```
console.log(dictionary.size());
```

Now, let's execute the following lines of code:

```
console.log(dictionary.keys());
console.log(dictionary.values());
console.log(dictionary.get('Tyrion'));
```

The output will be as follows, in the respective order:

```
["Gandalf", "John", "Tyrion"]
["gandalf@email.com", "johnsnow@email.com", "tyrion@email.com"]
tyrion@email.com
```

Finally, let's execute some more lines of code:

```
dictionary.remove('John');
```

Let's also execute the following ones:

```
console.log(dictionary.keys());
console.log(dictionary.values());
console.log(dictionary.getItems());
```

The output will be as follows:

```
["Gandalf", "Tyrion"]
["gandalf@email.com", "tyrion@email.com"]
Object {Gandalf: "gandalf@email.com", Tyrion: "tyrion@email.com"}
```

As we removed one element, the `dictionary` instance now contains only two elements. The highlighted line exemplifies how the `items` object is structured internally.

The hash table

In this section, you will learn about the `HashTable` class, also known as `HashMap`, a hash implementation of the `Dictionary` class.

Hashing consists of finding a value in a data structure in the shortest time possible. You have learned from previous chapters that if we would like to get a value from it (using a `get` method), we need to iterate through the structure until we find it. When we use a hash function, we already know which position the value is in, so we can simply retrieve it. A hash function is a function that given a key, and will return an address in the table where the value is.

For example, let's continue using the e-mail address book we used in the previous section. The hash function we will use is the most common one, called a "lose lose" hash function, where we simply sum up the ASCII values of each character of the key length.

Name/Key	Hash Function	Hash Value	Hash Table	
			[...]	
Gandalf	71 + 97 + 110 + 100 + 97 + 108 + 102	685	[399]	johnsnow@email.com
			[...]	
John	74 + 111 + 104 + 110	399	[645]	tyrion@email.com
Tyrion	84 + 121 + 114 +105 + 111 + 110	645	[...]	
			[685]	gandalf@email.com
			[...]	

Creating a hash table

We will use an array to represent our data structure to have a data structure very similar to the one we used in the diagram in the previous topic.

As usual, let's start with the skeleton of our class:

```
function HashTable() {
    var table = [];
}
```

Next, we need to add some methods to our class. We will implement three basic methods for every class:

- `put(key,value)`: This adds a new item to the hash table (or it can also update it)
- `remove(key)`: This removes the value from the hash table using the key
- `get(key)`: This returns a specific value searched by the key

The first method that we will implement before we implement these three methods is the hash function. This is a private method of the `HashTable` class:

```
var loseloseHashCode = function (key) {
    var hash = 0;                            //{1}
    for (var i = 0; i < key.length; i++) {   //{2}
        hash += key.charCodeAt(i);           //{3}
    }
    return hash % 37;                        //{4}
};
```

Given a `key` parameter, we will generate a number based on the sum of each char ASCII value that composes `key`. So, first we need a variable to store the sum (line {1}). Then, we will iterate through the `key` (line {2}) and add the ASCII value of the corresponding character value from the ASCII table to the `hash` variable (to do so, we can use the `charCodeAt` method from the JavaScript `String` class—line {3}). Finally, we return this `hash` value. To work with lower numbers, we will use the rest of the division (`mod`) of the hash number using an arbitrary number (line {4}).

> For more information about ASCII, please go to `http://www.asciitable.com/`.

Now that we have our hash function, we can implement the `put` method:

```
this.put = function (key, value) {
    var position = loseloseHashCode(key); //{5}
    console.log(position + ' - ' + key); //{6}
    table[position] = value; //{7}
};
```

First, for the given `key`, we need to find a position in the table using the hash function we created (line {5}). For information purposes, we will log the position on the console (line {6}). We can remove this line from the code as it is not necessary. Then, all we have to do is add the `value` parameter to `position`, which we found using the `hash` function (line {7}).

To retrieve a value from the `HashTable` instance is also simple. We will implement the `get` method for this purpose:

```
this.get = function (key) {
    return table[loseloseHashCode(key)];
};
```

First, we will retrieve the position of the given `key` using the hash function we created. This function will return the position of the value and all we have to do is access this position from the `table` array and return this value.

The last method we will implement is the `remove` method:

```
this.remove = function(key){
    table[loseloseHashCode (key)] = undefined;
};
```

To remove an element of the `HashTable` instance, we simply need to access the desired position (that we can get using the hash function) and assign the value `undefined` to it.

For the `HashTable` class, we do not need to remove the position from the `table` array as we did for the `ArrayList` class. As the elements will be distributed throughout the array, some positions will not be occupied by any value, having the undefined value by default. We also cannot remove the position itself from the array (this will shift the other elements), otherwise, next time we try to get or remove another existing element, the element will not be present in the position we get from the hash function.

Using the HashTable class

Let's test the `HashTable` class by executing some code:

```
var hash = new HashTable();
hash.put('Gandalf', 'gandalf@email.com');
hash.put('John', 'johnsnow@email.com');
hash.put('Tyrion', 'tyrion@email.com');
```

When we execute the previous code, we will get the following output on the console:

```
19 - Gandalf
29 - John
16 - Tyrion
```

The following diagram represents the `HashTable` data structure with these three elements in it:

Name/Key	Hash Value	Hash Table	
		[...]	
Gandalf	19	[16]	tyrion@email.com
John	29	[...]	
Tyrion	16	[19]	gandalf@email.com
		[...]	
		[29]	johnsnow@email.com
		[...]	
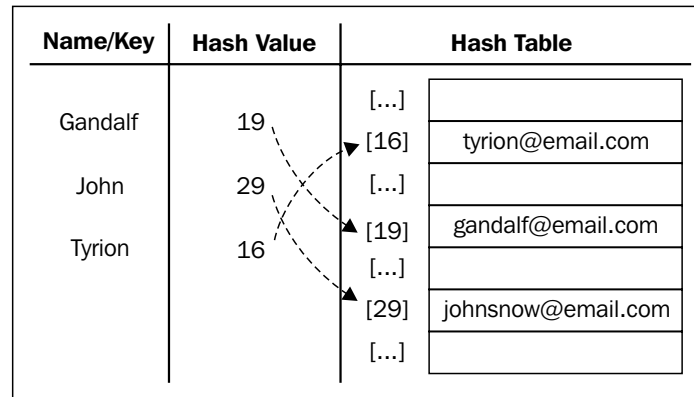
Now, let's test the `get` method:

```
console.log(hash.get('Gandalf'));
console.log(hash.get('Loiane'));
```

We will have the following output:

```
gandalf@email.com
undefined
```

As `Gandalf` is a key that exists in `HashTable`, the `get` method will return its value. As `Loiane` is not an existing key, when we try to access its position in the array (a position generated by the hash function) its value will be `undefined` (non-existent).

Next, let's try to remove `Gandalf` from `HashTable`:

```
hash.remove('Gandalf');
console.log(hash.get('Gandalf'));
```

The `hash.get('Gandalf')` method will give `undefined` as the output on the console, as `Gandalf` no longer exists in the table.

Hash table versus hash set

A hash table is the same thing as a hash map. We covered this data structure in this chapter.

In some programming languages, we also have the **hash set** implementation. The hash set data structure consists of a set, but to insert, remove, or get elements, we use a hash function. We can reuse all the code we implemented in this chapter for a hash set; the difference is that instead of adding a key-value pair, we will insert only the value, not the key. For example, we could use a hash set to store all the English words (without their definition). Similar to set, the hash set also stores only unique values, not repeated ones.

Handling collisions between hash tables

Sometimes, different keys can have the same hash value. We call it a collision, as we will try to set different values to the same position of the `HashTable` instance. For example, let's see what we get in the output with the following code:

```
var hash = new HashTable();
hash.put('Gandalf', 'gandalf@email.com');
hash.put('John', 'johnsnow@email.com');
hash.put('Tyrion', 'tyrion@email.com');
hash.put('Aaron', 'aaron@email.com');
hash.put('Donnie', 'donnie@email.com');
hash.put('Ana', 'ana@email.com');
hash.put('Jonathan', 'jonathan@email.com');
hash.put('Jamie', 'jamie@email.com');
hash.put('Sue', 'sue@email.com');
hash.put('Mindy', 'mindy@email.com');
hash.put('Paul', 'paul@email.com');
hash.put('Nathan', 'nathan@email.com');
```

The following will be the output:

```
19 - Gandalf
29 - John
16 - Tyrion
16 - Aaron
13 - Donnie
13 - Ana
5 - Jonathan
5 - Jamie
5 - Sue
32 - Mindy
32 - Paul
10 – Nathan
```

Note that `Tyrion` has the same hash value as `Aaron` (`16`). `Donnie` has the same hash value as `Ana` (`13`). `Jonathan`, `Jamie`, and `Sue` (`5`) have the same hash value as well and so do `Mindy` and `Paul` (`32`).

What will happen to the `HashTable` instance? Which values do we have inside it after executing the previous code?

To help us find out, let's implement a helper method called `print`, which will log on the console the values in the `HashTable` instance:

```
this.print = function () {
    for (var i = 0; i < table.length; ++i) {  //{1}
        if (table[i] !== undefined) {          //{2}
            console.log(i + ": " + table[i]); //{3}
        }
    }
};
```

First, we iterate through all the elements of the array (line {1}). For the positions that have a value (line {2}), we will log the position and its value on the console (line {3}).

Now, let's use this method:

```
hash.print();
```

We will have the following output on the console:

```
5: sue@email.com
10: nathan@email.com
13: ana@email.com
16: aaron@email.com
19: gandalf@email.com
29: johnsnow@email.com
32: paul@email.com
```

`Jonathan`, `Jamie`, and `Sue` have the same hash value, that is, `5`. As `Sue` was the last one to be added, `Sue` will be the one to occupy position 5 of `HashTable`. First, `Jonathan` will occupy it, then `Jamie` will overwrite it, and `Sue` will overwrite it again. The same will happen to the other elements that have a collision.

The idea of using a data structure to store all these values is obviously not to lose these values, it is to keep them all somehow. For this reason, we need to handle this situation when it happens. There are a few techniques to handle collisions: separate chaining, linear probing, and double hashing. We will cover the first two in this book.

Separate chaining

The **separate chaining** technique consists of creating a linked list for each position of the table and store the elements in it. It is the simplest technique to handle collisions; however, it requires additional memory outside the `HashTable` instance.

For example, if we use separate chaining in the code we used to do some testing in the previous topic, this would be the output:
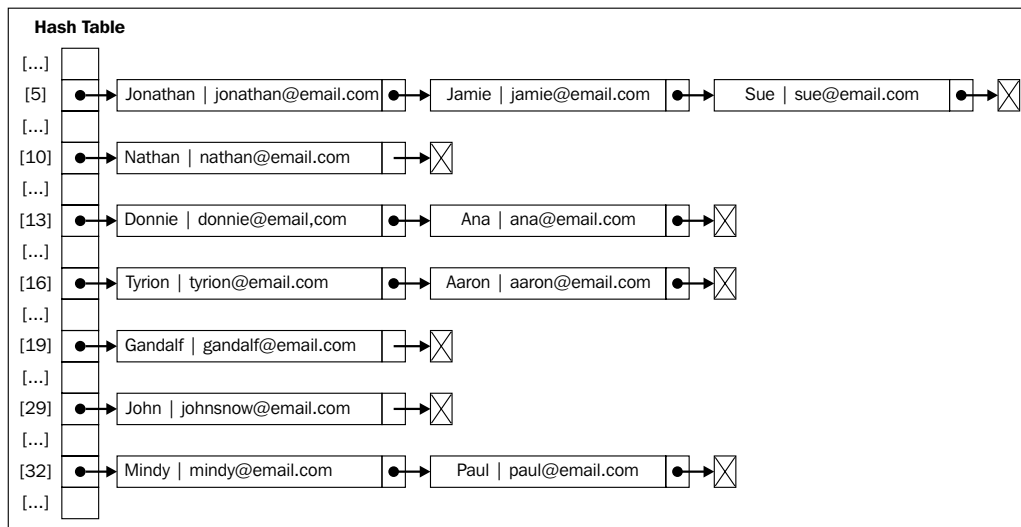
At position **5**, we would have a `LinkedList` instance with three elements in it; at positions **13**, **16**, and **32**, we would have `LinkedList` instances with two elements in it, and at positions **10**, **19**, and **29**, we would have `LinkedList` instances with a single element in it.

For separate chaining and linear probing, we only need to override three methods: `put`, `get`, and `remove`. These three methods will be different in each different technique we decide to implement.

To help us implement a `HashTable` instance using the separate chaining technique, we will need a new helper class to represent the element we will add to the `LinkedList` instance. We will call it the `ValuePair` class (declared inside the `HashTable` class):

```
var ValuePair = function(key, value){
    this.key = key;
    this.value = value;

    this.toString = function() {
        return '[' + this.key + ' - ' + this.value + ']';
    }
};
```

This class will simply store `key` and `value` in an `Object` instance. We will also override the `toString` method to help us later in outputting the results on the browser console.

The put method

Let's implement the first method, the `put` method, as follows:

```
this.put = function(key, value){
    var position = loseloseHashCode(key);

    if (table[position] == undefined) { //{1}
        table[position] = new LinkedList();
    }
    table[position].append(new ValuePair(key, value)); //{2}
};
```

In this method, we will verify that the position we are trying to add the element to already has something in it (line {1}). If this is the first time we are adding an element in this position, we will initialize it with an instance of the `LinkedList` class (which you learned about in *Chapter 5, Linked Lists*). Then, we will add the `ValuePair` instance (`key` and `value`) to the `LinkedList` instance using the `append` method we implemented in *Chapter 5* (line {2}).

The get method

Next, we will implement the `get` method to retrieve a specified value:

```
this.get = function(key) {
    var position = loseloseHashCode(key);

    if (table[position] !== undefined){ //{3}

        //iterate linked list to find key/value
        var current = table[position].getHead(); //{4}

        while(current.next){  //{5}
            if (current.element.key === key){ //{6}
                return current.element.value; //{7}
            }
            current = current.next; //{8}
        }

        //check in case first or last element
        if (current.element.key === key){ //{9}
            return current.element.value;
        }
    }
    return undefined; //{10}
};
```

The first verification we need to do is to see whether there is any element at the desired position (line {3}). If not, we return `undefined` to represent that the value was not found in the `HashTable` instance (line {10}). If there is a value in the position, we know that the instance is a `LinkedList` instance. Now, all we have to do is search for the element we want to find by iterating through the list. To do so, we need to get the reference of the head of the list (line {4}) and then we can iterate through it until we find the end of the list (line {5}, `current.next`, will be `null`).

The `Node` list contains the `next` pointer and the `element` attribute. The `element` attribute is an instance of `ValuePair`, so it has the attributes `value` and `key`. To access the `key` attribute of the `Node` list, we can use `current.element.key` and compare it to see whether it is the `key` we are searching for (line {6}). (This is the reason we are using the helper class `ValuePair` to store the elements. We cannot simply store the value itself, as we would not know which value corresponds to a particular key.) If it is the same `key` attribute, we return the `Node` value (line {7}), and if not, we continue iterating through the list by going to the next element of the list (line {8}).

If the element we are looking for is the first or last element of the list, it will not go inside the `while` loop. For this reason, we also need to handle this special case in line {9}.

The remove method

Removing an element from the `HashTable` instance using the separate chaining technique is a little bit different from the `remove` method we implemented earlier in this chapter. Now that we are using `LinkedList`, we need to remove the element from `LinkedList`. Let's check the `remove` method implementation:

```
this.remove = function(key){
    var position = loseloseHashCode(key);

    if (table[position] !== undefined){

        var current = table[position].getHead();
        while(current.next){
            if (current.element.key === key){ //{11}
                table[position].remove(current.element); //{12}
                if (table[position].isEmpty()){ //{13}
                    table[position] = undefined; //{14}
                }
                return true; //{15}
            }
            current = current.next;
        }

        //check in case first or last element
        if (current.element.key === key){ //{16}
            table[position].remove(current.element);
            if (table[position].isEmpty()){
                table[position] = undefined;
            }
            return true;
        }
    }

    return false; //{17}
};
```

In the `remove` method, we will do the same thing we did in the `get` method to find the element we are looking for. When iterating through the `LinkedList` instance, if the `current` element in the list is the element we are looking for (line {11}), we will use the `remove` method to remove it from `LinkedList` (line {12}). Then we will do an extra validation: if the list is empty (line {13} — there are no elements in it anymore), we will set the table `position` as `undefined` (line {14}), so we can skip this position whenever we look for an element or try to print its contents. At last, we return `true` to indicate that the element was removed (line {15}) or we return `false` at the end to indicate that the element was not present in `HashTable` (line {17}). Also, we need to handle the special case of first or last element (line {16}), as we did for the `get` method.

Overwriting these three methods, we have a `HashMap` instance with a separate chaining technique to handle collisions.

Linear probing

Another technique of collision resolution is linear probing. When we try to add a new element, if the position index is already occupied, then we try index +1. If index +1 is occupied, then we try index + 2, and so on.

The put method

Let's go ahead and implement the three methods we need to overwrite. The first one will be the `put` method:

```
this.put = function(key, value){
    var position = loseloseHashCode(key); // {1}

    if (table[position] == undefined) { // {2}
        table[position] = new ValuePair(key, value); // {3}
    } else {
        var index = ++position; // {4}
        while (table[index] != undefined){ // {5}
            index++; // {6}
        }
        table[index] = new ValuePair(key, value); // {7}
    }
};
```

As usual, we start by getting the position generated by the hash function (line {1}). Next, we verify that the position has an element in it (if it is already occupied, it will be line {2}). If not, we add the element to it (line {3} — an instance of the `ValuePair` class).

If the position is already occupied, we need to find the next position that is not (`position` is `undefined`), so we create an `index` variable and assign `position + 1` to it (line {4} — the increment operator `++` before the variable will increment the variable first and then assign it to `index`). Then we verify that the position if occupied (line {5}), and if it is, we increment `index` (line {6}) until we find a position that is not occupied. Then, all we have to do is assign the value we want to that position (line {7}).

> In some languages, we need to define the size of the array. One of the concerns of using linear probing is when the array is out of available positions. We do not need to worry about this in JavaScript as we do not need to define a size for the array and it can grow as needed automatically — this is part of JavaScript's built-in functionality.

If we run the inserts from the *Handling collisions* section again, this will be the result for the hash table using linear probing:

Hash Table	
[...]	
[5]	Jonathan | jonathan@email.com
[6]	Jamie | jamie@email.com
[7]	Sue | sue@email.com
[...]	
[10]	Nathan | nathan@email.com
[...]	
[13]	Donnie | donnie@email.com
[14]	Ana | ana@email.com
[...]	
[16]	Tyrion | tyrion@email.com
[17]	Aaron | aaron@email.com
[18]	
[19]	Gandalf | gandalf@email.com
[...]	
[19]	John | johnsnow@email.com
[...]	

Let's simulate the insertions in the hash table:

1. We will try to insert **Gandalf**. The hash value is **19** and as the hash table was just created, so position **19** is empty—we can insert the name here.

2. We will try to insert **John** at position **29**. It is also empty, so we can insert the name.

3. We will try to insert **Tyrion** at position **16**. It is empty, so we can insert the name.

4. We will try to insert **Aaron**, which also has a hash value of **16**. Position **16** is already occupied by **Tyrion**, so we need to go to *position + 1* (16 +1). Position **17** is free, so we can insert **Aaron** at **17**.

5. Next, we will try to insert **Donnie** in position **13**. It is empty, so we can insert it.

6. We will try to insert **Ana** also at position **13**, but this position is occupied. So we try position **14**, which is empty, so we can insert the name here.

7. Next, we will insert **Jonathan** at position **5**, which is empty, so we can insert the name.

8. We will try to insert **Jamie** at position **5**, but this position is occupied. So, we go to position **6**, which is empty, so we can insert the name.

9. We will try to insert **Sue** at position **5** as well, but is occupied. So we go to position **6**, which is also occupied. Then we go to position **7**, which is empty, so we can insert the name.

And so on.

The get method

Now that we have added our elements, let's implement the `get` function so that we can retrieve their values:

```
this.get = function(key) {
    var position = loseloseHashCode(key);

    if (table[position] !== undefined){ //{8}
        if (table[position].key === key) { //{9}
            return table[position].value; //{10}
        } else {
            var index = ++position;
            while (table[index] === undefined
|| table[index].key !== key){ //{11}
```

```
                index++;
            }
            if (table[index].key === key) { //{12}
                return table[index].value; //{13}
            }
        }
    }
    return undefined; //{14}
};
```

To retrieve a key's value, we first need to verify that the key exists (line {8}). If it does not exist, it means that the value is not in the hash table, so we can return `undefined` (line {14}). If it does exist, we need to check whether the value we are looking for is the one at the specified position (line {9}). If positive, we simply return its value (line {10}).

If not, we continue searching the following positions in the `HashTable` instance until we find a position that contains an element, and this element's key matches the key we are searching for (line {11}). Then, we verify that the item is the one we want (line {12}—just to make sure) and then we return its value (line {13}).

This is the reason we continue using the `ValuePair` class to represent the `HashTable` element—because we do not know at which position the element will actually be.

The remove method

The `remove` method will be exactly the same as the `get` method. The difference will be in lines {10} and {13}, which will be replaced with the following code:

```
table[index] = undefined;
```

To remove an element, we simply assign the value `undefined` to represent that the position is no longer occupied and it is free to receive a new element if needed.

Creating better hash functions

The "lose lose" hash function we implemented is not a good hash function as we can conclude (too many collisions). We would have multiple collisions if we use this function. A good hash function is composed by some factors: time to insert and retrieve an element (performance) and also a low probability of collisions. We can find several different implementations on the Internet or we can create our own.

Another simple hash function that we can implement and is better than the "lose lose" hash function is **djb2**:

```
var djb2HashCode = function (key) {
    var hash = 5381; //{1}
    for (var i = 0; i < key.length; i++) { //{2}
        hash = hash * 33 + key.charCodeAt(i); //{3}
    }
    return hash % 1013; //{4}
};
```

This consists of initializing the `hash` variable with a prime number (line {1}—most implementations use `5381`), then we iterate the `key` parameter (line {2}), multiply the `hash` value by `33` (used as a magical number), and sum it with the ASCII value of the character (line {3}).

Finally, we are going to use the rest of the division of the total by another random prime number (greater than the size we think the `HashTable` instance can have—in our case, let's consider 1000 as the size).

If we run the inserts from the *Handling collisions* section again, this will be the result we will get using `djb2HashCode` instead of `loseloseHashCode`:

```
798 - Gandalf
838 - John
624 - Tyrion
215 - Aaron
278 - Donnie
925 - Ana
288 - Jonathan
962 - Jamie
502 - Sue
804 - Mindy
54 - Paul
223 - Nathan
```

No collisions!

This is not the best hash function there is, but it is one of the most highly recommended hash functions by the community.

> There are also a few techniques to create a hash function for numeric keys. You can find a list and implementations at `http://goo.gl/VtdN2x`.

Summary

In this chapter, you learned about dictionaries, and how to add, remove, and get elements among other methods. We also learned the difference between a dictionary and a set.

We also covered hashing, how to create a hash table (or hash map) data structure, how to add, remove, and get elements, and also how to create hash functions. We learned how to handle collision in a hash table using two different techniques.

In the next chapter, we will learn a new data structure called a tree.

8
Trees

So far in this book, we have covered some sequential data structures. The first non sequential data structure we covered in this book was the **Hash Table**. In this chapter, we will learn another non sequential data structure called a **tree**, which is very useful for storing information that needs to be found easily.

A tree is an abstract model of a hierarchical structure. The most common example of a tree in real life would be a family tree, or a company organizational chart as we can see in the following figure:
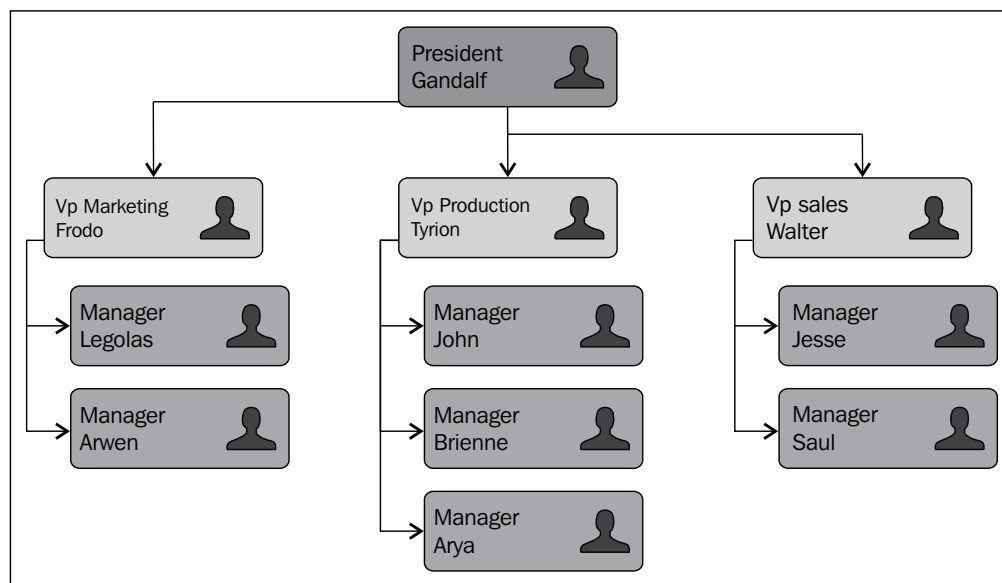
Trees terminology

A tree consists of nodes with a parent-child relationship. Each node has a parent (except for the first node at the top) and zero or more children:
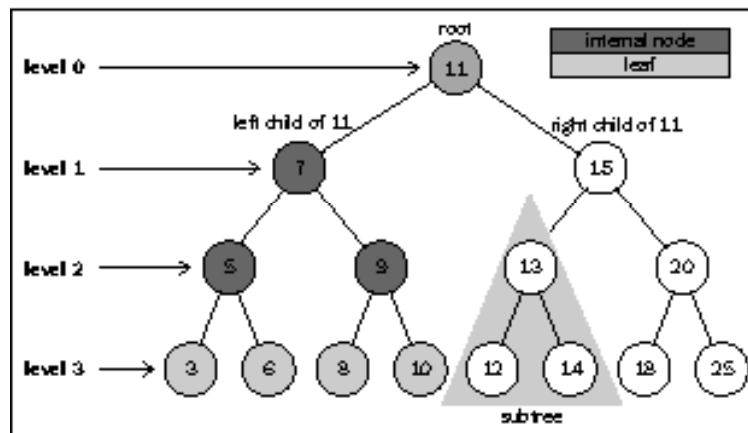
The top node of a tree is called the **root** (**11**). It is the node that does not have a parent. Each element of the tree is called node. There are internal nodes and external nodes. An internal node is a node with at least one child (**7**, **5**, **9**, **15**, **13**, and **20** are internal nodes). A node that does not have children is called an external node or leaf (**3**, **6**, **8**, **10**, **12**, **14**, **18**, and **25** are leaves).

A node can have ancestors and descendants. The ancestors of a node (except the root) are parent, grandparent, great-grandparent, and so on. The descendants of a node are child, grandchild, great-grandchild, and so on. For example, node **5** has **7** and **11** as its ancestors and **3** and **6** as its descendants.

Another terminology used with trees is the subtree. A subtree consists of a node and its descendants. For example, nodes **13**, **12**, and **14** consist a subtree of the tree from the preceding diagram.

The depth of a node consists of the number of ancestors. For example, node **3** has depth 3 because it has 3 ancestors (**5**, **7**, and **11**).

The height of a tree consists of the maximum depth of any node. A tree can also be broken down into levels. The root is on level 0, its children are on level 1, and so on. The tree from the preceding diagram has height 3 (maximum depth is 3 as shown in the preceding figure—level 3).

Now that we know the most important terms related to trees, we can start learning more about trees.

Binary tree and binary search tree

A node in a binary tree has at most two children: one left child and one right child. This definition allows us to write more efficient algorithms for inserting, searching, and deleting nodes to/from a tree. Binary trees are largely used in computer science.

A binary search tree is a binary tree, but it only allows you to store nodes with lesser values on the left side and nodes with greater values on the right side. The diagram in the previous topic exemplifies a binary search tree.

This will be the data structure we will be working on in this chapter.

Creating the BinarySearchTree class

Let's start by creating our `BinarySearchTree` class. First, let's declare its skeleton:

```
function BinarySearchTree() {

    var Node = function(key){ //{1}
        this.key = key;
        this.left = null;
        this.right = null;
    };

    var root = null; //{2}
}
```

The following diagram exemplifies how our **Binary Search Tree** (**BST**) will be organized in terms of data structure:
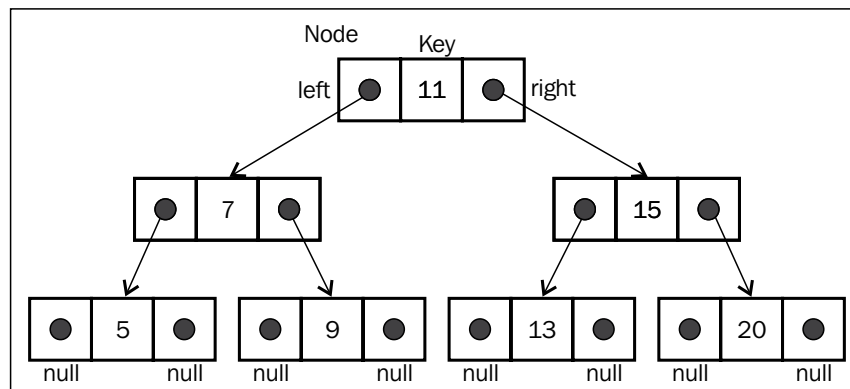
Just like in linked lists, we will work with pointers again to represent the connection between the nodes (called **edges** in tree terminology). When we worked with double linked lists, each node had two pointers: one to indicate the next node and another one to indicate the previous node. Working with trees, we will use the same approach (we will also work with two pointers). However, one pointer will point to the left child, and the other one will point to the right child. For this reason, we will declare a Node class that will represent each node of the tree ({1}). A small detail that is worth noticing is that instead of calling the node itself as a node or item as we did in the previous chapters, we will call it a key. A key is how a tree node is known in tree terminology.

We are going to follow the same pattern we used in the `LinkedList` class (from *Chapter 5*, *Linked Lists*). This means that we will also declare a variable so we can control the first node of the data structure. In the case of a tree, instead of the head, we have the root ({2}).

Next, we need to implement some methods. The following are the methods we will implement in our tree class:

- `insert(key)`: This inserts a new key in the tree
- `search(key)`: This searches for the key in the tree and returns `true` if it exists and returns `false` if the node does not exist
- `inOrderTraverse`: This visits all nodes of the tree using in-order traverse
- `preOrderTraverse`: This visits all nodes of the tree using pre-order traverse
- `postOrderTraverse`: This visits all nodes of the tree using post-order traverse
- `min`: This returns the minimum value/key in the tree
- `max`: This returns the maximum value/key in the tree
- `remove(key)`: This removes the key from the tree

We will implement each of these methods in the subsequent sections.

Inserting a key in a tree

The methods we will be implementing in this chapter are a little bit more complex than the ones we implemented in previous chapters. We will use a lot of recursion in our methods. If you are not familiar with recursion, please refer to the *Recursion* section in *Chapter 11*, *More about Algorithms*.

The following code is the first piece of the algorithm used to insert a new key in a tree:

```
this.insert = function(key){

    var newNode = new Node(key); //{1}

    if (root === null){ //{2}
        root = newNode;
    } else {
        insertNode(root,newNode); //{3}
    }
};
```

To insert a new node (or item) in a tree, there are three steps we need to follow.

The first step is to create the instance of the `Node` class that will represent the new node (`{1}`). Because of its constructor properties, we only need to pass the value we want to add to the tree and its pointers `left` and `right` will have a `null` value automatically.

Second, we need to verify that the insertion is a special case. The special case is if the node we are trying to add is the first one in the tree (`{2}`). If it is, all we have to do is point the root to this new node.

The third step is to add a node to a different position than the root. In this case, we will need a helper (`{3}`) private function to help us to do this, which is declared as follows:

```
var insertNode = function(node, newNode){
    if (newNode.key < node.key){ //{4}
        if (node.left === null){    //{5}
            node.left = newNode;    //{6}
        } else {
            insertNode(node.left, newNode); //{7}
        }
    } else {
        if (node.right === null){  //{8}
            node.right = newNode;  //{9}
        } else {
            insertNode(node.right, newNode); //{10}
        }
    }
};
```

The `insertNode` function will help us to find out where the correct place to insert a new node is. The following are the steps that describe what this function does:

- If the tree is not empty, we need to find a place to add a new node. For this reason, we will call the `insertNode` function passing the root and the node as parameters ({3}).

- If the node's key is less than the current node key (in this case, it is the root ({4})), then we need to check the left child of the node. If there is no left node ({5}), then we insert the new node there ({6}). If not, we need to descend a level in the tree by calling `insertNode` recursively ({7}). In this case, the node we will be comparing next time will be the left child of the current node.

- If the node's key is greater than the current node key and there is no right child ({8}), then we insert the new node there ({9}). If not, we will also need to call the `insertNode` function recursively, but the new node to be compared will be the right child ({10}).

Let's use an example so we can understand this process better.

Consider the following scenario: we have a new tree, and we are trying to insert its first key:

```
var tree = new BinarySearchTree();
tree.insert(11);
```

In this case, we will have a single node in our tree, and the root pointer will be pointing to it. The code that will be executed is in line {2} of our source code.

Now, let's consider we already have the following tree:
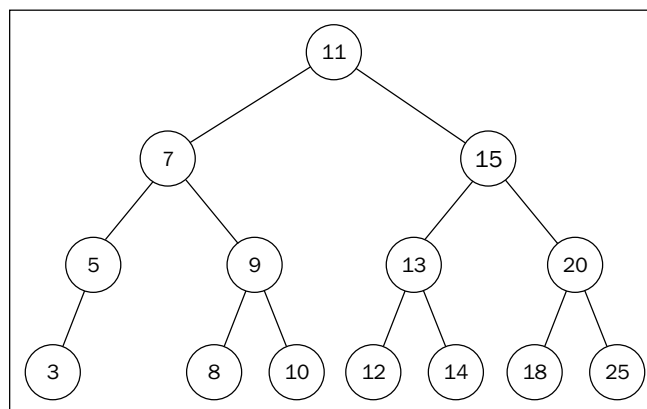
The code to create the tree seen in the preceding diagram is a continuation of the previous code (where we inserted key 11):

```
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);
```

And we would like to insert a new key with value 6, so we will execute the following code:

```
tree.insert(6);
```

The following steps will be executed:

1. The tree is not empty, and then the code from line {3} will be executed. The code will call the insertNode method (root, key[6]).

2. The algorithm is going to check line {4} (key[6] < root[11] is true) and then will check line {5} (node.left[7] is not null) and then will go to line {7} calling insertNode (node.left[7], key[6]).

3. We will go inside the insertNode method again, but with different parameters. It will check line {4} again (key[6] < node[7] is true) and then will check line {5} (node.left[5] is not null), and then will go to line {7} calling insertNode (node.left[5], key[6]).

4. We will go once more inside the insertNode method. It will check line {4} again (key[6] < node[5] is false), and then it will go to line {8} (node.right is null — node 5 does not have any right child descendents) and will execute line {9}, inserting key 6 as the right child of node 5.

5. After that, the stack of method calls will pop up and the execution will end.

This will be the result after key **6** is inserted in the tree:
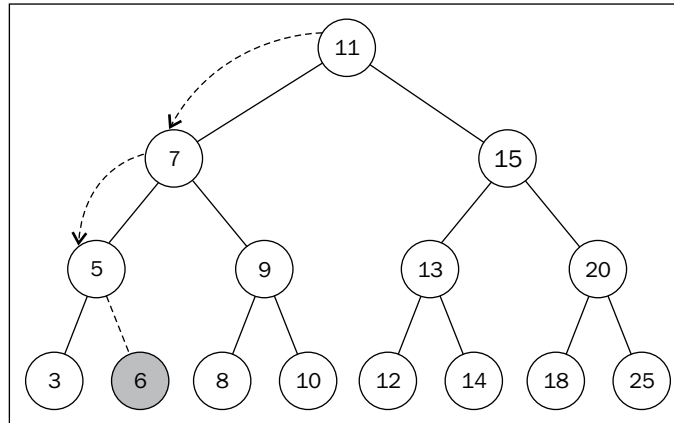
Tree traversal

Traversing (or walking) a tree is the process of visiting all nodes of a tree and performing an operation at each node. But how should we do that? Should we start from the top of the tree or from the bottom? From the left or the right side? There are three different approaches that can be used to visit all the nodes in a tree: in-order, pre-order, and post-order.

In the following sections, we will deep dive into the uses and implementations of these three types of tree traversals.

In-order traversal

An in-order traversal visits all the nodes of a BST in ascending order, meaning it visits the nodes from the smallest to largest. An application of in-order traversal would be to sort a tree. Let's check out its implementation:

```
this.inOrderTraverse = function(callback){
    inOrderTraverseNode(root, callback); //{1}
};
```

The `inOrderTraverse` method receives a callback function as a parameter. This function can be used to perform the action we would like to execute when the node is visited (this is known as the visitor pattern; for more information on this, refer to `http://en.wikipedia.org/wiki/Visitor_pattern`). As most algorithms we are implementing for the BST are recursive, we will use a private helper function that will receive a `node` and the `callback` function to help us with it (`{1}`):

```
var inOrderTraverseNode = function (node, callback) {
    if (node !== null) { //{2}
        inOrderTraverseNode(node.left, callback);  //{3}
        callback(node.key);                        //{4}
        inOrderTraverseNode(node.right, callback); //{5}
    }
};
```

To traverse a tree using the in-order method, first we need to check whether the `node` that was passed as parameter is `null` (this is the point where the recursion stops being executed—line {2}—the base case of the recursion algorithm).

Next, we visit the left node ({3}) calling the same function recursively. Then we visit the `node` ({4}) by performing an action with it (`callback`), and then we visit the right node ({5}).

Let's try to execute this method using the tree from the previous topic as an example:

```
function printNode(value){ //{6}
    console.log(value);
}
tree.inOrderTraverse(printNode); //{7}
```

But first, we need to create a callback function ({6}). All we are going to do is print the node's value on the browser's console. Then, we can call the `inOrderTraverse` method passing our callback function as a parameter ({7}). When we execute this code, the following will be the output in the console (each number will be outputted on a different line):

```
3 5 6 7 8 9 10 11 12 13 14 15 18 20 25
```

The following diagram illustrates the path the `inOrderTraverse` method followed:
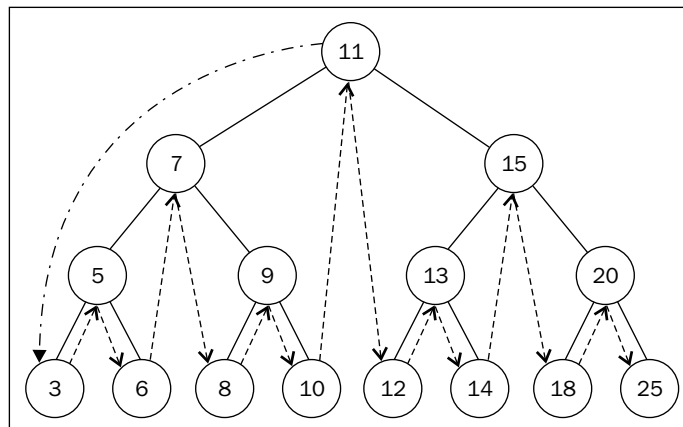
Pre-order traversal

A pre-order traversal visits the node prior to its descendants. An application of pre-order traversal could be to print a structured document.

Let's see its implementation:

```
this.preOrderTraverse = function(callback){
    preOrderTraverseNode(root, callback);
};
```

The `preOrderTraverseNode` method implementation is as follows:

```
var preOrderTraverseNode = function (node, callback) {
    if (node !== null) {
        callback(node.key); //{1}
        preOrderTraverseNode(node.left, callback); //{2}
        preOrderTraverseNode(node.right, callback); //{3}
    }
};
```

The difference between the in-order and the pre-order is that the pre-order will visit the node first (`{1}`) and then will visit the left node (`{2}`) and then the right node (`{3}`), while the in-order executes the lines in the following order: `{2}`, `{1}`, and `{3}`.

The following will be the output in the console (each number will be outputted on a different line):

```
11 7 5 3 6 9 8 10 15 13 12 14 20 18 25
```

The following diagram illustrates the path followed by the `preOrderTraverse` method:
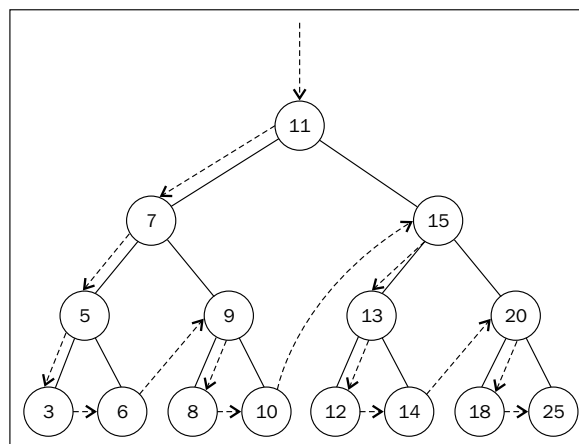
Post-order traversal

A post-order traversal visits the node after it visits its descendants. An application of post-order could be computing the space used by a file in a directory and its subdirectories.

Let's see its implementation:

```
this.postOrderTraverse = function(callback){
    postOrderTraverseNode(root, callback);
};
```

The `postOrderTraverseNode` implementation is as follows:

```
var postOrderTraverseNode = function (node, callback) {
    if (node !== null) {
        postOrderTraverseNode(node.left, callback);   //{1}
        postOrderTraverseNode(node.right, callback);   //{2}
        callback(node.key);                            //{3}
    }
};
```

In this case, the post-order traverse will visit the `left` node ({1}), and then the `right` node ({2}), and at last, it will visit the node ({3}).

As you can see, the algorithms for the in-order, pre-order, and post-order approaches are very similar; the only thing that changes is the order that lines {1}, {2}, and {3} are executed in each method.

This will be the output in the console (each number will be outputted on a different line):

```
3 6 5 8 10 9 7 12 14 13 18 25 20 15 11
```

The following diagram illustrates the path the `postOrderTraverse` method followed:
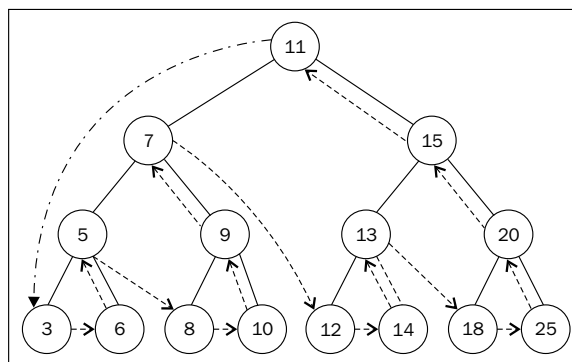
Searching for values in a tree

There are three types of searches that are usually performed in trees:

- Minimum values
- Maximum values
- Searching for a specific value

Let's take a look at each one.

Searching for minimum and maximum values

Let's use the following tree for our examples:
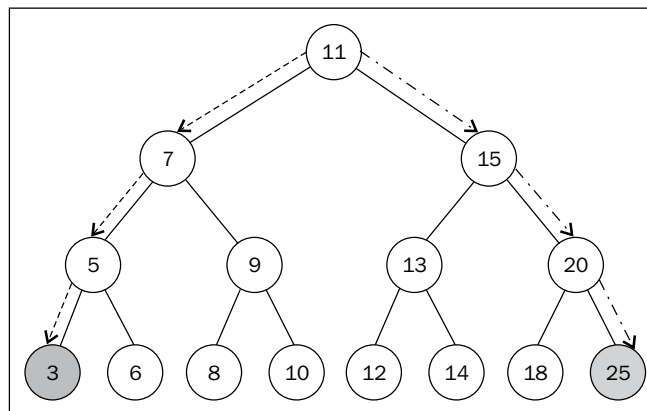
Just looking at the preceding figure, could you easily find the minimum and maximum values of the tree?

If you take a look at the left-most node in the last level of tree, you will find the value **3**, which is the lowest key from this tree. And if you take a look at the node that is furthest to the right (also in the last level of the tree), you will find the key **25**, which is the highest key in this tree. This information helps us a lot when implementing methods that will find the minimum and maximum nodes of the tree.

First, let's take a look at the method that will find the minimum key of the tree:

```
this.min = function() {
    return minNode(root); //{1}
};
```

The `min` method will be the method exposed to the user. This method calls the `minNode` method ({1}):

```
var minNode = function (node) {
    if (node){
        while (node && node.left !== null) { //{2}
            node = node.left;                 //{3}
        }

        return node.key;
    }
    return null;  //{4}
};
```

The `minNode` method allows us to find the minimum key from any node of the tree. We can use it to find the minimum key from a subtree or from the tree itself. For this reason, we call the `minNode` method passing the tree root ({1})—because we want to find the minimum key of the whole tree.

Inside the `minNode` method, we will traverse the `left` edge of the tree ({2} and {3}) until we find the node at the highest level of the tree (left end).

In a similar way, we also have the `max` method:

```
this.max = function() {
    return maxNode(root);
};

var maxNode = function (node) {
    if (node){
        while (node && node.right !== null) { //{5}
            node = node.right;
        }

        return node.key;
    }
    return null;
};
```

To find the maximum key, we will traverse the right edge of the tree ({5}) until we find the last node at the right end of the tree.

So for the minimum value, we always go the left side of the tree, and for the maximum value, we will always navigate to the right side of the tree.

Searching for a specific value

In previous chapters, we also implemented the `find`, `search`, or `get` methods to find a specific value in the data structure (similar to the `has` method we implemented in previous chapters). We will also implement the search method for the BST as well. Let's see its implementation:

```
this.search = function(key){
    return searchNode(root, key); //{1}
};

var searchNode = function(node, key){

    if (node === null){ //{2}
        return false;
    }
    if (key < node.key){ //{3}
        return searchNode(node.left, key); //{4}

    } else if (key > node.key){ //{5}
        return searchNode(node.right, key); //{6}

    } else {
        return true; //{7}
    }
};
```

The first thing we need to do is declare the `search` method. Following the pattern of other methods declared for our BST, we are going to use a helper function to help us (`{1}`).

The `searchNode` method can be used to find a specific key in the tree or any of its subtrees. This is the reason we call this method in {1}, passing the tree `root` as parameter.

Before we start the algorithm, we are going to validate that the `node` passed as parameter is valid (is not `null`). If it is, it means that the key was not found and we return `false`.

If the node is not `null`, we need to continue the verification. If the key we are looking for is lower than the current node ({3}), then we will continue the search using the left child subtree ({4}). If the value we are looking for is greater than the current node ({5}), then we continue the search from the right child of the current node ({6}). Otherwise, it means the key we are looking for is equal to the current node's key, and we return `true` to indicate we found the key ({7}).

We can test this method using the following code:

```
console.log(tree.search(1) ? 'Key 1 found.' : 'Key 1 not found.');
console.log(tree.search(8) ? 'Key 8 found.' : 'Key 8 not found.');
```

It will output the following:

```
Value 1 not found.
Value 8 found.
```

Let's go into more detail on how the method was executed to find the key `1`:

1. We called the `searchNode` method, passing the root as parameter ({`1`}). (`node[root[11]]`) is not `null` ({`2`}), therefore we go to line {`3`}.

2. (`key[1] < node[11]`) is `true` ({`3`}), therefore we go to line {`4`} and call the `searchNode` method again, passing (`node[7], key[1]`) as parameters.

3. (`node[7]`) is not `null` ({`2`}), therefore we go to line {`3`}.

4. (`key[1] < node[7]`) is `true` ({`3`}), so we go to line {`4`} and call the `searchNode` method again, passing (`node[5], key[1]`) as parameters.

5. (`node[5]`) is not `null` ({`2`}), therefore we go to line {`3`}.

6. (`key[1] < node[5]`) is `true` ({`3`}), therefore we go to line {`4`} and call the `searchNode` method again, passing (`node[3], key[1]`) as parameters.

7. (`node[3]`) is not `null` ({`2`}), therefore we go to line {`3`}.

8. (`key[1] < node[3]`) is `true` ({`3`}), therefore we go to line {`4`} and call the `searchNode` method again, passing (`null, key[1]`) as parameters. `null` was passed as a parameter because `node[3]` is a leaf (it does not have children, so the left child will have the value `null`).

9. (`null`) is `null` (in line {`2`}, the node to search in this case is `null`), therefore we return `false`.

10. After that, the stack of method calls will pop up and the execution will end.

Let's do the same exercise to search value `8`:

1. We called the `searchNode` method, passing `root` as parameter ({`1`}). (`node[root[11]]`) is not `null` ({`2`}), therefore we go to line {`3`}.

2. (`key[8] < node[11]`) is `true` ({`3`}), therefore we go to line {`4`} and call the `searchNode` method again, passing (`node[7], key[8]`) as parameters.

3. (`node[7]`) is not `null` ({`2`}), therefore we go to line {`3`}.

4. (`key[8] < node[7]`) is `false` ({`3`}), therefore we go to line {`5`}.

5. (`key[8] > node[7]`) is `true` ({5}), therefore we go to line {6} and call the `searchNode` method again, passing (`node[9], key[8]`) as parameters.

6. (`node[9]`) is not `null` ({2}), therefore we go to line {3}.

7. (`key[8] < node[9]`) is `true` ({3}), therefore we go to line {4} and call the `searchNode` method again, passing (`node[8], key[8]`) as a parameter.

8. (`node[8]`) is not `null` ({2}), therefore we go to line {3}.

9. (`key[8] < node[8]`) is `false` ({3}), therefore we go to line {5}.

10. (`key[8] > node[8]`) is `false` ({5}), therefore we go to line {7} and return `true` because `node[8]` is the key we were looking for.

11. After that, the stack of method calls will pop up and the execution will end.

Removing a node

The next and final method we will implement for our BST is the `remove` method. This is the most complex method we will implement in this book. Let's start with the method that will be available to be called from a tree instance:

```
this.remove = function(key){
    root = removeNode(root, key); //{1}
};
```

This method receives the desired `key` to be removed and it also calls `removeNode`, passing `root` and `key` to be removed as parameters ({1}). One thing very important to note is that the root receives the return of the method `removeNode`. We will understand why in a second.

The complexity of the `removeNode` method is due to the different scenarios that we need to handle and also because it is recursive.

Let's take a look at the `removeNode` implementation:

```
var removeNode = function(node, key){

    if (node === null){ //{2}
        return null;
    }
    if (key < node.key){ //{3}
        node.left = removeNode(node.left, key); //{4}
        return node; //{5}
```

```
    } else if (key > node.key){ //{6}
        node.right = removeNode(node.right, key); //{7}
        return node; //{8}

    } else { // key is equal to node.key

        //case 1 - a leaf node
        if (node.left === null && node.right === null){ //{9}
            node = null; //{10}
            return node; //{11}
        }

        //case 2 - a node with only 1 child
        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}
        }

        //case 3 - a node with 2 children
        var aux = findMinNode(node.right); //{18}
        node.key = aux.key; //{19}
        node.right = removeNode(node.right, aux.key); //{20}
        return node; //{21}
    }
};
```

As a stop point we have line {2}. If the node we are analyzing is `null`, it means the key does not exist in the tree, and for this reason, we return `null`.

Then, the first thing we need to do is to find the node in the tree. So if the key we are looking for has a lower value than the current node ({3}), then we go to the next node at the left edge of the tree ({4}). If the key is greater than the current node ({6}), then we will go the next node at the right edge of the tree ({7}).

If we find the key we are looking for (`key is equal to node.key`), then we have three different scenarios we have to handle.

Removing a leaf node

The first scenario is a leaf node that does not have a left or right child—{9}. In this case, all we have to do is get rid of this node by assigning `null` to it ({9}). But as we learned during the implementation of linked lists, we know that assigning `null` to the node is not enough and we also need to take care of the pointers. In this case, the node does not have any children, but it has a parent node. We need to assign `null` to its parent node and this can be done by returning `null` ({11}).

As the node already has value `null`, the parent pointer to the node will receive this value as well. And this is the reason we are returning the node value as the function return. The parent node will always receive the value returned from the function. An alternative to this approach could be passing the parent and the node as parameters of the method.

If we take a look back at the first lines of the code of this method, we will notice that we are updating the pointer values of the left and right pointer of the nodes in {4} and {7} and we are also returning the updated node in {5} and {8}.

The following diagram exemplifies the removal of a leaf node:
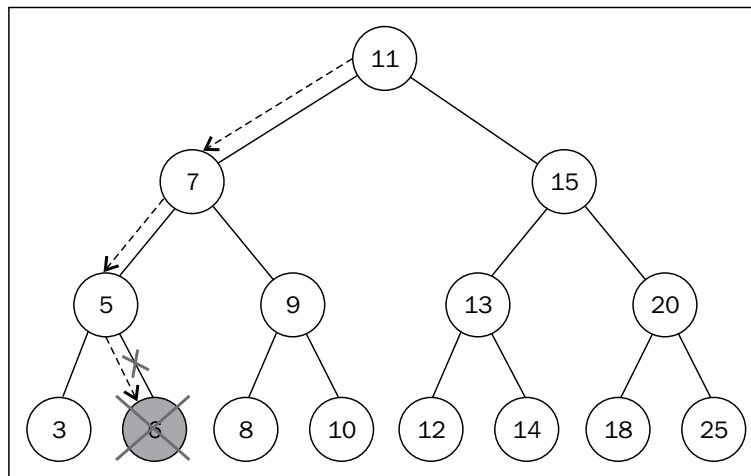
Removing a node with a left or right child

Now let's take a look at the second scenario, which is a node that has a left or right child. In this case, we need to skip this node and assign the parent pointer to the child node.

If the node does not have a left child ({12}), it means it has a right child. So, we change the reference of the node to its right child ({13}) and return the updated node ({14}). We will do the same if the node does not have the right child ({15})—we will update the node reference to its left child ({16}) and return the updated value ({17}).

The following diagram exemplifies the removal of a node with only a left child or a right child:
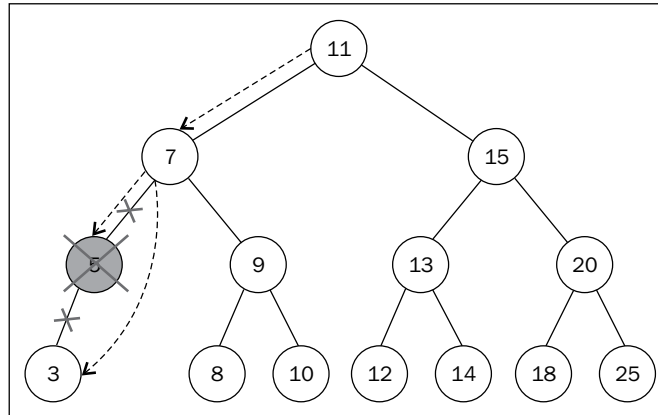
Removing a node with two children

Now comes the third scenario and the most complex one, which is the scenario where the node we are trying to remove has two children—the right and left one. To remove a node with children, there are four steps that need to be performed:

1. Once we find the node we want to remove, we need to find the minimum node from its right edge subtree (its successor—line {18}).

2. Then, we update the value of the node with the key of the minimum node from its right subtree ({19}). With this action, we are replacing the key of the node, which means it was removed.

3. However, now we have two nodes in the tree with the same key and this cannot happen. What we need to do now is remove the minimum node from the right subtree, since we moved it to the place of the removed node ({20}).

4. And finally, we return the updated node reference to its parent ({21}).

The implementation of the `findMinNode` method is exactly the same as the `min` method. The only difference is that in the `min` method, we return only the key and in the `findMinNode` method, we are returning the node.

The following diagram exemplifies the removal of a node with only a left child and right child:
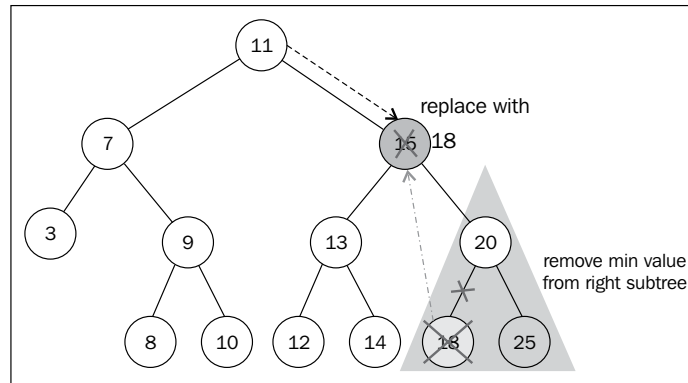
More about binary trees

Now that you know how to work with BST, you can dive into the study of trees if you would like to.

BST has an problem: depending on how many nodes you add, one of the edges of tree can be very deep, meaning a branch of the tree can have a high level, and another branch can have a low level, as shown in the following diagram:
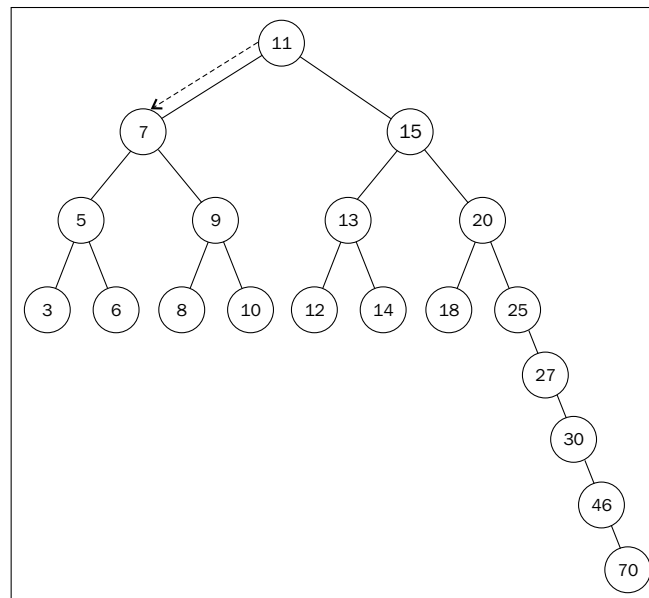
This can cause performance issues when adding, removing, and searching for a node on a particular edge of the tree. For this reason, there is a tree called **Adelson-Velskii and Landis' tree** (**AVL tree**). The AVL tree is a self-balancing BST tree, which means the height of both the left and right subtree of any node differs by 1 at most. This means the tree will try to become a complete tree whenever possible while adding or removing a node.

We will not cover AVL trees in this book, but you can find its source code inside the `chapter08` folder of this book's source code and take a look at it there.
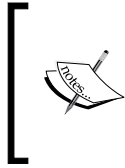
> Another tree that you should also learn about is the Red-Black tree, which is a special type of binary tree. This tree allows efficient in-order traversal of its nodes (`http://goo.gl/OxED8K`). You should also check out the Heap tree as well (`http://goo.gl/SFlhW6`).

Summary

In this chapter, we covered the algorithms to add, search, and remove items from a binary search tree, which is the basic tree data structure largely used in Computer Science. We also covered three traversal approaches to visit all the nodes of a tree.

In the next chapter, we will study the basic concepts of graphs, which is also a non linear data structure.

9
Graphs

In this chapter, you will learn about another nonlinear data structure called graph. This will be the last data structure we will cover before diving into sorting and searching algorithms.

This chapter will cover a considerable part of the wonderful applications of graphs. Since this is a vast topic, we could write a book like this just to dive into the amazing world of graph.

Graph terminology

A **graph** is an abstract model of a network structure. A graph is a set of **nodes** (or **vertices**) connected by **edges**. Learning about graphs is important because any binary relationship can be represented by a graph.

Any social network, such as Facebook, Twitter, and Google plus, can be represented by a graph.

We can also use graphs to represent roads, flights, and communications, as shown in the following image:
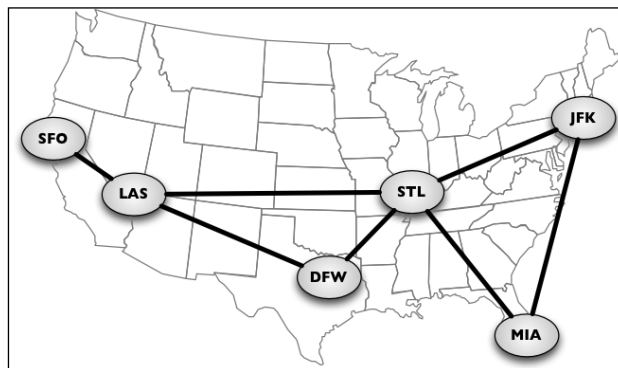
Let's learn more about the mathematical and technical concepts of graphs.

A graph *G = (V, E)* is composed of:

- *V*: A set of vertices
- *E*: A set of edges connecting the vertices in *V*

The following diagram represents a graph:
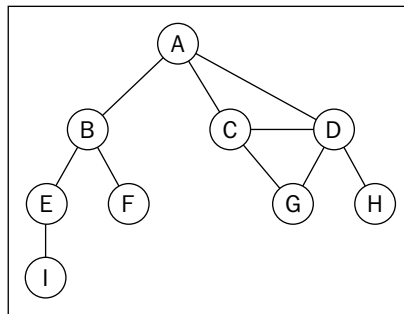
Let's cover some graph terminology before we start implementing any algorithms.

Vertices connected by an edge are called **adjacent vertices**. For example, **A** and **B** are adjacent, **A** and **D** are adjacent, **A** and **C** are adjacent, and **A** and **E** are *not* adjacent.

A **degree** of a vertex consists of the number of adjacent vertices. For example, **A** is connected to other three vertices, therefore, **A** has degree 3; **E** is connected to other two vertices, therefore, **E** has degree 2.

A **path** is a sequence of consecutive vertices *v1, v2, …, vk*, where *vi* and *vi+1* are adjacent. Using the graph from the previous diagram as an example, we have paths **A B E I** and **A C D G**, among others.

A **simple path** does not contain repeated vertices. As an example, we have the path **A D G**. A **cycle** is a simple path, except for the last vertex, which is the same as the first vertex: **A D C A** (back to **A**).

A graph is **acyclic** if it does not have cycles. A graph is **connected** if there is a path between every pair of vertices.

Directed and undirected graphs

Graphs can be **undirected** (where edges do not have a direction) or **directed** (**digraph**), where edges have a direction, as demonstrated here:
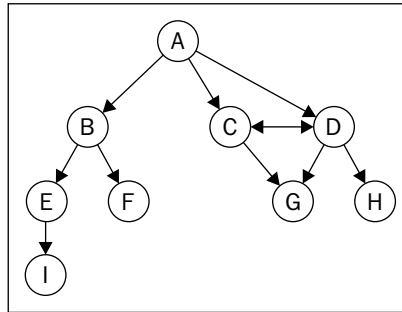
A graph is **strongly connected** if there is a path in both directions between every pair of vertices. For example, **C** and **D** are strongly connected while **A** and **B** are not strongly connected.

Graphs can also be **unweighted** (as we have seen so far) or **weighted**, where the edges have weights, as shown in the following diagram:
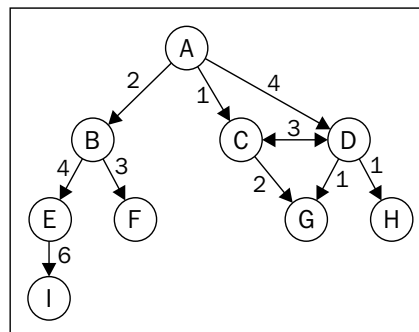
We can solve many problems in the Computer Science world using graphs, such as searching a graph for a specific vertex or searching for a specific edge, finding a path in the graph (from one vertex to another), finding the shortest path between two vertices, and cycle detection.

Representing a graph

There are a few ways we can represent graphs when it comes to data structures. There is no correct way of representing a graph among the existing possibilities. It depends on the type of problem you need to resolve and the type of graph as well.

The adjacency matrix

The most common implementation is the adjacency matrix. Each node is associated with an integer, which is the array index. We represent the connectivity between vertices using a two-dimensional array, as *array[i][j] === 1* if there is an edge from the node with index *i* to the node with index *j*; or as *array[i][j] === 0* otherwise, as demonstrated by the following diagram:
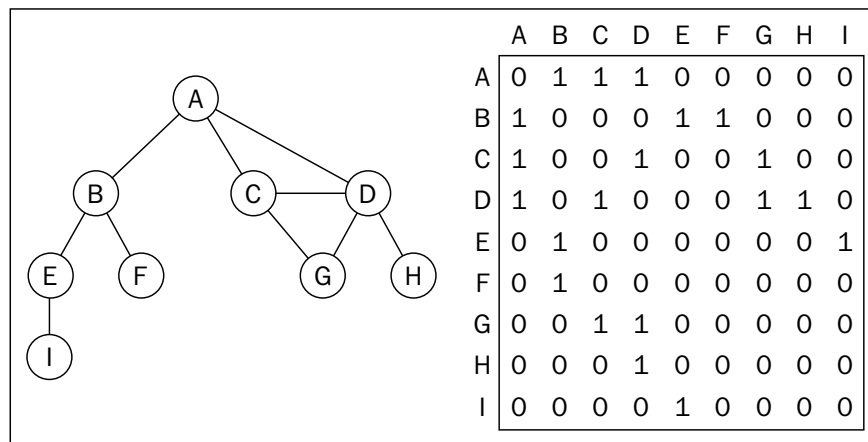
Graphs that are not strongly connected (**sparse graphs**) will be represented by a matrix with many zero entries in the adjacency matrix. This means we will waste space in the computer memory to represent edges that do not exist; for example, if we need to find the adjacent vertices of a given vertex, we will have to iterate through the whole row even if this vertex has only one adjacent vertex. Another reason this might not be a good representation is because the number of vertices in the graph may change and a two-dimensional array is inflexible.

The adjacency list

We can use a dynamic data structure to represent graphs as well, called an adjacency list. This consists of a list of adjacent vertices for every vertex of the graph. There are a few different ways we can represent this data structure. To represent the list of adjacent vertices, we can use a list (array), a linked list, or even a hash map or dictionary. The following diagram exemplifies the adjacency list data structure:
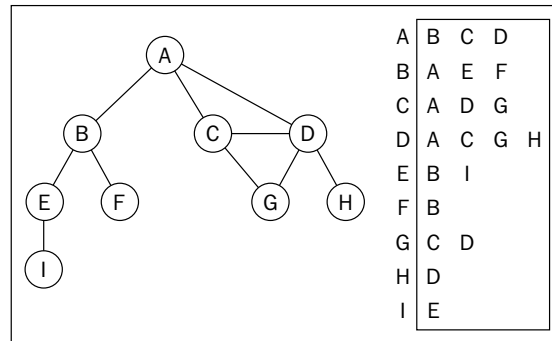
Both representations are very useful and have different properties (for example, finding out whether vertices *v* and *w* are adjacent is faster using adjacent matrix), although adjacency lists are probably better for most problems. We are going to use the adjacency list representation for the examples in this book.

The incidence matrix

We can also represent a graph using an incidence matrix. In an **incidence matrix**, each row of the matrix represents a vertex and each column represents an edge. We represent the connectivity between the two objects using a two-dimensional array, as *array[v][e] === 1* if the vertex *v* is incident upon edge *e*; or as *array[v][e] === 0* otherwise, as demonstrated in the following diagram:
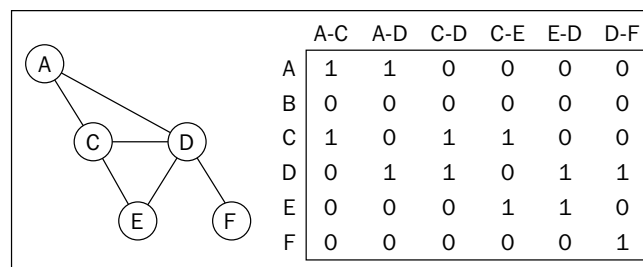
An incidence matrix is usually used to save space and memory when we have more edges than vertices.

Creating the Graph class

As usual, we are going to declare the skeleton of our class:

```
function Graph() {
    var vertices = []; //{1}
    var adjList = new Dictionary(); //{2}
}
```

We are going to use an array to store the names of all the vertices of the graph (line {1}) and we are going to use a dictionary (implemented in *Chapter 7, Dictionaries and Hashes*) to store the adjacent list (line {2}). The dictionary will use the name of the vertex as key and the list of adjacent vertices as a value. Both the vertices array and the adjList dictionary are private attributes of our Graph class.

Next, we are going to implement two methods: one to add a new vertex to the graph (because when we instantiate the graph, it will create an empty one) and another method to add edges between the vertices. Let's implement the addVertex method first:

```
this.addVertex = function(v){
    vertices.push(v); //{3}
    adjList.set(v, []); //{4}
};
```

This method receives a vertex v as parameter. We are going to add this vertex to the list of vertices (line {3}), and we are also going to initialize the adjacent list with an empty array by setting the dictionary value of the vertex v key with an empty array (line {4}).

Now, let's implement the addEdge method:

```
this.addEdge = function(v, w){
    adjList.get(v).push(w); //{5}
    adjList.get(w).push(v); //{6}
};
```

This method receives two vertices as parameters. First, we are going to add an edge from vertex v to vertex w (line {5}) by adding w to the adjacent list of v. If you want to implement a directed graph, line {5} is enough. As we are going to work with undirected graphs in most examples in this chapter, we also need to add the edge from w to v (line {6}).

Note that we are only adding new elements to the array, since we have already initialized it in line {4}.

Let's test this code:

```
var graph = new Graph();
var myVertices = ['A','B','C','D','E','F','G','H','I']; //{7}
for (var i=0; i<myVertices.length; i++){ //{8}
    graph.addVertex(myVertices[i]);
}
graph.addEdge('A', 'B'); //{9}
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');
```

To make our lives easier, let's create an array with all the vertices we want to add to our graph (line {7}). Then, we only need to iterate through the `vertices` array and add the values one by one to our graph (line {8}). Finally, we add the desired edges (line {9}). This code will create the graph we used in the diagrams presented so far in this chapter.

To make our lives even easier, let's also implement the `toString` method for this `Graph` class, so that we can output the graph on the console:

```
this.toString = function(){
    var s = '';
    for (var i=0; i<vertices.length; i++){ //{10}
        s += vertices[i] + ' -> ';
        var neighbors = adjList.get(vertices[i]); //{11}
        for (var j=0; j<neighbors.length; j++){ //{12}
            s += neighbors[j] + ' ';
        }
        s += '\n'; //{13}
    }
    return s;
};
```

We are going to build a string with the adjacent list representation. First, we are going to iterate the list of `vertices` array (line {10}) and add the name of the vertex to our string. Then, we are going to get the adjacent list for this vertex (line {11}) and we are also going to iterate it (line {12}) to get the name of the adjacent vertex and add it to our string. After we have iterated the adjacent list, we are going to add a new line to our string (line {13}), so we can see a pretty output on the console. Let's try this code:

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

This will be the output:

```
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
```

A pretty adjacent list! From this output, we know that vertex `A` has the following adjacent vertices: `B`, `C`, and `D`.

Graph traversals

Similar to the tree data structure, we can also visit all the nodes of a graph. There are two algorithms that can be used to traverse a graph called **breadth-first search** (**BFS**) and **depth-first search** (**DFS**). Traversing a graph can be used to find a specific vertex or to find a path between two vertices, check whether the graph is connected, check whether it contains cycles, and so on.

Before we implement the algorithms, let's try to better understand the idea of traversing a graph.

The idea of graph traversal algorithms is that we must track each vertex when we first visit it and keep track of which vertices have not yet been completely explored. For both traversal graph algorithms, we need to specify which is going to be the first vertex to be visited.

To completely explore a vertex, we need to look at each edge of this vertex. For each edge connected to a vertex that has not been visited yet, we mark it as discovered and add it to the list of vertices to be visited.

In order to have efficient algorithms, we must visit each vertex twice, at the most, when each of its endpoints are explored. Every edge and vertex in the connected graph will be visited.

The BFS and DFS algorithms are basically the same, with only one difference, which is the data structure used to store the list of vertices to be visited:

Algorithm	Data structure	Description
DFS	Stack	By storing the vertices in a stack (learned in *Chapter 3*, *Stacks*), the vertices are explored by lurching along a path, visiting a new adjacent vertex if there is one available.
BFS	Queue	By storing the vertices in a queue (learned in *Chapter 4*, *Queues*), the oldest unexplored vertices are explored first.

When marking the vertices that we have already visited, we use three colors to reflect their status:

- **White**: This represents that the vertex has not been visited
- **Grey**: This represents that the vertex has been visited but not explored
- **Black**: This represents that the vertex has been completely explored

This is why we must visit each vertex twice, at the most, as mentioned earlier.

Breadth-first search (BFS)

The BFS algorithm will start traversing the graph from the first specified vertex and will visit all its neighbors (adjacent vertices) first, as it is visiting each layer of the graph at a time. In other words, it visits the vertices first widely and then deeply, as demonstrated by the following diagram:
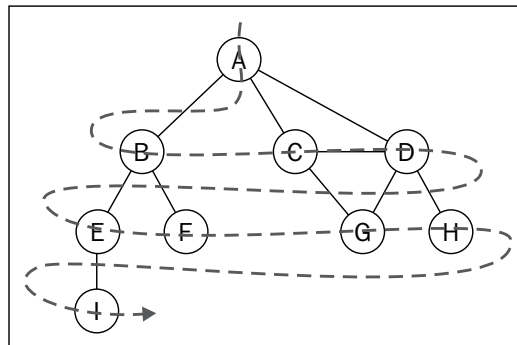
These are the steps followed by the BFS algorithm starting at vertex *v*:

1. Create a queue *Q*.
2. Mark *v* as discovered (grey) and enqueue *v* into *Q*.
3. While *Q* is not empty, perform the following steps:

   1. Dequeue *u* from *Q*.
   2. Mark *u* as discovered (grey).
   3. Enqueue all unvisited (white) neighbors *w* of *u*.
   4. Mark *u* as explored (black).

Let's implement the BFS algorithm:

```
var initializeColor = function(){
    var color = [];
    for (var i=0; i<vertices.length; i++){
        color[vertices[i]] = 'white'; //{1}
    }
    return color;
};

this.bfs = function(v, callback){

    var color = initializeColor(), //{2}
        queue = new Queue();        //{3}
    queue.enqueue(v);               //{4}

    while (!queue.isEmpty()){       //{5}
        var u = queue.dequeue(),        //{6}
            neighbors = adjList.get(u); //{7}
        color[u] = 'grey';                      //{8}
        for (var i=0; i<neighbors.length; i++){ //{9}
            var w = neighbors[i];               //{10}
            if (color[w] === 'white'){          //{11}
                color[w] = 'grey';              //{12}
                queue.enqueue(w);               //{13}
            }
        }
        color[u] = 'black'; //{14}
        if (callback) {     //{15}
            callback(u);
        }
    }
};
```

For both BFS and DFS, we will need to mark the vertices visited. To do so, we will use a helper array called `color`. As and when we start executing the BFS or DFS algorithms, all vertices have the color white (line {1}), so we can create a helper function called `initializeColor`, which will do this for us for both the algorithms we are implementing.

Let's dive into the BFS method implementation. The first thing we will do is use the `initializeColor` function to initialize the `color` array with the color `white` (line {2}). We also need to declare and create a `Queue` instance (line {3}) that will store the vertices that need to be visited and explored.

Following the steps we explained at the beginning of this chapter, the `bfs` method receives a vertex that will be used as the point of origin for our algorithm. As we need a starting point, we will enqueue this vertex into the queue (line {4}).

If the queue is not empty (line {5}), we will remove a vertex from the queue by dequeuing it (line {6}) and we will get its adjacency list that contains all its neighbors (line {7}). We will also mark this vertex as `grey`, meaning we discovered it (but have not finished exploring it yet).

For each neighbor of `u` (line {9}), we will obtain its value (the name of the vertex—line {10}), and if it has not been visited yet (color set to `white`—line {11}), we will mark that we have discovered it (color is set to `grey`—line {12}), and will add this vertex to the queue (line {13}), so it can be finished exploring when we dequeue it from the queue.

When we finish exploring the vertex and its adjacent vertices, we mark is as explored (color is set to `black`—line {14}).

The `bfs` method we are implementing also receives a callback (we used a similar approach in *Chapter 8*, *Trees*, for tree traversals). This parameter is optional, and if we pass any `callback` function (line {15}), we will use it.

Let's test this algorithm by executing the following code:

```
function printNode(value){ //{16}
    console.log('Visited vertex: ' + value); //{17}
}
graph.bfs(myVertices[0], printNode); //{18}
```

First, we declared a `callback` function (line {16}) that will simply output in the browser's console the name (line {17}) of the vertex that was completely explored by the algorithm. Then, we will call the `bfs` method, passing the first vertex (A—from the `myVertices` array that we declared at the beginning of this chapter) and the `callback` function. When we execute this code, the algorithm will output the following result in the browser's console:

```
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
```

As you can see, the order of the vertices visited is the same as shown by the diagram at the beginning of this section.

Finding the shortest paths using BFS

So far, we have only demonstrated how the BFS algorithm works. We can use it for more things than just outputting the order of vertices visited. For example, how would we solve the following problem?

Given a graph *G* and the source vertex *v*, find the distance (in number of edges) from *v* to each vertex *u* *Î G* along the shortest path between *v* and *u*.

Given a vertex *v*, the BFS algorithm visits all vertices with distance 1, and then distance 2, and so on. So, we can use the BFS algorithm to solve this problem. We can modify the `bfs` method to return some information for us:

- The distances *d[u]* from *v* to *u*
- The predecessors *pred[u]*, which is used to derive a shortest path from *v* to every other vertex *u*

Let' see the implementation of an improved BFS method:

```
this.BFS = function(v){

    var color = initializeColor(),
        queue = new Queue(),
        d = [],      //{1}
      pred = []; //{2}
    queue.enqueue(v);

    for (var i=0; i<vertices.length; i++){ //{3}
        d[vertices[i]] = 0;                     //{4}
        pred[vertices[i]] = null;           //{5}
    }

    while (!queue.isEmpty()){
        var u = queue.dequeue(),
            neighbors = adjList.get(u);
        color[u] = 'grey';
        for (i=0; i<neighbors.length; i++){
            var w = neighbors[i];
            if (color[w] === 'white'){
                color[w] = 'grey';
                d[w] = d[u] + 1;            //{6}
              pred[w] = u;                 //{7}
                queue.enqueue(w);
            }
        }
        color[u] = 'black';
    }
    return { //{8}
        distances: d,
        predecessors: pred
    };
};
```

What has changed in this version of the BFS method?

> The source code of this chapter contains two `bfs` methods: `bfs` (the first one we implemented) and `BFS` (the improved one).

We also need to declare the `d` array (line {1}), which represents the distances and the `pred` array (line {2}), which represents the predecessors. The next step would be initializing the `d` array with `0` (zero—line {4}) and the `pred` array with `null` (line {5}) for every vertex of the graph (line {3}).

When we discover the neighbor `w` of a vertex `u`, we set the predecessor value of `w` as `u` (line {7}) and we also increment the distance (line {6}) between `v` and `w` by adding `1` and the distance of `u` (as `u` is a predecessor of `w`, we have the value of `d[u]` already).

At the end of the method, we can return an object with `d` and `pred` (line {8}).

Now, we can execute the `BFS` method again and store its return value in a variable:

```
var shortestPathA = graph.BFS(myVertices[0]);
console.log(shortestPathA);
```

As we executed the `BFS` method for the vertex `A`, this will be the output on the 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"]
```

This means that vertex `A` has distance of `1` edge from vertices `B`, `C`, and `D`; a distance of `2` edges from vertices `E`, `F`, `G`, and `H`; and a distance of `3` edges from vertex `I`.

With the predecessors array, we can build the path from vertex `A` to the other vertices by using the following code:

```
var fromVertex = myVertices[0]; //{9}
for (var i=1; i<myVertices.length; i++){ //{10}
    var toVertex = myVertices[i], //{11}
        path = new Stack();        //{12}
    for (var v=toVertex; v!== fromVertex;
            v=shortestPathA.predecessors[v]) { //{13}
        path.push(v);                          //{14}
    }
    path.push(fromVertex);       //{15}
    var s = path.pop();          //{16}
    while (!path.isEmpty()){      //{17}
        s += ' - ' + path.pop(); //{18}
    }
    console.log(s); //{19}
}
```

We will use vertex A as the source vertex (line {9}). For every other vertex (except vertex A—line {10}), we will calculate the path from vertex A to it. To do so, we will get the value of the toVertex method from the vertices array (line {11}), and we will create a stack to store the path values (line {12}).

Next, we will follow the path from toVertex to fromVertex (line {13}). The v variable will receive the value of its predecessor and we will be able to take the same path backwards. We will add the v variable to the stack (line {14}). Finally, we will add the origin vertex to the stack as well (line {15}) to have the complete path.

After this, we will create an s string and we will assign the origin vertex to it (this will be the last vertex added to the stack, so it is the first item to be popped out—line {16}). While the stack is not empty (line {17}), we will remove an item from the stack and will concatenate it to the existing value of the s string (line {18}). Finally (line {19}), we simply output the path on the browser's console.

After executing the previous code, we will get the following output:

```
A - B
A - C
A - D
A - B - E
A - B - F
A - C - G
A - D - H
A - B - E - I
```

Here, we have the shortest path (in number of edges) from A to the other vertices of the graph.

Further studies on the shortest paths algorithms

The graph we used in this example is not a weighted graph. If we want to calculate the shortest path in weighted graphs (for example, what the shortest path is between city A and city B—an algorithm used in GPS and Google Maps), BFS is not the indicated algorithm.

There is **Dijkstra's algorithm**, which solves the single-source shortest path problem for example. The **Bellman–Ford algorithm** solves the single-source problem if edge weights are negative. The **A* search algorithm** provides the shortest path for a single pair of vertices using heuristics to try to speed up the search. The **Floyd–Warshall algorithm** provides the shortest path for all pairs of vertices.

As mentioned on the first page of this chapter, the subject of graphs is an extensive topic, and we have many solutions for the shortest path problem and its variations. But before we start studying these other solutions, you need to learn the basic concepts of graphs, which we covered in this chapter. These other solutions will not be covered in the book, but you can have an adventure of your own exploring the amazing graph world.

Depth-first search (DFS)

The DFS algorithm will start traversing the graph from the first specified vertex and will follow a path until the last vertex of this path is visited, and then will backtrack and then follow the next path. In another words, it visits the vertices first deeply and then widely, as demonstrated in the following diagram:
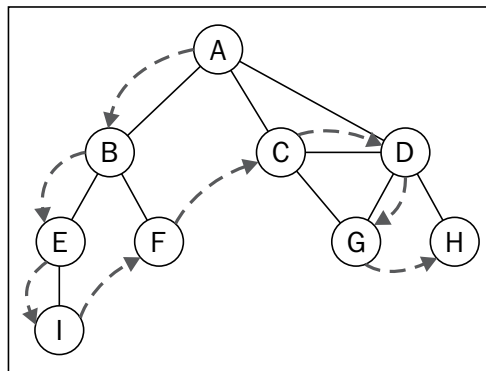
The DFS algorithm does not need a source vertex. In the DFS algorithm, for each unvisited vertex *v* in graph *G*, visit the vertex *v*.

To visit vertex *v*, do the following:

1. Mark *v* as discovered (grey).
2. For all unvisited (white) neighbors *w* of *v*:
    1. Visit vertex *w*.
3. Mark *v* as explored (black).

As you can see, the DFS steps are recursive, meaning the DFS algorithm uses a stack to store the calls (a stack created by the recursive calls).

Let's implement the DFS algorithm:

```
this.dfs = function(callback){
    var color = initializeColor(); //{1}

    for (var i=0; i<vertices.length; i++){ //{2}
        if (color[vertices[i]] === 'white'){ //{3}
            dfsVisit(vertices[i], color, callback); //{4}
        }
    }
};

var dfsVisit = function(u, color, callback){
    color[u] = 'grey'; //{5}
    if (callback) {     //{6}
        callback(u);
    }
    var neighbors = adjList.get(u);         //{7}
    for (var i=0; i<neighbors.length; i++){ //{8}
        var w = neighbors[i];               //{9}
        if (color[w] === 'white'){          //{10}
            dfsVisit(w, color, callback);   //{11}
        }
    }
    color[u] = 'black'; //{12}
};
```

The first thing we need to do is create and initialize the `color` array (line {1}) with the value `white` for each vertex of the graph. We did the same thing for the BFS algorithm. Then, for each non-visited vertex (lines {2} and {3}) of the `Graph` instance, we will call the recursive private function `dfsVisit`, passing the vertex, the `color` array, and the `callback` function (line {4}).

Whenever we visit the `u` vertex, we mark it as discovered (`grey`—line {5}). If there is a `callback` function (line {6}), we will execute it to output the vertex visited. Then, the next step is getting the list of neighbors of vertex `u` (line {7}). For each unvisited (color `white`—lines {10} and {8}) neighbor `w` (line {9}) of `u`, we will call the `dfsVisit` function, passing `w` and the other parameters (line {11}—add the vertex `w` to the stack so it can be visited next). At the end, after the vertex and its adjacent vertices were visited deeply, we **backtrack**, meaning the vertex is completely explored, and is marked `black` (line {12}).

Let's test the `dfs` method by executing the following code:

```
graph.dfs(printNode);
```

This will be its output:

```
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
```

The order is the same as demonstrated by the diagram at the beginning of this section. The following diagram demonstrates the step-by-step process of the algorithm:
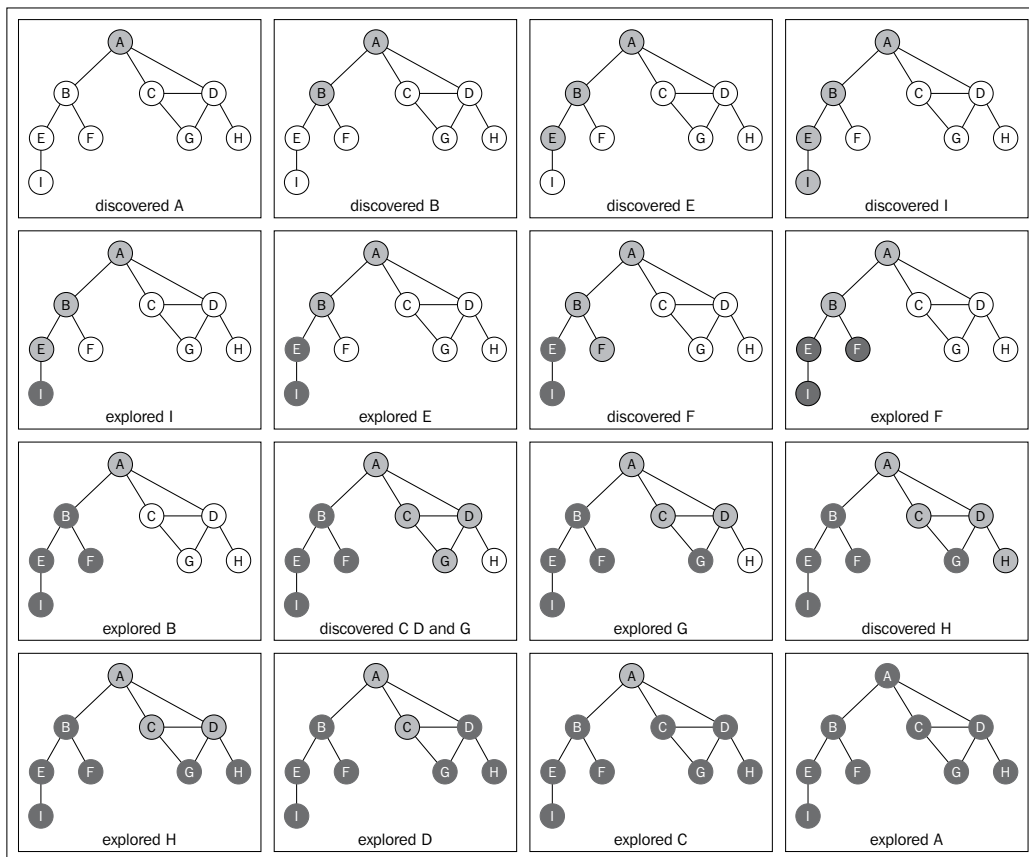
In this graph that we used as an example, line {4} will be executed only once, because all the other vertices have a path to the first one that called the dfsVisit function (vertex A). If vertex B is the first one to call the function, then line {4} would be executed again for another vertex (for example, vertex A).

Exploring the DFS algorithm

So far, we have only demonstrated how the DFS algorithm works. We can use it for more functions than just outputting the order of vertices visited.

Given a graph *G*, the DFS algorithm traverses all vertices of *G* and constructs a forest (a collection of **rooted trees**) together with a set of source vertices (**roots**), and outputs two arrays: discovery time and finish explorer time. We can modify the dfs method to return some information for us:

- The discovery time *d[u]* of *u*
- The finish time *f[u]* when *u* is marked black
- The predecessors *p[u]* of u

Let's see the implementation of the improved BFS method:

```
var time = 0; //{1}
this.DFS = function(){
    var color = initializeColor(), //{2}
        d = [],
        f = [],
        p = [];
    time = 0;

    for (var i=0; i<vertices.length; i++){ //{3}
        f[vertices[i]] = 0;
        d[vertices[i]] = 0;
        p[vertices[i]] = null;
    }
    for (i=0; i<vertices.length; i++){
        if (color[vertices[i]] === 'white'){
            DFSVisit(vertices[i], color, d, f, p);
        }
    }
    return {              //{4}
        discovery: d,
        finished: f,
        predecessors: p
    };
```

```
    };

    var DFSVisit = function(u, color, d, f, p){
        console.log('discovered ' + u);
        color[u] = 'grey';
        d[u] = ++time;      //{5}
        var neighbors = adjList.get(u);
        for (var i=0; i<neighbors.length; i++){
            var w = neighbors[i];
            if (color[w] === 'white'){
                p[w] = u;                           //{6}
                DFSVisit(w,color, d, f, p);
            }
        }
        color[u] = 'black';
        f[u] = ++time;      //{7}
        console.log('explored ' + u);
    };
```

As we want to track the time of discovery and time when we finished exploring, we need to declare a variable to do it (line {1}). We cannot pass time as a parameter because variables that are not objects cannot be passed as a reference to other JavaScript methods (passing a variable as a reference means that if this variable is modified inside the other method, the new values will also be reflected in the original variable). Next, we will declare the d, f, and p arrays too (line {2}). We also need to initialize these arrays for each vertex of the graph (line {3}). At the end of the method, we will return these values (line {4}) so we can work with them later.

When a vertex is first discovered, we track its discovery time (line {5}). When it is discovered as an edge from u, we also keep track of its predecessor (line {6}). At the end, when the vertex is completely explored, we track its finish time (line {7}).

What is the idea behind the DFS algorithm? The edges are explored out of the most recently discovered vertex *u*. Only edges to non-visited vertices are explored. When all edges of *u* have been explored, the algorithm backtracks to explore other edges where the vertex *u* was discovered. The process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then we repeat the process for a new source vertex. We repeat the algorithm until all vertices from the graph are explored.

There are two things we need to check for the improved DFS algorithm:

- The `time` variable can only have values from one to two times the number of vertices of the graph (`2|V|`)
- For all vertices `u`, *d[u] < f[u]* (meaning the discovered time needs to have a lower value than the finish time—meaning all the vertices have been explored)

With these two assumptions, we have the following rule:

$$1 \le d[u] < f[u] \le 2\,|\,V\,|$$

If we run the new DFS method for the same graph again, we will get the following discovery/finish time for each vertex of the graph:
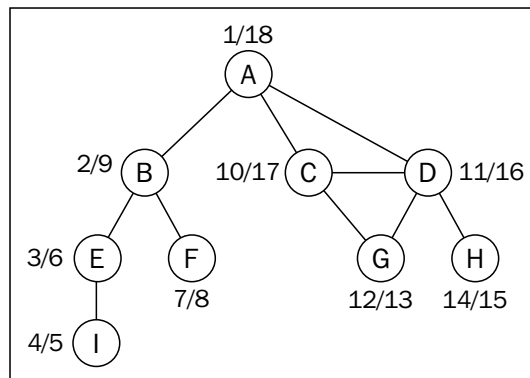
But what can we do with this information? Let's see in the following section.

Topological sorting using DFS

Given the following graph, suppose each vertex is a task that you need to execute:
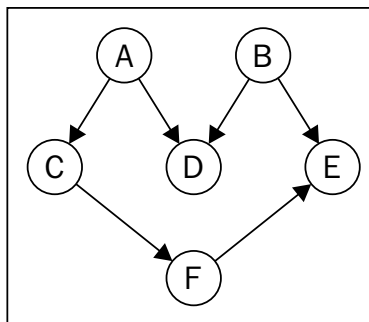
> This is a directed graph, meaning there is an order that the tasks need to be executed. For example, task *F* cannot be executed before task *A*. Note that the previous graph also does not have a cycle, meaning it is an acyclic graph. So, we can say that the previous graph is a **directed acyclic graph** (**DAG**).

When we need to specify the order that some tasks or steps need to be executed in, it is called **topological sorting** (or **topsort** or even **toposort**). This problem is present in different scenarios of our lives. For example, when we start a Computer Science course, there is an order of disciplines we can take before taking any other discipline (you cannot take Algorithms II before taking Algorithms I). When we are working in a development project, there are some steps that need to be executed in order, for example, first we need to get the requirements from the client, then develop what was asked by the client, and then deliver the project. You cannot deliver the project and after that gather the requirements.

Topological sorting can only be applied to DAGs. So, how can we use topological sorting using DFS? Let's execute the DFS algorithm for the diagram presented at the beginning of this topic:

```
graph = new Graph();
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');
var result = graph.DFS();
```

This code will create the graph, apply the edges, execute the improved DFS algorithm, and store the results inside the `result` variable. The following diagram demonstrates the discovery and finish times of the graph after DFS is executed:
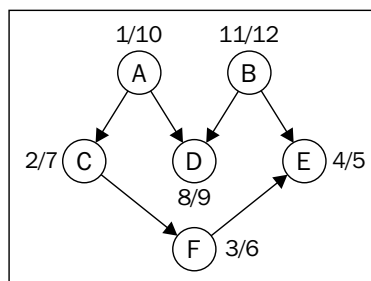
Now all we have to do is sort the finishing time array in decreased order of finishing time and we have the topological sorting for the graph:

```
B - A - D - C - F - E
```

Note that the previous toposort result is only one of the possibilities. There might be different results if we modify the algorithm a little bit, for example, the following result is one of the many other possibilities:

```
A - B - C - D - F - E
```

This could also be an acceptable result.

Summary

In this chapter, we covered the basic concepts of graphs. We learned the different ways we can represent this data structure and we implemented an algorithm to represent a graph using adjacency list. You also learned how to traverse a graph using BFS and DFS approaches. This chapter also covered two applications of BFS and DFS, which are finding the shortest path using BFS and topological sorting using DFS.

In the next chapter, you will learn the most common sorting algorithms used in Computer Science.

10
Sorting and Searching Algorithms

Suppose we have a telephone agenda (or a notebook) that does not have any sorting order. When you need to add a contact with telephone numbers, you simply write it down in the next available slot. Suppose you also have a high number of contacts in your contact list. On any ordinary day, you need to find a particular contact and their telephone numbers. But as the contact list is not organized in any order, you have to check it contact by contact until you find the desired one. This approach is horrible, don't you agree? Imagine that you have to search for a contact in the *Yellow Pages* and it is not organized! It could take forever!

For this reason, among others, we need to organize sets of information, such as the information we have stored in data structures. Sorting and searching algorithms are widely used in daily problems we have to solve. In this chapter, you will learn about the most commonly used sorting and searching algorithms.

Sorting algorithms

From this introduction, you should understand that you need to learn how to sort first and then search the information given. In this section, we will cover some of the most well-known sorting algorithms in Computer Science. We will start with the slowest one, and then we will cover some better algorithms.

Before we get started with the sorting algorithms, let's create an `array` (list) to represent the data structure we want to sort and search:

```
function ArrayList(){

    var array = []; //{1}

    this.insert = function(item){ //{2}
        array.push(item);
    };

    this.toString= function(){ //{3}
        return array.join();
    };
}
```

As you can see, the `ArrayList` is a simple data structure that will store the items in an array ({1}). We only have an `insert` method to add elements to our data structure ({2}), which simply uses the native `push` method of the JavaScript `Array` class that we covered in *Chapter 2*, *Arrays*. Finally, to help us verify the result, the `toString` method ({3}) will concatenate all the array's elements into a single string so we can easily output the result in the browser's console by using the `join` method from the native JavaScript `Array` class.

> The `join` method joins the elements of an array into a string and returns the string.

Note that this `ArrayList` class does not have any method to remove data or insert it into specific positions. We want to keep it simple so we can focus on the sorting and searching algorithms. We will add all the sorting and searching methods to this class.

Now we can get started!

Bubble sort

When people first start learning sorting algorithms, they usually learn the bubble sort algorithm first, because it is the simplest of all the sorting algorithms. However, it is one of the worst-case sorting algorithms with respect to runtime, and you will see why.

The bubble sort algorithm compares every two adjacent items and swaps them if the first one is bigger than the second one. It has this name because the items tend to move up into the correct order like bubbles rising to the surface.

Let's implement the bubble sort algorithm:

```
this.bubbleSort = function(){
    var length = array.length;          //{1}
    for (var i=0; i<length; i++){       //{2}
        for (var j=0; j<length-1; j++ ){ //{3}
            if (array[j] > array[j+1]){  //{4}
                swap(j, j+1);            //{5}
            }
        }
    }
};
```

First, let's declare a variable called `length`, which will store the size of the `array` ({1}). This step will help us to get the size of the `array` on {2} and {3}, and this step is optional. Then, we will have an outer loop ({2}) that will iterate the array from its first position to the last one, controlling how many passes are done in the array (should be one pass per item of the array, as the number of passes is equal to the size of the array). Then, we have an inner loop ({3}) that will iterate the array starting from its first position to the penultimate item that will actually do the comparison between the current item and the next one ({4}). If the items are out of order (the current one is bigger than the next one), then we swap them ({5}), meaning that the value of position `j+1` will be transferred to position `j` and vice versa.

Now we need to declare the `swap` function (a private function that is available only to the code inside the `ArrayList` class):

```
var swap = function(index1, index2){
    var aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
};
```

To make the swap, we need a temporary variable to store the value of one of the items in. We will use this method for other sorting methods as well, and this is the reason we declare this swap code into a function so that we can reuse it.

The following diagram illustrates the bubble sort in action:
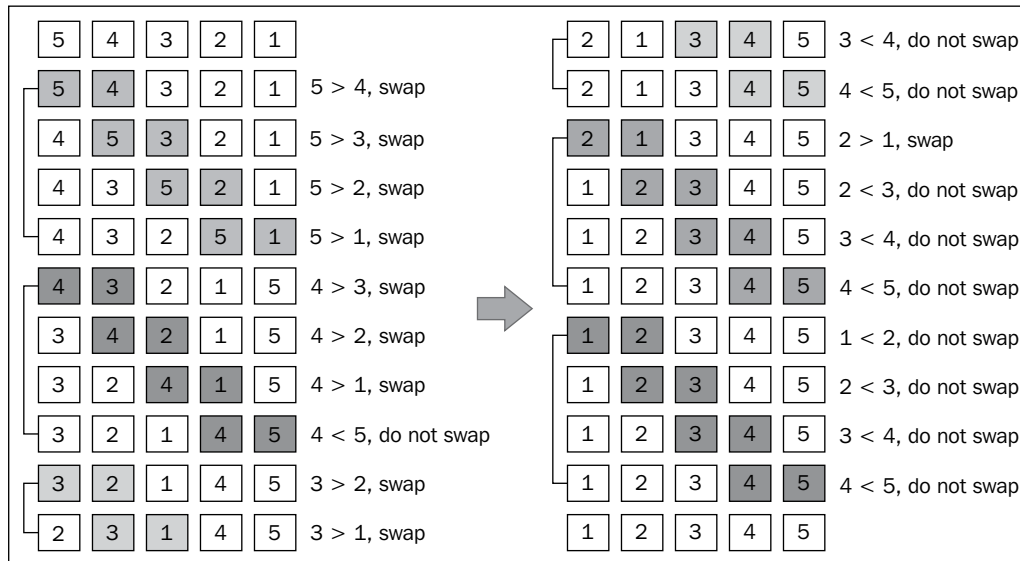
Each different section in the preceding diagram represents a pass made by the outer loop ({2}), and each comparison between two adjacent items is made by the inner loop ({3}).

To test the bubble sort algorithm and get the same results shown by the diagram, we are going to use the following code:

```
function createNonSortedArray(size){ //{6}
    var array = new ArrayList();
    for (var i = size; i> 0; i--){
        array.insert(i);
    }
  return array;
}

var array = createNonSortedArray(5); //{7}
console.log(array.toString());        //{8}
array.bubbleSort();                   //{9}
console.log(array.toString());        //{10}
```

To help us test the sorting algorithms you will learn in this chapter, we are going to create a function that will automatically create a non-sorted array with the size that is passed by the parameter (`{6}`). If we pass `5` as parameter, the function will create the following array for us: `[5, 4, 3, 2, 1]`. Then, all we have to do is call this function and store its return value in a variable that will contain the instance of the `ArrayList` class initialized with some numbers (`{7}`). Just to make sure we have an unsorted array, we will output the array's content on `console` (`{8}`), call the bubble sort method (`{9}`), and output the array's content on `console` again so we can verify that the array was sorted (`{10}`).
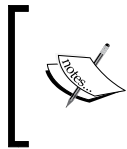
> You can find the complete source code of the `ArrayList` class and the testing code (with additional comments) on the source code that you downloaded from the support page (or from the GitHub repository).

Note that when the algorithm executes the second pass of the outer loop (the second section of the the previous diagram), the numbers 4 and 5 are already sorted. Nevertheless, on the subsequent comparisons, we keep comparing them even the comparison it is not needed. For this reason, we make a small improvement on the bubble sort algorithm.

Improved bubble sort

If we subtract the number of passes from the inner loop, we will avoid all the unnecessary comparisons done by the inner loop (`{1}`):

```
this.modifiedBubbleSort = function(){
    var length = array.length;
    for (var i=0; i<length; i++){
        for (var j=0; j<length-1-i; j++ ){ //{1}
            if (array[j] > array[j+1]){
                swap(j, j+1);
            }
        }
    }
};
```

The following diagram exemplifies how the improved bubble sort works:
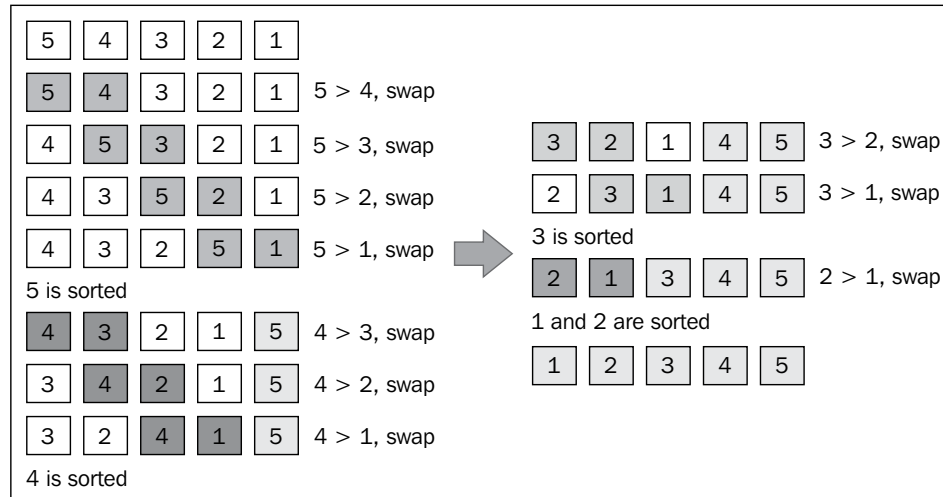
> Note that we do not compare the numbers that are already in place. Even though we have made this small change to improve the bubble sort algorithm a little bit, it is not a recommended algorithm. It has a complexity of $O(n^2)$.

We will talk more about the big O notation in the next chapter.

Selection sort

The selection sort algorithm is an in-place comparison sort algorithm. The general idea of the selection sort is to find the minimum value in the data structure and place it in the first position, then find the second minimum value and place it in the second position, and so on.

The following is the source code for the selection sort algorithm:

```
this.selectionSort = function(){
    var length = array.length,          //{1}
        indexMin;
    for (var i=0; i<length-1; i++){     //{2}
        indexMin = i;                   //{3}
        for (var j=i; j<length; j++){   //{4}
            if(array[indexMin]>array[j]){ //{5}
```

```
                indexMin = j;              //{6}
            }
        }
        if (i !== indexMin){               //{7}
            swap(i, indexMin);
        }
    }
};
```

First, we declare some variables we are going to use in the algorithm ({1}). Then, we have an outer loop ({2}) that will iterate the array and control the passes (which n^{th} value of the array we need to find next—the next min value). We assume that the first value of the current pass is the minimum value of the array ({3}). Then, starting from the current i value to the end of the array ({4}), we compare whether the value in the position j is less than the current minimum value ({5}); if true, we change the value of the minimum to the new minimum value ({6}). When we get out of the inner loop ({4}), we will have the nth minimum value of the array. Then, if the minimum value is different from the original minimum value ({7}) we swap them.

To test the selection sort algorithm, we can use the following code:

```
array = createNonSortedArray(5);
console.log(array.toString());
array.selectionSort();
console.log(array.toString());
```

The following diagram exemplifies the selection sort algorithm in action, based on our array that is used in the preceding code [5, 4, 3, 2, 1]:
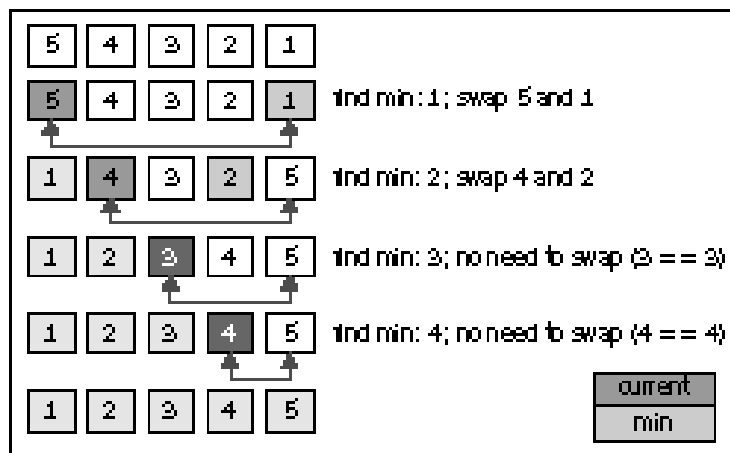
The arrows on the bottom of the array indicate the positions currently in consideration to find the minimum value (inner loop {4}), and each step of the preceding diagram represents the outer loop ({2}).

The selection sort is also an algorithm of complexity $O(n^2)$. Like the bubble sort, it contains two nested loops, which are responsible for the quadratic complexity. However, the selection sort performs worse than the insertion sort algorithm you will learn next.

Insertion sort

The insertion sort algorithm builds the final sorted array one item at a time. It assumes that the first element is already sorted. Then, a comparison with the second item is performed—should the second item stay in its place or be inserted before the first item? So, the first two items get sorted and the comparison takes place with the third item (should it be inserted in the first, second, or third position?), and so on.

The following code represents the insertion sort algorithm:

```
this.insertionSort = function(){
    var length = array.length,          //{1}
        j, temp;
    for (var i=1; i<length; i++){       //{2}
        j = i;                          //{3}
        temp = array[i];                //{4}
        while (j>0 && array[j-1] > temp){ //{5}
            array[j] = array[j-1];      //{6}
            j--;
        }
        array[j] = temp;                //{7}
    }
};
```

As usual, the first line of the algorithm is used to declare the variables we will use in the source code ({1}). Then, we will iterate the array to find the correct place for the ith item ({2}). Note that we start from the second position (index 1), instead of position 0 (we consider the first item already sorted). Then, we will initiate an auxiliary variable with the value of i ({3}), and we will also store the value in a temporary value ({4}) so we can insert it in the correct position later. The next step is finding the correct place to insert the item. As long the j variable is bigger than 0 (because the first index of the array is 0—there is no negative index) and the previous value in the array is bigger than the value we are comparing ({5}), we shift the previous value to the current position ({6}) and decrease j. At the end, we will insert the value in its correct position.

The following diagram exemplifies the insertion sort in action:

3	5	1	4	2		1	3	?	5	2	5 > 4, shift
3	5	1	4	2	5 to be inserted	1	3	4	5	2	3 < 4, insert 4
3	5	1	4	2	3 < 5, insert 5	1	3	4	5	2	2 to be inserted
3	5	1	4	2	1 to be inserted	1	3	4	?	5	5 > 2, shift
3	?	5	4	2	5 > 1, shift	1	3	?	4	5	4 > 2, shift
?	3	5	4	2	3 > 1, shift	1	?	3	4	5	3 > 2, shift
1	3	5	4	2	reached index 0, insert 1	1	2	3	4	5	1 < 2, insert 2
1	3	5	4	2	4 to be inserted	1	2	3	4	5	
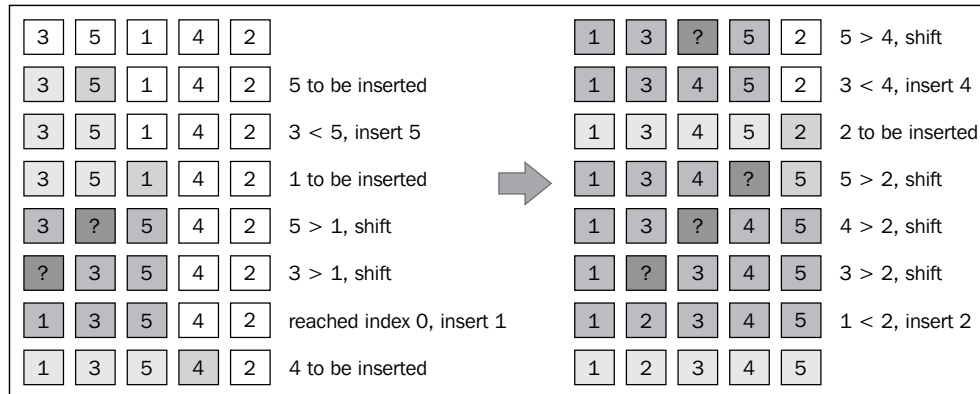
For example, suppose the array we are trying to sort is `[3, 5, 1, 4, 2]`. These values will be carried out in the steps performed by the insertion sort algorithm, as described in the following steps:

1. Value 3 is already sorted, so we start sorting the second value of the array, which is value 5. Value 3 is less than value 5, so 5 stays in the same place (meaning the second position of the array). Values 3 and 5 are already sorted.

2. The next value to be sorted and inserted in the correct place is 1 (currently in the third position of the array). 5 is greater than 1, so 5 is shifted to the third position. We need to analyze whether 1 should be inserted in the second position—is 1 greater than 3? No, so value 3 gets shifted to second position. Next, we need to verify that 1 should be inserted in the first position of the array. As 0 is the first position and there is not a negative position, 1 needs to be inserted on the first position. Values 1, 3, and 5 are sorted.

3. We move to the next value: 4. Should value 4 stay in the current position (index 3) or does it need to be moved to a lower position? 4 is less than 5, so 5 gets shifted to index 3. Should we insert 4 in the index 2? Value 4 is greater than 3, so 4 is inserted in position 3 of the array.

4. The next value to be inserted is 2 (position 4 of array). Value 5 is greater than 2, so 5 gets shifted to index 4. Value 4 is greater than 2, so 4 also gets shifted (position 3). Value 3 is also greater than 2, and 3 also gets shifted. 1 is less than 2, so 2 is inserted on the second position of the array. Thus, the array is sorted.

This algorithm has a better performance than the selection and bubble sort algorithms when sorting small arrays.

Merge sort

The merge sort algorithm is the first sorting algorithm that can be used in the real world. The three first sorting algorithms you learned in this book do not give a good performance, but the merge sort gives a good performance, with a complexity of *O(n log n)*.

> The JavaScript `Array` class defines a `sort` function (`Array.prototype.sort`) that can be used to sort arrays using JavaScript (with no need to implement the algorithm ourselves). ECMAScript does not define which sorting algorithm needs to be used, so each browser can implement its own algorithm. For example, Mozilla Firefox uses merge sort as the `Array.prototype.sort` implementation, while Chrome uses a variation of quick sort (which we will learn next).

The merge sort is a divide and conquer algorithm. The idea behind it is to divide the original array into smaller arrays until each small array has only one position and then merge these smaller arrays into bigger ones until we have a single big array at the end that is sorted.

Because of the divide and conquer approach, the merge sort algorithm is also recursive:

```
this.mergeSort = function(){
    array = mergeSortRec(array);
};
```

Like in the previous chapters, whenever we implemented a recursive function, we always implemented a helper function that was going to be executed. For the merge sort, we will do the same. We are going to declare the `mergeSort` method that will be available for use and the `mergeSort` method will call `mergeSortRec`, which is a recursive function:

```
var mergeSortRec = function(array){
    var length = array.length;
    if(length === 1) {        //{1}
        return array;         //{2}
    }
    var mid = Math.floor(length / 2),     //{3}
        left = array.slice(0, mid),       //{4}
        right = array.slice(mid, length); //{5}

    return merge(mergeSortRec(left), mergeSortRec(right)); //{6}
};
```

The merge sort will transform a bigger array into smaller arrays until they have only one item in them. As the algorithm is recursive, we need a stop condition, which is if the array has size equal to 1 ({1}). If positive, we return the array with size 1 ({2}) because is already sorted.

If the array is bigger than 1, then we will split it into smaller arrays. To do so, first we need to find the middle of the array ({3}), and once we find the middle, we will split the array into two smaller arrays that we will call left ({4}) and right ({5}). The left array comprises of elements from index 0 to the middle index, and the right array consists from the middle index to the end of the original array.

The next steps will be to call the merge function ({6}), which will be responsible for merging and sorting the smaller arrays into bigger ones until we have the original array sorted and back together. To keep breaking the original array into smaller pieces, we will recursively call mergeSortRec again, passing the left smaller array as a parameter and another call for the right array:

```
var merge = function(left, right){
    var result = [], // {7}
        il = 0,
        ir = 0;

    while(il < left.length && ir < right.length) { // {8}
        if(left[il] < right[ir]) {
            result.push(left[il++]);  // {9}
        } else{
            result.push(right[ir++]); // {10}
        }
    }

    while (il < left.length){    // {11}
        result.push(left[il++]);
    }

    while (ir < right.length){   // {12}
        result.push(right[ir++]);
    }

    return result; // {13}
};
```

The `merge` function receives two arrays and merges them into a bigger array. During the merge is when the sorting happens. First, we need to declare a new array that is going to be created for the merge and also declare two variables ({7}) that are going to used to iterate the two arrays (the `left` and `right` arrays). While we can iterate through the two arrays {8}, we are going to compare whether the item from the `left` array is less than the item from the `right` array. If positive, we add the item from the `left` array to the merged result array and also increment the variable that is used to iterate the array ({9}); otherwise, we add the item from the `right` array and increment the variable that is used to iterate the array ({10}). Next, we will add every remaining item from the `left` array ({11}) to the merged result array and do the same for the remaining items from the `right` array ({12}). At the end, we return a merged array as the result ({13}).

If we execute the `mergeSort` function, this is how it is going to be executed:
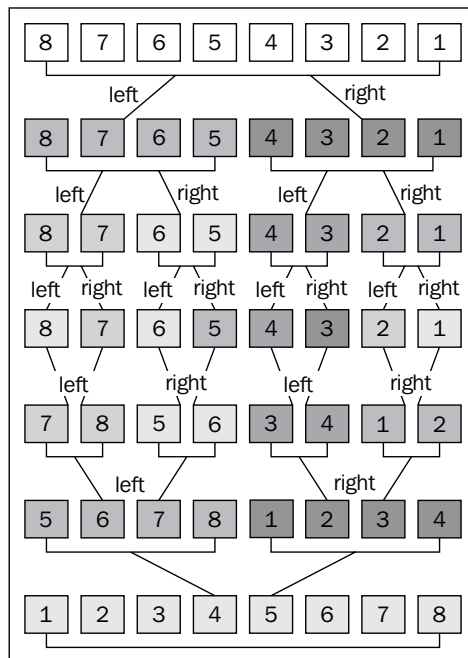
Note that first the algorithm splits the original array until it has smaller arrays with a single element, and then it starts merging. While merging, it does the sorting as well until we have the original array completely back together and sorted.

Quick sort

The quick sort is probably the most used sorting algorithm. It has a complexity of *O(n log n)*, and it usually performs better than other *O(n log n)* sorting algorithms. Like the merge sort, it also uses the divide and conquer approach, dividing the original array into smaller ones (but without splitting them as the merge sort does) to do the sorting.

The quick sort algorithm is a little bit more complex than the other ones you have learned so far. Let's learn it step by step as follows:

1. First, we need to select an item from the array that is called as a pivot, which is the middle item in the array.

2. We will create two pointers—the left one will point to the first item of the array and the right one will point to the last item of the array. We will move the left pointer until we find an item that is bigger than the pivot and we will also move the right pointer until we find an item that is less than the pivot and we will swap them. We repeat this process until the left pointer passes the right pointer. This process helps to have values lower than the pivot before the pivot and values greater than the pivot after the pivot. This is called the partition operation.

3. Next, the algorithm repeats the previous two steps for smaller arrays (sub-arrays with smaller values, and then sub-arrays with greater values) until the array is completely sorted.

Let's start the implementation of the quick sort:

```
this.quickSort = function(){
    quick(array,  0, array.length - 1);
};
```

Like the merge sort, we will start declaring the main method that will call the recursive function, passing the array that we want to sort along with indexes 0 and its last position (because we want to have the whole array sorted, not only a subset of it):

```
var quick = function(array, left, right){

    var index; //{1}

    if (array.length > 1) { //{2}

        index = partition(array, left, right); //{3}
```

```
        if (left < index - 1) {                  //{4}
            quick(array, left, index - 1);     //{5}
        }

        if (index < right) {                  //{6}
            quick(array, index, right);       //{7}
        }
    }
};
```

First we have the `index` variable (`{1}`), which will help us to separate the sub-array with smaller and greater values so we can recursively call the `quick` function again. We will obtain the `index` value as return value of the `partition` function (`{3}`).

If the size of the array is bigger than 1 (because an array with a single element is already sorted—line `{2}`), we will execute the partition operation on the given sub-array (the first call will pass the complete array) to obtain `index` (`{3}`). If a sub-array with smaller elements exists (`{4}`), we will repeat the process for the sub-array (`{5}`). We will do the same thing for the sub-array with greater values. If there is any sub-array with greater values (`{6}`), we will repeat the quick sort process (`{7}`) as well.

The partition process

Now, let's take a look at the `partition` process:

```
var partition = function(array, left, right) {

    var pivot = array[Math.floor((right + left) / 2)], //{8}
        i = left,                                       //{9}
        j = right;                                      //{10}

    while (i <= j) {                //{11}
        while (array[i] < pivot) {  //{12}
            i++;
        }
        while (array[j] > pivot) {  //{13}
            j--;
        }
        if (i <= j) { //{14}
            swapQuickStort(array, i, j); //{15}
            i++;
```

```
            j--;
        }
    }
    return i; //{16}
};
```

The first thing we need to do is choose the `pivot` element. There are a few ways we can do it. The simplest one is selecting the first item of the array (the leftmost item). However, studies show that this is not a good selection if the array is almost sorted, causing the worst behavior of the algorithm. Another approach is selecting a random item of the array or the middle item. For this implementation, we will select the middle item as `pivot` ({8}). We will also initiate the two pointers: `left` (low — line {9}) with the first element of the array and `right` (high — line {10}) with the last element of the array.

While the `left` and `right` pointers do not cross each other ({11}), we will execute the partition operation. First, until we find an element that is greater than `pivot` ({12}), we will shift the `left` pointer. We will do the same with the `right` pointer until we find an element that is less than `pivot`, we will shift the `right` pointer as well ({13}).

When the `left` pointer is greater than the `pivot` and the `right` pointer is lower than the `pivot`, we compare whether the `left` pointer index is not bigger than the `right` pointer index ({14}), meaning the left item is greater than the right item (in value). We swap them ({15}) and shift both pointers and repeat the process (start again at line {11}).

At the end of the partition operation, we return the index of the left pointer that will be used to create the sub-arrays in line {3}.

The `swapQuickStort` function is very similar to the `swap` function we implemented at the beginning of this chapter. The only difference is that this one also receives the `array` that will suffer the swap as a parameter:

```
var swapQuickStort = function(array, index1, index2){
    var aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
};
```

Quick sort in action

Let's see the quick sort algorithm in action step by step:

3	5	1	6	4	7	2	
3	5	1	**6**	4	7	2	pivot = 6
L						R	set pointer left (i) and right (j)
3	5	1	**6**	4	7	2	3 < 6, shift left pointer
	L					R	
3	5	1	**6**	4	7	2	5 < 6, shift left pointer
		L				R	
3	5	1	**6**	4	7	2	1 < 6, shift left pointer
			L			R	6 == 6, stop
3	5	1	**6**	4	7	2	6 > 2, stop
			L			R	swap 6 and 2
							shift pointers
3	5	1	2	4	7	**6**	4 < 6, shift left pointer
				L	R		
3	5	1	2	4	7	**6**	7 < 6, stop (left pointer)
					L R		7 < 6, shift right pointer
3	5	1	2	4	7	**6**	4 < 6, stop
				R	L		left > right, stop and start over
							(start from index 0 to index left -1)

Given the array [3, 5, 1, 6, 4, 7, 2], the preceding diagram represents the first execution of the partition operation.

The following diagram exemplifies the execution of the partition operation for the first sub-array of lower values (note that **7** and **6** are not part of the sub-array):

3	5	1	2	4	7	6	pivot = 1
L				R			set pointer left (i) and right (j)
3	5	1	2	4	7	6	3 > 1, stop
L			R				4 > 1, shift right pointer
3	5	1	2	4	7	6	2 > 1, shift right pointer
L		R					
3	5	1	2	4	7	6	1 == 1, stop
L		R					swap 3 and 1
							shift pointers
1	5	3	2	4	7	6	5 > 1, shift right pointer
	LR						
1	5	3	2	4	7	6	1 == 1, stop
R	L						left > right, stop and start over
							(start from index left to index 4)
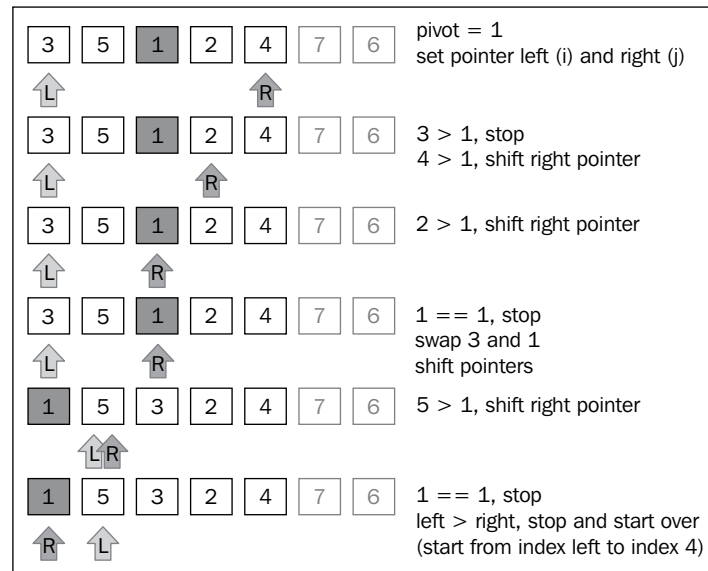
Next, we continue creating sub-arrays, as seen in the following diagram, but now with greater values than the sub-array of the preceding diagram (the lower sub-array with value **1** does not need to be partitioned because it only contains one item):

1	5	3	2	4	7	6	pivot = 3
	L			R			set pointer left (i) and right (j)
1	5	3	2	4	7	6	5 > 3, stop
	L			R			4 > 3, shift right pointer
1	5	3	2	4	7	6	2 < 3, stop
	L		R				swap 5 and 2
							shift pointers
							3 == 3, stop (left pointer)
1	2	3	5	4	7	6	3 == 3, stop (right pointer)
		LR					swap 3 and 3
							shift pointers
1	2	3	5	4	7	6	left > right, stop and start over
	R		L				(start from index right and left -1)
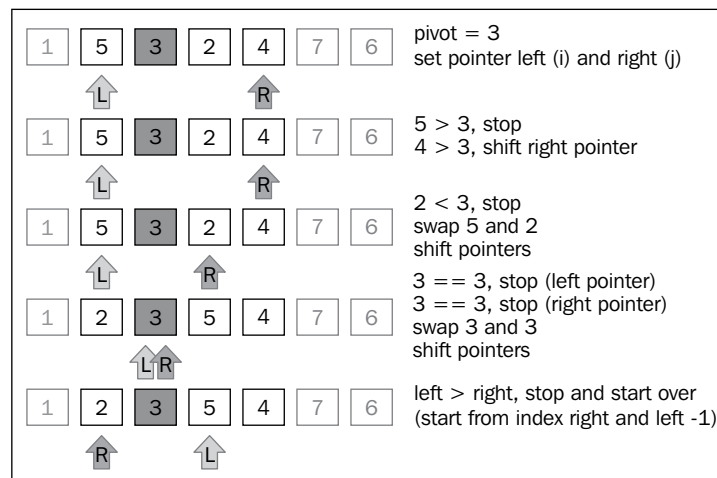
The lower sub-array `[2, 3]` from sub-array (`[2, 3, 5, 4]`) continues to be partitioned (line {5} from the algorithm):
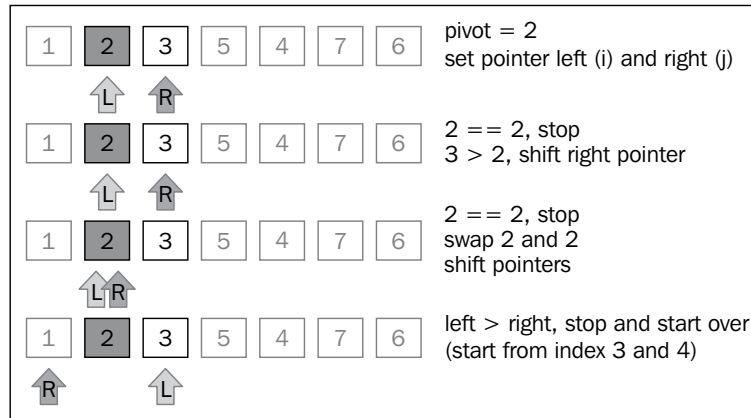
Then the greater sub-array `[5, 4]` from the sub-array `[2, 3, 5, 4]` also continues to be partitioned (line {7} from the algorithm), as shown in the following diagram:
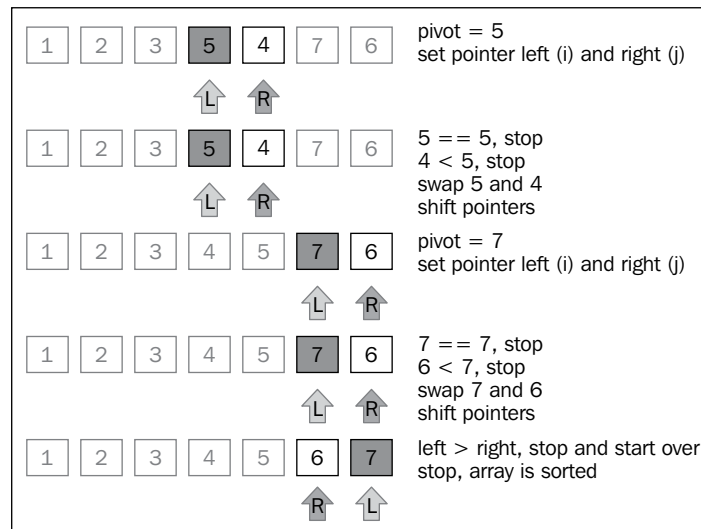
At the end, the greater sub-array `[6, 7]` will also suffer the `partition` operation, completing the execution of the quick sort algorithm.

Searching algorithms

Now, let's talk about searching algorithms. If we take a look at the algorithms we implemented in previous chapters, such as the `search` method of the `BinarySearchTree` class (*Chapter 8*, *Trees*) or the `indexOf` method of the `LinkedList` class (*Chapter 5*, *Linked Lists*), these are all search algorithms, and of course, each one was implemented according to the behavior of its data structure. So we are already familiar with two-searches algorithm, we just do not know their "official" names yet!

Sequential search

The sequential or linear search is the most basic search algorithm. It consists of comparing each element of the data structure with the one we are looking for. It is also the most inefficient one.

Let's take a look at its implementation:

```
this.sequentialSearch = function(item){
    for (var i=0; i<array.length; i++){ //{1}
        if (item === array[i]){          //{2}
            return i;                    //{3}
        }
    }
    return -1;  //{4}
};
```

The sequential search will iterate through the array (`{1}`), and will compare each item with the value we are searching for (`{2}`). If we find it, then we can return something to indicate we found it. We can return the item itself, the value `true`, or its index (`{3}`). In the preceding implementation, we are returning the index of the item. If we don't find the item, we can return `-1` (`{4}`), indicating the index does not exist; the values `false` and `null` are among other options.

Suppose we have the array `[5, 4, 3, 2, 1]` and we are looking for the value 3, then the following diagram shows the steps of the sequential search:
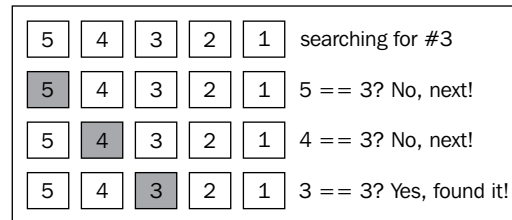
Binary search

The binary search algorithm works similar to the number guessing game, where someone says "I'm thinking of a number between 1 and 100". We begin by responding with a number and the person will say higher, lower, or that we got it right.

To make the algorithm work, the data structure needs to be sorted first. These are the steps that the algorithm follows:

1. A value is selected in the middle of the array.
2. If the item is the one we are looking for, we are done (the value is right).
3. If the value we are looking for is less than the selected one, then we go to the left and go back to 1 (lower).
4. If the value we are looking for is bigger than the selected one, then we go to the right and go back to 1 (higher).

Let's see its implementation:

```
this.binarySearch = function(item){
    this.quickSort();  //{1}

    var low = 0,                 //{2}
        high = array.length - 1, //{3}
        mid, element;

    while (low <= high){ //{4}
        mid = Math.floor((low + high) / 2); //{5}
        element = array[mid];               //{6}
        if (element < item) {               //{7}
```

```
            low = mid + 1;                   //{8}
        } else if (element > item) {         //{9}
            high = mid - 1;                   //{10}
        } else {
            return mid;                      //{11}
        }
    }
    return -1; //{12}
};
```

To get started, the first thing we need to do is sort the array. We can use any algorithm we implemented in the *Sorting algorithms* section. Quick sort was chosen for this implementation ({1}). After the array is sorted, we will set the `low` ({2}) and `high` ({3}) pointer (which will work as boundaries).

While `low` is lower than `high` (line {4}), in this case, `low` is greater than `high` means the value does not exist and we return -1 ({12}), we find the middle index ({5}) and hence have the value of the middle item ({6}). Then, we start comparing whether the selected value is less than the value we are looking for ({7}) and we need to go lower ({8}) and start over. If the selected value is greater than the value we are looking for ({9}) and we need to go higher ({10}) and start over. Otherwise, it means the value is equal to the value we are looking for, therefore we return its index ({11}).

Given the array in the following diagram, let's try to search for value **2**. These are the steps the algorithm will perform:
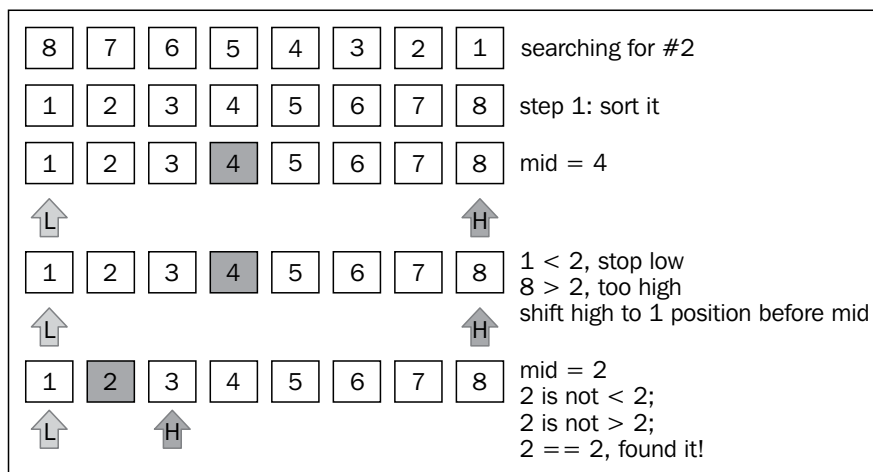
> The `BinarySearchTree` class we implemented in *Chapter 8*, *Trees*, has the `search` method, which is exactly the same as the binary search, but is applied to tree data structures.

Summary

In this chapter, you learned about sorting and searching algorithms. You learned the bubble, selection, insertion, merge, and quick sort algorithms, which are used to sort data structures. You also learned the sequential search and binary search (which required the data structure to be sorted already).

You can apply any logic you learned in this chapter to any data structure or any type of data. You just need to make the necessary modifications on the source code.

In the next chapter, you will learn some advanced techniques used in algorithms and also more about the *big O* notation that was mentioned in this chapter.

Index

N

node
  leaf node, removing 144
  removing 142, 143
  removing, with left/right child 144
  removing, with two children 145
Node.js
  URL 11
Node Packages Modules
  URL 7
null set 91

O

object 27
object-oriented programming (OOP) 27
operators
  about 17
  arithmetic 17
  assignment 18
  bitwise 18
  comparison 18
  delete 19
  logical 18
  typeof 19

P

partition process, quick sort
      algorithm 186, 187
path 150
pop method 51
post-order traversal 137
pre-order traversal 136
printTitle function 28
priority queue
  about 64
  example 65, 66
projects, JavaScript
  reference link 7
push method 51

Q

queue
  about 59
  creating 60, 61

Queue class
  about 62
  using 63, 64
quick sort algorithm
  about 185, 186
  executing 188-191
  partition process 186, 187

R

Red-Black tree
  about 147
  URL 147
reduce method 43
remove method 93, 94, 107
return statement 26
root 128
rooted trees 167

S

script tag 13
searching algorithms
  about 191
  binary search 192, 193
  sequential search 191
searching methods
  about 43, 44
  indexOf method 46
  lastIndexOf method 46
search method 191
selection sort algorithm 178-180
separate chaining
  about 117, 118
  get method 119
  put method 118
  remove method 120, 121
sequential search 191
set
  about 91
  add method 93
  clear method 94
  creating 92
  has(value) method 93
  remove method 94
  Set class, using 96
  size method 95
  values method 96

V

values method 96, 108
variable, value types
  arrays 14
  Booleans 14
  dates 14
  functions 14
  null 14
  numbers 14
  objects 14
  regular expressions 14
  strings 14
  undefined 14
variable scope 16
variables 14
visitor pattern
  URL 134

W

web servers (XAMPP)
  used, for environment setup 10, 11
WebStorm
  about 29
  URL 29
weighted graph 151
while loop 25

X

XAMPP
  URL, for installing 10

Thank you for buying
Learning JavaScript Data Structures and Algorithms

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Ext JS 4 First Look

ISBN: 978-1-84951-666-2   Paperback: 340 pages

A practical guide including examples of the new features in Ext JS 4 and tips to migrate from Ext JS 3

1. Migrate your Ext JS 3 applications easily to Ext JS 4 based on the examples presented in this guide.

2. Full of diagrams, illustrations, and step-by-step instructions to develop real-word applications.

3. Driven by examples and explanations of how things work.
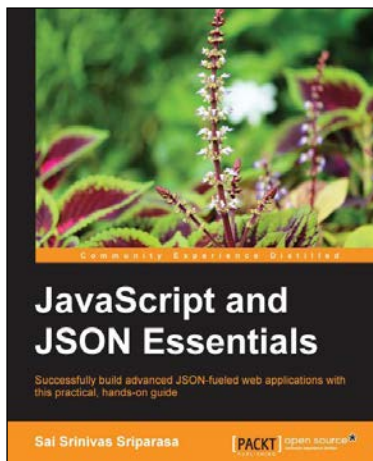
JavaScript and JSON Essentials

ISBN: 978-1-78328-603-4   Paperback: 120 pages

Successfully build advanced JSON-fueled web applications with this practical, hands-on guide

1. Deploy JSON across various domains.

2. Facilitate metadata storage with JSON.

3. Build a practical data-driven web application with JSON.

Please check **www.PacktPub.com** for information on our titles
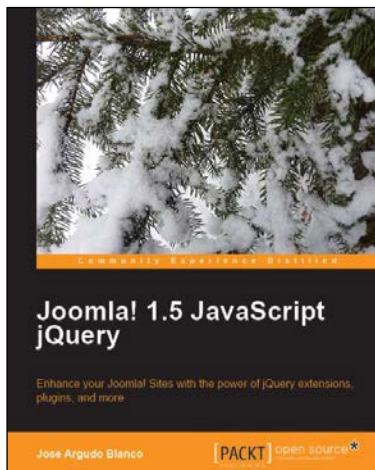
Joomla! 1.5 JavaScript jQuery

ISBN: 978-1-84951-204-6   Paperback: 292 pages

Enhance your Joomla! Sites with the power of jQuery extensions, plugins, and more

1. Build impressive Joomla! Sites with JavaScript and jQuery.

2. Create your own Joomla!, jQuery-powered, extensions.

3. Enhance your site with third-party features, code-highlighting, Flicker, and more using Joomla! plugins.

4. Detailed explanations with step-by-step guidance and practical examples.
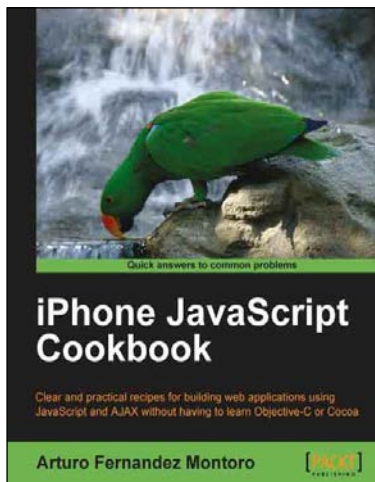
iPhone JavaScript Cookbook

ISBN: 978-1-84969-108-6   Paperback: 328 pages

Clear and practical recipes for building web applications using JavaScript and AJAX without having to learn Objective-C or Cocoa

1. Build web applications for iPhone with a native look feel using only JavaScript, CSS, and XHTML.

2. Develop applications faster using frameworks.

3. Integrate videos, sound, and images into your iPhone applications.

4. Work with data using SQL and AJAX.

Please check **www.PacktPub.com** for information on our titles